29 AGOSTO 2018

The NESTORE specific ontology

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

www.nestore-coach.eu

DELIVERABLE ID:	**WP2/D2.3/TASK NUMBER 2.5**
DELIVERABLE TITLE:	The NESTORE specific ontology
RESPONSIBLE PARTNER:	ROPARDO
CONTRIBUTORS:	Ciprian Cândea, Marius Staicu, Gabriela Cândea, Claudiu Zgripcea (ROPARDO), Silvia Orte (EURECAT), Filippo Palumbo, Paolo Baronti, Alberto Gotta (CNR)
NATURE	Report
DISSEMINATION LEVEL:	PU
FILE:	Deliverable D2.3_v2.docx
REVISION:	V 2.0
DUE DATE OF DELIVERABLE:	2018.07.31
ACTUAL SUBMISSION DATE:	2018.08.31
CALL	European Union’s Horizon 2020 Grant agreement: No 769643
TOPIC	SC1-PM-15-2017 - Personalised Medicine

Document History

REVISION	**DATE**	**MODIFICATION**	**AUTHOR**
0.1	2018.07.09	Document Creation, initial TOC	Ropardo
0.2	2018.07.26	Updated content	CNR
0.3	2018.08.02	Updated content	Eurecat
1.0	2018.08.04	Updated content	Ropardo
2.0	2018.08.14	Updated content	Ropardo

Approvals

DATE	**NAME**	**ORGANIZATION**	**ROLE**
2018.07.20	Claudiu Zgripcea	ROPARDO	Reviewer
2018.07.026	Filippo Palumbo	CNR	Reviewer
2018.08.02	Silvia Orte	EURECAT	Reviewer
2018.08.28	Giovanna Rizzo	CNR	WP Leader

2018.09.01	Giuseppe Andreoni	POLIMI	Scientific Coordinator

Short Abstract

The deliverable D2.3 *The NESTORE specific ontology* is the results of the work from task 2.5 *Conversion of the knowledge into ontology*, and provides a formal description of the integrated knowledge provided by Task 2.4 in order to guarantee interoperability and integration of NESTORE data model to the UniversAAL platform.

The ontology has been developed utilizing the uAAL definitions and using the uAAL eclipse ontology tool editor. The ontology covers the knowledge domain concerning the defintions from D2.1 - Models for Healthy Older People:

- Physiological status and Physical Activity Behaviour
- Nutrition
- Cognitive and Mental Status and Social Behaviour

Within the ontology, different types of hierarchies, taxonomies and relationships with respect to the aforementioned areas are defined. The formal languages used to represent knowledge are: Unified Modeling Language and then it will be translated in Java language (in order to be complied with universAAL platform). The modularization and ontology structure are specified by graphical representation.

The selected tool editor, uAAL eclipse plugin covers the defined requirements, described in D6.1 Nestore Platform Requirements[], with respect to the architecture, the knowledge model and the browsing capabilities. The developed ontology will be the basic structure for the development of the uAAL Bridge that it will be implemented within the context of Task 6.6. The NESTORE uAAL ontology , will utilize a semantics repository technology that will permit the extended exploitation of the ontology offered capabilities.

NESTORE is increasing the uAAL ontologies collection with 40% and expand the uAAL domain usage for Physiological Status and Physical Activity Behaviour (8 ontologies), Nutrition (3 ontologies) and Cognitive and Mental Status and Social Behaviour (4 ontologies).

The ontology is available as sources code and documentation on the

https://git.nestore-coach.eu/uaal/ontology

Key Words

NESTORE Ontology, universAAL, Turtle, UML, Java, Nutrition Ontology, Physiological status and Physical Activity Behaviour Ontology, Cognitive and Mental Status and Social Behaviour Ontology

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

Table of Contents

Table of Figures

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

1. Ontology Technologies

1.1 Ontology Languages

Based on the language structure ontology languages can be grouped into three categories, frame based, description logics based and first order logic based.

1.1.1 Frame based languages

Frames are designed to help an Artificial Intelligence system in recognizing specific instances of patterns [6]. Frames usually contain properties called attributes or slots and these may contain default values (subject to override by detecting a different value for an attribute), refer to other frames (component relationships) or contain methods for recognizing pattern instances.

Similar to other knowledge representation systems and languages, frames strive to resemble the way people store knowledge. It seems like we are storing our knowledge in rather large parts, and that different parts are highly interconnected. Regarding the frame-based knowledge representations, the knowledge describing a particular concept is organized as a frame. The frame usually contains a name and a set of slots/attributes.

The slots describe the frame with attribute-value pairs <slotname value> or alternatively a triple containing frame name, slot name and value in some order. In various frame systems the slots are heterogeneous structures that have facets describing the properties of the slot. Each slot may contain a primitive value (a text string or an integer), or it may be another frame. Many of existing systems grant multiple values for slots and some systems support procedural attachments. These attachments can be used to compute the value of the slots, or they can be triggers used to produce consistency checking or updates of other slots. The triggers can be operated by updates on slots.

In general Frame based languages are particularly useful when the application has the following requirements:

- Creating ontologies for domains in which the closed-world assumption is appropriate.
- The application focuses on data acquisition.
- The application domain requires constraints on slot values.
- When classes should be used as property values.

F logic [1] and Open Knowledge Base Connectivity (OKBC) [3] are frame based languages.

1.1.2 First order logic

First-order logic is a formal logical system used in mathematics, philosophy, linguistics, and computer science. It is known also like: first-order predicate calculus, the lower predicate calculus, quantification theory, and predicate logic [4]. It is the most important knowledge representation formalism.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

First-order logic captures some of the essence of human reasoning by providing a notion of logical consequence as already mentioned. It also provides a notion of universal truth in the sense that a logical statement can be universally valid (and thus called a tautology), meaning that it is a statement which is true regardless of any preconditions.

A logical consequence of a theory is a statement which is true in all models of the theory. How to derive logical consequences from a theory – a process called deduction or inferencing – is obviously central to the study of logic. Deduction allows getting the knowledge which is not explicitly given but implicitly represented by a theory.

CycL and Knowledge Interchange Format (KIF) are examples of languages that support expressions in first-order logic, and, in particular, allow general predicates.

1.1.3 Description logics

Description logics (DL) are a family of formal knowledge representation languages. They are more expressive than propositional logic but have more efficient decision problems than first-order predicate logic [2].

The description logic theory consists of statements about concepts, individuals, and their relations. Individuals correspond to constants in first-order logic, and concepts correspond to unary predicates.

This knowledge representation language is used in Artificial Intelligence for formal reasoning on the concepts of an application domain (known as terminological knowledge). It provides a logical formalism for Ontologies and the Semantic Web.

The following requirements of applications may make a Description Logics language a preferred choice:

- Creating robust terminologies in which classes are defined.
- Need for DL reasoning to ensure logical consistency of ontologies.
- Controlled terminologies are published on the Semantic Web and accessed by other applications.
- Applications in which classification is a paradigm for reasoning.

Examples of description logics based languages include DAM+OIL+ KL-ONE, RACER and OWL-DL.

1.2 The OWL language

The Web Ontology Language (OWL) for example is a family of knowledge representation languages based on the RDF (Resource Description Framework) for authoring ontologies endorsed by the World Wide Web Consortium. A variety of commercial and open source tools (like Protégé) are available and allow an easier creation and work with ontologies.

The OWL family contains three versions: OWL-Lite, OWL-DL and OWL-Full. Regarding to their names they contain different number of constructions to build up an ontology [9]. The mightiness of each OWL-Level covers the simpler version of its predecessor (OWL-Lite < OWL-DL < OWL-Full).

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

A first-level schema definition language for explicit specification of data models called RDF Schema (RDFS). RDFS extends the language so that not only data but also data models can be represented in terms of RDF model. The most important language keywords added by RDFS are: *rdfs:Class* (the class of all classes), *rdfs:Datatype* (the type of all datatypes), *rdfs:Resource* (the root class for all resources), *rdfs:Literal* (the root datatype), *rdfs:domain* and *rdfs:range* that can be used for specifying certain relationships between a property (an instance of *rdf:Property*) and classes or datatypes, namely the domain and range of properties, *rdfs:subClassOf* that can be used for specifying hierarchical relationships between classes, and *rdfs:subPropertyOf* (similarly for specifying hierarchical relationships between properties).

The Web Ontology Language (OWL) is built on top of RDFS and introduces a set of new language constructs that provide us with the power of Description Logic (DL) for definition of ontologies. OWL divides rdfs:Resource into four disjoint subclasses, namely

- owl:Class (the class of all abstract classes),
- owl:Thing (the class of all individuals),
- owl:ObjectProperty (the class of properties that only accept individuals as value) and
- owl:DatatypeProperty (the class of properties that only accept literal values).

It additionally defines (among others):

- owl:FunctionalProperty as the class of all properties – no matter if an object property or a datatype property – that are not multi-valued and accept at most one value
- owl:SymmetricProperty, owl:TransitiveProperty, and owl:InverseFunctionalProperty as subclasses of owl:ObjectProperty and equivalent to their counterparts in the theory of functions / relations in mathematics
- owl:equivalentProperty and owl:inverseOf for defining further relationships (in addition to rdfs:subPropertyOf) between properties
- owl:equivalentClass, owl:disjointWith, and owl:complementOf for defining further relationships (in addition to rdfs:subClassOf) between classes
- owl:intersectionOf and owl:unionOf for constructing classes as intersection or union of other classes
- owl:oneOf for constructing classes by enumerating all of their instances
- owl:sameAs and owl:differentFrom for defining relationships between individuals

OWL defines another special construct for defining classes that is used in universAAL very intensively, namely owl:Restriction. Each restriction defines the class of all individuals that satisfy a condition on a certain property, e.g., the class of all individuals that are colored green or the class of all individuals that have at most three moons. To define a restriction, we have to specify a certain property as the value of owl:onProperty as well as exactly one of the following language keywords: owl:minCardinality (the minimum number of values), owl:maxCardinality (the maximum number of values – see the above example with "three" moons), owl:cardinality (the exact number of values), owl:hasValue (the exact value – see the above example with "green"), owl:allValuesFrom (exactly one specific type for all acceptable values), and owl:someValuesFrom (the mandatory type for some acceptable values).

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

1.2.1 OWL Lite

At the beginning, OWL Lite was aimed to support those users primarily needing a classification hierarchy and simple constraints. Although OWL Lite upholds cardinality constraints, it only allows cardinality values of 0 or 1.

It's started from idea that it easier to provide support for OWL Lite than to more articulated relatives, allowing quick migration path for systems utilizing thesauri and other taxonomies. Development of OWL Lite tools has therefore proven not quite easy as development of tools for OWL DL, and OWL Lite is not widely used.

1.2.2 OWL DL

OWL DL was designed to provide as maximum expressiveness as possible to retain computational completeness (either ϕ or ¬ϕ belong), decidability (there is an effective procedure to determine whether ϕ is derivable or not), and the availability of practical reasoning algorithms. OWL DL includes all OWL language construction, but they can be used only under certain restrictions (for example, number restrictions may not be placed upon properties which are declared to be transitive). OWL DL is so named due to its correspondence with description logic, a field of research that has studied the logics that form the formal foundation of OWL.

1.2.3 OWL Full

OWL Full is based on a different semantics from OWL Lite or OWL DL; it contain hierarchical restriction, and was aimed to preserve some suitability with RDF Schema.

In OWL Full a class can be handled concurrently as a collection of individuals and as an individual in its own right, but this is not allowed in OWL DL. OWL Full permit an ontology to boost the meaning of the pre-defined (RDF or OWL) vocabulary. It is improbable that any reasoning software will be able to support complete reasoning for OWL Full.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

2. Ontologies in universAAL

Ontologies can be represented in a standard format such as serialized RDF, but in universAAL Ontologies have a specific code representation so they can be handled by the Middleware.

The universAAL platform [10] enables interoperability in open distributed systems without limiting the scope of the system [8]. The possibility is assured by shifting all specifics to plug-able units modelled through ontologies. The universAAL platform is based on the ontological framework of the Semantic Web, especially the following four specifications [11]:

- A meta model for data representation called the Resource Description Framework (RDF). RDF provides a standard model for representing data that is independent from the data itself (all sorts of data can be serialized in terms of RDF. The universAAL platform uses the Turtle syntax. The RDF model basically says that data can be represented either as a literal or as a resource. Resources can be identified by URIs / IRIs and can be described based on their properties. The types of resources and the names of their properties are also resources, and as such, must be identified by URIs / IRIs. The names of resource types and resource properties follow the rule formulated in the last bullet of the previous enumeration. To describe a resource, statements must be built. A statement comprises a subject (the resource to be described), a predicate (a distinct property of that resource), and an object (the value of that property, which might be a literal or a resource).

- A first-level schema definition language for explicit specification of data models called RDF Schema (RDFS).

- The Web Ontology Language (OWL) which builds on top of RDFS and introduces a set of new language constructs that provide us with the power of Description Logic (DL) for definition of ontologies.

- The SPARQL Protocol And RDF Query Language (SPARQL) that is a SQL-like query language for querying data represented in terms of RDF. The usage of SPARQL in universAAL is, however, currently very limited (planned to be used more in future versions) versions[1].

In the Middleware representation Ontologies [12] are composed by an Ontology class that defines the whole Ontology, a Factory for serialization, and a collection of classes representing the concepts.

Resources[2] are how the concepts are represented. They are the nodes in the mesh. They are identified by a URI. They can inherit from other resources, and have properties that link to other resources or datatypes. In the Middleware representation a Resource is like a class. In addition to being defined in the Ontology class, there is a class representing it that can be instantiated to be used from the code, and contains constants that identify its URI and properties URI, and helper methods to handle it. All concept Resources that can be instantiated and handled from code extend from universAAL´s ManagedIndividual (or another super-class which will eventually extend ManagedIndividual), which extends from Resource. If the concept is not supposed to be instantiated it can be defined as abstract, despite this is not a feature of Ontologies.

Properties[2] are links between the concepts. They are also identified by URIs and can also inherit from other properties. They can have restrictions upon them, like cardinality. For each property there usually are helper methods in its Resource class to get, set and possibly add a value, although it is always possible to use the generic method setProperty. It always checks that the value set satisfies the restrictions. When cardinality is greater than 1, multiple values can be set as a List.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

Datatypes[2] are the native data formats, like Boolean, Integer and so on. They are always present by default and don´t have properties. These types are available through TypeMapper to obtain their URI.

Enumerations[2] are sets of instances of Resources, representing different specific values that a property can point to. They cannot be instantiated and therefore are created as abstract Ontological Resources, with convenient methods to initialize them in the Ontology class. They have a class representing them too, but it´s different from those representing normal Resources.

2.1 uAAL Ontology support

Plug-gable ontology artifacts to support context-awareness and personalization, ontologies are used as the data model of unviersAAL, shaping the information shared through the middleware buses.

There are some ontologies developed[3] in universAAL:

- ont.activityhub: ISO 11073-10471 Activity Hub (https://github.com/universAAL/ontology/wiki/ActivityHub)
- ont.av: Audio & Video (https://github.com/universAAL/ontology/wiki/Audio-and-Video)
- ont.cryptographic: Cryptographic methods, algorithms and high level services. (https://github.com/universAAL/ontology/wiki/Cryptography)
- ont.dependability: Dependability (https://github.com/universAAL/ontology/wiki/Dependability)
- ont.device: Unified devices ontology: Typical home automation, AmI and AAL devices (https://github.com/universAAL/ontology/wiki/Devices)
- ont.health.disease: Diseases and illnesses (https://github.com/universAAL/ontology/wiki)
- ont.health.measurement: Health-specific measurements (https://github.com/universAAL/ontology/wiki/Healht-Measurement)
- ont.impaiment: Model of user impairments (https://github.com/universAAL/ontology/wiki/Impairment)
- ont.lighting: Lighting (https://github.com/universAAL/ontology/wiki)
- ont.measurement: Measurement concepts (https://github.com/universAAL/ontology/wiki/Measurement)
- ont.medication: Medication (https://github.com/universAAL/ontology/wiki/Medication)
- ont.multimedia: Multimedia appliances (https://github.com/universAAL/ontology/wiki/Multimedia)
- ont.personalhealthdevices: Some health-related sensors for personal use. (https://github.com/universAAL/ontology/wiki/PersonalHealthDevices)
- ont.phWorld: Basic concepts of the physical world (https://github.com/universAAL/ontology/wiki/Physical-World)
- ont.profile: All Profiles: Users, AAL Services, uSpaces... (https://github.com/universAAL/ontology/wiki/Profile)
- ont.profile.ui.mainmenu: UI Main menu subprofile (https://github.com/universAAL/ontology/wiki/MainMenu-subprofile)
- ont.profile.ui.preferences: UI Preferences subprofile (https://github.com/universAAL/ontology/wiki/UI-Preferences-subprofile)

[3] https://github.com/universAAL/ontology/wiki (online, 16 July 2018)

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

- ont.profile.health: User's Health subprofile, and concepts like illness (https://github.com/universAAL/ontology/wiki/ProfileHealth)
- ont.recommendations: Recommendations ontology for providing more input to UI Handlers in the rendering process (https://github.com/universAAL/ontology/wiki/Recommendations)
- ont.security: Authentication and Authorization security (https://github.com/universAAL/ontology/wiki/Security)
- ont.sysinfo: System information events (https://github.com/universAAL/ontology/wiki/Sysinfo)
- ont.unit: Units of measurement and prefixes (https://github.com/universAAL/ontology/wiki/Unit)

2.2 uAAL URI of class and property names

2.2.1 URIs of class and property names

URIs / IRIs are used in the Semantic Web as the unique identifier. Each of the syntaxes used for the serialization of RDF provides means for defining the namespace prefixes. In Turtle (used within universAAL), this is done the following way: complete URIs are written in Turtle in a pair of <>:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
```

All names that have no prefix belong to a default namespace. In Turtle, names from the default namespace start with a ":" provided that such an empty prefix is previously defined in the corresponding Turtle segment.

Subtypes of rdfs:Literal

For some types of literal values, (for example integers, floats, doubles, and booleans), the Turtle parser is expected to recognize the values when they are not in quotation marks; that is, true is equivalent to `"true"^^xsd:boolean` in Turtle.

2.2.2 URIs of individuals

All resources that are neither classes nor properties are classified as individuals. Individuals might be known or anonymous. Known individuals must be referred to using their URIs / IRIs. The representation of anonymous individuals might differ in different syntaxes. In Turtle, "[]" is used to indicate an anonymous individual.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

2.3 uAAL Ontology editor

The universAAL Studio tool is intended to be used by developers of services and platform components that need to create or extend an own ontology. The goal of the tool is to simplify the development of ontologies, by generating most of the required Java code based on a more high-level description.

The benefits of this approach are:

- simpler and quicker development and maintenance of ontologies,
- avoiding manually introduced mistakes in the code, and
- more uniform implementation code.

The use of this tool is not mandatory, but highly recommended[4]. The alternative to using the tool (in combination with the Model Transformation Tool) is to manually code the ontologies as Java code.

The universAAL Studio tools are implemented as Eclipse plug-ins and provide integration with other universAAL Studio tools (3rd party and those developed in universAAL) – installation guide can be accessed at - https://github.com/universAAL/tools.eclipse-plugins/wiki/universAAL-Studio-overview-and-installation

2.3.1 Functionalities overview

The universAAL Studio support for ontology modelling allows the creation or extending ontologies by building a model of the ontology and then generating the Java code that represent the model [5].

The tools are built around the open source Eclipse projects Papyrus (for modelling) and MOFScript (for code generation), with extensions that include UML profiles for ontology modelling and transformations that generate the Java code.

The UML profiles for OWL and RDF included with the tool are based the Ontology Definition Metamodel (OMD) from OMG, and have been adapted from the implementation of http://code.google.com/p/twouse/ .

The process of creating a new ontology to be used in universAAL follows three basic steps:

1. Create an ontology project using a wizard (see Ontology Project Wizard for details about the wizard)
2. Design a model of the ontology in UML
3. Generate the Java representation of the ontology and Maven POM files from the model

[4] https://github.com/universAAL/tools.eclipse-plugins/wiki/Ontology-modelling (online, 16 July 2018)

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

2.3.2 Ontology concepts hidden for the designer

The UML metamodel does not provide enough expressiveness to unambiguously model an ontology (subset) using the standard UML language (classes, associations etc). Therefore it is necessary to a UML profile for OWL that extends the UML language with OWL concepts, allowing the designer to mark model elements. The OMT hides much of the complexity and required work through the use of template projects and modelling palettes in Eclipse.

2.3.2.1 Core modelling concepts

The modelling approach to ontology design is closely related to the way an ontology is defined in Java. This subsection describes the overall concept mappings

- UML Model: is the topmost modelling level. Using the Ontology Project Wizard will create the model correctly.
- UML Package: represents an Ontology. A package is marked as a "owlOntology" using stereotypes from the OWL UML Library. Using the Ontology Project Wizard the package is created and labeled using the names filled in the wizard
- UML Class: represents a ontology concept. A Class can have properties (attributes) that are typed (datatype or object), and associations that are directed. A class will map to a Java class (and file), and will be included in the Ontology and Activator files.
- UML Properties: this is an attribute of the class or interface and represents a property of the ontology concept. There are two types of properties that can be associated with a class: ObjectProperty and DatatypeProperty. Using the elements found in the Ontology Modelling Tool palette, you the types and required stereotypes settings are automatically set to their default values (e.g. isFunctional="true")
- UML Association: an association is similar to a property and will result in a property in the owner class (assuming only unidirectional associations). *Note*: the name of the property will be the label on the association end (default by tool is the end class name), and this is also where you can set the multiplicity of the relation (0..1, 1, 0..*, 1..*). In Papyrus this member end is shown to the right in the Properties view.
- UML Generalization: a generalization is a way to describe inheritance using UML, and will result in a class extend relationship in java.

2.3.2.2 Ontology Modelling Process

To create a new ontology, all the required effort has to be done on the UML model. However, the label and naming conventions must be followed:

- Naming: org.universaal.ontology + <ontologyname></ontologyname>
- UML Classes are used for ontology classes. The classes can be set as "Abstract" in the properties view
- Generalizations/specializations are used for subclassing
- Enumerations have enumeration literals. Also found in the palette
- Associations have cardinality and member end names

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

- Properties
    - Name: lowercaseUppercase
    - Stereotype describes type
    - ObjectProperty: another ontology class
    - DatatypeProperty: UML primitive types
    - Tagged value: isFunctional can be set true/false. The palette elements have default value true
    - Cardinality should be set for each property in the properties view
    - The property's default value can used for upper/lower Integers
        - name: "values", value: "lower..upper"

When modelling ontologies, there will be usually reused elements from existing ontologies, either by generalizing from them (using them as super-classes) or by referring to them in object properties and associations.

To get access to classes from another model, that model must first be imported into the model. The template model that is used for creating an ontology model with the Ontology Wizard, comes with the upper ontologies of universAAL pre-imported. The list of imported models can be seen as Package Import entries in the Model Explorer view of Papyrus.

2.4 universAAL studio limitations

During development of the NESTORE ontology using tools and guidelines provided by uAAL we observe some problems encountered with universAAL Studio.

For each property *p* defined for a UML class *D* and having a value of type *R* the OWL ontology generated by the universAAL Studio plugin includes the following (in turtle format):

ns:p rdfs:domain ns:D .

This turtle line states that the domain of property *p* is the class *D* that defines it.

Now suppose you need to define two properties with the same name *p* on two different classes *E* and *F* and suppose you want these properties to represent the same concept (and have the same type). If you define two such properties in the universAAL Studio plugin, the resulting ontology will contain:

ns:p rdfs:domain ns:E .

ns:p rdfs:domain ns:F .

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

The ontology generation algorithm makes no distinction of the two properties and states that property *p* has both domain *E* and domain *F*. This is unfortunate since facts like

IndividualA a ns:E .

IndividualA ns:p ValueX .

would allow an inferencing engine to deduce that

IndividualA a ns:F .

which might not be the case if *E* and *F* are intended to represent disjoint entity sets.

2.4.1 Solution 1

A possible solution would be to define two distinct properties *p1* and *p2* on the two classes *E* and *F*. universAAL Studio would then produce an ontology with the following lines:

ns:p1 rdfs:domain ns:E .

ns:p2 rdfs:domain ns:F .

These properties are now completely separate and don't create any problems but we are not able to treat them as referring to the same concept. We could work around this by defining a new property *p* in the resulting ontology such that both *p1* and *p2* are subproperties of *p*:

ns:p1 rdfs:subPropertyOf ns:p .

ns:p2 rdfs:subPropertyOf ns:p .

This way whenever we assert a fact about *p1* or *p2* we also assert it for *p*. If we state

IndividualA a ns:E .

IndividualA ns:p1 ValueX .

we can infer that

IndividualA ns:p ValueX .

but we cannot infer

IndividualA a ns:F .

Unfortunately we cannot have universAAL Studio add the subproperty rules for us.

2.4.2 Solution 2

Another approach requires moving property *p* to a superclass *D* of *E* and *F* in the UML diagram.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

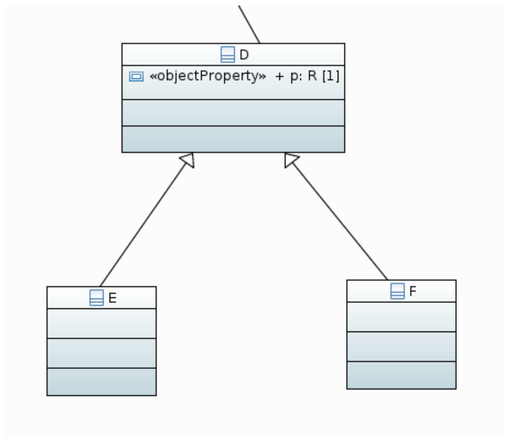

Figure 1 uAAL Studio Solution 2

This way the generated ontology includes:

ns:p :domain ns:D .

ns:E a owl:Class ;

ns:E :subClassOf ns:D .

ns:F a owl:Class ;

ns:F :subClassOf ns:D .

With this ontology in place and the facts

IndividualA a ns:E .

IndividualA ns:p ValueX .

we can only infer

IndividualA a ns:D .

This is fine until we want to define some other property *q* that is shared by *E* with some other class *G* (with *G* not the same as *F*).

In this case we cannot move that property to *D* but we need another class *C* that is a superclass of *E* and *G*.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

2.5 Ontology Transformation to Java

An mentioned before the universAAL platform are using the defined ontologies as Java classes and to simplify the translation between UML definiton to Java the Model Transformation Tool can be used. [13]

The main benefit of using this tool is that it automates (part of) the implementation work. The alternative for the user would usually be to hand-code the implementation based on the model or some equivalent design. The modelling and transformation approach also have the benefit that maintenance is simplified.

One important part is related to the validation of the defined ontology, this activity can be used during the development of the ontology and is a mandatory step during the transformation process to Java classes.

The transformation tool is calling the validation feature which checks the UML ontology model for errors. The tool will reports on errors that will cause problems for the code generation provided by the UML ontology to Java transformation, such as:

- Classes and properties without names
- Classes and properties using the same name
- Circular derivation/generalization
- Classes without a direct or indirect derivation from an upper ontology class

It gives warning for issues that might be problems, such as:

- No properties defined for a class
- Multiple derivation/generalization

3. NESTORE Ontologies

The aim of this document is to provide a formal description of the integrated knowledge provided by task 2.1, 2.2, 2.3, and 2.4, in order to guarantee interoperability and integration of NESTORE data model to the universAAL platform. To reach this goal, we developed the NESTORE ontology using and extending uAAL ontologies already available and validated (https://github.com/universAAL/ontology/wiki).

The knowledge about the three main areas of intervention of NESTORE, described in Deliverable D2.1 (Annexes 1, 2, and 3), has been used as the basis for the formal definition of the ontology. The result of the development consists of a set of packages (one for each subdomain) containing UML and Java code (more details on the produced Java code can be found in the next Section) and a detailed description of each modeled domain. The produced material can be found in the wikipages of the NESTORE git repository: https://git.nestore-coach.eu/uaal/ontology/wikis/home.

3.1 Definitions

The NESTORE ontologies have been organized, following the recommendations of the domain experts, in three main areas, as follows.

Physiological Status and Physical Activity Behaviour (8):

- Anthropometric Characteristics (https://git.nestore-coach.eu/uaal/ontology/wikis/Anthropometric-characteristics)
- Cardiovascular System (https://git.nestore-coach.eu/uaal/ontology/wikis/Cardiovascular-system)
- Respiratory System (https://git.nestore-coach.eu/uaal/ontology/wikis/Respiratory-system)
- Musculoskeletal System (https://git.nestore-coach.eu/uaal/ontology/wikis/Musculoskeletal-system)
- Cardiorespiratory Exercise Capacity (https://git.nestore-coach.eu/uaal/ontology/wikis/Cardiorespiratory-exercise-capacity)
- Strength-Balance-Flexibility Exercise Capacity (https://git.nestore-coach.eu/uaal/ontology/wikis/Strength-balance-flexibility-exercise-capacity)
- Physical Activity Behavior (https://git.nestore-coach.eu/uaal/ontology/wikis/Physical-activity-behaviour)
- Sleep Quality (https://git.nestore-coach.eu/uaal/ontology/wikis/Sleep-quality)

Nutrition:

- Blood Parameters (https://git.nestore-coach.eu/uaal/ontology/wikis/Blood-parameters)
- Energy Expenditure (https://git.nestore-coach.eu/uaal/ontology/wikis/Energy-expenditure)

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

- Nutrition Habits (https://git.nestore-coach.eu/uaal/ontology/wikis/Nutrition-habits)

Cognitive and Mental Status and Social Behaviour:

- Cognitive Status (https://git.nestore-coach.eu/uaal/ontology/wikis/Cognitive-status)
- Mental Status (https://git.nestore-coach.eu/uaal/ontology/wikis/Mental-status)
- Mental Behaviors and States (https://git.nestore-coach.eu/uaal/ontology/wikis/Mental-behaviors-and-states)
- Social Behavior (https://git.nestore-coach.eu/uaal/ontology/wikis/Social-behaviour)

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

For each subdomain, a dedicated wikipage has been produced presenting a brief introduction on the subdomain, a diagram of the UML classes, and a short description of each element composing the ontology.

Many of the domains/subdomains we were attempting to describe entailed two main concepts: variables and variable measurements. The former identify and characterize the quantity that is being measured in the context of each subdomain and how it relates to other variables (for instance variables can influence, depend on or be calculated from other variables). The latter represent actual variable measurements. As such they are related to a certain instant in time (or time interval) and concern some human that is being examined or monitored. Variable measurements are also associated with a measuring instrument and several different instruments/devices can be used to measure the same variable. A final information we modeled is the association between the variable measurement and the actual variable concept (i.e., we wanted to state that a "body height measurement" is a measure of variable "body height").

3.2 Handling limitatations

While attempting to model Nestore domains/subdomain with universAAL Studio we came across some limitations of this tool.

Although the Profile ontology is available on the universAAL github site (https://github.com/universAAL/ontology/wiki/Profile) and this ontology is capable of modeling users, it is not provided with a UML file. This prevents it from being imported into the universAAL Studio and can only be used through Java code. To circumvent this issue we developed a basic and unpretentious "NestoreUser" ontology with a very simple User class. This ontology is imported and used by the subdomains in order to associate variable measurements with a user. It can be easily replaced by a more sophisticated user ontology when such option becomes available. On the other hand we imported and used the Unit ontology (https://github.com/universAAL/ontology/wiki/Unit) to represent measurement units since it came with a UML file.

The only way to represent facts about individuals via the universAAL Studio plugin is to create an Enumeration and populate it with enumeration literals. The OWL ontology generated from the Java classes will represent the Enumeration as a class and state that the enumeration literals are members of that class (via the rdf:type property). It is not possible to generate other types of facts about individuals (i.e., stating that individual A has a certain property with value B) except for class membership. This limitation prevented us from expressing facts concerning dependence relationships among specific variables. For instance we couldn't say that

ns:body_height_variable ns:influences ns:body_mass_index_variable

Nevertheless we modeled a relation named "influences" from AnthropometricVariable to itself with [0..*] cardinality. This way we can express the fact that such dependency may exist but we do not say which anthropometric variables influence which other variables.

Also via universAAL Studio it is not possible to state that classes are disjoint or that the elements of an Enumeration are all distinct or that they are all the elements of a class. This prevented us from expressing facts that can actually be expressed via the OWl language.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

Another problem we came across is the way associations are modeled by universAAL Studio. Suppose you want to model the measurement instrument for class BodyHeightValue in order to represent the device used to measure the height of a person and suppose you have a class BodyHeightMeasurementInstrument that represents the set of devices used to measure body height. Suppose you use universAAL Studio to model an association named measurementInstrument from BodyHeightValue to BodyHeightMeasurementInstrument. The algorithm used by universAAL Studio to generate the OWL ontology will generate the statement

ns:measurementInstrument rdf:domain ns:BodyHeightValue

Now suppose you also have to model body weight and define an association measurementInstrument from BodyWeightValue to BodyWeightMeasurementInstrument. The resulting OWL ontology will also contain the statement

ns:measurementInstrument rdf:domain ns:BodyWeightValue

We are modeling a measurementInstrument property on two different classes. is unfortunate the since ontology generated by universAAL Studio may lead to unexpected inferences. For instance if a user of the ontology were to state that

MyBodyHeightValue ns:measurementInstrument ns:tape_meter

a reasoniong engine will be able to deduce

MyBodyHeightValue rdf:type ns:BodyHeightValue

MyBodyHeightValue rdf:type ns:BodyWeightValue

One way out of this is to name the two associations differently (e.g., bodyHeightMeasurementInstrument and bodyWeightMeasurementInstrument) and lose the information that the two properties represent the same concept. The benefits of this approach must be balanced with the fact that universAAL Studio does not allow to state that a property is a subproperty of another.

Another option is to define a superclass of BodyHeightValue and BodyWeightValue (call it AnthropometricValue), a superclass of BodyHeightMeasurementInstrument and BodyWeightMeasurementInstrument (call it AnthropometricMeasurementInstrument) and define property measurementInstrument from the former to the latter. With this solution you lose model details. In our modeling we used the first approach when we judged it was important to represent more details about the domain.

We used the second approach when the range of the property being modeled was the same so we didn't need to distinguish.

3.3 Nestore ontology sample

All the defined NESTORE ontologies, documentation and source code can be accessed here:

https://git.nestore-coach.eu/uaal/ontology/

As sample in the present document we include one sub-domain ontology definition.

Many of the domains/subdomains that are describe entailed two main concepts: variables and variable measurements.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

The former identify and characterize the quantity that is being measured in the context of each subdomain and how it relates to other variables (for instance variables can influence, depend on or be calculated from other variables).

The latter represent actual variable measurements. As such they are related to a certain instant in time (or time interval) and concern some human that is being examined or monitored. Variable measurements are also associated with a measuring instrument and several different instruments/devices can be used to measure the same variable.

A final information we modeled is the association between the variable measurement and the actual variable concept (i.e., we wanted to state that a "body height measurement" is a measure of variable "body height").

3.3.1 Sleep quality ontology

The ontology describing the main factors related to sleep and can be found it here: https://git.nestore-coach.eu/uaal/ontology/wikis/Sleep-quality

Diagram

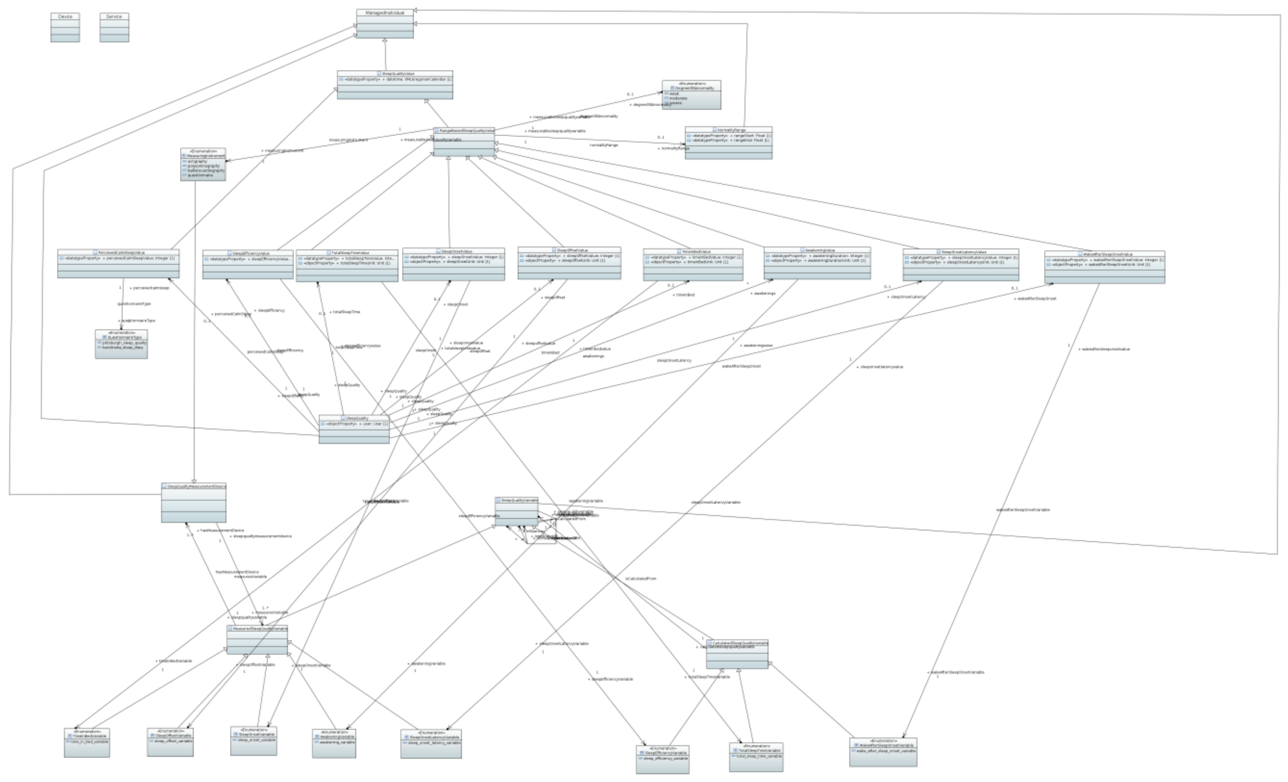

Figure 2 Sleep-quality

Sleep Quality

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

The root concept of this ontology. It represents a set of measurements/data related to a user.

SleepQualityVariable

The superclass of all sleep quelity variables. It defines dependency relations among variables, including *influences, isIndicatorOf, and correlatesWith*.

Subclass **MeasuredSleepQualityVariable** includes variables that can be measured via instruments / devices. Subclass **CalculatedSleepQualityVariable** includes variables that are calculated from other sleep quality variables. Further subclasses down the hierarchy (defined below) represent concrete variables that are linked to the corresponding measured value (see class **SleepQualityValue** below).

SleepOffsetVariable

Sleep Offset is the time at which the subject awakes and does not manage to fall asleep again.

SleepOnsetVariable

Sleep Onset is the time at which the subject falls asleep for the first time.

TimeInBedVariable

Time in Bed is the time spent in bed including the wake periods before the sleep onset, in between sleep onset and sleep offset or after sleep offset.

AwakeningVariable

An awakening period during sleep time.

SleepOnsetLatencyVariable

Sleep Onset Latency represents the time that it takes to accomplish the transition from full wakefulness to sleep, normally to the lightest of the non-REM sleep stages.

TotalSleepTimeVariable

Total Sleep Time is the time between sleep session start and sleep session end minus the time classified as awake.

SleepEfficiencyVariable

Sleep Efficiency is commonly defined as the ratio of total sleep time to time in bed.

WakeAfterSleepOnsetVariable

Wake After Sleep Onset is the total duration of wake time after sleep onset and it is calculated as the amount of time elapsed between sleep start and sleep end scored as wake.

SleepQualityValue

The base class of all measured values. Subclass **RangeBasedSleepQualityValue** represents those measured values that can have a normality range (expressed by class **NormalityRange**) and a degree of abnormality (enumeration **DegreeOfAbnormality**)

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

X-Value

A specific measurement relating to a certain **SleepQualityVariable**. Where applicable it has properties decsribing its value, measurement unit, and measurement device.

SleepQualityMeasurementDevice

The superclass of all measurement device classes. It represents measurement devices/instruments that can be used to measure **MeasuredSleepQualityVariables**.

X-MeasurementDevice

A specific **SleepQualityMeasurementDevice** subclass that includes measurement devices/instruments used to measure a specific **MeasuredSleepQualityVariable**.

4. Conclusion

The universAAL platform enables interoperability in open distributed systems without limiting the scope of the system. This is possible because of shifting all specifics to plug-able units called ontologies. For this reason, NESTORE defined the necessary ontologies for the project scope.

For NESTORE coaching applications, defining the model of the domain as ontology give to the consortium support for explicitness and share-ability.

NESTORE contributes with a number of 15 ontologies to the uAAL repository. uAAL repository is based on a number of 37 ontologies that are mainly focused on the environmental sensors described as ontologies. NESTORE is increasing the uAAL ontologies collection with 40% and expand the uAAL domain usage.

NESTORE expand the uAAL usage to 3 new domains:

- Physiological Status and Physical Activity Behaviour with 8 ontologies
- Nutrition with 3 ontologies
- Cognitive and Mental Status and Social Behaviour with 4 ontologies

AAL is an application domain with a lot of overlapping subdomains [7], like the eHealth domain, the home entertainment domain, the home automation domain, the household appliance domain, the energy control and saving domain and many more.

NESTORE ontologies contribute to the eHealth domain providing a structured knowledge, built on expertise of the NESTORE experts (exercise physiologists, nutritionists, psychologists, geriatricians), able to characterize the person in terms of both status and behaviour.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643

References

[1] Angele J., Lausen G. (2004) Ontologies in F-logic. In: Staab S., Studer R. (eds) Handbook on Ontologies. International Handbooks on Information Systems. Springer, Berlin, Heidelberg, https://doi.org/10.1007/978-3-540-24750-0_2

[2] Baader, F., Horrocks, I., Sattler, U., 2008. Description logics. In: van Harmelen, F., Lifschitz, V., Porter, B., (Eds.), Handbook of Knowledge Representation. Elsevier Academic Press, Amsterdam, pp. 135 –179.

[3] Chaudhri, K., Farquhar, A., Fikes, R., Karp, P.D., Rice J. P., (2000). OKBC: A Programmatic Foundation for Knowledge Base Interoperability. Proceedings of the National Conference on Artificial Intelligence. (https://www.researchgate.net/publication/2333835_OKBC_A_Programmatic_Foundation_for_Knowledge_Base_Interoperability)

[4] Enderton, Herbert B., "Second-order and Higher-order Logic", The Stanford Encyclopedia of Philosophy (Fall 2015 Edition), Edward N. Zalta (ed.), ttps://plato.stanford.edu/archives/fall2015/entries/logic-higher-order/.

[5] Hanke S. et al. (2011) universAAL – An Open and Consolidated AAL Platform. In: Wichert R., Eberhardt B. (eds) Ambient Assisted Living. Springer, Berlin, Heidelberg

[6] Malone, T.W., Grant, K.R., Turbak, F.A., Brobst, S.A., Cohen, M. D. , Intelligent information-sharing systems, Communications of the ACM, 1987 (https://dspace.mit.edu/bitstream/handle/1721.1/2157/SWP-1850-21289506-CISR-147.pdf)

[7] Memon, M., Rahr Wagner, S., Pedersen Fischer, C., Aysha Beevi, F.H., Overgaard Hansen, F., Ambient Assisted Living Healthcare Frameworks, Platforms, Standards, and Quality Attributes, Sensors 14 (2014) 4312 – 4341. doi:10.3390/s140304312

[8] Mosmondor, M. universAAL: Technical insights. In AAL Interoperability Days (MACSI 2014), European commission, Brussels, Belgium, February 2014.

[9] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. OWL web ontology language semantics and abstract syntax. W3C Recommendation, 2004.

[10] Ram, R., Furfari, F., Girolami, M., Ibañez-Sánchez, G., Lázaro-Ramos, J.-P., Mayer, C., Zentek, T. (2013). universAAL: Provisioning Platform for AAL Services. In A. van Berlo, K. Hallenborg, J. M. C. Rodríguez, D. I. Tapia, & P. Novais (Eds.), Ambient Intelligence - Software and Applications (pp. 105–112). Heidelberg: Springer International Publishing.

[11] https://github.com/universAAL/tools.eclipse-plugins/wiki/Ontology-modelling

[12] https://github.com/universAAL/middleware/wiki/Data-Representation

[13] https://github.com/universAAL/tools.eclipse-plugins/wiki/Transformation-OWL-UML-Java

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 769643