Publications Office
of the European Union

Initiative for developing ePROCUREMENT Ontology

ePROCUREMENT UML conceptual model conventions

Disclaimer

This report was prepared for the Publications Office of the European Union by Infeurope.

Document metadata

Reference	WP 1.2: eProcurement UML conceptual model conventions
Corporate Author	Publications Office of the European Union
Author	Eugeniu Costetchi
Reviewers	Natalie Muric, Ioannis Rousochatzakis, George Vernardos
Contractor	Infeurope S.A.
Framework contract	10688/35368
Work package	WP 1.2
Delivery date	17 April 2020

Abstract

In the eProcurement ontology initiative, the conceptual model is represented in Unified Modelling Language (UML). It is a visual representation language that facilitates understanding and convergence between stakeholders towards a common conceptualisation of the model.

UML does not define a formal semantics that would permit to determine, from the class diagrams, whether an ontology is consistent; or to determine the correctness of the ontology implementation. Semantics in such cases becomes a subject to interpretation by the stakeholders involved in the development process and later by the users in the application and integration tasks.

On the other hand, UML is closer than more logic-oriented approaches to the programming languages in which enterprise applications are implemented. The UML Conceptual Model of the eProcurement domain serves as the single source of truth, which means that the formal eProcurement ontology is derived from it through a model transformation process.

For this reason, the current specification establishes conventions for interpretation of the UML-based conceptual model. It provides the UML modelling constraints and a set of conventional and technical recommendations for naming and structuring the UML class diagrams.

Contents

1 Introduction

This document provides a working specification of the guidelines and conventions for the eProcurement conceptual model, such that it qualifies as suitable input for the transformation scripts meant to generate the formal eProcurement ontology.

The business context and the project overview are detailed in Costetchi [7], while the next section provides the motivation and situates the current document within the project.

1.1 Context

In the eProcurement ontology project, the conceptual model was decided [10] to be represented in *Unified Modelling Language (UML)* [3]. It is a visual representation language that facilitates understanding and convergence between stakeholders towards a common conceptualisation of the model.

Generally, the primary application of UML [11] for ontology design is in the specification of class diagrams initially conceived for object-oriented software. UML does not define a formal semantics that would permit to determine, from the class diagrams, whether an ontology is consistent, or to determine the correctness of the ontology implementation. Semantics in such cases becomes a subject to interpretation by the stakeholders involved in the development process and later by the users in the application and integration tasks [12].

On the other hand, UML is closer than more logic-oriented approaches to the programming languages in which enterprise applications are implemented. For this reason the current specification establishes conventions for interpretation of the UML-based conceptual model.

The UML Conceptual Model of the eProcurement domain serves as the single source of truth, which means that the formal eProcurement ontology is derived from it through a model transformation process. It is possible to generate automatically the formal ontology in RDF format [30] from the XMI (v.2.5.1) serialisation [1] of an UML (v.2.5) model [6], provided that a set of clear transformation rules are established [8], and that a set of modelling conventions is respected.

This document provides the UML modelling constraints and a set of conventional and technical recommendations for naming and structuring the UML class diagram

elements: packages, classes, data types, enumerations, enumeration items, class attributes, association relation and dependency relation. There are additional elements which will be addressed contextually in the following sections.

1.2 Requirements

The eProcurement conceptual model must fulfil mainly four fundamental objectives.

- Facilitate understanding/comprehension of the represented system
- Promote efficient conveyance of system details between team members and external stakeholders.
- Provide a point of reference for system designers to gather system specifications and documentation.
- Serve as input for development of a (more) formal model.

In order to support objectives a conceptual model should fulfil the following requirements.

- Be available to all team members, to facilitate collaboration and iteration.
- Be easily changeable, as a continuous reflection of up-to-date information.
- Contain both visual and written forms of diagramming, to better explain the abstract concepts it may represent.
- Establish relevant terms and concepts that will be used throughout the project.
- Define said terms and concepts.
- Provide a basic structure for entities of the project.
- Reduce the ambiguity while maintaining a simple and concise encoding.

1.3 Key words for Requirement Statements

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119 [4].

The key words “MUST (BUT WE KNOW YOU WON’T)”, “SHOULD CONSIDER”, “REALLY SHOULD NOT”, “OUGHT TO”, “WOULD PROBABLY”,

"MAY WISH TO", "COULD", "POSSIBLE", and "MIGHT" in this document are to be interpreted as described in RFC 6919 [26].

2 Preliminary definitions

In this document three closely related and mostly overlapping domains are used interchangeably. They come from the worlds of: *enterprise architecture*, *software engineering and design* and *formal knowledge representation*. It is therefore necessary to establish the terms, especially the common ones, by defining what they mean in each of these domains.

2.1 Conceptual Model

A *conceptual model* is a representation of a system that uses concepts to form said representation. A *concept* can be viewed as an idea or notion; a unit of thought [17]. However, what constitutes a unit of thought is subjective, and this definition is meant to be suggestive, rather than restrictive. That is why each concept needs to be well named by providing preferred and alternative labels, and a clear and precise definition supported by examples and explanatory notes.

The conceptual model comprises representations of concepts, their qualities or attributes and relationships to other concepts. Most commonly these are association and generalisation relations. In addition, behaviour can be represented, ranging from the concept level up to the level of the system as a whole. Behavioural aspects, however, fall out of the scope in the current specification, which addresses at the structural elements mainly.

2.2 Unified Modelling Language (UML)

The *Unified Modelling Language (UML)* is a general-purpose, developmental, modelling language in the field of software engineering that is intended to provide a standard way to visualise the design of a system [3].

This set of specifications is based on the assumption that the conceptual models are represented with UML. Moreover, for the purposes of this convention only the structural elements of UML are considered, and in particular those comprising a class diagram. Next, the most important structural elements are introduced.

A *class* represents a discrete concept within the domain being modelled. It is a description of a set of individual objects that share the same attributes, behaviour, relationships. Graphically, a class is rendered as a rectangle [3].

An *instance* or *individual object* is a discrete (run-time) entity with identity, state and invocable behaviour and which can be distinguished from other (run-time) entities. We say that an individual object instantiates a class and represents a concrete (run-time) manifestation of that class. Conversely, a class represents the abstract concept by which to understand and describe the multitude of instantiated individual objects.

A *property* is a structural feature which represents some named part of the structure of a class and characterises it in a particular fashion. It can be an attribute of a classifier or a member end of a relation.

An attribute is a named property of a class that describes the type and a range of values that instances of the property may hold. An attribute may be conceptualised as a slot shared by all objects of that class that is filled by values through instantiation [3].

When building abstractions very few classes stand alone. Instead most of them are connected to each other in a number of different ways. In UML, there are three kinds of *relationships* that are important in this specification: *dependencies*, which represent using relationships among classes (including refinement, trace, and bind relationships); *generalisations*, which link generalised classes to their specialisations; and *associations*, which represent structural relationships among objects. Each of these relationships provides a different way of combining your abstractions [3].

When a class participates in an association, it has a specific role that it plays in that relationship. A *role* is the face the class at the near end of the association presents to the class at the other end of the association. It is possible to explicitly name the role a class plays in an association [3].

An association represents a structural relationship among objects. In many modelling situations, it's important for you to state how many objects may be connected across an instance of an association. This "how many" is called the *multiplicity* of an association's role, and is written as an expression that evaluates to a range of values or an explicit value. It is possible to show a multiplicity of exactly one [1], zero or one [0..1], many [0..*], one or more [1..*], or even an exact number (for example, 3) [3]. Multiplicity applies not only to associations, but to dependency relation as well,

and also, to class attributes.

A *stereotype* represents an extensibility mechanism that is foreseen in UML. It allows for the possibility to create new domain specific kinds of elements that are derived from the existing standard ones. In the simplest form, they act as annotations on the UML building blocks, but can redefine entirely the visual representation of the UML element. For example some elements may be considered, optional, recommended or required in the context of information exchange. This is possible by creating the three stereotypes and applying them accordingly [3].

2.3 Formal ontology

Much has been discussed about what an ontology is and is not [13]. In computational context, an ontology encompasses a representation, formal naming and definition of the categories, properties and relations between the concepts, data and entities that substantiate one, many or all domains of discourse.

Here we adopt Studer et al. [29] definition that *"an ontology is a formal, explicit specification of a shared conceptualization"*. In this specification we adopt *Web Ontology Language (OWL 2)* [9, 21, 22] to specify the formal ontologies. OWL 2 is a knowledge representation language, with formally defined meaning, designed to formulate, exchange and reason with knowledge about a domain of interest.

OWL 2 ontologies can be used along with information written in *Resource Description Framework (RDF)* [30]. RDF is a standard model for data interchange on the Web. And OWL 2 ontologies themselves are primarily exchanged as RDF documents.

An RDF document is composed of RDF statements. The *RDF statement*, or *triple*, is a three-slotted structure of the form $< subject - predicate - object >$. The RDF statement asserts that some relationship, indicated by the predicate, holds between the resources denoted by the subject and object [30]. The subject is always a resource identified by an URI, while the object may be either a URI resource or a literal value. Next, the relevant OWL 2 concepts are introduced.

Classes provide an abstraction mechanism for grouping resources with similar characteristics. Classes can be understood as sets of individuals, called the class extension. The individuals in the class extension are called the instances of the class [9].

Individuals in OWL 2 represent actual objects from the domain. There can be named individuals, which are given an explicit name to refer to the same object; and anonymous individuals, which do not have an explicit name and are used locally.

Datatypes are entities that refer to sets of data values. Thus, datatypes are analogous to classes, the main difference being that the former contain data values such as strings and numbers, rather than individuals [21].

Literals represent data values such as particular strings or integers. They can also be understood as individuals denoting data values. Literals can be either plain (no datatype) or typed [21].

In OWL 2 *properties* are defined as that which can take the *predicate* role in an RDF statement, and are distinguished as object properties and datatype properties. *Object properties* represent relationships between pairs of individuals. *Data properties* represent relationships between an individual and a literal. In some knowledge representation systems, functional data properties are called attributes [21].

3 Conventional constraints

Defining naming and structural conventions for concepts in an ontology and then strictly adhering to these conventions doesn't only makes the ontology easier to understand, but also helps avoid some common modelling mistakes.

UML is a language without formal semantics. Moreover, it is quite flexible and permissive with ways in which a concept can be expressed. Also, there are many alternatives in naming concepts. Often there is no particular reason to choose one or another alternative. However, we need to define a set of naming conventions for classes, relations, attributes, controlled lists and adhere to it [19].

In theory any name can be assigned to a concept, relationship or property. In practice, there are two types of constraints on the kind of names that should be used: *technical* and *conventional*. This section deals with conventional constraint, while the technical constraints are addressed in Section 4.

3.1 What is in a name?

The naming conventions apply to the *element names* in the conceptual model. These names are intended for further use as human-readable denominations, called *labels*;

and machine-readable denominations, called *identifiers*. The identifiers serve as a basis for generating URIs [2] to ensure unambiguous reference to a formal construct; while the labels are meant to ease the comprehension by human-readers. For this reason we will consider that mostly the conventional recommendations provided here apply to them and none of the technical constraints.

The names should also belong to and be organised by *namespaces*. They can be provided as a short prefix to the elements name, for example "org:Organisation", "epo:Notice" or "skos:Concept". Namespaces are addressed in detail in Section 4.1.

In [25] a simple convention is proposed: that the identifier of a conceptual model element is the name of the element, where spaces have been removed. For example, the identifier of the "Legal Entity" class is "LegalEntity". Note that the casing is important and is addressed in Section 3.2.

It is recommended that the names and descriptions for classes and properties are expressed in British English [10]. In addition, a mechanism for providing a multilingual labelling system should be adopted.

It is recommended to avoid abbreviations in concept names. Also the words employed in the meta-model such as "class", "property", "attribute", "connector" etc. should be avoided as well. Names which are nonsensical, unpronounceable, hard to read, or easily confusable with other names should not be employed [5].

3.2 Case sensitivity

We can greatly improve the readability of an ontology if we use consistent capitalisation for concept names. For example, it is common to capitalise class names and use lower case for property names. And so, the names of classes, data-types and enumerations must begin with a capital letter while the names of class attributes, enumeration items, association and dependency relations, including their source and target roles, must begin with a lower case character.

All the names are case sensitive. This means that the class "Buyer" and the attribute "buyer" are two different names. Nonetheless such similarities are strongly discouraged and more elaborated names are highly encouraged. For example, a simple elaboration is to use suffixes or prefixes.

3.3 Delimitation

In UML, the spaces in names are allowed and using them may be the most intuitive solution for many ontology developers. It is however, important to consider other systems with which your system may interact. If those systems do not use spaces or if your presentation medium does not handle spaces well, it can be useful to use another method [19].

It is recommended that the element names should avoid using spaces and instead follow a camel-case convention. *CamelCasing* is the practice of writing phrases such that the word or abbreviation in the middle of the phrase begins with a capital case.

Exceptionally, if the conceptual model authors must maintain high readability of the UML diagrams, spaces may be tolerated and handled by the conversion script. In the conversion process, spaces are trimmed and phrases turned into camel-case form. For example " Pre-award catalogue request " is transformed into "PreAwardCatalogueRequest".

3.4 Name uniqueness

In the formal ontology, each class, property or individual in the formal ontology must be uniquely identifiable in the formal ontology. Therefore the elements of the conceptual model, classes, attributes, connectors, instance, should have unique names.

This means that there cannot exist a class and an attribute with the same name (such as a class "Buyer" and a property "buyer"). Neither there can exist a class and an instance or an instance and a relation with the same name.

Names that reduce to the same identifier are considered unique. For example "Legal Entity" and "LegalEntity" are different labels but they reduce to the same identifier "LegalEntity". In such cases the names are considered equal and the UML elements replicated.

Nevertheless, name uniqueness is a recommendation but sometimes it is useful to replicate an UML element. In such cases, when the names are reused, the assumption is that the two UML elements represent manifestations of the same meaning. This is especially important for relations and is explained in Section 3.8.

3.5 Suffix and prefix

Some ontology engineering methodologies suggest using prefix and suffix conventions in the names to distinguish between classes and attributes. Two common practices are to add a "has-" or a suffix "-of" to attribute names. Thus, our attributes become "hasAwardStatus" and "hasBuyer" if we chose the "has-" convention. The attributes become "awardStatusOf" and "buyerOf" if we chose the "of-" convention. This approach allows anyone looking at a term to determine immediately if the term is a class or an attribute. However, the term names become slightly longer [19].

Here it is recommended that the names of class attributes employ the "has-" suffix.

Other common suffixes are the prepositions "-by" and "-to". The organisation ontology [27] exemplifies their usage in cases such as "embodiedBy" and "conformsTo". However, if the preposition can be avoided, then do so [24].

It is recommended to use prepositions in the ontology terms only if necessary.

Optionally common and descriptive prefixes and suffixes for related properties or classes may be used. While they are just labels and their names have no inherent semantic meaning, it is still a useful way for humans to cluster and understand the vocabulary. For example, properties about languages or tools might contain suffixes such as "Language" (e.g. "displayLanguage") or "Tool" (e.g. "validationTool") for all related properties [10].

3.6 Classes

When choosing class names, it is conventional to use simple nouns or noun phrases. In case the class refers to actions, states, relations or qualities, which are usually expressed in natural language by verbs or adjectives then they must be nominalised. We often form nouns from other parts of speech, most commonly from a verb or an adjective. We can then use the noun phrase instead of the verb or adjective to create a more formal style. This process is called nominalisation.

A class name represents a collection of objects. For example, a class "Language" actually represents all languages. Therefore, it could be more natural for some model designers to call the class "Languages" rather than "Language". In practice, however, the singular is used more often for class names, while the plural for sets and collections [19]. Therefore, it is required that the class names must always use the singular lexical form.

When building the class hierarchy, names of direct subclasses of a class should consistently either all include or not include the name of the superclass. For example, if we are creating two subclasses of the "Wine" class to represent red and white wines, the two subclass names should be either "Red Wine" and "White Wine" or "Red" and "White", but not "Red Wine" and "White" [19].

Class specialisations with a single child must be avoided. This means that there should be at least two sibling subclasses specified in the model. By default the classes are not disjunctive, however, if required, the transformation script may generate disjunctive classes by default.

Circular inheritance must be avoided. This means that if there is a B that specialises a class A then A may not specialise B or any of the sub-classes of B.

3.7 Relations

When establishing relations between concepts it is conventional to use verbs of action, state, process or relation such as such as "includes", "replaces", "manages". It is required to use a verb or a verb phrase for relationship terms. It should be in *lowerCamelCase* such that $< subject - predicate - object >$ triples may actually be read as natural language clauses, e.g. "EconomicOperator offers ProcuredItem" [10].

The verb phrase must be in present tense, if needed inflected as third person singular. If an additional level of specificity is needed a qualifying nominal phrase may be appended.

Relationships are usually bi-directional and the inverse one should be provided where it makes sense. Adjust the verb phrases in the predicates as appropriate, usually, by employing the *active and passive voice* in the term formulation brings the desired result. For example, "uses/isUsedBy" and "refersTo/isReferredToBy" or "offers/isOfferedBy" [10].

The name of the inverse relation should not be semantically inverted verb, such as in case of "buys/sells" , "open/closes". The semantically inverted dichotomies must be modelled in separate connectors because they represent different relations. For example the dichotomy "buys/sells" should be modelled as two pairs: "buys/isBoughtBy" and "sells/isSoldBy".

When the relation is of different nature, more like an attribute, then prefixing and

suffixing techniques can be employed. For example, in the Organisation Ontology [27], the concepts of an "Organisation" and a "Site" are defined along with two relationships that are the inverse of each other: "Organisation hasSite Site" and "Site siteOf Organisation" [24].

It is recommended that each relationship includes a definition of its inverse.

Models should define such inverse pairs for relationships although this does not extend to attributes. For example, Dublin Core[15] includes a property of "dateAccepted", there is no inverse property that would link a given date, which is expressed as a simple value, to all the documents accepted for publication on that date.

3.8 Relations reusability

The relation names should be chosen so that there is a balance of accuracy and precision on one hand and the relation reusability on the other hand. These two dimensions are inversely correlated: the higher the reuse the lower the accuracy and vice versa.

On one hand, if we choose more generic predicates such as "isSpecifiedIn" this tends towards maximising relation reusability across the model. Yet at the same time the risk of overloading the relation meaning also increases.

On the other hand, the above risk could be mitigated by simply appending the range class to the relation name: such "isSpecifiedInContract" and such "isSpecifiedInProcedure" following the following naming pattern: `verbPhrase + [RangeClassName] Qualifier`. This ensures predicate uniqueness and maximum level of specificity at the cost of reusability across and beyond the model. The latter can be achieved through inference, but an additional predicate inheritance structure must be defined. Another risk is that a change or evolution of the name of the class has a direct impact on all incoming predicates, and thus raising the chances of errors. This in turn may decrease the model agility and elasticity.

Optionally, the transformation process from the conceptual model to the formal ontology, may contain a mechanism of appending the name of the range class to the predicate name in order to automatically produce a predicate with higher specificity, shall this be required.

3.9 Attributes

When creating attribute names, it is conventional to use simple nouns such as "name", "weight", "colour". Attributes are a special type of relations that describe an entity in terms of its qualities. And so, to be consistent with the above convention and in order to increase the clarity, it is recommended to employ the prefix "has-" for each attribute even if it does not add much to the term's meaning. So, it is preferred to use terms such as "hasName", "hasWeight" and "hasColour".

It is recommended to use simple nouns for attribute names prepended with the verb "has-".

To avoid laborious mechanical work of adding the prefix, it is possible to rely on the convention that the attribute names starting with a capital letter must be read as having the "has-" prefix. It means that the transformation script will prepend the "has-" prefix to all attributes starting with a capital letter.

By default, the attribute multiplicity is "1". This should be read as any number which is expressed as "0..*". In special cases, a list of custom default multiplicities is defined for the transformation script. That means that some data types or classes that are used as attribute types are paired with a default multiplicity, for example "1..1", "0..1", "2..2".

3.10 Controlled lists

The controlled list is a carefully selected list of words and phrases and is often employed in the modelling practices. The controlled list has a name and a set of terms. For example the list of grammatical genders can be named "Gender" and comprise the terms "masculine", "feminine", "neuter" and "utrum".

It is required that the controlled lists are named using nouns or nominal phrases starting with a capital letter. The enumeration items must start with a lower case.

As a rule of thumb, but not always, the relationship between the controlled list as a whole and its comprising elements can be informally conceptualised as a class-instance, class-subclass, set-item, or part-whole.

3.11 Notes, descriptions and comments

Large emphasis is set on the naming conventions. Nonetheless, most often, a good name is insufficient for an accurate or easy comprehension by human-readers. To mitigate this and increase the conceptual richness, practitioners may wish to provide human readable definitions, notes, examples and comments grasping the underlying assumptions, usage examples, additional explanations and other types of information.

It is recommended that each element is defined by a crisp, one-line definition. The definition starts with a capital letter and ends with a period.

A description may provide complementary information concerning the usage of the element or its relation to relevant standards. For example, a description may contain recommendations about which controlled vocabularies to use, describe the underlying assumptions and additional explanations for reducing ambiguity. Descriptions may contain multiple paragraphs separated by blank lines. The descriptions should not paraphrase the definitions.

In case the model editor provides concrete examples of possible element values or instances then they shall be provided as a comma-separated list. Each example value is enclosed in quotes and is optionally followed by a short explanation enclosed in parentheses [25].

4 Technical constraints

4.1 Namespaces

In order to enable the reuse of names defined in other models and reuse of unique references for names that support easy identification, namespace management must be considered. We adopt XML approach to defining and managing namespaces as it is inherent in both XMI and OWL2 standards. Hence, a *namespace* is a set of symbols that are used to organise objects of various kinds, so that these objects may be referred to by name and uniquely identifiable.

Namespaces are commonly structured as hierarchies to allow reuse of names in different contexts [16]. This mechanism can be implemented in UML through partitioning the model using packages, which are described in 4.3.

A namespace organises a collection of names obeying three constraints: each name is (1) unique, (2) assigned in a consistent way, and (3) assigned according to a common definition [28]. An (expanded) *name* in a namespace is a pair consisting of a *namespace name*, also called *base URI* or just *base*, and a *local name*, also called *local segment* [5, 18]. The combination of universally managed URI with the vocabulary local name is effective in avoiding name clashes. For example, in the expanded name "http://www.w3.org/ns/org#Organization", "http://www.w3.org/ns/org#" is the namespace name and "Organization" is the local name.

Unlike in the XML specifications, the constraints on the local name are slightly relaxed, allowing token delimitation by space character (see Section 3.3). This provides an additional level of readability to the conceptual model users. Nevertheless, the local names must be *normalised strings*, which mean that only single occurrences of space character are permitted. Other delimiting characters, such as tab, line feed, carriage return must be replaced by an occurrence of space and trimmed. In the transformation process, when URIs are generated, the spaces are removed anyway and conform with the XML conventions (see Section 3.1).

```
name = <namespace name>/<local name>
name = <namespace name>#<local name>
```

URI references are often inconveniently long, so expanded names should not be used directly. Instead *qualified names* should be used while expanded names are strongly discouraged. A *qualified name* is a name subject to namespace interpretation. Syntactically, they are either *prefixed names* or *unprefixed names*.

```
qualified name = [<namespace prefix>:]<local name>
```

The namespace name is usually applied as a *prefix* to the local name but it may be missing as well. [16] specifies a declaration syntax which permits to bind prefixes to namespace names and also to bind a default namespace that applies to unprefixed element names. For example we can bind the namespace name "http://www.w3.org/ns/org#" to the prefix "org", which we can then use to create the same name as such "org:Organization". The prefix is subject to namespace interpretation and resolved to an URI [16].

It is recommended that the UML element names indicate the namespace prefix by prepending it to the name delimited by colon character (:). In case the namespace is not specified (no delimiter) then the name of the package containing the current element is used as namespace prefix. For example if a class "Contract" is placed in a package "epo" then the name of the containing package is used as the namespace

prefix and resolved to "epo:Contract". If the delimiter (:) is used without any prefix, then the empty string prefix is resolved to the default namespace as defined in [16].

4.2 Character encoding

In the formal ontology, the names must conform to RDF [30] and XML[5] format specifications. Both languages effectively require that terms begin with an upper or lower case letter from the ASCII character set, or an underscore (_). This tight restriction means that, for example, terms may not begin with a number, hyphen or accented character [24]. Although underscores are permitted, they are discouraged as they may be, in some cases, misread as spaces. A formal definition of these restrictions is given in the XML specification document [5].

It is required that the names use words beginning with an upper or lower case letter (A–Z, a–z) or an underscore (_) for all terms in the model. Digits (0–9) are allowed in the subsequent character positions. Also, as mentioned above, spaces are permitted in the local segment of the name.

Encoded UTF-8 and UTF-16 names are supported [5, 20] but we recommend avoiding any character encodings in the element names. Encoded characters are mostly not readable and require a decoding to become human friendly. Also unexpected results may occur in the transformation script. This recommendation does not apply to the content strings such as descriptions, notes and comments, which may use any character encoding.

4.3 uml:Package

Packages should be used to define a logical partition of the model. They serve as the primary method for the vertical slicing of the conceptual model as described in the layering and slicing section of Costetchi [7].

Packages may form hierarchies. In this case the hierarchical relation is interpreted as *meronymy*, denoting a constituent parts of the package. Formally they are translated into owl:import statements.The module corresponding to the parent package imports modules corresponding to the child packages.

No empty packages shall be present in the model. A package is empty if it contains no child elements.

Package names shall be short lowercase normalised strings representing an acronym or a short name. They may serve as proxies for the namespace prefixes and used to resolve the name of any comprised elements when the prefix is not provided. This however should not be used as a primary naming method, but rather for suggesting corrections in the element name.

4.4 uml:Class

uml:Class is transformed into an owl:Class. Each uml:Class must have a name and should have a description representing the human readable class definition in the domain context.

Optionally, in case there is a technical possibility to distinguish between UML notes, then additionally editorial, history notes or simply comments should be provided as class descriptors.

It is recommended that a uml:Class has relations, attributes, or both. A class must not miss both, attributes and relations associated with it. It is mandatory to avoid using the same name in a class, attribute or a relation.

Classes may use << *abstract* >> *stereotype*. It means that no instances are allowed of this class in the datasets. This is not covered by the OWL 2 [21] but can be expressed in SHACL data shapes [14].

4.5 uml:Class attributes

uml:Attribute is mostly transformed into owl:DataProperty and in some controlled cases into owl:ObjectProperty.

Each uml:Attribute must have a name and attribute type. The name is used to generate URI and label while the type is used to define the range restriction.

An attribute may contain an alias, which is used as an alternative label; and it may have initial value provided which is transferred into a definition.

It is recommended that the attribute type is one of the XSD and RDF datatypes compatible[1] with OWL 2. Exceptionally, generic data types such as “Numeric”, “String”, “Date” can be used. In such cases the transformation script uses a correspondence table defining which XSD data type shall be used for each atomic UML

[1] `https://www.w3.org/2011/rdf-wg/wiki/XSD_Datatypes`

type. If the datatype is not found in the correspondence table then it is considered invalid.

The attribute multiplicity should be specified indicating the minimum and maximum cardinality. The default [1] multiplicity shall be interpreted as unspecified as expressed as [0..*] in the OWL model.

It is recommended to avoid duplicate attributes names across multiple classes. Unless, by design, attributes with the same name are shared across multiple classes.

It is mandatory to avoid using the same name in an attribute and in a relation, unless there is an additional rule that handles intentional exceptions.

All attribute data types must be defined in the model for reference, regardless if they are reused from other models or specific to the local model. In the case of reused external models, the local (re-)definitions serve merely as proxies as explained in Section 4.7.

It is recommended that the attribute type is an atomic data type. It is possible to use a uml:Enumeration as an attribute type. These cases are transformed into owl:ObjectProperty in a manner similarly to uml:Dependency described in Section 4.9.

It is recommended to avoid using another class as the attribute type. An acceptable exception for this is with a controlled set of classes. The list of allowed classes must be explicitly indicated in the transformations script. These cases are transformed into owl:ObjectProperty in a manner similarly to uml:Association described in Section 4.8. For the eProcurement project the set of exceptions is: Identifier, Amount, Quantity, Measure, Code. These were initially defined as composite datatypes and then transformed into classes.

4.6 uml:Enumeration

In UML the controlled lists, discussed in Section 3.10 are represented as uml:Enumeration. They are transformed into instances of a SKOS model [17].

Each uml:Enumeration element is transformed into skos:ConceptScheme and each enumeration item (represented by an uml:Attribute) is transformed into a skos:Concept. An enumeration must not be empty.

In an enumeration element, the name shall be interpreted as the controlled list name;

it must be a normalised string. Each attribute name is used as a local segment in the generation of the concept URI. The attribute type is ignored and by default is considered to be skos:Concept. The attribute alias is transformed into skos:Concept preferred label. The attribute initial value is transformed into the alternative label of the concept. If the attribute alias is longer than the attribute initial value, then it is considered that the two fields have been swapped by mistake.

In case no attribute alias is specified then the attribute name is used as preferred label of the skos:Concept. This happens as skos:prefLabel is a mandatory property in the SKOS model.

It is possible to employ the enumerations for class properties in two manners: (a) either define an attribute (uml:Attribute) that indicates as type the enumeration name (b) or draw a dependency (uml:Dependency) relation from the class to the enumeration and provide a relation role. An intermediary, and discouraged solution, is to define attributes with the generic type "Code" and, in addition, draw a dependency relation with the same role name to the specific enumeration. In these cases an additional verification should be performed for matching the role on the dependency relation to the attribute name.

4.7 uml:Datatype

This convention draws the distinction between primitive (or atomic) types (consisting of single literal value) and composite types (consisting of multiple attributes) [25]. In fact, the composite datatypes must be defined as classes and handled as such. For example: AmountType, Identifier, Quantity and Measure are to be treated as classes even if conceptually they could be seen as composite data types.

It is recommended to employ the primitive datatypes that are already defined in XSD [23] and RDF [30], which cover the standard and most common types. Thus definitions of custom data types shall be avoided unless the model really needs them. Such cases are, however, rare.

The data types defined in the UML model (and custom ones) are resolved into their XSD equivalent using the correspondences from Table 1. Note that the family of string datatypes is mapped to *rdf:langString*. This means that the instance data should provide a language tag for the textual data indicating how it should be read. This enables multilingual data specification. Also, note that Date is mapped to xsd:date and DateTime is mapped to xsd:dateTime. However the xsd:date is not

included in the OWL2 interpretation and instead a strong preference is expressed fro xsd:dateTime. Therefore it is recommended to follow the OWL2 specification, although the xsd:date is a valid datatype in the RDF data and in SPARQL queries.

Table 1: UML to XSD datatype correspondences

UML	XSD
Boolean	xsd:boolean
Float	xsd:float
Integer	xsd:integer
Char, Character, String	rdf:langString
Short	xsd:short
Long	xsd:long
Decimal	xsd:decimal
Date	xsd:date
DateTime	xsd:dateTime

It is recommended to use OWL 2 compliant XSD and RDF standard data types. They may be useful in indicating a specific data type which is not possible with UML ones. For example making a distinction between a general string (xsd:string) and a literal with a language tag (rdf:langString) or XML encoded ones such as rdf:HTML and rdf:XMLLiteral.

For the model consistency, it is recommended that the proxy data types be defined in the model for the XSD[2] and RDF data types[3] used in the model. The proxies must follow the standard namespace convention using the "rdf" and "xsd" prefixes.

4.8 uml:Association

The uml:Association connectors represent relations between source and target classes. The association connector cannot be used between other kinds of UML elements.

A generic UML connector may have a name applied to it, and it may have source/target roles specified in addition. This provides flexibility to how the domain knowledge may be expressed in UML, however this freedom increases the level of ambiguity as

[2] `https://www.w3.org/2011/rdf-wg/wiki/XSD_Datatypes`
[3] `https://www.w3.org/TR/rdf11-concepts/#section-Datatypes`

well. Therefore, we foresee two distinct ways to express properties: using the connector generic name, or using the connector source/target ends.

First, if a connector name is specified then no source or target roles can be provided. The name must be valid as it is used to generate the OWL property URI. The minimum and maximum cardinality of the relation must be specified as target multiplicity.

The second, and recommended approach is if the connector has no name then the target role must be specified. Or the converse, if a target role is specified then no connector name can be specified. Optionally a source role may be provided. In this case the relation direction must be changed from "Source-¿Target" to "Bidirectional". Or conversely, if the connector direction is "Bidirectional" then source and target roles must be provided. No other directions are permitted.

The target and source multiplicity must be specified accordingly indicating the minimum and maximum cardinality.

It is recommended that each association has a definition. The definition is then used for each role as they stand for the same meaning manifested in the inverse direction. Additional, specific definition, can be specified along the target and source roles.

4.9 uml:Dependency

The dependency connector may be used between uml:Class and uml:Enumeration boxes, oriented from the class towards the enumeration. It indicates the class has an owl:ObjectProperty whose range is a controlled vocabulary. The connector must have direction "Source-¿Target". No other directions are acceptable.

The connector must have a valid name and no source/target roles are acceptable. The multiplicity must be specified at the target of the connector.

In the transformation process, for the reasoning purposes, the range of the property must be expressed as a range restriction using owl:oneOf the values from the enumeration Concept scheme. This is also valuable for generating SHACL shapes.

4.10 uml:Generalization

The uml:Generalization connector signifies a class-subClass relation and is transformed into rdfs:subClassOf relation standing between source and target classes.

The connector must have no name or source/target roles specified in the UML model.

In case a model class should inherit a class from an external model then proxies must be created for those classes. For example if “Buyer” specialises an “org:Organization” then a proxy for “org:Organization” must be created in the “org” package.

In this specification, the subclasses are assumed disjoint by default, unless otherwise specified in the transformations script, or explicitly marked on the generalisation relation with ¡¡non-disjoint¿¿ stereotype. For the converse case the ¡¡disjoint¿¿ stereotype shall be used.

In case two classes are equivalent, then the << *equivalent* >> or << *complete* >> stereotype should be used as a marker.

Bibliography

[1] Xml metadata interchange (xmi) specification: Version 2.5.1. Standard formal/2015-06-07, Object Management Group (OMG), 2015. URL `http://www.omg.org/spec/XMI/2.5.1`.

[2] T. Berners-Lee, R. T. Fielding, and L. M. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986, Jan. 2005. URL `https://rfc-editor.org/rfc/rfc3986.txt`.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005. ISBN 0321267974.

[4] S. O. Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119, Mar. 1997. URL `https://rfc-editor.org/rfc/rfc2119.txt`.

[5] T. Bray, M. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Paoli. Extensible markup language (XML) 1.0 (fifth edition). W3C recommendation, W3C, Nov. 2008. http://www.w3.org/TR/2008/REC-xml-20081126/.

[6] S. Cook, C. Bock, P. Rivett, T. Rutt, E. Seidewitz, B. Selic, and D. Tolbert. Unified modeling language (UML) version 2.5.1. Standard formal/2017-12-05, Object Management Group (OMG), Dec. 2017. URL `https://www.omg.org/spec/UML/2.5.1`.

[7] E. Costetchi. eProcurement ontology architecture and formalisation specifications. Recommendation, Publications Office of the European Union, April 2020.

[8] E. Costetchi. eProcurement uml conceptual model to owl ontology transformation. Recommendation, Publications Office of the European Union, April 2020.

[9] M. Dean and G. Schreiber. OWL web ontology language reference. W3C recommendation, W3C, Feb. 2004. http://www.w3.org/TR/2004/REC-owl-ref-20040210/.

[10] M. Dekkers, E. Stani, B. Wyns, and F. Barthelemy. D02.01 - specification of the process and methodology to develop the eprocurement ontology with initial draft of the eprocurement ontology for 3 use cases. Deliverable SC378DI07171, Publications Office of the European Union, 2017.

[11] M. Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[12] M. Grunninger. Enterprise modelling. In *Handbook on enterprise architecture*, pages 515–541. Springer, 2003.

[13] N. Guarino, D. Oberle, and S. Staab. What is an ontology? In *Handbook on ontologies*, pages 1–17. Springer, 2009.

[14] H. Knublauch and D. Kontokostas. Shapes constraint language (SHACL). W3C recommendation, W3C, July 2017. https://www.w3.org/TR/2017/REC-shacl-20170720/.

[15] J. Kunze and T. Baker. The dublin core metadata element set. Technical report, RFC 5013, August, 2007.

[16] A. Layman, T. Bray, H. Thompson, D. Hollander, and R. Tobin. Namespaces in XML 1.0 (third edition). W3C recommendation, W3C, Dec. 2009. http://www.w3.org/TR/2009/REC-xml-names-20091208/.

[17] A. Miles and S. Bechhofer. SKOS simple knowledge organization system reference. W3C recommendation, W3C, Aug. 2009. http://www.w3.org/TR/2009/REC-skos-reference-20090818/.

[18] R. Moats. Urn syntax, 1997.

[19] N. F. Noy, D. L. McGuinness, et al. Ontology development 101: A guide to creating your first ontology, 2001.

[20] J. Paoli, F. Yergeau, M. Sperberg-McQueen, T. Bray, E. Maler, and J. Cowan. Extensible markup language (XML) 1.1 (second edition). W3C recommendation, W3C, Aug. 2006. http://www.w3.org/TR/2006/REC-xml11-20060816/.

[21] B. Parsia, P. Patel-Schneider, and B. Motik. OWL 2 web ontology language structural specification and functional-style syntax (second edition). W3C recommendation, W3C, Dec. 2012. http://www.w3.org/TR/2012/REC-owl2-syntax-20121211/.

[22] P. Patel-Schneider, B. Parsia, and B. Motik. OWL 2 web ontology language structural specification and functional-style syntax. W3C recommendation, W3C, Oct. 2009. http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/.

[23] D. Peterson, A. Malhotra, S. Gao, M. Sperberg-McQueen, P. V. Biron, and H. Thompson. W3C xml schema definition language (XSD) 1.1 part 2: Datatypes. W3C recommendation, W3C, Apr. 2012. http://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/.

[24] PwC EU Services. D3.1 - process and methodology for core vocabularies. Deliverable, ISA programme of the European Commission, 2011.

[25] PwC EU Services. e-government core vocabularies handbook. Report, ISA programme of the European Commission, 2015. URL `https://ec.europa.eu/isa2/library/e-government-core-vocabularies-handbook_en`.

[26] E. Rescorla, R. Barnes, and S. Kent. Further Key Words for Use in RFCs to Indicate Requirement Levels. RFC 6919, Apr. 2013. URL `https://rfc-editor.org/rfc/rfc6919.txt`.

[27] D. Reynolds. The organization ontology. W3C recommendation, W3C, Jan. 2014. http://www.w3.org/TR/2014/REC-vocab-org-20140116/.

[28] P. Saint-Andre and J. Klensin. Uniform resource names (urns). *Internet Engineering Task Force (IETF), RFC*, 8141, 2017.

[29] R. Studer, V. R. Benjamins, and D. Fensel. Knowledge engineering: principles and methods. *Data & knowledge engineering*, 25(1-2):161–197, 1998.

[30] D. Wood, R. Cyganiak, and M. Lanthaler. RDF 1.1 concepts and abstract syntax. W3C recommendation, W3C, Feb. 2014. http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.