SEVENTH FRAMEWORK PROGRAME

Specific Targeted Research Project

Project Number:	**FP7–SMARTCITIES–2013(ICT)**
Project Acronym:	**VITAL**
Project Number:	**608682**
Project Title:	**Virtualized programmable InTerfAces for innovative cost-effective IoT depLoyments in smart cities**

D3.1.2 Virtual Models, Data and Metadata for ICOs V2

Document Id:	VITAL-D312-231015-Draft
File Name:	VITAL-D312-231015-Draft.pdf
Document reference:	Deliverable 3.1.2
Version:	Draft
Editors:	Anne Helmreich, Salma Abdulazis, Zeeshan Jan, Aqeel Kazmi, Martin Serrano
Organisation:	NUI Galway
Date:	23 / 10 / 2015
Document type:	Deliverable
Security:	PU (Public)

DOCUMENT HISTORY

Rev.	Author(s)	Organisation(s)	Date	Comments
V01	Aqeel Kazmi	NUI Galway	12/08/2015	Initial version of the document
V02	Nathalie Mitton	INRIA	21/08/2015	Updated section 6.5
V03	Angelos Lenis	SiLo	06/09/2015	Updated Sections 6.7 and 6.8
V04	Aqeel Kazmi	NUIG	29/09/2015	Updates to various sections
	Paola Dal Zovo, Lorenzo Bracco	REPLY	08/10/2015	Updated Section 6.3
V05	Miguel Montero	ATOS	09/10/2015	Updates to Trust and CEP sections
V06	Aqeel Kazmi	NUIG	09/10/2015	Integration of partner contributions.
	Yusuf Yaslan	ITU	09/10/2015	Section 4.2.3
V07	Aqeel Kazmi	NUIG	10/10/2015	Updates to Section 4 & 5.
V08	Aqeel Kazmi	NUIG	11/10/2015	Updates to Section 6.
V09	Angelos Lenis	SiLo	12/10/2015	Updated Section 6.8.
V10	Aqeel Kazmi	NUIG	12/10/2015	Updated Section 6.
V11	Aqeel Kazmi	NUIG	15/10/2015	Updated JSON-LD contexts & examples.
V12	Katerina Roukounaki	AIT	20/10/15	Technical Review.
V13	Elisa Herrmann	ATOS	20/10/2015	Update to section 6.6
V14	Aqeel Kazmi	NUIG	20/10/2015	Review Comments Addressed.
V15	Salvatore Guzzo Bonifacio	INRIA	21/10/2015	Updated section 6.5
V16	Angelos Lenis	SiLo	21/10/2015	Updated Sections 6.7 and 6.8
V17	Salvatore Guzzo Bonifacio	INRIA	22/10/2015	Minor fixes
V18	Aqeel Kazmi	NUIG	22/10/2015	Minor Fixes and Quality Review
	Aqeel Kazmi	NUIG	22/10/2015	Circulated for Approval
Draft	Martin Serrano	NUIG	23/10/2015	EC Submitted

OVERVIEW OF UPDATES/ENHANCEMENTS OVER D3.1.1

Section	**Description**
Section 1	Updates to the introduction and scope of the second version of this deliverable.
Section 2	Updates to linked data and semantic descriptions.
Section 3	Updates to sensors and measurements.
Section 4	Updates to smart cities section.
Section 5	Updates to IoT systems and services.
Section 6	Updates to VITAL system and services.
Section 7	Updates to conclusion and further directions.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

TERMS AND ACRONYMS

ACL	Access Control List
API	Application programming interface
CEP	Complex Event Processing
COMANTO	Context Management Ontology
COMPOSE	Collaborative Open Market to Place Objects at your Service
CQELS	Continuous Query Evaluation over Linked Stream
CRUD	Create Read Update Delete
DCMI	Dublin Core Metadata Initiative
DCO	Delivery Context Ontology
DOLCE-Lite	Descriptive Ontology for Linguistic and Cognitive Engineering Lite
GEO	Basic Geo Ontology
GPS	Global Positioning System
GSN	Global Sensor Network
hRest	HTML for REST
HTTP	Hypertext Transfer Protocol
ICO	Internet Connected Object
ICT	Information and communication technology
IoT	Internet of Things
IRI	Internationalised Resource Identifier
JSON	JavaScript Object Notation
JSON-LD	JSON for Linked Data
LODE	Ontology for Linking Open Descriptions of Events
MSM	Minimal Service Model
MUO	Measurements Unit Ontology
N3	Notion 3
OAE	Ontology of Adverse Events
OM	Ontology of Units of Measure
OnTraJaCS	Ontology Based Traffic Control Systems
OWL	Web Ontology Language
OWL-S	Web Ontology Language for Web Services
PPI	Platform Provider Interface
PPO	Privacy Preference Ontology
QUDT	Quantities, Units, Dimensions and Data Types Ontologies
RDF	Resource Description Framework
REST	Representational State Transfer

S4AC	Social Semantic SPARQL Security for Access Control
SA-REST	Semantic Annotations for REST
SAWSDL	Semantically Annotated WSDL
SOAP	Simple Object Access Protocol
SOUPA	Standard Ontology for Ubiquitous and Pervasive Applications
SPARQL	SPARQL Protocol and RDF Query Language
SSN	Semantic Sensor Network
SSO	Stimulus-Sensor-Observation pattern
TAC	Triple Access Control
TAO	Trust Assertion Ontology
UO	Units of Measurement Ontology
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UTO	Urban Traffic Ontology
W3C	World Wide Web Consortium
WAC	Web Access Control
WADL	Web API Definition Language
WSDL	Web Service Definition Language
WSMO	Web Service Modelling Ontology

1 INTRODUCTION

This deliverable specifies the data models and ontologies used in the VITAL project. These models and ontologies are used in three different ways. First, they are used by applications to communicate data between them and the VITAL platform, e.g. querying sensor data. Second, they are used by the so-called platform provider interface (PPI), which specifies an interface between VITAL and external IoT systems that should be integrated into VITAL, e.g. a description of the services offered by the IoT system. Third, they are used inside the VITAL platform to communicate data between VITAL services, e.g. security level information, etc.

Due to this diverse set of users (and usage cases), VITAL has to cover a wide area of data models. It has to specify how basic IoT sensors and sensor measurements are modelled, e.g. using the Semantic Sensor Network (SSN) ontology. It has to define how to describe IoT systems and IoT services, e.g. using the Minimal Service Model (MSM). It has to identify data models for Smart City applications, e.g. for smart transport systems. And it has to specify metadata for the VITAL system and its components, e.g. to model security and monitoring information.

In the first version of this deliverable (D3.1.1) we analysed existing ontologies and data models that could be used as the basis of the VITAL data models and extended them as required by VITAL. We subdivided the work into different areas – sensors and sensor measurements, Smart Cities, IoT systems and services, VITAL systems and services. For each of these areas we discussed which ontologies were used for modelling the required data (and why), presented necessary additional data items and how to model them, and showed examples of the resulting descriptions. The present version (D3.1.2) of the deliverable is an enhancement to the first version (D3.1.1). In this version of the deliverable (D3.1.2) we present fine-tuned data models, incorporate practical experiences and new requirements, and integrate new data items that we identified during the project lifetime.

The document is intended for: (a) users of the VITAL platform that want to learn about the used data models to develop applications, (b) IoT platform providers that want to learn how to integrate their platform with VITAL, (c) the partners of the VITAL consortium, allowing them to develop their components in a way that ensures easy integration and interoperability, (d) external researchers and developers that want to design and/or use data models in Smart City systems beyond VITAL, as well as (e) the project reviewers to better understand the work done in the project.

This document is structured as follows. First we provide a brief overview on semantic and linked data technologies. Then we present the different ontologies – as well as necessary extensions – for the VITAL data model, e.g. for modelling sensor measurements and sensor descriptions, for Smart Cities, for IoT systems and services (including the data required for the PPI), as well as for the VITAL system itself and its services. We finish the deliverable with a short conclusion and outlook on further work during the VITAL project and beyond.

2 LINKED DATA AND SEMANTIC DESCRIPTIONS

This chapter provides background information required to understand the remainder of this deliverable. We give a short overview on linked data and its technologies. First, we explain the basic concepts of Linked Data. After that we briefly present the most commonly used technologies of Linked Data, namely RDF, SPARQL and Ontologies. Finally, we discuss how Linked Data will be used in VITAL.

The term 'Linked Data' is usually applied to a set of techniques for publishing and interlinking structured data on the Web. With Linked Data different kinds of data sources can be integrated (see [HeBi11]). Linked Data is based on four principles, explained in [HeBi11] and [HKHD09]:

1. The first principle is to **use URIs as names for things**. After the identification of items in a domain of interest, they are described in the data set with their properties and relationships. Each thing must be assigned a globally unique name, usually an HTTP URI (since this makes it easy to enforce global uniqueness). The fact that each thing has a globally unique name makes it possible to integrate local data sets with remote sets that have been developed independently.
2. The second principle is to **use HTTP URIs**, so clients can retrieve a description of the names thing or resource using the HTTP protocol. For humans the description can be provided as HTML, for machines it can be provided as RDF triples.
3. The third principle is to **use standards to provide information**. This usually means that standards like RDF or SPARQL are used to model and access data. We explain the most important standards later in this chapter.
4. The last principle is to **link to other URIs** and to enable the possibility to discover more things. That means that there should be external links pointing to other data sources on the Web. By following these links, a larger (distributed) data space can be explored automatically.

In a nutshell, Linked Data enables the implementation of generic applications operating over a huge, interconnected (distributed) data space by using Web standards and a common data model.

2.1 RDF

The most common data model used in the context of Linked Data is the Resource Description Framework (RDF) [GaSc14], which we introduce in this subsection. We first describe the basics of RDF, before discussing some of its advantages. RDF is a popular standard for describing things (known as resources or entities). By itself it is a graph-based data model that represents information as labelled directed graphs. This graph is built of triples that describe the data. Each triple `(s, p, o)` consists of a subject `s`, a predicate `p` and an object `o`. Take for example the information "The sky has the colour blue". This would be modelled as a triple with "Sky" as the subject, "has colour" as the predicate and "blue" as the object. Note that the predicate in the middle always denotes the relationship between subject and object. Both the subject and the predicate are identified by URIs (assigning them globally unique identifiers)[1]

[1] A subject can also be identified by a so-called blank node. A blank node is a local identifier.

while the object can be a URI or a literal value (i.e. a string or a number). The triple in our example could e.g. look like this (given in N-Triple notation):

```
<http://example.com/Sky> <http://example.com/hasColour> "blue"
```

Using RDF in a Linked Data context has some advantages. In the following we briefly outline the most important ones of these advantages. The reader is referred to [HeBi11] for more detailed information. Possible advantages are:

- If the identifiers of data items (both used as subjects and objects) and vocabulary terms (used as predicates) are HTTP URIs, the RDF data model can be used at global scale and anybody is able to refer to anything.
- Each RDF triple is included in the Web of Data and can be the starting point for explorations in the data space, because any URI can be looked up in an RDF graph over the Web.
- It is possible to set RDF links between data from different sources.
- Sets of triples can be merged in a single graph to combine information.
- Terms taken from different vocabularies can be mixed in a RDF graph.

As RDF is just a data model and not a data format, there are a number of data formats that can be used to write RDF data, either directly as triples or as nodes that can be mapped to RDF triples, e.g. RDF/XML [GaSc14], RDFa [HASB13], Turtle, N-Triples and JSON-LD. In the VITAL Project JSON-LD is used.

2.2 JSON-LD

Many developers have little or no experience with Linked Data, RDF or common RDF serialization formats such as N-Triples and Turtle. This produces extra overhead in the form of a steeper learning curve when integrating new systems to consume linked data. To counter this the project consortium decided to use a format based on a common serialization format such as XML or JSON. Thus, the two remaining options are RDF/XML and JSON-LD [SLK+14] [jsld]. JSON-LD was chosen over RDF/XML as the data format for all Linked Data items in VITAL. JSON-LD is a JSON-based serialisation for Linked Data with the following design goals:

- **Simplicity**: There is no need for extra processors or software libraries, just the knowledge of some basic keywords.
- **Compatibility**: JSON-LD documents are always valid JSON documents, so the standard libraries from JSON can be used.
- **Expressiveness**: Real-world data models can be expressed because the syntax serialises a directed graph.
- **Terseness**: The syntax is readable for humans and developers need little effort to use it.
- **Zero Edits**: Most of the time JSON-LD can be devolved easily from JSON-based systems.
- **Usable as RDF**: JSON-LD can be mapped to / from RDF and can be used as RDF without having any knowledge about RDF.

From the above, terseness and simplicity are the main reasons JSON-LD was chosen over RDF/XML. JSON-LD also allows for referencing external files to provide context. This means contextual information can be requested on-demand and makes JSON-LD better suited to situations with high response times or low bandwidth usage requirements. We think that using JSON-LD will reduce the complexity of VITAL development by (1) making it possible to reuse a large number of existing tools and (2) reduce the inherent complexity of RDF documents. Ultimately, this will increase VITAL's uptake and success. In the following we provide a short overview of the main JSON-LD features and concepts. More information can be found in [SLK+14].

The data model underlying JSON-LD is a labelled, directed graph. There are a few important keywords, such as `@context`, `@id`, `@value`, and `@type`. These keywords are the core part of JSON-LD. Four basic concepts should be considered:

- **Context**: A context in JSON-LD allows using shortcut terms to make the JSON-LD file shorter and easier to read (as well as increasing its resemblance with pure JSON). The context maps terms to IRIs. A context can also be externalised and reused for multiple JSON-LD files by referencing its URI.
- **IRIs**: Internationalised Resource Identifiers (IRIs) are used to identify nodes and properties in Linked Data. In JSON-LD two kinds of IRIs are used: absolute IRIs and relative IRIs. JSON-LD also allows defining a common prefix for relative IRIs using the keyword `@vocab`.
- **Node Identifiers**: Node identifiers (using the keyword @id) reference nodes externally. As a result of using `@id`, any RDF triples produced for this node would use the given IRI as their subject. If an application follows this IRI it should be able to find some more information about the node. If no node identifier is specified, the RDF mapping will use blank nodes.
- **Specifying the Type**: It is possible to specify the type of a distinct node with the keyword `@type`. When mapping to RDF, this creates a new triple with the node as the subject, a property `rdf:type` and the given type as the object (given as an IRI).

2.3 SPARQL

Assuming that there is RDF data, then a developer needs a language to query it. SPARQL (SPARQL Protocol and RDF Query Language) is an RDF query language and it is used to retrieve and manipulate data, which is stored in RDF. There are four query variations that SPARQL can distinguish: `SELECT`, `CONSTRUCT`, `ASK` and `DESCRIBE` queries. The most basic constructs of a SPARQL query are graph patterns, explained in [HaSe13]. A basic graph pattern is similar to a RDF triple except that the subject and predicate can be variables as well. The example in Figure 1 shows a query that will return the names of all pairs of people that know each other.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?name1 ?name2
FROM <http://example.org/foaf>
WHERE {
      ?person1 foaf:knows ?person2.
      ?person1 foaf:name ?name1.
      ?person2 foaf:name ?name2
}
```

Figure 1: SPARQL example [LWS13]

2.4 Ontologies

In addition to RDF and SPARQL another very important technology in Linked Data are ontologies. This section provides a brief introduction of ontologies and their meaning.

Linked Data can be used in cases where data originates from different sources. To integrate all data, be it from one or different sources, there have to be some rules. Some rules determine how the RDF graph is to be built and how triples may be connected or not. These rules are given by ontologies. An ontology specifies formally the conceptualisation of a domain of interest. As the conceptualisation is formal, a computer can automatically reason on it. There are practical ontologies for different domains of interest. An ontology consists of concepts (also referred to as classes), relations (also called properties), instances and axioms. It defines basic terms and relationships.

To specify ontologies, the W3C published the Web Ontology Language (OWL[2]), which builds on RDF. OWL facilitates mechanism for creating concepts, instances, relations and axioms. Concepts can have super and sub concepts. Axioms provide information about classes and properties. This topic is explained in detail by [DSW06] and [NoMc01].

There are many ontologies that have already been developed. Reuse of existing ontologies is crucial. If an existing ontology within the domain of use does not meet all the requirements and some new data models arise they should be attached to the existing ontology. This ontology grows by doing so and helps users on any of its nodes to reach every other node in this ontology graph. The user will get much more information as just of his own model – if he/she wants to.

2.5 Conclusion

Linked Data (using RDF, SPARQL and ontologies) helps to describe and integrate data that is provided by different organisations in an interoperable way. This is ideally suited for VITAL. VITAL envisages that a multitude of (independent) organisations

[2] www.w3.org/2004/OWL

and entities deploy and operate different sensors and IoT systems, which produce data (and offer functionality) that should be integrated in a platform agnostic way.

VITAL uses Linked Data standards for modelling and accessing data. RDF is used as the basic data model. JSON-LD is used as the data format. SPARQL is used to query data using complex sub graph patterns. Finally, ontologies are used to specify our data formally. To use the potential of Linked Data for interconnecting with as many external information sources as possible (making their information readily available for VITAL developers), we use the good practice of reusing as many ontologies as possible. We require ontologies in different areas. First, VITAL must allow to model sensors and sensor measurements, which are the basis of any IoT system. Second, VITAL has to provide means to model entities that are relevant to Smart Cities. Third, VITAL must allow to model IoT systems and services that are integrated into the VITAL platform. And finally, it must provide ontologies to model the VITAL system itself. In the remainder of this deliverable we discuss how to model these different parts using which ontologies. Table 1 provides an overview of the main ontologies used in VITAL, their namespaces and the prefixes that we use to refer to them.

Table 1: Ontology/ Language Prefixes

Prefix	Ontology / Language	Namespace
dcn	Delivery Context ontology	http://www.w3.org/2007/uwa/context/deliveryContext.owl#
dul	DOLCE+DnS Ultralite ontology	http://www.ontologydesignpatterns.org/ont/dul/DUL.owl#
geo	Basic Geo (WGS84) ontology	http://www.w3.org/2003/01/geo/wgs84_pos#
hrest	hRESTS ontology	http://www.wsmo.org/ns/hrests#
msm	Minimal Service Model ontology	
owl	Web Ontology Language	http://www.w3.org/2002/07/owl#
rdfs	RDF Schema ontology	http://www.w3.org/2000/01/rdf-schema#
sawsdl	Semantic Annotations for WSDL and XML Schema ontology	http://www.w3.org/ns/sawsdl#
ssn	Semantic Sensor Network ontology	http://purl.oclc.org/NET/ssnx/ssn#
time	OWL Time ontology	http://www.w3.org/2006/time#
vital	VITAL ontology	http://vital-iot.eu/ontology/ns/#
wsl	WSMO-Lite ontology	http://www.wsmo.org/ns/wsmo-lite#
xsd	XML Schema Definition	http://www.w3.org/2001/XMLSchema#
qudt	Quantities, Units, Dimensions and Data Types Ontologies	http://qudt.org/schema/qudt#
foaf	Friend of a Friend	http://xmlns.com/foaf/
s4ac	Social Semantic SPARQL Security for Access Control	http://ns.inria.fr/s4ac/v2#
otn	Ontology of Transportation Networks	http://www.pms.ifi.lmu.de/rewerse-wga1/otn/OTN.owl

3 SENSORS AND MEASUREMENTS

Networked sensors and their measurements are one of the most important parts of any data model for IoT systems and therefore also an integral part of the VITAL data model. To better define the semantics of IoT data, a number of ontologies have been developed on multiple layers of abstraction [w3c]:

(1) **sensor-centric ontologies** like the Ontonym sensor ontology [SKD+09], the Sensor Data Ontology [ELS07], and OntoSensor [GoRu06];
(2) **observation-centric ontologies** like the Semantic Sensor Network (SSN) Ontology [CBB+12], the Sensei Observation and Measurement Ontology [WeBa09], and stimuli-centered ontologies [SJB+09], as well as
(3) **context-centric ontologies** like COMANTO [SRA06] and SOUPA [CPF+04].

Clearly it is impossible to specify a single ontology that defines the semantics of all possible data items, as they are in many cases application (domain) specific. This has lead to the development of rather abstract and complex ontologies that try to fit all possible cases by providing a conceptual framework only, omitting concrete instances like specific sensor models, etc. Such ontologies try to impose an overarching structure onto IoT systems and their data, e.g. specifying abstract metadata classes for stimuli, observations, measurements, sensors and features of interest.

In practice such ontologies must be combined with additional ontologies to define concrete instances of abstract concepts. As an example, while a generic sensor ontology may specify how to model what a sensor is measuring, additional definitions must be used to model a concrete location sensor.

In the following we first present, SSN as the generic sensor ontology that the VITAL consortium has selected to be used in VITAL. We then discuss a number of further ontologies as well as additional definitions that are used in VITAL to create concrete data items.

3.1 The W3C Semantic Sensor Network Ontology

The Semantic Sensor Network (SSN) ontology [CBB+12] defines a conceptual framework for describing sensors and observations. It was developed by the W3C Semantic Sensor Network Incubator group (SSN-XG). In the following, the ontology is explained in more detail to provide an understanding about its meaning by presenting what it describes, the used pattern (SSO), and the four main perspectives.

Overall the SSN ontology is able to describe:

- sensors including their accuracy and capabilities,
- observations,
- methods for sensing,
- concepts for operating and survival ranges, and
- deployments.

A sensor could be anything that observes, be it an electronic object, a virtual object or a human. The ranges are used in the definition of sensors conjoined with the performance of these sensors. The description of deployment includes the deployment lifetime as well as the sensing purpose of the deployed macro instrument.

3.1.1 Stimulus-Sensor-Observation Pattern

The ontology is built around an ontology design pattern called Stimulus-Sensor-Observation (SSO) Pattern. This pattern describes the relationships between sensors, stimuli and observations.

A sensor is modelled as a physical object (`dul:PhysicalObject`) and is able to observe, transform and represent data. How it observes is defined in sensing methods (imported from other ontologies).

A stimulus (`dul:Event`) is represented by a change or state which can be detected and used by a sensor to measure some property. It is comparable to a proxy for an observation property. A property, in turn, is an observable characteristic of a real-world entity (`ssn:FeatureOfInterest`).

Observations are the connectors in the SSO pattern. They can link between the act of sensing, a stimulus event, a sensor, a method, a result, an observed feature and a property – and put them in an interpretative context. An observation is modelled as a social construct (from `dul:Situation`).

3.1.2 Perspectives

For a better understanding of the ontology in terms of sensors and observations, there are four main perspectives in the SSN ontology, explained in the following.

Sensor perspective

A sensor is described with a stimulus, a sensing method, observations and capabilities. The environment can influence the performance of a sensor. This is referred to as measurement capabilities. Such capabilities are i.e. accuracy, measurement range, measurement precision, and measurement resolution. One sensor can have many measurement capabilities, representing the capabilities of the sensor in different conditions (observable properties of the sensor-environment).

Observation perspective

The observation perspective describes an observation. An observation includes a context for interpreting incoming stimuli and puts the observation event (including the stimulus) into an interpreting context. A context includes observed features, properties, the observing sensor, the result and the used sensing method. As an observation is a social construct, a stimulus event can be abstracted from its (potential) interpretations. The sensing method can be a principle underlying a sensor or a description of how the observations were done.

System perspective

A system can have sub-systems or sub-concepts like devices and sensing devices. There are operating and survival restrictions for a system, similar to the ones for sensors and measurement capabilities, which are observable properties of the system. The operating range specifies the range in which the system aims to operate. The survival range is a range in which a sensor can bare without release a lasting damage. The process of combining all life-cycle phases of a deployed system is named deployment. It includes installation, maintenance and decommissioning of a system.

Feature and property perspective

The feature and property perspective focuses on properties, more specifically on the sensors that sense some distinct property or on the observations that were made about a distinct property. Figure 2 provides the key concepts and relations of the SSN ontology.

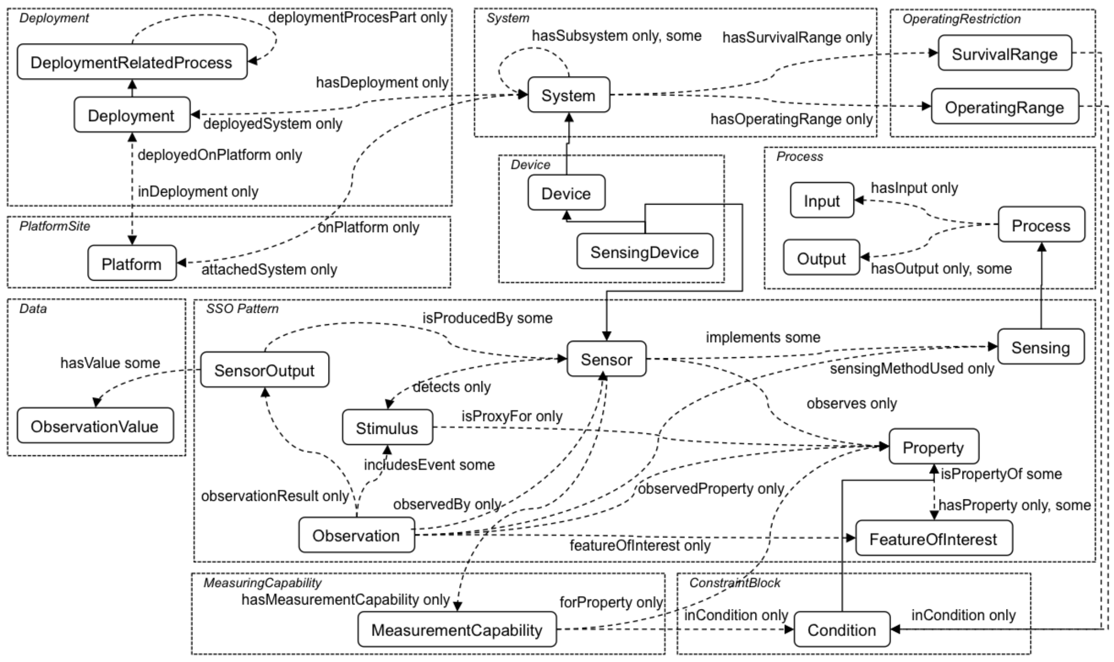

Figure 2: The SSN Ontology – key concepts and relations [CBB+12]

3.1.3 Conclusion

To sum up, the SSN ontology was built to describe sensors, sensing and measurement capabilities of sensors as well as the resulting observations and deployment. The ontology covers big parts of the SensorML and O&M standards. SSN restricts itself to a conceptual view. It does not include specific definitions for concrete sensors or domain areas. Therefore, to realise a concrete IoT system based on the SSN ontology additional domain specific ontologies must be imported. For example you would use a temperature ontology on top of the SSN ontology to model a temperature sensor. VITAL uses the SSN ontology as the basis for modelling all entities and activities related to sensor data and metadata in a Smart City. Necessary extensions with additional ontologies (e.g. for location modelling or domain specific measurements) are discussed in the remainder of this chapter.

3.2 Basic Concepts and Ontologies

In this section we present ontologies to model basic concepts like time, location and unit of measurement. These will be used throughout the VITAL system. Thereafter we discuss how to model sensors and sensor measurements in more detail.

3.2.1 Time

Temporal aspects are essential for any system addressing real world phenomena, e.g. smart city IoT systems. Timestamps can be used to describe when a sensor reading was taken or when it was valid. Multiple readings can be ordered by the time of their occurrence. Clients may specify that they are interested in the current state of an environment or in the state it had one week ago. To model this, VITAL has to provide an ontology for time as well as temporal properties and relations. A well-established ontology for this is OWL Time [HoPa06]. OWL Time allows describing of temporal properties and relationships. It also supports time intervals as well as durations, which are useful for example, when describing imprecise measurement times as well as complex event specifications.

In VITAL, all timestamps, temporal properties and relations are described using OWL Time. An example for a timestamp in a sensor measurement is given in Figure 3: .

```
{
  "@context": {
    "time": "http://www.w3.org/2006/time#",
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "time:inXSDDateTime": {
      "@type": "xsd:dateTime"
    }
  },
  "@type": "time:Instant",
  "time:inXSDDateTime": "2014-08-20T16:47:32+01:00"
}
```

Figure 3: Example Time Instant Specification with OWL Time

3.2.2 Location

Location in the physical world is another basic concept that has to be modelled in VITAL. There is a multitude of different location models and ontologies available today, including geographical and symbolic location models. VITAL follows a practical approach to allow easy usage of the system while at the same time being flexible enough for advanced use cases.

WGS84 coordinates are used as the basic location model, since they are the de-facto standard for outdoor localisation using the GPS system. To model them, the basic Geo (WGS84) vocabulary[3] is used. Figure 4: provides an example. In addition, symbolic names are often used as locations. VITAL allows using symbolic names instead of WGS84 coordinates. VITAL uses the Linked GeoData system[4] to model more complex location concepts, including symbolic names, cell-based locations and inter-location relationships. To map between symbolic names and WGS84 coordinates, VITAL uses the GeoNames system[5]. If a location is given as a symbolic name, VITAL automatically tries to retrieve the WGS84 coordinates for it and enriches the location data with them. The same is true the other way round.

```
{
  "@context": {
    "geo": "http://www.w3.org/2003/01/geo/wgs84_pos#"
  },
  "@type": "geo:Point",
  "geo:lat": "55.701",
  "geo:long": "12.552",
  "geo:alt": "4.33"
}
```

Figure 4: Example WGS84 Location with the Basic Geo Vocabulary

3.2.3 Unit Of Measurement

Different properties in the VITAL data models represent physical magnitudes like “length” or “weight”. Each one of these properties should be associated with an unambiguous unit of measurement, e.g. “metre” or “kg”. Otherwise, cultural differences may lead to clients interpreting values incorrectly, e.g. by assuming a length is given in “feet”, while it is actually given in “metre”. While RDF allows specifying the type of a given property value by tagging it with a special type identifier (e.g. "13"^^<http://www.w3.org/2001/XMLSchema#int>), it does not support tagging values directly with a unit of measurement. At the same time, there is currently no single accepted ontology to model units of measurements in linked data. A number of potential ontologies were found and four were chosen for detailed evaluation. We discuss these four in the following.

Quantities, Units, Dimensions and Data Types Ontologies (QUDT) [HKHS14] is a family of ontologies developed by TopQuadrant and sponsored by NASA designed to formalize quantities, units of measurement, dimensions and types in RDF/OWL formats. Due to this ambitious goal it is incredibly broad and precise. It models base types such as length and time and builds derived types as a hierarchy (e.g. `Velocity = Length / Time`, `Kilometres Per Hour = Kilometres Travelled / Time taken`). Where appropriate, it also references similar ontologies with `sameAs` and `exactMatch` relationships as well as DBpedia entries where appropriate. This gives items associated with QUDT types a huge amount of semantic information.

[3] http://www.w3.org/2003/01/geo/

[4] http://linkedgeodata.org/

[5] http://www.geonames.org/

Similar to QUDT, the Units of Measurement Ontology (UO) [UO] establishes a hierarchy of base and derived types. It contains a large number of types but is missing some common derived types such as “Kilometres Per Hour”. It also fails to link to external resources such as DBpedia where relevant which makes it less useful that QUDT.

The Ontology of units of Measure (OM) [OM] stands out from the other evaluated ontologies by using URIs as the value for the majority of each type’s information. This immediately gives feedback to the user that values are unique. Thanks to each type using the same signature, this lets the user easily compare types for similarities using simple methods such as string comparison rather than needing to traverse the type hierarchy. Each of these URIs (e.g. `om:kilometre`, `om:dimension`, `om:length-dimension`) can be followed to obtain more information including other users of this value. OM has a large number of derived types available but – just as UO – lacks links to external resources such as DBpedia and other ontologies.

Finally, the Measurements Unit Ontology (MUO) [PoBe08] aims to solve the same problem as the other evaluated ontologies by establishing a hierarchy of base and derived types. However, it appears to have been abandoned before completion as of time of writing.

VITAL chooses QUDT as the ontology for units of measurements due to the impressive scope and amount of information available on each type as well as the reputation of the maintainer and sponsoring party. QUDT is also actively maintained, with the latest version that was released in March 2014. In addition, the links to DBpedia provide a tight integration with a lot of further data items, both DBpedia itself and in data sources linking to it – even more so since VITAL will use DBpedia to link to information about cities (see Section 4.1). After the re-evaluation between Om and QUDT, QUDT remains the best choice. An example for using QUDT is given in Figure 7: .

3.3 Sensors

After presenting ontologies for basic concepts we can now discuss how to model sensors, sensor measurements and their descriptions in VITAL. In general, VITAL reuses and extends the SSN ontology and the Delivery Context (DC) ontologies.

A sensor is modelled as a `VitalSensor`, a subclass of `ssn:Sensor` and `dcn:Device`[6]. According to the SSN, a “sensor can do (implements) sensing: that is, a sensor is any entity that can follow a sensing method and thus observe some Property of a FeatureOfInterest. Sensors may be physical devices, computational methods, a laboratory setup with a person following a method, or any other thing that can follow a Sensing Method to observe a Property” [ssn]. By using the SSN ontology, VITAL can immediately describe sensors in detail, including aspects like the properties that they observe, sensor locations, and sensor observations. The SSN ontology also allows to model non-functional aspects of a sensor, e.g. its accuracy or reliability, by adding a `ssn:hasMeasurementProperty` property to the sensor description that points to a `ssn:MeasurementCapability` (or a subclass of it). The reader is referred to the SSN ontology specification at [ssn] for more information about the ontology. The DC ontology defines a `dcn:Device` as a class that “represents a device in the delivery context” [FoLe09]. By using the DC ontology,

[6] Note that `dcn` is the prefix of one of the DC ontologies.

VITAL can reuse a highly detailed set of ontologies describing many aspects of devices, including their software, their hardware as well as their networking.

```
Class: VitalSensor
    SubClassOf: ssn:Sensor, dcn:Device
```

3.3.1 Additional Ontology Definitions

In addition to the SSN and DC ontologies, VITAL defines an additional property for sensors, *hasLastKnownLocation*. This property is a sub property of dul:lastLocation as specified in the SSN ontology description. It links to a location, which is the last known location of the sensor. Note that this property replaces dul:hasLocation to make the property's semantic less ambiguous in VITAL. The property does not imply that this is the actual *current* location of the sensor. If the sensor is mobile, it could have moved to a new location after the description was created. If the property is not available in a sensor description, then the location of the sensor may not be known. However, a client may be able to access the current location using a so-called localizer service (see Section 6.5.4).

```
ObjectProperty: hasLastKnownLocation
    SubPropertyOf: dul:hasLocation
    Domain: VitalSensor
    Range: dul:Entity
```

Note that the location of a sensor can be modelled with different types as specified before, e.g. as a geo:Point in case GPS coordinates are used. To be as flexible as possible, we use the generic dul:Entity class to represent all different location types here. It is taken from the DOLCE+DnS Ultralite (DUL) ontology and defines it as "[a]nything: real, possible, or imaginary, which some modeller wants to talk about for some purpose." [dul]

In addition to the basic description of a sensor, some VITAL services may require additional information. Properties and classes to model this information are described in Chapter 6 of this deliverable.

3.3.2 JSON-LD Definitions

Note that since Linked Data in VITAL is always formatted as JSON-LD we can introduce some additional definitions (in a JSON-LD context section as shown in

Figure 5:) that do not change the used ontology or the resulting RDF triples but align the JSON-LD representation more closely to 'normal' JSON and thus makes it easier for developers to work with the data. We use this approach repeatedly in this document. As can be seen in the figure, we first specify that all JSON keys that do not specify a prefix will be expanded to URIs in the VITAL ontology namespace. This results in more compact files with less clutter. Then we define that the key uri will be

mapped to a JSON-LD node identifier (`@id`). The node identifier is used to create the URI that is used as the subject in RDF triples.

```
{
  "@context": {
    "@vocab": "http://vital-iot.eu/ontology/ns/",
    "vital": "http://vital-iot.eu/ontology/ns/",
    "lsm": "http://lsm.deri.ie/ont/lsm.owl#",
    "ssn": "http://purl.oclc.org/NET/ssnx/ssn#",
    "geo": "http://www.w3.org/2003/01/geo/wgs84_pos#",
    "msm": "http://iserve.kmi.open.ac.uk/ns/msm#",
    "hrest": "http://www.wsmo.org/ns/hrests#",
    "hard": "http://www.w3.org/2007/uwa/context/hardware.owl#",
    "soft": " http://www.w3.org/2007/uwa/context/software.owl#",
    "id": "@id",
    "type": "@type",
    "name": "rdfs:label",
    "description": "rdfs:comment",
    "ssn:madeObservation": {
      "@type": "@id"
    },
    "status": {
      "@type": "@id"
    },
  }
}
```

Figure 5: JSON-LD Context for Sensor

Similarly, we specify that the key `type` will be mapped to the JSON-LD keyword `@type`. This results in an RDF triple being created that specifies the RDF type of a node. To further simplify the JSON-LD file, we specify that the key `name` will be mapped to `rdfs:label` and the key `description` will be mapped to `rdfs:comment`. We then specify a number of prefixes that can be used in the JSON-LD description to reduce the length of keys by specifying them as so-called terms. Terms are automatically expanded using the provided prefix URI. As an example, the term `geo:lat` would be expanded to `http://www.w3.org/2003/01/geo/wgs84_pos#lat`. Finally we specify type specifications that declare that a value given for the key `ssn:madeObservation` or `hrest:hasAddress`[7] should be mapped to a node id instead of a string. All these mappings are completely transparent to developers and can be ignored by clients. They are only relevant if the JSON-LD file is mapped to RDF triples internally. Together, they reduce the complexity of the resulting JSON-LD file and make it both smaller and easier to read and understand for JSON developers.

3.3.3 Sensor Description Example

[7] see Section 5.2 for more information about *hrest* and the hRESTS ontology.

To illustrate the resulting sensor descriptions, we give a short example for a sensor description (in JSON-LD format) In

Figure 6: . The example uses an external version of the context specification for VITAL sensors that we have just described. This further reduces the size of the JSON-LD file. VITAL clients can access the external context file at `http://vital-iot.eu/contexts/sensor.jsonld` if necessary. However, we expect that this will not be necessary for most scenarios and normal operation. The context file can also be cached on the client side, because it is independent of a specific VITAL

```
{
  "@context": "http://vital-iot.eu/contexts/sensor.jsonld",
  "name": "TemperatureSensor No.123",
  "type": "VitalSensor",
  "description": "This is an example sensor",
  "id": "http://www.example.com/vital/sensor/123",
  "hasLastKnownLocation": {
    "type": "geo:Point",
    "geo:lat": "53.2719",
    "geo:long": "-9.0489"
  },
  "ssn:observes": [
     {
      "type": "openiot:Temperature",
      "id": "http://www.example.com/vital/sensor/123/temperature"
     }
  ]
}
```

sensor and can be reused.

Figure 6: Example Sensor Description

In the example, we describe a sensor with its name, its type (a `VitalSensor`), its URI in the VITAL system and its description. The sensor provides a last known location as a GPS (WGS84) location (of type `geo:Point`). It also specifies that it observes two different parameters, the light level and the temperature, as well as URIs to retrieve current measurements for both. Finally, a sensor *may* specify observations it made in the past, in this case an observation with ID `http://www.example.com/ vital/sensor/123/obsvn/1`.

3.4 Sensor Measurements

Similarly to sensors, VITAL uses the SSN ontology to model sensor measurements. A measurement is modelled as an `ssn:Observation`. An example can be seen in Figure 7: . Figure 8: shows the JSON-LD context used by the example. The observation contains a link to an observed property (using `ssn: observationProperty`) to specify what the observation is measuring. In addition, it specifies when the measurement was taken (`ssn:observationResultTime`), at which location (`dul:hasLocation`) in WGS84 format, the quality of the measurement (`ssn:observationQuality`), as well as the measured value (`ssn:observationResult`), 21.0 Degree Celcius, with the unit of measurement specified with the QUDT ontology.

```
{
  "@context": "http://vital-iot.eu/contexts/measurement.jsonld",
  "uri": "http://www.example.com/vital/sensor/123/obsvn/1",
  "type": "ssn:Observation",
  "ssn:observationProperty": {
    "type": "http://lsm.deri.ie/OpenIoT/Temperature"
  },
  "ssn:observationResultTime": {
    "inXSDDateTime": "2014-08-20T16:47:32+01:00"
  },
  "dul:hasLocation": {
    "type": "geo:Point",
    "geo:lat": "55.701",
    "geo:long": "12.552",
    "geo:alt": "4.33"
  },
  "ssn:observationQuality": {
    "ssn:hasMeasurementProperty": {
      "type": "Reliability",
      "hasValue": "HighReliability"
    }
  },
  "ssn:observationResult": {
    "type": "ssn:SensorOutput",
    "ssn:hasValue": {
      "type": "ssn:ObservationValue",
      "value": "21.0",
      "qudt:unit": "qudt:DegreeCelsius"
    }
  }
}
```

Figure 7: Example Measurement

```
{
  "@context": {
    "@vocab": "http://vital-iot.eu/ontology/ns/",
    "vital": "http://vital-iot.eu/ontology/ns/",
    "lsm": "http://lsm.deri.ie/ont/lsm.owl#",
    "ssn": "http://purl.oclc.org/NET/ssnx/ssn#",
    "geo": "http://www.w3.org/2003/01/geo/wgs84_pos#",
    "time": "http://www.w3.org/2006/time#",
    "qudt": "http://qudt.org/vocab/unit#",
    "xsd": "http://www.w3.org/2001/XMLSchema#",
    "id": "@id",
    "type": "@type",
    "time:inXSDDateTime": {
      "@type": "xsd:dateTime"
    },
    "qudt:unit": {
      "@type": "@id"
    }
  }
}
```

Figure 8: JSON-LD Context for Measurement

3.5 Conclusion

Sensors and sensor measurements are a central part of each IoT data model. VITAL uses (and extends) the well-established SSN ontology to model these information items, including meta-information like the type of measurements a sensor can perform or the quality of a measurement. To model basic concepts like time, location and unit of measurement, VITAL relies on additional ontologies, like OWL Time and the basic geo ontology. Together with the SSN ontology this provides the means to specify sensors and measurements in a platform agnostic way, as required by VITAL. It also allows us to link them with other systems and information sources that use the same or similar ontologies, like the FP7 project OpenIoT. This also enables the reuse of existing tools and software components for VITAL, without the need to implement everything from scratch. An example for this is the VITAL data management component, which will be based on OpenIoT.

However, we cannot yet describe real physical instances of sensors (or measurements), since we do not have a common ontology for specific values for our descriptions. As an example, while we know how to specify what property of the real world a sensor is observing (by using `ssn:observes`) we do not have a specification for modelling real instances of such properties, e.g. temperature, speed or light level. These are application domain specific. This issue is discussed further in the next chapter.

4 SMART CITIES

‘Smart City’ is quite a broad term that represents many different aspects of a city's population, economy and [CDN09]. For this document, we define a Smart City as one that uses ICT to integrate its different capabilities and enhance the quality of life for its population. Smart cities have many capabilities that must be modelled as part of a smart city ontology. The most active and relevant producers of data come under the following headings (see also VITAL deliverable D2.1 [SoKa13]):

- **Transport** – e.g. dynamic route calculation informed of accidents and congestion, integration of transport methods such as bus, tram and subway for public transport users.
- **Energy** – e.g. optimization of light levels, reporting of faults.
- **Emergency Services** – e.g. using real-time traffic data to inform emergency response units and to provide real-time information on accident and crimes.
- **Waste Management** – e.g. detection of full bins as well as automated reporting of missing/damaged bins.
- **Air and Water** – e.g. reporting of high pollution in air and water, radiation levels reporting of water leaks, automatic irrigation of parks and green areas.
- **Recreation** – e.g. provide data on local social events as well as nearby facilities, produce data on large events such as concerts and sports games to inform public transport and law enforcement systems of potential increased activity.
- **Smarter Working** – Mobile workers require optimal working environments to be available at short notice and without any difficulties to use. Along with the available work desks in an area additional information is also shown associated with each workspace e.g. anticipated air quality, temperature, humidity, and footfall in the requested time window and location etc.

Other capabilities such as smart homes (including advanced integration cases such as those designed for OAPs and the disabled) and smart business and industry are mostly consumers but may also provide alerts (e.g. factories may provide notice when beginning a water-intensive task).

The VITAL use cases (see VITAL deliverable D2.2 [SoKa14]) so far focus on (smart) transport and traffic management and smart working. Therefore, in the following we discuss how to model data items and properties that are relevant for smart traffic scenarios. Clearly, VITAL is not restricted to smart transport scenarios. A user who would like to use VITAL for other smart city aspects can do so by specifying additional ontology elements. Due to the nature of Linked Data, these additional elements can be added at any time without the need to redesign the system. In the following we first discuss briefly how VITAL includes general information about cities that may be useful for different aspects of smart city applications. After that we focus on modelling smart transport and smart working related information.

4.1 Cities

VITAL obtains the majority of its semantic information on cities from DBpedia, using the classic DBpedia dataset for most information with the option of using DBpedia live for information that updates more frequently. It is also encouraged to link real places and services in cities back to DBpedia to improve the amount of knowledge available. For example, while Camden Road would be modelled as an `otn:Road` as part of a smart transport system, it should also link to `dbpedia:Camden_Road`.

4.2 Smart Transport

Smart Transport is an important topic in Smart Cities and covers a wide range of domains. Some examples include tracking pedestrian congestion in an area through footfalls per minute, smart traffic light systems or route recommendation to empty parking spaces capable of reserving a spot en route. There are also opportunities for integrated solutions with other smart city services such as automatic contact of emergency services in the case of an accident and the ability to detour traffic away from the scene through the use of traffic light control.

In this section, we'll first discuss how traffic infrastructure is modelled in VITAL before going on to discuss what kind of services can be built on top of this infrastructure. Finally, we'll discuss how VITAL will showcase the power of integrated IoT smart city solutions.

4.2.1 Transport Infrastructure

VITAL models transport infrastructure using a combination of ontologies. The core of these is the *Ontology of Transportation Networks* (OTN) [OTN]. This ontology allows easy modelling of a transport network graph with connections between infrastructure such as bus and train stations as well as events such as accidents and blocked passages.

Modelling public transport was one of OTN's main goals during its design so it is trivial to integrate different public transport methods. For example, a journey that requires a 20 min train journey, 3 minutes walking to a regional bus station, a 1 hour bus journey, 2 minutes walking to a city bus service, a 5 minute bus journey and 4 minutes walking to destination can be described as a route through a single graph. This graph representation of public transport can be used by services such as those described in Section 4.2.2 to plan journeys. Another example would be providing automatic detours in the case of accidents and construction and using an ambulance's projected route to change traffic lights to red based on its current location with dynamic recalculation if the ambulance changes from the suggested route.

VITAL also aims to extend OTN to allow new public and private transport types. Some common examples include locations of taxi offices and bicycle racks but this will also allow the addition of less common methods of transport such as rickshaws, which are extremely popular in some countries and tour buses, which may be preferable to tourists.

Each node in the OTN graph is also augmented with approximate longitude and latitude, corresponding DBpedia information and URIs for any sensors located at this location.

4.2.2 Transport Services

Route recommendation services aim to provide a user with an optimal route to their destination, generally using live traffic data and other available information such as construction works and accidents to reach this solution. There are a number of high-profile examples of these such as Google Maps, Bing Maps and Nokia's Here Maps. However, while these solutions attempt to take external factors such as those previously mentioned into account they cannot take advantage of any knowledge of other drivers' routes. Smart cities do not need to suffer this limitation.

The Ontology based Traffic Jam Control System (OnTraJaC) [HaMa12] is designed to recommend routes to avoid traffic jams and support shorter trip times. It achieves this by taking into account all subscribed vehicles during route recommendation and is able to recommend a route to each user that does not conflict with others.

This approach should produce the optimal route for each subscribed user but also has some interesting opportunities for future expansion and integration with other Smart City services. For example, this system could be used to tweak subscribed user's routes when they conflict with the path of an emergency vehicle. In Section 4.2.1 it was suggested that traffic light infrastructure could be integrated with emergency response systems. However, this would not affect users driving along the emergency vehicles path, only those on other routes that would cross it at a junction. By combining this approach with the route recommendation service, subscribed users could be recommended to turn off the emergency vehicles projected route just before the emergency vehicle reaches their position. This would prevent the emergency vehicle having to potentially overtake them and would be safer for both subscribed users and emergency response teams.

While there are ontologies that support similar functionality to existing personal route recommendation systems – such as the Urban Traffic Ontology [UTO] – these should in theory suggest the same route as OnTraJaCS would for the case of `n = 1`, where `n` equals the number of subscribed users taken into account during route calculation. Therefore, we have decided there is no need for any other route recommendation ontology and instead we should simple accommodate a scenario where OnTraJaCS operates with knowledge of only one user.

To support public transport route planning many ontologies were examined but most have their own model of how the transport network should be modelled. Since VITAL has chosen to use a combination of ontologies to represent transport infrastructure and public transport facilities, chiefly OTN, it is non-trivial to integrate these ontologies with existing ones. For the purpose of transport and traffic scenarios and use cases (specifically in Camden Town), VITAL models the following classes:

```
Class: Line

Class: BusArrival
  SubClassOf: ssn:Property

Class: RailArrival
  SubClassOf: ssn:Property
```

```
Class: TubeArrival
  SubClassOf: ssn:Property

Class: AvailableBikes
  SubClassOf: ssn:Property
```

To describe general description of Line, VITAL supports two new properties. These properties are:

```
ObjectProperty: name
      Domain: Line
      Range: dul:Entity

ObjectProperty: direction
      Domain: Line
      Range: dul:Entity
```

4.2.3 Traffic Management

Safety on the roads is one of the main issues faced by the governments. In order to ensure safe and efficient use of the road network, traffic management is generally performed by specialized Traffic Control Centres (TCC). These centres maintain and operate Intelligent Transportation Systems in the cities and include Traffic Monitoring and Supervision Cameras, Radar Detectors, Sensors, Variable Message Signs (VMS, VTS), and similar systems. Still most of the monitoring and operational tasks are done by human effort in these centres. There are traffic operators and call centre staff monitoring cameras and observing data coming from the sensors in order to predict incidents. Therefore there is a high demand for intelligent smart traffic management systems. Some of the important smart traffic management functionalities are; traffic speed prediction, congestion control, incident detection, identification sensor validation, automatic detection of sensor failures. Due to the number of cars in traffic and long-last travel times especially in big cities giving instant travel time is mostly insufficient. Instead, it is required to provide estimated travel time for long trips and for dense traffic situations. Most of the time people need predictions in order to schedule their travel in big cities. Therefore traffic prediction functionality plays an important role in city dwellers life. The instant travel information and predictions are obtained by using the sensor data. Hence the reliability of the sensor measurements is very important. A faulty sensor could cause a failure, and must be detected as fast as possible. On the other hand detection of an incident in traffic is another important functionality that a smart traffic management system should include. The incident detection can be obtained on the sharp decrease in the speed of the vehicles passing by a sensor.

Smart traffic management system's functionalities will benefit from VITAL. The sensor data reaching the VITAL platform will be obtained by the smart traffic management systems in JSON-LD data format. In order to achieve some of these functionalities using VITAL a use case application for Istanbul is planned. The application will use the functionalities of the platform and will be used for traffic prediction, incident detection, and identification of faulty sensors. There have been no data models selected for traffic management application as yet. The detailed descriptions of the application and it's integration with VITAL is given in Deliverable 6.2.1.

4.3 Smart Working

The structural development in advanced economies is influencing the change in working patterns with (increasingly) employees more likely to work on the move. Mobile workers require optimal working environments to be available at short notice and without any difficulties to use. Owners of these suitable environments require optimal occupancy. To meet these requirements, the smarter working application (powered by VITAL) will use the following classes (as sub classes of `ssn:Property`):

```
Class: AvailableDesks
    SubClassOf: ssn:Property

Class: Availability
    SubClassOf: ssn:Property
```

Each available workspace in the system that meets the specified criteria is shown on a map and/or list. Additional information is also shown associated with each workspace e.g. anticipated air quality, temperature, humidity, and footfall in the requested time window and location etc. In order to model additional information, following classes are defined as sub classes of `ssn:Property`:

```
Class: CarbonMonoxide
    SubClassOf: ssn:Property

Class: Ozone
    SubClassOf: ssn:Property


Class: Footfall
    SubClassOf: ssn:Property
```

Should the additional information need to be added and modelled by introducing new classes, they will be added as the scenarios and use cases make progress.

4.4 Conclusion

Modelling the infrastructure and available facilities and services of a smart city is a non-trivial task. VITAL therefore aims to cover two of the most important components: Smart Transport (and Traffic management) and Smart Working. Smart

Transport includes modelling the cities transport infrastructure and services such as route recommendation, but also explores integration with other smart services such as emergency response to showcase the power of integrated smart city solutions. Future services can then be integrated with VITAL's traffic infrastructure from the beginning, which will allow scenarios such as automatic route calculation to entities such as burst water pipes and fallen electricity lines for repair teams. Smart Working includes modelling available workspaces to support mobile workers.

5 IOT SYSTEMS AND SERVICES

VITAL integrates existing IoT systems (i.e. deployed platforms) and allows clients (applications as well as VITAL system services) to access (meta) data and services of such systems. So far the consortium has selected four platforms for which example deployments will be integrated as a proof of concept: X-GSN, Reply H1, INRIA FIT and Xively. To integrate systems and work with them, VITAL needs a set of models to describe IoT services, their data and services. For data, we will use the models presented in Chapter 3. In the remainder of this chapter we present the models used for an IoT system itself as well as for generic IoT services.

5.1 IoT Systems

VITAL models an IoT system as a subclass of `ssn:System` with a number of additional properties. An IoT system description always includes a basic set of properties that describe general aspects of the system, e.g. its operator. In addition, a system description may specify a set of *IoT services* that it offers.

```
Class: IotSystem
    SubClassOf: ssn:System
```

5.1.1 General Metadata

To describe general metadata about the system, VITAL supports three new properties. These properties are:

- **`status`**, pointing to a status description for the system (this property must be available 0..1 times),
- **`operator`**, pointing to the entity that is responsible for operating this system (this property must be available 0..n times),
- **`serviceArea`**, pointing to the spatial context of this system, e.g. the city it is operating in (this property must be available 0..n times).

```
ObjectProperty: status
    Domain: IotSystem
    Range: OperationalState
```

```
ObjectProperty: operator
    Domain: IotSystem
    Range: dul:Entity
```

```
ObjectProperty: serviceArea
    Domain: IotSystem
    Range: dul:Entity
```

Note that `status` provides only coarse-grained aggregated information about the overall state of the system. The status of an IoT system might change during its lifecycle, the consortium has decided to represent its values as observations of a virtual sensor. Thus, VITAL compliant systems that want to expose their current operational stat must manage a virtual sensor of type MonitoringSensor (sub class of `VitalSensor`):

```
Class: MonitoringSensor
  SubclassOf: VitalSensor
```

```
{
  "@context": "http://vital-iot.eu/contexts/system.jsonld",
  "id": "http://example.com",
  "type": "vital:VitalSystem",
  "name": "Sample IoT system",
  "description": "This is a VITAL compliant IoT system.",
  "operator": "http://example.com/people#john_doe",
  "serviceArea": "http://dbpedia.org/page/Camden_Town",
  "sensors":
    [
      "http://example.com/sensor/1",
      "http://example.com/sensor/2"
    ],
  "services":
    [
      "http://example.com/service/1",
      "http://example.com/service/2",
      "http://example.com/service/3"
    ],
  "status": "vital:Running"
}
```

Figure 9: Example IoT System Description

More detailed information about the state of a system may be retrieved from an optional monitoring service. In case an IoT system does not provide any such service, the system description may need to be modified manually.

In addition, a `VitalSystem` supports properties that are independent of VITAL, namely `rdfs:label` (to specify a human readable name for a system) and `rdfs:comment` (to give a human readable description of a system).

Figure 9 shows an example for such an IoT system description. Note that the example is using the external JSON-LD context shown in

```
{
  "@context": {
    "@vocab": "http://vital-iot.eu/ontology/ns/",
    "vital": "http://vital-iot.eu/ontology/ns/",
    "rdfs": "http://www.w3.org/2000/01/rdf-schema#",
    "msm": "http://iserve.kmi.open.ac.uk/ns/msm#",
    "hrest": "http://www.wsmo.org/ns/hrests#",
    "id": "@id",
    "type": "@type",
    "name": "rdfs:label",
    "description": "rdfs:comment",
    "status": {
      "@type": "@id"
    },
    "operator": {
      "@type": "@id"
    },
    "serviceArea": {
      "@type": "@id"
    },
    "services": {
        "@id": "vital:providesService",
        "@type": "@id"
    },
    "sensors": {
      "@id": "vital:managesSensor",
      "@type": "@id"
    }
  }
}
```

Figure 10.

Figure 10: JSON-LD Context Specification for Systems

In the example, the system description first associates a `name` ("Camden Reply System") and a `description` (stating that this is a deployment of the Reply H1 IoT system for Camden Town) with this system. Note that – similar to before – `name`, `description` and `uri` are identifiers that have been specified in the context to make the JSON-LD file cleaner and easier to read for developers with previous experience with JSON. The context maps them to `rdfs:label`, `rdfs:comment`, and `@id` (JSON-LD node identifiers), respectively. In the example, the node identifier `http://www.reply.com/camden` identifies this system description and – following

Linked Data good practice – specifies the location where a client can retrieve the description via HTTP.

The system in the example is operated by CTU (`operator`). The `serviceArea` is identified by its DBPedia URI. This allows a client to query the DBPedia for more information about Camden and to integrate this information. The system is currently in the Running state (see Section 5.1.2 below for a more detailed explanation of `OperationalState`).

5.1.2 OperationalState

`OperationalState` specifies the operational state of a system (or other entities, e.g. services). So far, the following states are defined in VITAL as sub classes of `OperationalState`:

- **`Operational`**: The entity is currently not running (and may remain in this state indefinitely) but could be started if needed (please note that this does not guarantee that starting it will succeed),
- **`StartingUp`**: The entity is in the process of starting and will (probably) be running soon,
- **`Running`**: The entity is currently operating and available to use,
- **`ShuttingDown`**: The entity is in the process of shutting down and will soon be stopped,
- **`Unavailable`**: The entity is currently not operating and cannot be started at this time. Usually this implies that a manual intervention is necessary to return the entity to the `Operational` state.

Should additional states be needed, they will be added in later versions of this deliverable.

```
Class: Operational
    SubClassOf: OperationalState


Class: StartingUp
    SubClassOf: OperationalState


Class: Running
    SubClassOf: OperationalState


Class: ShuttingDown
    SubClassOf: OperationalState
```

```
Class: Unavailable
    SubClassOf: OperationalState
```

5.1.3 Provided IoT Services

In addition to the metadata discussed so far, an IoT system may offer a set of IoT services to access its functionalities. Typical examples of such services are data access services, data stream management, monitoring, discovery, and configuration services etc. Each system may offer different services, making it impossible to define a fixed set of services. VITAL therefore provides the means to specify a large variety of IoT services with generic service specification ontology. More information about this can be found in Section 5.2. To allow an IoT system to link to descriptions of provided IoT services, VITAL introduces a new property `providesService`.

```
ObjectProperty: providesService
    Domain: IotSystem
    Range: msm:Service
```

Figure 11 shows an example that provides description of an IoT system that offers three IoT services.

The first service (of type `ConfigurationService`) allows clients to get and set configurations of an IoT system. The second service (of type `MonitoringService`) allows clients to call a number of monitoring functionalities offered by an IoT system e.g. the status of an IoT system, the status of sensors that an IoT system manages, performance metrics of an IoT system, etc. The third service (of type `ObservationService`) allows clients to retrieve ICO observations made available by an IoT system.

In the following we describe IoT services and the ontologies used for them in more detail. To do so, we first discuss existing ontologies. Then, we specify how to describe IoT services in VITAL.

```
[
  {
    "@context": "http://vital-iot.eu/contexts/service.jsonld",
    "id": "http://example.com/service/1",
    "type": "vital:ConfigurationService",
    "operations":
    [
      {
        "type": "vital:GetConfiguration",
        "hrest:hasAddress": "http://example.com/service/1",
        "hrest:hasMethod": "hrest:GET"
      },
      {
        "type": "vital:SetConfiguration",
        "hrest:hasAddress": "http://example.com/service/1",
        "hrest:hasMethod": "hrest:POST"
      }
    ]
  },
  {
    "@context": "http://vital-iot.eu/contexts/service.jsonld",
    "id": "http://example.com/service/2",
    "type": "vital:MonitoringService",
    "msm:hasOperation":
    [
      {
        "type": "vital:GetSystemStatus",
        "hrest:hasAddress": "http://example.com/system/status",
        "hrest:hasMethod": "hrest:POST"
      },
      {
        "type": "vital:GetSensorStatus",
        "hrest:hasAddress": "http://example.com/sensor/status",
        "hrest:hasMethod": "hrest:POST"
      },
      {
        "type": "vital:GetSupportedPerformanceMetrics",
        "hrest:hasAddress": "http://example.com/system/performance",
        "hrest:hasMethod": "hrest:GET"
      },
      {
        "type": "vital:GetPerformanceMetrics",
        "hrest:hasAddress": "http://example.com/system/performance",
        "hrest:hasMethod": "hrest:POST"
      },
      {
        "type": "vital:GetSupportedSLAParameters",
        "hrest:hasAddress": "http://example.com/system/sla",
        "hrest:hasMethod": "hrest:GET"
      },
      {
        "type": "vital:GetSLAParameters",
        "hrest:hasAddress": "http://example.com/system/sla",
        "hrest:hasMethod": "hrest:POST"
      }
    ]
  },
  {
    "@context": "http://vital-iot.eu/contexts/service.jsonld",
    "id": "http://example.com/service/3",
    "type": "vital:ObservationService",
    "operations":
    [
      {
        "type": "vital:GetObservations",
        "hrest:hasAddress": "http://example.com/sensor/observation",
        "hrest:hasMethod": "hrest:POST"
      }
    ]
  }
]
```

Figure 11: IoT System with Provided Services Description

5.2 IoT Services

In VITAL, an IoT system does not only provide access to IoT data (e.g. sensor measurements) but may offer a set of distinct and heterogeneous IoT services. An IoT service may be generic, e.g. a service to discovery ICOs or to access filtered data, or application specific, e.g. a service to reserve a parking space in a Smart City IoT system. In fact, VITAL models all functionality that can be exposed by an IoT system and can be accessed and used by a client as an IoT service, including data access, e.g. reading a sensor measurement. VITAL therefore specifies a flexible data model to specify all different kinds of IoT services. This allows extending the system in case a provider of an IoT system wants to expose new application specific services. In addition to this, VITAL provides concrete instances of IoT service specifications for generic (system) IoT services like the aforementioned discovery and filtering services. Service providers *can* extend these instances for their specific IoT system, but they *should* use instances specified by VITAL to model services if possible. In the following we briefly discuss different existing ontologies and modelling approaches for services, before describing the VITAL IoT service model in more detail.

5.2.1 Service Ontologies And Semantic Description Languages

There are a number of existing ontologies and specification languages to model services, among them the Web Service Definition Language (WSDL) and the Web API Definition Language (WADL) for specifying (SOAP-based) web services and (REST-based) web APIs (also known as RESTful services). Both WSDL and WADL are not using ontologies. Ontology-based approaches (see [VHM+14] for a recent survey) can be classified in three main groups: approaches that aim at extending generic WSDL-based service descriptions with semantic data, approaches that concentrate on extending descriptions of RESTful services with semantic data, and approaches that concentrate on providing a conceptual framework for service descriptions. In the following we briefly discuss some of the existing approaches and discuss their applicability to the VITAL use case. In general, VITAL aims at modelling IoT services with a focus on simplicity, minimalism, reuse as well as support from an active community.

WSDL-based Semantic Service Description Languages

WSDL-based approaches to describe semantic web services, like OWL-S [MBH+04], Semantically Annotated WSDL (SAWSDL) [FaLa07], and the Web Service Modelling Ontology (WSMO) [LPR05] focus on generic and complex semantic extensions of WSDL to describe SOAP-based web services. As an example, OWL-S consists of three parts that are used to specify three aspects of web services: the service profile (specifying what the service does for potential clients, e.g. name, description and quality of service levels), the process model (specifying how to use a service on a semantic level, e.g. input and output messages), and the service grounding (specifying details on how to access a service on a technical level, e.g. protocols and addresses). To fully specify the service grounding, OWL-S relies on an additional (external) specification. Most commonly WSDL is used. Despite the long availability of approaches such as OWL-S (having been published as a W3C member submission in 2004), they have received limited uptake and do not take into account newer developments and technologies such as REST.

Light-weight Semantic Service Description Languages

To accommodate for this, a number of newer specifications have been developed. Microformats such as Semantic Annotations for REST (SA-REST) [GRS10] and HTML for restful Services (hRESTS) [KGV08] embed additional metadata into (X)HTML-based textual descriptions of (usually RESTful) web services. This metadata can be used to describe services in more detail and allows creation of machine understandable descriptions that can be used for discovery and automatic orchestration. MicroWSMO [KVF09] extends hRESTS with the ability to enhance a service description with external semantic descriptions as well as transformation routines to map between raw and semantic data. It is similar to SAWSDL but simplified to be used for RESTful services, only.

Higher Layer Conceptual Integration Frameworks

WSMO-Lite [VKVF08] is a lightweight subset of WSMO that provides a number of semantic annotations to describe web services. It can be used in conjunction with different lower layer ontologies and languages, e.g. on top of SAWSDL/WSDL for SOAP services and on top of MicroWSMO/hRESTS for RESTful services.

The Minimal Service Model (MSM)[8] [KGV08] [PML+10] is a very lightweight ontology for the semantic modelling of Web service descriptions. It consists of only a few general concepts (e.g. a service and an operation) that are not detailed further in MSM. Instead, MSM provides a conceptual framework to integrate semantic descriptions that have been created using different other approaches. To actually specify a service, MSM must be combined with other ontologies, e.g. hRESTS, MicroWSMO and WSMO-Lite. While WSMO-Lite integrates semantic annotations, MSM integrates the basic concepts of a service description.

5.2.2 VITAL IoT Service Model

There is currently no single, standardised way to model IoT services. Based on the related work discussed before, the VITAL consortium decided to base its semantic IoT service model on existing work in the domain of web services. As discussed before, VITAL aims at providing a semantic model that is generic – yet simple and minimal, reuses existing ontologies as much as possible and allows to link with an active community as well as other current projects.

After careful consideration, the consortium selected to use the MSM as the basis of its IoT modelling ontology. MSM is small and easy to understand while at the same time providing integration with other languages like SAWSDL, WSMO-Lite and hRESTS. MSM is also widely used, e.g. by SOA4All [KNSP09], iServe [isrv], as well as the FP7 project COMPOSE [com]. COMPOSE focuses on transforming the IoT into an Internet of Services. By aligning VITAL with COMPOSE, applications using the VITAL platform will be able to participate in the open service marketplace envisioned in COMPOSE. At the same time, VITAL will be able to easily integrate all services offered by the COMPOSE marketplace. In addition, using MSM also allows a direct integration with iServe [isrv], an online service warehouse supporting service publication, analysis, and discovery. Nevertheless, the consortium will keep

[8] http://iserve.kmi.open.ac.uk/ns/msm/msm-2013-05-03.html

monitoring all usage and experiences with the IoT service ontologies and may extend the used model to integrate further ontologies, e.g. OWL-S.

With this in mind, we can specify the following: In the VITAL system an IoT service is modelled as a RESTful (web) service that is described by Linked Data using the MSM ontologies. This allows publishing a description of the IoT service that can e.g. be used for discovery or for automatic composition tasks. An example for such an IoT service is given in Figure 12: and Figure 13: . Other than for earlier examples, to showcase how the JSON-LD file is mapped to RDF triples, we provide the same example twice, first in N3[9] notation, then in JSON-LD format using the context shown in Figure 14: .

```
@prefix : <http://vital-iot.eu/ontology/ns/> .
@prefix hrest: <http://www.wsmo.org/ns/hrests#> .
@prefix msm: <http://iserve.kmi.open.ac.uk/ns/msm#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix vital: <http://vital-iot.eu/ontology/ns/> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<http://example.com/service/1> a vital:ConfigurationService;
rdfs:label "IoT System Configuration Service";

    msm:hasOperation [ a vital:GetConfiguration ;
            rdfs:label "Get configurations";

            hrest:hasAddress
"http://example.com/service/1"^^hrest:URITemplate ;
            hrest:hasMethod "hrest:GET" ],
        [ a vital:SetConfiguration ;
            rdfs:label "Set configurations";


            hrest:hasAddress
"http://example.com/service/1"^^hrest:URITemplate ;
            hrest:hasMethod "hrest:POST" ].
```

Figure 12: Example IoT Service for Configuration (N3)

In the example, we specify an IoT service for IoT system configuration that allows a user to request the current configuration of the system using their identifier (ID) by sending a HTTP GET request to the URI representing the configuration service. Using VITAL ontology, we first specify a service `vital:ConfigurationService` that has two operations `vital:GetConfiguration` and `vital:SetConfiguration`. In addition, we assign a label to the service that helps to describe the service for human readers. The operations again specify labels intended for human readers and in addition specify how to access the operations using the hRESTs ontology by giving its address template and the used HTTP method (in this case GET and POST). The address template allows to specify not only a single, fixed URI but a set of URIs (of type `hrest:URITemplate`) that can be

[9] http://www.w3.org/DesignIssues/Notation3.html

used with this operation. In this example, the address template specifies that the operation can be called with the ID of configuration service. As an example, calling the operation `GetConfiguration` with address `http://example.com/service/1` would get the configurations of the IoT system.

```
{
    "@context": "http://vital-iot.eu/contexts/service.jsonld",
    "id": "http://example.com/service/1",
    "type": "vital:ConfigurationService",
    "name" : "IoT System Configuration Service",
    "operations":
    [
      {
        "type": "vital:GetConfiguration",
        "hrest:hasAddress": "http://example.com/service/1",
        "hrest:hasMethod": "hrest:GET"
      },
      {
        "type": "vital:SetConfiguration",
        "hrest:hasAddress": "http://example.com/service/1",
        "hrest:hasMethod": "hrest:POST"
      }
    ]
}
```

Figure 13: Example IoT Service for Configuration

Figure 13: shows the same example again, this time in JSON-LD format. Note that we could omit the `type` entry to achieve a more compact description. However, we did not do this to make it easier to compare the two representations of the example.

```
{
  "@context": {
    "@vocab": "http://vital-iot.eu/ontology/ns/",
    "vital": "http://vital-iot.eu/ontology/ns/",
    "rdfs": "http://www.w3.org/2000/01/rdf-schema#",
    "msm": "http://iserve.kmi.open.ac.uk/ns/msm#",
    "hrest": "http://www.wsmo.org/ns/hrests#",
    "id": "@id",
    "type": "@type",
    "name": "rdfs:label",
    "description": "rdfs:comment",
    "hrest:hasAddress": {
      "@type": "hrest:URITemplate"
    },
    "operations": {
      "@id": "msm:hasOperation",
      "@type": "@id"
    }
  }
}
```

Figure 14: JSON-LD Context Specification for Services

Using this combination of commonly used ontologies, IoT systems can specify their generic services towards the VITAL platform as well as to client applications. The VITAL discovery service can use this information to allow clients to discover IoT services at runtime. The VITAL orchestration service can use it to provide semi-automatic service orchestrations or to provide users with lists of equivalent IoT services. Clearly, this IoT service model can be extended, e.g. with the ability to specify other communication protocols. The consortium has decided to extend the IoT service model with the following services.

5.2.3 Configuration Service

IoT systems (integrated with VITAL platform) may provide configuration functionalities. In order to model these functionalities, `ConfigurationService` class is defined as a sub class of `msm:Service` along with two operations: `GetConfiguration` (to access existing configurations) and `SetConfiguration` (to set new configurations):

```
Class: ConfigurationService
  SubclassOf: msm:Service


Class: GetConfiguration
  SubclassOf: msm:Operation


Class: SetConfiguration
  SubclassOf: msm:Operation
```

Examples of ConfigurationService and its operations can be found in deliverable D3.2.2 (See section 3.2).

5.2.4 Monitoring Service

An IoT system can allow VITAL to monitor a number of monitoring functionalities. For example, the status of an IoT system, the status of sensors that an IoT system manages, performance metrics of an IoT system, SLA parameters related to an IoT system, etc. These functionalities are exposed by a MonitoringService class a sub class of `msm:Service` with a number of operations:

- **`GetSystemStatus:`** To access to status description of an IoT system.
- **`GetSensorStatus:`** To access to status description of a sensor within an IoT system.
- **`GetSupportedPerformanceMetrics:`** To access the performance metrics supported by an IoT system.
- **`GetPerformanceMetrics:`** To access the performance metrics of an IoT system.

- **`GetSupportedSLAParameters:`** To access the SLA parameters supported by an IoT system.
- **`GetSLAParameters:`** To access the SLA parameters of an IoT system.

```
Class: MonitoringService
  SubclassOf: msm:Service


Class: GetPerformanceMetrics
  SubclassOf: msm:Operation


Class: GetSensorStatus
  SubclassOf: msm:Operation


Class: GetSLAParameters
  SubclassOf: msm:Operation


Class: GetSupportedPerformanceMetrics
  SubclassOf: msm:Operation


Class: GetSupportedSLAParameters
  SubclassOf: msm:Operation


Class: GetSystemStatus
  SubclassOf: msm:Operation
```

Examples of MonitoringService and its operations can be found in deliverable D3.2.2 (See section 3.2).

5.2.5 Observation Service

The VITAL platform can use both a pull and push based mechanism to obtain observations made by a sensor. An IoT system with various sensors can provide/support both mechanism by providing an observation service. An IoT system must support at least one of these two mechanisms in order to allow access to sensor observations. This service is modelled as ObservationService sub class of `msm:service` with the following operations:

- **`GetObservations:`** If the IoT system supports a pull-based mechanism, it must provide this operation to pull observations.

- **`SubscribeToObservationStream:`** To subscribe to a specific stream of observations in the case of push-based mechanism supported by an IoT system.
- **`UnsubscribeFromObservationStream:`** To unsubscribe from a specific stream of observations (for which there is an active subscription) in the case of push-based mechanism supported by an IoT system.

```
Class: ObservationService
  SubclassOf: msm:Service


Class: GetObservations
  SubclassOf: msm:Operation


Class: SubscribeToObservationStream
  SubclassOf: msm:Operation


Class: UnsubscribeFromObservationStream
  SubclassOf: msm:Operation
```

5.3 Conclusion

One of the central aspects of the VITAL system is to integrate existing (background) IoT systems. To do so, we require semantic data models to describe such IoT systems, including metadata about the systems, their operational state and the services that they provide. Note that VITAL does not allow a system to directly specify what sensor data or ICOs it provides – nor is it possible to include actual sensor measurements into the description. Instead, every aspect of the system is described as an IoT service. To offer access to sensor measurements, a system can specify that it provides access to a data service. To allow pull based data access, a system can specify that it offers a stream management service. This greatly simplifies the system description and reduces its changing rate, while providing a very flexible way to describe (and integrate) systems. The actual interfaces of services are not yet fixed. This is the focus of deliverable D3.2.1, which will be available in M15. However, the general design of VITAL is based on Web technologies. Services are (so far) always RESTful HTTP services. To allow continuous (stream) access, technologies such as Server Side Events or WebSockets will be integrated into VITAL.

In the following section we describe the last area that must be modelled in VITAL, the VITAL system itself as well as its services.

6 VITAL SYSTEM AND SERVICES

In addition to (meta) data and services of background IoT systems that are integrated in VITAL, we also need to model the VITAL system itself. This is the aim of this chapter. Note that the chapter does not specify the exact interfaces of systems or services (e.g. the PPI). Instead, its main goal is to discuss data items that are required by these interfaces. Interfaces have been specified in D3.2.1 and D3.2.2.

6.1 VITAL System

A VITAL system is modelled as a collection of RESTful services and an associated semantic metadata description. From the viewpoint of its semantic description, a VITAL system is similar to an IoT system, discussed before. A VITAL system is therefore modelled as a subclass of `IotSystem` with an additional property.

6.1.1 VitalSystem

A `VitalSystem` is defined as a subclass of `IotSystem` that supports an additional VITAL-specific property `providesSystem`. The property `providesSystem` is a sub property of `ssn:hasSubSystem` that points to a set of *IotSystems*, which are registered in this `VitalSystem`. Note that since a `VitalSystem` is a subset of `IotSystem`, `providesSystem` can also point to another VITAL system, making it possible to create hierarchical VITAL systems.

```
Class: VitalSystem
    SubClassOf: IotSystem

ObjectProperty: providesSystem
    SubPropertyOf: ssn:hasSubSystem
    Domain: VitalSystem
    Range: IotSystem
```

In the following we describe a short example of a VITAL system description in JSON-LD format (see

Figure 15:). Note that the example is similar to the example for an IoT system given in Figure 9. Due to the similarities the example uses the same generic external context specification for systems (see

Figure 10) as the IoT system example.

```
{
  "@context": "http://vital-iot.eu/contexts/system.jsonld",
  "id": "http://example2.com",
  "type": "vital:VitalSystem",
  "name": "Sample 2 IoT system",
  "description": "This is a VITAL compliant IoT system example.",
  "operator": "http://example.com/people#adam_murphy",
  "serviceArea": "http://dbpedia.org/page/Camden_Town",
  "sensors":
  [
    "http://example.com/sensor/1",
    "http://example.com/sensor/2",
    "http://example.com/sensor/3",
    "http://example.com/sensor/4"
  ],
  "services":
  [
    "http://example.com/service/1",
    "http://example.com/service/2",
    "http://example.com/service/3",
  ],
  "status": "vital:Running"
}
```

Figure 15: Example VITAL Description

In the example VITAL system, the description first associates a `name` ("Camden VITAL System") and a `description` (stating that this is a VITAL deployment for Camden Town) with this system. The example system is operated by CTU (as specified by `operator`). The `status` is set to `Running`. Similar to the example for an IoT system, the `serviceArea` is identified by providing its DBPedia URI, allowing a client to query the DBPedia for more information about Camden. The example system description further specifies that this VITAL instance contains four sensors. Finally, the VITAL system description specifies three VITAL services, similar to the ones presented for our example IoT system before. Please note that the exact interfaces of VITAL services are defined in D3.1.2 and D3.2.2. Therefore, the examples given here may differ slightly.

In the following we describe the currently planned VITAL services. We also provide models for data required by these services. Since VITAL services are currently under development, the exact data that they need is not always clear. Therefore, this part of the deliverable will be clarified and expanded in future versions. For now the information given should be understood as an outlook and basis for discussions in the consortium.

6.2 Data Access

Data access is one of the basic services of any IoT system, including VITAL. VITAL will provide two different service types with different data access patterns. The first one is a pull-based data access service. The second one is a push-based linked data stream service. For pull-based access, a client specifies the system entity, e.g. the ICO or sensor reading type that it wants to retrieve using its URI. The data access service is modelled as ObservationService sub class of `msm:service`. See Section 5.2.5 for details on observation service.

6.3 Security and Access Control

6.3.1 Access Control

There are many ontologies available for access control, each utilizing different approaches that provide unique benefits. We evaluated a number of these based on maturity and features offered.

WebAccessControl [WAC] is an ontology created by the Read Write Web Community Group that allows giving access to serialized RDF documents to users and groups, identifying each as HTTP URLs. The main issue with WAC is that it was designed to grant access to the entire document, which makes it unsuited to platforms where more fine-grained permissions are required.

PPO [PPO] and TAC [TAC] are both ontologies designed to extend WAC and allow defining access on a triple-by-triple basis. Which these are more powerful than WAC and allow setting access for each triple, they do not define what exactly access is beyond a Boolean-type value. Both of these ontologies could potentially be extended to better serve VITAL's usage scenario.

Social Semantic SPARQL Security For Access Control Ontology (S4AC[10]) [S4AC] was developed in response to WAC's shortcomings like the two aforementioned ontologies. However, S4AC allows setting fine-grained create, read, update and delete permissions for each data item. This is highly compatible with the permissions model used for traditional REST APIs.

After evaluating all of the above vocabularies we decided to use S4AC for VITAL. The main reason for this decision is that S4AC was the only one to use the CRUD approach for permissions. This kind of fine-grained control is important for a system like VITAL where much of the information is highly confidential.

Also, the S4AC access control modelling provides a solid foundation for a versatile solution. In fact, S4AC allows to use SPARQL ASK clauses where the condition to be satisfied can be specified in a very flexible way, considering also any desired aspect of the evaluation context.

6.3.2 Users and Authentication in VITAL

Users in VITAL are represented through the class User, as an extension of the Friend of a friend ontology [FOAF].

The Agent class is the class of agents; things that do stuff, which can represent people, organizations or groups. The class can be used when these are overly specific.

```
Class: User
    SubClassOf: foaf:Agent
ObjectProperty: providesSystem
```

[10] http://ns.inria.fr/s4ac/v2/s4ac_v2.html

```
SubPropertyOf: ssn:hasSubSystem
Domain: VitalSystem
```

```
{
  "@context": "http://vital-iot.eu/contexts/user.jsonld",
  "@id": "http://www.example.com/vital/users/34423",
  "vital:userID": 34423,
  "@type": "foaf:Person",
  "foaf:name": "Joe Bloggs"
}

{
  "@context": "http://vital-iot.eu/contexts/user.jsonld",
  "@id": "http://www.example.com/vital/users/999",
  "vital:userID": 999,
  "@type": "foaf:Person",
  "foaf:name": "Dave Barlow"
}

{
  "@context": "http://vital-iot.eu/contexts/user.jsonld",
  "@id": "http://www.example.com/vital/user/23",
  "vital:userID": 23,
  "@type": "foaf:Group",
  "foaf:name": "Sensor 100-200 Administrators",

  "foaf:member": [
    "http://www.example.com/vital/users/999"]
}
```

```
Range: IotSystem
```

Figure 16: Examples of Users and Groups in VITAL

Figure 16 shows some examples for modelling users and groups in VITAL. The first two examples show definitions for two users. The third example shows defining a group that contains the administrators, in this case user 999, for a set of sensors. A unique identifier, Vital:UserID, is an integer used as an indirect identifier to represent a user.

Users and groups are distinguished by their type, which is `foaf:Person` for users and `foaf:Group` for groups. Groups may also contain `foaf:member` which are collections of URIs pointing to users belonging to that group.

A new VITAL system will always contain a group with `foaf:name` “Administrators” and `foaf:member` with a URI pointing to the initial user who setup the system. The members of this group have full access to the entire system. By creating new groups associated with access control policies defined in section 6.3.3 a VITAL administrator can easily create fine-grained permissions for members.

6.3.3 Integrated Security Model

VITAL uses an identity management framework and authentication and authorization mechanisms for controlling access to VITAL services. This access control is based on pattern-matching on the resource URL; it will be enhanced with access control mechanisms at a finer level based on flexible policies. For instance, it shall be possible to selectively authorize direct or indirect access to observations of a specific sensor.

For this VITAL uses a system similar to an Access Control List [ACL], using resources URIs as keys. When a resource is requested via a REST endpoint, the URI of that resource along with the user’s ID and operation type (using CRUD) is sent to the access control server after the user’s identity has been verified. The server then checks if the user’s ID is present in the list of authorized users and groups and if the operation type is authorized. Finally, the server responds with a success or fail.

If the requested URI has no permissions associated with it the server will assume only members of the Administrators group can access it. If the resource contains the `foaf:member` property but not the `s4ac:hasAccessPrivilege` property the members of the User Set will be granted Read-Only permissions. In the reverse case, defining permissions without adding members to the members field has no effect.

```
{
  "@id": "http://www.example.com/vital/access-control/patterns/34",
  "pattern": "http://www.example.com/vital/sensor/123/obsvn/*",
  "vital:hasAccessPolicy": [
    {
      "@type": "s4ac:AccessPolicy",
      "s4ac:hasAccessPrivilege": [ "s4ac:Read" ],
      "s4ac:hasAccessConditionSet": {
        "@type": "s4ac:DisjunctiveAccessConditionSet",
        "s4ac:hasAccessCondition": [
          {
            "foaf:member": [
              "http://www.example.com/vital/users/34423",
              "http://www.example.com/vital/users/23",
              "http://www.example.com/vital/users/500"
            ]
          }
        ]
      }
    },
    {
      "@type": "s4ac:AccessPolicy",
      "s4ac:hasAccessPrivilege": [
        "s4ac:Create", "s4ac:Read", "s4ac:Update", "s4ac:Delete"
      ],
      "s4ac:hasAccessConditionSet":
      {
          "s4ac:hasAccessCondition": [
          {
            "foaf:member": [
              "http://www.example.com/vital/users/457"
            ]
          }
        ]
      }
    }
  ]
```

Figure 17: Access Control Policy for Single Sensor's Observations

Figure 17: describes the access control policy for all observations of Sensor 123. The pattern property describes the pattern that should match this access control policy. While the fine-grained access control system has not yet been implemented this example is based on the assumption that the policy evaluation will always pick the most specific match for the provided resource URL and for the subject.

In this example, the users or groups with the IDs 34423, 23 and 500 have read-only access to sensor 123’s observations, while 457 has full access.

6.4 Trust

VITAL trust will focus on securing the main value of the platform, that is information, while trying to provide a mechanism to users and applications which let choose the more reliable data sources. These data sources are the ICOs and services provided by the platforms connected to vital through PPIs.

The model of Trust has two main parts, one related to the entities being evaluated and other related to the parameters or properties used to calculate the Trust level.

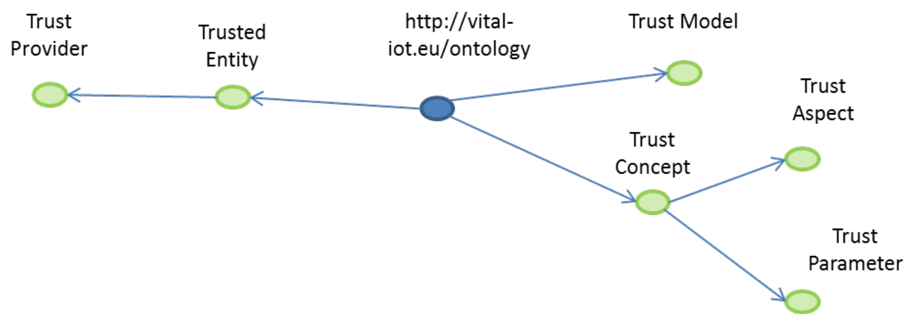

Figure 18: Trust Model.

The parameters evaluated in VITAL in order to calculate the Trust level of the entities are the SLA parameters, as this can be easily applied to any service or sensor.

Trust Parameter: It is a concept that can be measured directly without needing to combine other measures. They can be considered unitary concepts which are grouped in a Trust Aspect. Into the VITAL Ontology a Trust Parameter is an observation:

Trust parameter is a class of type **"ssn:observation"** which are shown in Figure 19.

```
    "type": "http://vital-iot.eu/ontology/ns/ResponseTime",
    "type": "http://vital-iot.eu/ontology/ns/StatusCode",
    "type": "http://vital-iot.eu/ontology/ns/UptimeAvailability",
    "type": "http://vital-iot.eu/ontology/ns/MaxNumReq",
    "type": "http://vital-iot.eu/ontology/ns/ResponseTime",
    "type": "http://vital-iot.eu/ontology/ns/TimeToRestore",
    "type": "http://vital-iot.eu/ontology/ns/MaxNumReqPUser",
    "type": "http://vital-iot.eu/ontology/ns/DataSheetVoxSynt",
    "type": "http://vital-iot.eu/ontology/ns/DataCorrelation",
    "type": "http://vital-iot.eu/ontology/ns/ValIntoThreshold",
    "type": "http://vital-iot.eu/ontology/ns/Variance",
    "type": "http://vital-iot.eu/ontology/ns/StdDeviation",
    "type": "http://vital-iot.eu/ontology/ns/RandomVal",
```

Figure 19: Types of Trust Parameters.

Trust Aspect: it represents a property to be evaluated in order to calculate the trust, which is composed by a group of Trust parameters.

At the time of writing this document there is one only Trust Aspect, which is the composition of the Trust parameters related to the numeric stability of the data. The name of this aspect is “DataStability”, it is a class into Vital ontology which contains a list of observation of SLA type (see Figure 20).

```
{
  "@context": "http://vital-iot.eu/contexts/Aspect.jsonld",
  "name":"DataStability",
  "type":”http://vital-iot.eu/ontology/ns/TrustAspect”
  “uri”:http://www.example.com/vital/sensor/123/dataStability”
  “ssn:observers”:[
    {
      "type": " http://vital-iot.eu/ontology/ns/ValIntoThreshold",
      "id": "http://example.com/sensor/1/ ValIntoThreshold "
    },
    {
      "type": " http://vital-iot.eu/ontology/ns/Variance",
      "id": "http://example.com/sensor/1/Variance"
    },
    {
      "type": " http://vital-iot.eu/ontology/ns/StdDeviation",
      "id": "http://example.com/sensor/1/StdDeviation"
    },
    {
      "type": " http://vital-iot.eu/ontology/ns/RandomVal",
      "id": "http://example.com/sensor/1/RandomVal"
    },
  ]
}
```

Figure 20: Example of DataStability aspect.

Trust Concept: it is any entity, which can be included in a trust model. It can be a Trust Aspect or a Trust Parameter.

Name: name of the concept.

Weight: it is a number that represents the weight of the trust concept within the trust model. Figure 21 provides an example.

```
  {
    "@context": "http://vital-iot.eu/contexts/concept.jsonld",
    "name":"DataStability",
    "type":”http://vital-iot.eu/ontology/ns/TrustConcept”,
    “uri”:http://www.example.com/vital/sensor/123/dataStability”,
    “vital:weight”:”0,5”
  },
  {
    "@context": "http://vital-iot.eu/contexts/concept.jsonld",
    "name":"DataStability",
    "type":”http://vital-iot.eu/ontology/ns/TrustConcept”,
    “uri”:http://www.example.com/vital/sensor/123/UptimeAvailability”,
    “vital:weight”:”0,3”
  }
]
```

Figure 21: Trust Concept example

Trust Model: it is the model used to evaluate the trustworthiness of a service. It will contain a set of properties or parameters and the weight in the evaluation.

Name: name of the model

Aggregates: A set of Trust concepts, which are used to calculate the trust within the model. Figure 22 provides an example for Trust Model.

```
{
  "@context": "http://vital-iot.eu/contexts/Model.jsonld",
  "name":"model2",
  "type":”http://vital-iot.eu/ontology/ns/TrustModel”,
  “uri”:”http://www.example.com/vital/trustModel2”,
  “vital:TrustConcept”:[
    {
      “uri”:http://www.example.com/vital/sensor/123/dataStability”
    },
    {

“uri”:http://www.example.com/vital/sensor/123/UptimeAvailability”
    }
  ]
}
```

Figure 22: Trust Model example.

Trusted Entity: this is a concept that defines all entities that can be trusted, these entities have to be identified in one only way and provide information about how it should be evaluated. Figure 23 provides an example for Trusted Entity. In VITAL a trusted entity can be a sensor or a service**.**

ID: the identifier

Trust Model: the model used to evaluate the entity,

Trust Level: it is the result of the trust evaluation.

```
{
  "@context": "http://vital-iot.eu/contexts/TrustEntity.jsonld",
  "name":"sensor12",
  "type":”http://vital-iot.eu/ontology/ns/TrustedEntity”,
  “uri”:http://www.example.com/vital/sensor12/trusted”,
  “vital:model”:
  {
    “type”: ”http://vital-iot.eu/ontology/ns/TrustModel”,
    “uri”: http://www.example.com/vital/trustModel2
  },
  “vital:TrustLevel”:”0,87”
}
```

Figure 23: Trusted Entity example.

At the time of writing this document there is any platform connected to VITAL which performs Trust evaluation of its data sources, the following concept is introduced to cope with new platforms that will be able to provide information about the level of trust of their sensors and services.

Trust Provider: It is an entity, which can be trusted and is able to evaluate trust of other entities.

AppliesModel: it represents the model applied by the provider to evaluate the trust of its associated entities.

TrustEntities: it is a list of the entities evaluated by the provider.

6.5 Discovery and Filtering

The VITAL discovery and filtering services will rely on metadata to be provided by the DMS. The full description and implementation of these two modules are investigated in WP4 and documented in D4.1 and D4.1.2. Please refer to these deliverables for details, we only report here the general idea and the connections between the DMS and Discovery and Filtering modules. The Discovery and Filtering services concentrate on ICOs, especially sensor devices. They propose basic mechanisms and enhanced mechanisms to provide more accurate results based on a sensor's location (if available), its movement pattern (if known) and its network connectivity (i.e. whether it is connected intermittently or continuously). The current version of the discovery modules tries to predict the position of the ICO at the date of the request and when it will be able to report its date. For this purpose, we have extended a sensor description (`ssn:Sensor`) with the following properties:

- `hasMovementPattern`, a mandatory property that links to an instance of `MovementPattern`,
- `hasNetworkConnection`, an optional property that links to an instance of `NetworkConnection`.
- `hasLocalizer`, an optional property that links to an IoT service specification that provides access to the current location of the sensor.

Note that the location of a sensor is already modelled in VITAL using the `hasLastKnownLocation` property.

```
ObjectProperty: hasMovementPattern
    Domain: ssn:Sensor
    Range: MovementPattern
```

```
ObjectProperty: hasNetworkConnection
    SubPropertyOf: ssn:hasSubSystem
    Domain: ssn:Sensor
    Range: NetworkConnection
```

```
ObjectProperty: hasLocalizer
    SubPropertyOf: ssn:hasSubSystem
    Domain: ssn:Sensor
    Range: msm:Service
```

6.5.1 MovementPattern

A movement pattern specifies how a device is moving or is expected to move. VITAL currently defines three basic movement patterns: `Stationary` (i.e. no movement at all), `Mobile` (i.e. no additional information is known), and `Predicted` (i.e. there is data available to predict a sensor's future mobility). Note that it is possible to include multiple values for a movement pattern, stating e.g. that a sensor movement pattern is mobile and predicted.

Stationary

If a sensor is stationary, its `hasMovementPattern` property points to an instance of `Stationary`. In this case the sensor may also have a `hasLastKnownLocation` property, pointing to its location. If no such property is present, the sensor location is (currently) unknown. Note that this might change, so a client cannot assume that the sensor location will not be known in the future.

```
Class: Stationary
    SubClassOf: MovementPattern
```

Mobile

A `Mobile` pattern implies that a sensor may change its location dynamically. Note that this does not necessarily imply that the location is known. If the description of a mobile sensor includes a `hasLastKnownLocation` property, then the provided location *may* be out-dated (which is why we call the property 'last known location'). A system *should* update the last know location field in the sensor description but it *can* do so when it chooses. This reduces the load on a system, which otherwise would need to update a sensor description every time a new location reading becomes available. A sensor description *should not* include a last known location if the system providing it does not update it. A mobile sensor *should* include a `hasLocalizer` property in its description, pointing to a localisation service.

```
Class: Mobile
    SubClassOf: MovementPattern
```

Predicted

If the movement pattern of a sensor is set to `Predicted`, then the system has additional information about the sensors movement pattern to predict future movements and locations.

```
Class: Predicted
    SubClassOf: MovementPattern
```

It is currently unclear which exact movement pattern data is needed as this depends on the algorithms that will use them. This is therefore left for future work. An example for a possible prediction pattern is given in

Figure 24: . The example shows a sensor with a predicted movement pattern that is based on linear interpolation using a predicted speed and movement direction.

```
{
  "@context": "http://vital-iot.eu/contexts/sensor.jsonld",

  "name": "TemperatureSensor No.123",
  "type": "vital:VitalSensor",
  "description": "This is an example sensor",
  "id": "http://www.example.com/vital/sensor/123",
  "hasMovementPattern": {
    "type": "Predicted",
    "hasPredictedSpeed": {
      "value":"3.1",
      "qudt:unit": "qudt:KilometerPerHour"
    },
    "hasPredictedDirection": {
      "type":"NormalVector",
      "geo:lat": "53.2719",
      "geo:long": "-9.0489"
      },
    }
  },
  "ssn:observes": {
    "type": "http://lsm.deri.ie/OpenIoT/Temperature",
    "id": "http://www.example.com/vital/sensor/123/temperature"
  }
}
```

Figure 24: Example Sensor Description with Predicted Movement Pattern

6.5.2 NetworkConnection

The NetworkConnection specifies how a sensor is connected at the moment, e.g. whether the connection is stable or not. It does not necessarily provide a fully detailed description of the network quality of service that is available. However, it always specifies the expected connection stability. To realise this, so far `NetworkConnection` supports two properties:

- **`hasStability`**, a mandatory property linking to a `ConnectionStability` instance,
- **`hasNetworkSupport`**, an optional property that is a sub property of `net:networkSupport` taken from the Network part of the DC ontology [FoLe09], linking to an instance of `net:NetworkSupport` from the same ontology.

```
ObjectProperty: hasStability
    Domain: NetworkConnection
    Range: ConnectionStability

ObjectProperty: hasNetworkSupport
    SubPropertyOf net:networkSupport
    Domain: ssn:Sensor
    Range: net:NetworkSupport
```

An example for these properties in given in Figure 25: . This example also contains an example for `ConnectionStability`. The described sensor has a stable, continuous network connection and is connected to a single, wired network. More information about this network could be added, e.g. the bandwidth, etc.

```
{
  "@context": "http://vital-iot.eu/contexts/sensor.jsonld",

  "name": "TemperatureSensor No.123",
  "type": "vital:VitalSensor",
  "description": "This is an example sensor",
  "id": "http://www.example.com/vital/sensor/123",
  "hasMovementPattern": {
    "type": "Stationary",
  },
  "hasNetworkConnection": {
    "hasStability": {
      "type": "Continuous"
    },
    "hasNetworkSupport": {
      "net:connectedNetworks": {
        "type": "net:WiredNetwork"
      }
    }
  }
}
```

Figure 25: Example Sensor Description with Network Connection Data

6.5.3 ConnectionStability

`ConnectionStability` allows specifying whether a sensor is connected to a communication network continuously, intermittently or not at all (which will usually not happen). To do so we define three subclasses `Continuous`, `Intermittent` and `Disconnected`. An example for a stable connection is given in Figure 25: above.

```
Class: Continuous
    SubClassOf: ConnectionStability


Class: Intermittent
    SubClassOf: ConnectionStability


Class: Disconnected
    SubClassOf: ConnectionStability
```

6.5.4 Localizer

The property `hasLocalizer` is used to refer to a *localizer service*. Such a service provides (REST-full) access to the current location of a sensor, which is especially important for mobile sensors. The service is modelled as an IoT Service as specified in Section 5.2. Note that the localizer service can be realised in different ways, depending on the sensor at hand. As an example, the sensor could provide the service itself, e.g. by giving access to its local GPS receiver. Alternatively the sensor could be localised using an external tracking system and the localizer service could be provided by the IoT system that manages the sensor. An example for a sensor-based localizer service is given in Figure 26**Error! Reference source not found.**.

```
ObjectProperty: hasLocalizer
    Domain: ssn:Sensor
    Range: msm:Service
```

```
{
  "@context": "http://vital-iot.eu/contexts/sensor.jsonld",

  "name": "TemperatureSensor No.123",
  "type": "vital:VitalSensor",
  "description": "This is an example sensor",
  "id": "http://www.example.com/vital/sensor/123",
  "hasMovementPattern": {
    "type": "Mobile"
  },
  "hasLocalizer": {
    "type": "GpsService",
    "msm:hasOperation": {
      "type": "GetLocation",
      "hrest:hasMethod": "hrest:GET",
      "hrest:hasAddress":
        "http://www.example.com/vital/sensor/123/location/"
    },
  },
  "ssn:observes": {
    "type": "http://lsm.deri.ie/OpenIoT/Temperature",
    "id": "http://www.example.com/vital/sensor/123/temperature"
  }
}
```

Figure 26: Example Sensor Description With Localizer

6.5.5 Queries

In addition to all data items described before, the discovery and filtering services require a way to specify a requested data query, either for ICOs in the case of the discovery service or for data in the case of the filtering service. Due to the nature of Linked Data we do not require a new data model for this. Instead VITAL reuses existing query languages that are well known in the Linked Data community. For onetime queries, VITAL reuses SPARQL, which allows specifying complex sub graph patterns. A SPARQL endpoint will be made available as a RESTful service. Similarly, for continuous (streaming) queries, VITAL reuses the CQELS query language [cql], which extends SPARQL with support for data streams and time windows.

6.6 Complex Event Processing

The CEP module will provide the CEP filtering services by means CEP instances, these services are divided in two different types, the static ones and the continuous. For this purpose we have extended the `msm:Service` description with a new service:

```
Class: CEPFitleringService
  SubclassOf: msm:Service
```

The static filters provide two different operations the filter static data operation the filter static query operation and to that end we have extended `msm:Operation` in order to provide the required functionality:

```
Class: FilterStaticData
  SubclassOf: msm:Operation

Class: FilterStaticQuery
  SubclassOf: msm:Operation
```

Filters are a new kind of vital sensor and to that end we have extended the `vital:`**`VitalSensor`** with the CEPFilterSensor and also we have extended these new sensors with two different kind of sensors, the one for filtering static data and the one for filtering static query:

```
Class: CEPFilterSensor
     SubClassOf: vital:VitalSensor

Class: CEPFilterStaticDataSensor
     SubClassOf: vital:CEPFilterSensor

Class: CEPFilterStaticQuerySensor
     SubClassOf: vital:CEPFilterSensor
```

The CEPFilterStaticDataSensor filters the data provided as an input of the filter for this purpose we need to add a new property to this kind of sensors; to that end we provide a new property:

```
ObjectProperty: data
     Domain: vital:CEPFilterStaticDataSensor
     Range: Range: xsd:string
```

The CEPFilterStaticQuerySensor filters the data received as a response of a query provided as an input of the filter for this purpose we need to add a new property to this kind of sensors; to that end we provide a new property:

```
ObjectProperty: query
     Domain: vital:CEPFilterStaticQuerySensor
     Range: xsd:string
```

In order to provide the continuous filtering functionality we have extended the `msm:Operation` with four new operations. These operations allow us to get, create and delete the CEP filtering instances:

```
Class: GetContinuousFilters
     SubclassOf: msm:Operation

Class: GetContinuousFilter
     SubclassOf: msm:Operation

Class: DeleteContinuousFilter
     SubclassOf: msm:Operation

Class: CreateContinuousFilter
     SubclassOf: msm:Operation
```

The CEP module also provides CEP instances integrated into the Vital platform as virtual ICOs, the CEPICOs. For this purpose we have extended the `msm:Service` description with a new service:

```
Class: CEPICOManagementService
     SubclassOf: msm:Service
```

In order to be able to create, delete or get this CEPICOs we have extended the `msm:Operation` with four new operations. These operations allow us to manage all the CEPICOs instances:

```
Class: GetCEPICOs
     SubclassOf: msm:Operation

Class: GetCEPICO
```

```
    SubclassOf: msm:Operation

Class: CreateCEPICO
    SubclassOf: msm:Operation

Class: DeleteCEPICO
    SubclassOf: msm:Operation
```

The most important of a CEP instance, either as a CEP sensor or as a CEP filter sensor are the Dolce Rules that models the CEP behaviour. In order to add the Dolce Rule Specification property to the sensor we have added a new property:

```
ObjectProperty: dolceSpecification
    Domain: vital:CEPSensor vital:CEPFilterSensor
    Range: xsd:string
```

CEP sensors and CEP filter sensors detects complex events over specified observations so a new measurement type is needed to express this kind of events to that end a new class is provided:

```
Class: ComplexEvent
```

6.7 Monitoring

For purposes of monitoring, IoT systems expose an IoT Service with type ‘Monitoring Service’. This service provides access to a set of performance metrics as they are measured by the IoT system. These additional metrics alongside the metadata descriptions of Systems, Services and Sensors is exploited by higher-level modules like the Management Platform to monitor the overall health of a Vital installation.

6.7.1 Performance Metrics

Performance Metrics of the monitoring service are modelled as virtual sensor measurements. This makes it possible to reuse much of the data models defined before in Chapter 3.4. To retrieve these virtual sensor measurements, clients can contact RESTful interfaces of the MonitoringService exposed by systems, similar to the ones used for normal measurements.

The monitoring service can provide information about components like sensors, services and systems. The prototype version of VITAL defines a core set of performance parameters that can be extended as required. For each metric we specify a new type of observation as follows:

- **`SysLoad`**, for a measurement of the load (CPU) of the component,
- **`SysUptime`**, for a measurement of the uptime of a component,
- **`UsedMem`**, for a measurement of the used memory of a component,
- **`AvailableMem`**, for a measurement of the available memory of a component,
- **`ServedRequests`**, for a measurement of the total number of requests that a component has served between the last re-start and the time of the measurement,
- **`PendingRequests`**, for a measurement of the number of requests that were served at the time of the measurement,
- **`MaxRequests`**, for a measurement of the maximum number of requests that a component can serve simultaneously,
- **`Errors`**, for a measurement of the total number of errors that have occurred between the last re-start and the time of the measurement.

These metrics are integrated in the Vital Ontology as new observation types. An example of these types is described in Figure 27 where a response of the GetSupportedPerformanceMetrics operation of the MonitoringSevice is displayed. While this is example does not display a JSON-LD response, it showcases the actual namespaces of the observation types as used inside the Vital platform.

```
[{
    "type": "http://vital-iot.eu/ontology/ns/SysLoad",
    "id": "http://example.iot.system/sensor/monitoring/sysLoad"
}, {
    "type": "http://vital-iot.eu/ontology/ns/SysUptime",
    "id": "http://example.iot.system/sensor/monitoring/sysUptime"
}, {
    "type": "http://vital-iot.eu/ontology/ns/MaxRequests",
    "id": "http://example.iot.system/sensor/monitoring/maxRequests"
}, {
    "type": "http://vital-iot.eu/ontology/ns/Errors",
    "id": "http://example.iot.system/sensor/monitoring/errors"
}, {
    "type": "http://vital-iot.eu/ontology/ns/ServedRequests",
    "id": "http://example.iot.system/sensor/monitoring/servedRequests"
}, {
    "type": "http://vital-iot.eu/ontology/ns/AvailableMem",
    "id": "http://example.iot.system/sensor/monitoring/availableMem"
}, {
    "type": "http://vital-iot.eu/ontology/ns/UsedMem",
    "id": "http://example.iot.system/sensor/monitoring/usedMem"
}, {
    "type": "http://vital-iot.eu/ontology/ns/PendingRequests",
    "id": "http://example.iot.system/sensor/monitoring/pendingRequests"
}]
```

Figure 27: `GetSupportedPerformanceMetrics` example response

Figure 28: shows an example of a virtual measurement as reported by the monitoring service. The measurement specifies the uptime of a system with the identifier `http://www.example.com/vital/system/123` as `2023546`

milliseconds (approximately 34 minutes). Other measurements can be constructed similarly.

```
{
  "@context": "http://vital-iot.eu/contexts/measurement.jsonld",

  "uri": "http://www.example.com/vital/sensor/monitoring/obsvn/42",
  "type": "ssn:Observation",
  "ssn:observationProperty": {
    "type": "Uptime"
  },
  "ssn:observationResultTime": {
    "inXSDDateTime": "2014-08-23T14:03:11+01:00"
  },
  "observationSubject": "http://www.example.com/vital/system/123",
  "ssn:observationResult": {
    "type": "ssn:SensorOutput",
    "ssn:hasValue": {
      "type": "ssn:ObservationValue",
      "value": "2023546",
      "qudt:unit": "qudt:MilliSecond"
    }
  }
}
```

Figure 28: Example Virtual Measurement for System Uptime

6.7.2 Activity Logging

In addition to updates about the state of a system component, the monitoring service may also provide updates about activities occurring in the system. These updates could be stored in an activity log, e.g. to log a user logging in or a system starting up. Such updates can be modelled similarly to state updates as virtual measurements by specifying new observation types.

6.7.3 Sensor Hardware and Software

Since VITAL reuses the DC ontologies, it already includes a detailed data model for hardware and software components like a battery, a keyboard, an operating system, etc. These models can be used to provide a detailed description of both the components and the status of a sensor. To do so, a sensor description can include the DC ontology properties `hard:deviceHardware` and `soft:deviceSoftware`, pointing to an instance of `hard:DeviceHardware` and `soft:DeviceSoftware` respectively. These may link to additional information about the sensor's hardware and software parts, including their current state (e.g. using `hard:status` to link to an instance of `hard:HardwareStatus`) . An example for this is shown in Figure 29: . The example description specifies a sensor that is currently active, has a build in memory size of 128 kByte and a CPU with a maximum speed of 10 MHerz.

```
{
  "@context": "http://vital-iot.eu/contexts/sensor.jsonld",

  "name": "TemperatureSensor No.123",
  "type": "VitalSensor",
  "description": "This is an example sensor",
  "uri": "http://www.example.com/vital/sensor/123",
  "deviceHardware": {
    "hard:status": "hard:HardwareStatus_ON",
    "hard:builtInMemory": {
      "size": 131072
    },
    "hard:cpu": {
      "type": "hard:CPU",
      "maxCpuFrequency": 10
    }
  }
}
```

Figure 29: Example Sensor Description with Hardware Components

6.8 Orchestration

The Vital Orchestrator module acts as a consumer of data and metadata from other Vital modules, like PPIs (VitalSystems) or the DMS module. By utilising existing services in dynamic workflows it provides new higher-level services to consumers. It implements one new type of service the vital:OrchestrationService, in the Vital platform, for managing meta-services (create new ones, undeploy unnecessary ones etc). The service description in JSON-LD is documented in the Figure 30.

```
{
    "@context": "http://vital-iot.eu/contexts/service.jsonld",
    "id": "http://some.service.id",
    "type": "vital:OrchestratorService",
    "operations": [{
        "type": "vital:GetOperationList",
        "hrest:hasAddress": "http://some.url/vital-orchestrator-
web/rest/operation",
        "hrest:hasMethod": "hrest:GET"
    }, {
        "type": "vital:GetOperation",
        "hrest:hasAddress": "http://some.url/vital-orchestrator-
web/rest/operation/{id}",
        "hrest:hasMethod": "hrest:GET"
    }, {
        "type": "vital:CreateOperation",
        "hrest:hasAddress": "http://some.url/vital-orchestrator-
web/rest/operation",
        "hrest:hasMethod": "hrest:POST"
    }, {
        "type": "vital:UpdateOperation",
        "hrest:hasAddress": "http://some.url/vital-orchestrator-
web/rest/operation/{id}",
        "hrest:hasMethod": "hrest:PUT"
    }, {
        "type": "vital:DeleteOperation",
       "hrest:hasAddress": "http://some.url/vital-orchestrator-
web/rest/operation/{id}",
        "hrest:hasMethod": "hrest:DELETE"
    }, {
        "type": "vital:ExecuteOperation",
        "hrest:hasAddress": "http://some.url/vital-orchestrator-
web/rest/execute/operation/",
        "hrest:hasMethod": "hrest:POST"
    }, {
        "type": "vital:GetWorkflowList",
        "hrest:hasAddress": "http://some.url/vital-orchestrator-
web/rest/workflow",
        "hrest:hasMethod": "hrest:GET"
    }, {
        "type": "vital:GetWorkflow",
        "hrest:hasAddress": "http://some.url/vital-orchestrator-
web/rest/workflow/{id}",
        "hrest:hasMethod": "hrest:GET"
    }, {
        "type": "vital:CreateWorkflow",
        "hrest:hasAddress": "http://some.url/vital-orchestrator-
web/rest/workflow",
        "hrest:hasMethod": "hrest:POST"
    }, {
        "type": "vital:UpdateWorkflow",
        "hrest:hasAddress": "http://some.url/vital-orchestrator-
web/rest/workflow/{id}",
        "hrest:hasMethod": "hrest:PUT"
    }, {
        "type": "vital:DeleteWorkflow",
        "hrest:hasAddress": "http://some.url/vital-orchestrator-
web/rest/workflow/{id}",
        "hrest:hasMethod": "hrest:DELETE"
    }, {
        "type": "vital:ExecuteWorkflow",
        "hrest:hasAddress": "http://some.url/vital-orchestrator-
web/rest/execute/workflow",
        "hrest:hasMethod": "hrest:POST"
    }, {
        "type": "vital:GetMetaServiceList",
        "hrest:hasAddress": "http://some.url/vital-orchestrator-
web/rest/metaservice",
        "hrest:hasMethod": "hrest:GET"
    }, {
        "type": "vital:GetMetaService",
        "hrest:hasAddress": "http://some.url/vital-orchestrator-
web/rest/metaservice/{id}",
        "hrest:hasMethod": "hrest:GET"
    }, {
        "type": "vital:DeployMetaService",
        "hrest:hasAddress": "http://some.url/vital-orchestrator-
web/rest/metaservice",
        "hrest:hasMethod": "hrest:POST"
    }, {
        "type": "vital:UndeployMetaService",
        "hrest:hasAddress": "http://some.url/vital-orchestrator-
web/rest/metaservice/{id}",
        "hrest:hasMethod": "hrest:DELETE"
    }]
}
```

Figure 30: Description of vital:Orchestration Service in JSON-LD

6.9 Conclusion

VITAL handles VITAL system deployments as special cases of IoT system deployments. This allows creation of hierarchical systems that contain several VITAL systems. As an example, a smart city like Istanbul might operate a local installation of VITAL that integrates all IoT systems in Istanbul. In addition, this VITAL system might be integrated into a larger, transnational VITAL system that may e.g. also include London and other cities. Besides easier administration, this approach enables e.g. the ability to fine-tune access rights and security, possibly hiding specific services and ICOs when integrating the system into the transnational one.

As discussed before, VITAL services that have been developed, their data items have been precisely defined. In this chapter we provided a second version of data items that have been identified so far as well as the current state of our analysis for data models in different areas.

7 CONCLUSIONS

This deliverable provides the basis for the semantic (meta-) data models used in the VITAL system. We build upon Linked Data principles and technologies to provide interoperable and platform agnostic data models that are based on existing ontologies. This allows VITAL applications to integrate other data sources in the Web, resulting in a large and varied set of usable data items. Although we analysed a large number of ontologies during the design of the VITAL data models, the work is not finished. Firstly, for some system components like VITAL services, the actual data needed were not fully clear during the time of this deliverable's first version (D3.1.1). While these services have been developed, new data item are defined in this version. In addition, for other areas e.g. trust and CEP, which were not clear in D3.1.1, new data items and ontologies have been included. Secondly, there are other areas still not clear e.g. Istanbul use case. Thus, it is envisaged that more data models will be added as the project progresses.

8 REFERENCES

[ACL] "Access Control List", http://www.cisco.com/c/en/us/td/docs/ios/12_2/security/configuration/guide/fsecur_c/scfacls.html, last visited 28 August 2014.

[CBB+12] M. Compton, P. Barnaghi, L. Bermudez, R. Garcia-Castro, O. Corcho, S. Cox, J. Graybeal, M. Hauswirth, C. Henson, A. Herzog, V. Huang, K. Janowicz, W. D. Kelsey, D. Le Phuoc, L. Lefort, M. Leggieri, H. Neuhaus, A. Nikolov, K. Page, A. Passant, A. Sheth, K. Taylor, "The SSN Ontology of the W3C Semantic Sensor Network Incubator", in Web Semantics: Science, Services and Agents on the World Wide Web, vol. 17, pp. 25-32, Elsevier, 2012.

[CDN09] A. Caragliu, C. Del Bo, P. Nijkamp, “Smart cities in Europe”, in: Serie Research Memoranda 0048 (VU University Amsterdam, Faculty of Economics, Business Administration and Econometrics), 2009.

[com] COMPOSE Consortium, “COMPOSE: Collaborative Open Market to Place Objects at your Service Website”, http://www.compose-project.eu/, last visited 18 August 2014.

[CoWi01] B. Corona, S. Winter, "Datasets for pedestrian navigation services" in Angewandte Geographische Informationsverarbeitung, Proceedings of the AGIT Symposium, 2001

[CPF+04] H. Chen, F. Perich, T. Finin, and A. Joshi, “Soupa: Standard ontology for ubiquitous and pervasive applications,” in Mobile and Ubiquitous Systems: Networking and Services, International Conference on, 2004.

[cql] “Continuous Query Evaluation over Linked Stream (CQELS) – CQELS Language”, https://code.google.com/p/cqels/wiki/CQELS_language, last visited 25 August 2014.

[DSW06] J. Davis, R. Studer, P. Warren, "Semantic Web Technologies: Trends and Research in Ontology based Systems", John Wiley & Sons, 2006.

[dul] "DOLCE+DnS Ultralite (DUL) – The DOLCE+DnS Ultralite ontology", http://ontologydesignpatterns.org/wiki/Ontology:DOLCE+DnS_Ultralite, last visited 24 August 2014.

[ELS07] M. Eid, R. Liscano, and A. E. Saddik, "A universal ontology for sensor networks data," in Computational Intelligence for Measurement Systems and Applications (CIMSA 2007), IEEE International Conference on, 2007, pp. 59–62.

[FaLa07] J. Farrell, H. Lausen, "Semantic Annotations for WSDL and XML Schema", W3C Recommendation, http://www.w3.org/TR/sawsdl/, last visited 20 August 2014, August 2007.

[FOAF] D. Brickley, L. Miller, Friend of a Friend (FOAF) project (of Semantic Web project), http://www.foaf-project.org/, 2000, last visited 28 August 2014.

[FoLe09] J. M. Cantera Fonseca, R. Lewis, "Delivery Context Ontology", W3C Working Draft 16, http://www.w3.org/TR/2009/WD-dcontology-20090616/, last visited 25 August 2014, June 2009.

[Gal06] S. Galizia, "WSTO: A Classification-Based Ontology for Managing Trust in Semantic Web Services" in Proceedings of the 3rd European Semantic Web Conference (ESWC 2006), pp. 697-711, Budva, Montenegro, June 2006.

[GaSc14] F. Gandon, G. Schreiber, "RDF 1.1 XML Syntax", W3C Recommendation, http://www.w3.org/TR/rdf-syntax-grammar/, last visited 24 August 2014, 25 February 2014.

[GCB14] F. Gao, E. Curry, S. Bhiri, "Complex Event Service Provision and Composition based on Event Pattern Matchmaking", in Proceedings of DEBS'14, Mumbai, India, May 2014.

[GoRu06] C. Goodwin and D. J. Russomanno, "An ontology-based sensor network prototype environment," in Information Processing in Sensor Networks, Fifth International Conference on, 2006.

[GRS10] K. Gomadam, A. Ranabahu, A. Sheth, "SA-REST: Semantic Annotation of Web Resources", W3C Member Submission, http://www.w3.org/Submission/SA-REST/, last visited 20 August 2014, 05 April 2010.

[HaMa12] Mohamed H. Haggag and Doaa R. Mahmoud, "OnTraJaCS: Ontology based Traffic Jam Control System", in International Journal of Computer Applications vol. 60 (2), p. 6-16, December 2012.

[HASB13] I. Herman, B. Adida, M. Sporny, M. Birbeck, "RDFa 1.1 Primer - Second Edition: Rich Structured Data Markup for Web Documents", W3C Working Group Note, http://www.w3.org/TR/rdfa-primer/, last visited 24 August 2014, 22 August 2013.

[HaSe13] S. Harris, A. Seaborne, eds., "SPARQL 1.1 Query Language", W3C Recommendation, http://www.w3.org/TR/sparql11-query/, last visited 20 August 2014, 21 March 2013.

[HCD06] F. K. Hussain, E. Chang, T. S. Dillon, “Trust ontology for service-oriented environment”, in Proceedings of the ACS/IEEE International Conference on Computer Systems and Applications (AICCSA), pp. 320-325, Dubai, UAE, 2006.

[HeBi11] Tom Heath, Christian Bizer, “Linked Data. Envolving the Web into a Global Data Space”, Morgan & Claypool, 2011.

[HKHD09] B. Heitmann, S. Kinsella, C. Hayes, S. Decker, “Implementing Semantic Web Applications: Reference Architecture and Challenges”, in Proceedings of the 5th Workshop on Semantic Web Enabled Software Engineering, at the International Semantic Web Conference (ISWC09), http://ceur-ws.org/Vol-524/swese2009_2.pdf, 2009.

[HKHS14] R. Hodgson, P. J. Keller, J. Hodges, J. Spivak, “QUDT - Quantities, Units, Dimensions and Data Types Ontologies”, http://qudt.org/, last visited 22 August 2014, March 18, 2014.

[HKO+10] M. Houda, M. Khemaja, K. Oliveira, M. Abed, "A public transportation ontology to support user travel planning" in Research Challenges in Information Science (RCIS), 2010 Fourth International Conference on, IEEE, May 2010, pp. 127-136, DOI: 10.1109/RCIS.2010.5507372

[HoPa06] Jerry R. Hobbs, Feng Pan, “Time Ontology in OWL”, W3C Working Draft, http://www.w3.org/TR/owl-time, last visited 21 August 2014, 27 September 2006.

[HuFo06] J. Huang, M. S. Fox, “An ontology of trust: formal semantics and transitivity”, in Proceedings of the 8th international conference on Electronic commerce (ICEC '06): The new e-commerce: innovations for conquering current barriers, obstacles and limitations to conducting successful business on the internet, pp. 259-270, USA, 2006.

[isrv] “iServe: Where Linked Data Meets Services“, http://iserve.kmi.open.ac.uk/, last visited 18 August 2014.

[jsls] “JSON for Linking Data”, http://json-ld.org/, last visited 24 August 2014.

[KGV08] J. Kopecký, K. Gomadam, T. Vitvar, “hRESTS: an HTML Microformat for Describing RESTful Web Services”, in Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence (WI-08), Sydney, Australia, December 2008.

[KNSP09] R. Krummenacher, B. Norton, E. Simperl, C. Pedrinaci, “SOA4All: Enabling Web-scale Service Economies”, in Proceedings of the 2012 IEEE Sixth International Conference on Semantic Computing (ICSC09), pp. 535-542, Berkeley, CA, USA, 2009

[KVF09] J. Kopecky, T. Vitvar, D. Fensel, “D3.4.3 MicroWSMO and hRESTS”, Deliverable 3.4.3, EU FP7 Project SOA4All, http://sweet.kmi.open.ac.uk/pub/microWSMO.pdf, last visited 20 August 2014, March 2009.

[LPR05] H. Lausen, A. Polleres, D. Roman, et al., “Web Service Modeling Ontology (WSMO)”, W3C Member Submission, http://www.w3.org/Submission/WSMO/, last visited 20 August 2014, 3 June 2005.

[LWS13] M. Leggieri, C. von der Weth, M. Serrano, “Semantic Representations of Internet-Connected Objects”, EU PF7 Project OpenIoT Deliverable 3.1.2 (D312-130315-v07), March 2013.

[MBH+04] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, Evren Sirin, Naveen Srinivasan, Katia Sycara, “OWL-S: Semantic Markup for Web Services”, W3C Member Submission, http://www.w3.org/Submission/OWL-S/, last visited 18 August 2014, 22 November 2004.

[NoMc01] N. F. Noy, D. L. McGuinness, “Ontology Development 101: A Guide to Creating Your First Ontology”, Stanford Knowledge Systems Laboratory Technical Report KSL-01-05 and Stanford Medical Informatics Technical Report SMI-2001-0880, http://protege.stanford.edu/publications/ontology_development/ontology101.pdf, last visited 20 August 2014, March 2001.

[OM] “Ontology of units of Measure (OM)”, http://www.wurvoc.org/vocabularies/om-1.6/, last visited 25 August 2014.

[OTN] “Ontology of Transportation Networks”, Deliverable A1-D4, Project REWERSE: Reasoning on the Web with Rules and Semantics, originally available at http://rewerse.net/deliverables/m18/a1-d4.pdf, Unavailable as of 27 August 2014, cached version available at https://webcache.googleusercontent.com/search?q=cache:gi0h1sBXLHYJ:rewerse.net/deliverables/m18/a1-d4.pdf #36, last visited 27 August 2014.

[PML+10] C. Pedrinaci, D. Liu, M. Maleshkova, D. Lambert, J. Kopecký, J. Domingue, “iServe: a linked services publishing platform”, in: Proceedings Workshop on Ontology Repositories and Editors for the Semantic Web, at the 7th Extended Semantic Web Conference, Heraklion, Greece, June 2010.

[PoBe08] L. Polo, D. Berrueta, “MUO Measurement Units Ontology”, Working Draft, http://idi.fundacionctic.org/muo/muo-vocab.html, last visited 25 August 2014, April 2008.

[PPO] “Privacy Preference Ontology (PPO)”, http://vocab.deri.ie/ppo, last visited 21 August 2014.

[S4AC] “S4AC Vocabulary Specification 0.2 Namespace Document 6 October 2011”, http://ns.inria.fr/s4ac/v2/s4ac_v2.html, last visited 21 August 2014.

[Sac13] O. Sacco, “Trust Assertion Ontology”, DERI, NUI Galway, 2013. http://vocab.deri.ie/tao, last visited 20 August 2014.

[SJB+09] C. Stasch, K. Janowicz, A. Bröring, I. Reis, and W. Kuhn, “A Stimulus-Centric Algebraic Approach to Sensors and Observations,” in Proc. 3rd International Conference on GeoSensor Networks (GSN’09), pp. 169–179, 2009.

[SKD+09] G. Stevenson, S. Knox, S. Dobson, and P. Nixon, “Ontonym: a collection of upper ontologies for developing pervasive systems,” in Proceedings 1st Workshop on Context, Information and Ontologies (CIAO’09), pp. 1–8, 2009.

[SLK+14] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, N. Lindstroem, “JSON-LD 1.0 - A JSON-based Serialization for Linked Data”, W3C Recommendation, 2014, http://www.w3.org/TR/json-ld/, last visited 21 August 2014.

[SNH+10] W. Sherchan, S. Nepal, J. Hunklinger, A. Bouguettaya, “A Trust Ontology for Semantic Services”, in Proceedings of the IEEE 7th International Conference on Services Computing (SCC 2010), pp. 313-320, USA, 2010.

[SoKa13] J. Soldatos, J. Kaldis, “D2.1 – Report on Stakeholder’s and Virtualization Requirements”, VITAL Project Deliverable D2.2, December 2013.

[SoKa14] J. Soldatos, J. Kaldis, “D2.2 – Reference and Validation Scenarios for IoT Virtualization”, VITAL Project Deliverable D2.2, April 2014.

[SRA06] M. Strimpakou, I. Roussaki, and M. E. Anagnostou, “A context ontology for pervasive service provision,” in Advanced Information Networking and Applications (AINA 2006), 20th International Conference on, 2006.

[ssn] The W3C Semantic Sensor Network Incubator Group, “Semantic Sensor Network Ontology”, http://www.w3.org/2005/Incubator/ssn/ssnx/ssn, last visited 22 August 2014.

[TAC] “TripleAccessControl Ontology 0.1 Namespace Document 15 September 2011”, http://ns.bergnet.org/tac/0.1/triple-access-control.html, last visited 21 August 2014.

[UO] “Units of Measurement Ontology”, http://bioportal.bioontology.org/ontologies/UO, last visited 25 August 2014.

[VHM+14] R. Verborgh, A. Harth, M. Maleshkova, S. Stadtmüller, T. Steiner, M. Taheriyan, R. Van de Walle, “Survey of Semantic Description of REST APIs”, book chapter in: REST: Advanced Research Topics and Practical Applications (C. Pautasso, E. Wilde, R. Alarcon), pp. 69-89, Springer New York, 2014.

[VKVF08] T. Vitvar, J. Kopecky, J. Viskova, D. Fensel, “WSMO-Lite Annotations for Web Services”, in Proceedings of the 5th European Semantic Web Conference, Tenerife, Spain, 2008.

[w3c] The W3C Semantic Sensor Network Incubator Group, “W3C SSN Incubator Group Review of Sensor and Observation Ontologies”, http://www.w3.org/2005/Incubator/ssn/wiki/Incubator_Report#Review_of_Sensor_and_Observation_ontologies, last visited 20 February 2014.

[WAC] “WebAccessControl”, http://www.w3.org/wiki/WebAccessControl, last visited 21 August 2014.

[WeBa09] W. Wei and P. Barnaghi, “Semantic annotation and reasoning for sensor data,” in Smart sensing and context (EuroSSC’09), 4th European conference on, 2009, pp. 66–76.