Bi-directional Transformation between Normalized Systems Elements and Domain Ontologies in OWL

Marek Suchánek[1][a], Herwig Mannaert[2], Peter Uhnák[3][b] and Robert Pergl[1][c]

[1] *Faculty of Information Technology, Czech Technical University in Prague, Thákurova 9, Prague, Czech Republic*

[2] *Normalized Systems Institute, University of Antwerp, Prinsstraat 13, Antwerp, Belgium*

[3] *NSX bvba, Wetenschapspark Universiteit Antwerpen, Galileilaan 15, 2845 Niel, Belgium*

Keywords: Ontology, Normalized Systems, Transformation, Model-driven Development, Ontology Engineering, Software Modelling.

Abstract: Knowledge representation in OWL ontologies gained a lot of popularity with the development of Big Data, Artificial Intelligence, Semantic Web, and Linked Open Data. OWL ontologies are very versatile, and there are many tools for analysis, design, documentation, and mapping. They can capture concepts and categories, their properties and relations. Normalized Systems (NS) provide a way of code generation from a model of so-called NS Elements resulting in an information system with proven evolvability. The model used in NS contains domain-specific knowledge that can be represented in an OWL ontology. This work clarifies the potential advantages of having OWL representation of the NS model, discusses the design of a bi-directional transformation between NS models and domain ontologies in OWL, and describes its implementation. It shows how the resulting ontology enables further work on the analytical level and leverages the system design. Moreover, due to the fact that NS metamodel is metacircular, the transformation can generate ontology of NS metamodel itself. It is expected that the results of this work will help with the design of larger real-world applications as well as the metamodel and that the transformation tool will be further extended with additional features which we proposed.

1 INTRODUCTION

Ontologies are widely used in the software engineering domain to describe the meaning of data in a machine-actionable and yet flexible format (Bhatia et al., 2016). The usage of ontologies and related technologies such as the Web Ontology Language (OWL) or the Resource Description Framework (RDF) can vary – Semantic Web, integration of databases, or ontology-driven conceptual modelling. An ontology forms a graph of terms, their relations and properties that represent knowledge of a particular domain. Graphs of different ontologies can also be connected using relations of their terms. This versatility and modularity bring many exciting opportunities in terms of integrations and transformations (Isotani et al., 2015). As ontologies are crucial in current software engineering, a discipline of ontology engineering is becoming more prominent(Hitzler et al., 2016).

Normalized Systems (NS) theory (Mannaert et al., 2016) targets the evolvability of systems as fine-grained modular structures. Although the theory is applicable to various domains, for example, civil engineering, it describes how to build evolvable-proven software using so-called Elements. The realization of this theory exists in the form of code expanders and related tool Prime Radiant to streamline the process. Using expanders, the NS model, composed of Elements and related entities, is used together with custom code fragments to produce the evolvable enterprise information system. When change occurs in a model or technologies, the application can be easily expanded again with the new changes. Normalized Systems has been applied to several real-world and large-scale applications (Huysmans and Verelst, 2013; De Bruyn, 2011; Oorts et al., 2014).

[a] https://orcid.org/0000-0001-7525-9218
[b] https://orcid.org/0000-0003-1057-6073
[c] https://orcid.org/0000-0003-2980-4400

Suchánek, M., Mannaert, H., Uhnák, P. and Pergl, R.
Bi-directional Transformation between Normalized Systems Elements and Domain Ontologies in OWL.
DOI: 10.5220/0009356800740085
In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2020)*, pages 74-85
ISBN: 978-989-758-421-3
Copyright © 2020 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

Currently, two problems lie in the NS models. First, NS models tie domain-specific knowledge such as entities, their attributes, processes, or views on entities together with implementation-specific configurations such as form-rendering details, visibility of attributes, or timed triggers and integrations with other systems. The second and related issue is the absence of explicit compatibility or mappings with different modelling languages that are often used for domain analysis such as UML, OntoUML, ORM, or BPMN. Elimination of those issues would allow to re-use domain models from other languages and to create models without the knowledge of implementation details.

In this work, we present first systematic approach of providing model interoperability of the domain-specific part of any NS model using transformation to OWL ontology and back. By choosing OWL we open the NS model to a wide set of tools and further ontology engineering. Bi-directionality of transformation is essential as models exist on both sides. First, we want to transform NS models (including the NS metamodel itself) into ontology in OWL. On the other hand, we may edit and compose models in OWL using other models and ontology mappings. The general hypothesis is that if we carefully select domain-specific parts and expose them using OWL, it will (1) allow analysis and further work with the model on ontology level free from implementation details, (2) make mappings from other modelling languages possible through existing OWL technologies, and (3) provide useful insights to NS metamodel.

This paper starts in Section 2 by introducing relevant topics with references to related work. Based on the related work, Section 3 describes the design of the bi-directional transformation and discusses various options and possible outcomes. Design serves for its implementation that is summarized in Section 4 and then evaluated in Section 6 together with proposals of future steps.

2 METHODOLOGY AND RELATED WORK

In this section, we present overview of the most relevant topics related to our work. The overview provides a necessary context for our approach.

2.1 Conceptual Modelling in Software Engineering

Conceptual modelling is a vital activity used in software engineering as well as other domains to promote understanding and communication related to the selected problem domain. It is primarily intended for human beings to model the domain using specific modelling language and with a focus on certain aspects. That, together with abstractions, helps to understand and overview the true essence (Verdonck et al., 2015). Some of the modelling languages, e.g. OntoUML, are called “ontology-driven” because they are tightly related to general concepts from some upper ontology – in case of OntoUML, it is UFO (Unified Foundation Ontology) (Guizzardi, 2005).

Notwithstanding, conceptual models are intended for humans; they are often used for other purposes as well. Some tools provide transformations of modelling languages to other languages or various formats including OWL. The more critical part of software engineering related to models is model-driven development. It uses models to automatically generate code fragments, skeletons, parts of applications, or even complex information systems. (Embley and Thalheim, 2012)

The usual problem with model-driven development (MDD) is that it does not handle changes over time well, and consistency between model and implementation is easily broken. It uses multiple layers of models: computation-independent, platform-independent, and platform-specific. MDD tries to limit the inconsistency issue but add levels of complexity to the whole process. (Tolvanen and Kelly, 2016)

2.2 Ontology Engineering

Ontology engineering is a discipline of computer and information science which deals with methods for designing and building ontologies. It covers the use of various formal languages for expressing ontologies, not just OWL – but also, for example, SHACL, OntoUML, Gellish. The main goals are to offer a direction towards solving the inter-operability problems brought about by semantic obstacles and well-describe the knowledge of a particular domain. The most often uses of ontology engineering are currently in fields of life sciences, data/system integration, and artificial intelligence. (Hitzler et al., 2016)

2.3 Normalized Systems

Normalized Systems Theory (Mannaert et al., 2016) targets the development of highly evolvable information systems where combinatorial effects are eliminated or systematically under control. The book (Mannaert et al., 2016) also describes how to build such software systems based on four elemen-

tary principles: Separation of Concerns, Data Version Transparency, Action Version Transparency, and Separation of States. It results in a fine-grained modular structure composed of so-called Elements. To make the development of evolvable information systems possible and efficient, it applies a code generation technique producing skeletons from the NS model and custom code fragments.

NSX spin-off uses this theory in practice to develop and maintain evolvable enterprise information systems for various customers. For the code generators, the Expanders are used that take the NS model, custom code fragments, technology and other configurations, and using code templates expand the information system. To automate the whole process, Prime Radiant tool serves to design the models in components, configure the application, and maintain deployments. (NSX bvba, 2019)

There are five types of NS Elements: Data, Task, Flow, Connector, and Trigger. Data Elements carry structural information about entities and their attributes and relationships. Transformation of Data Elements is the primary concern of this paper. Remaining Elements are concerned with behavior, orchestration, and interaction. The core of NS metamodel describes these NS Elements and is itself described by them, thus forming a metacircular model. Prime Radiant tool is also an evolvable application. (NSX bvba, 2019; Mannaert et al., 2019)

2.4 The Web Ontology Language

The Web Ontology Language (OWL) is a declarative language for expressing ontologies, i.e. sets of precise descriptive statements about some domain of interest (Hitzler et al., 2009). The overlap between ontologies and conceptual modelling is significant. Both share the same goal to capture the semantics. Additionally, conceptual models capture some domain ontology using a modelling language. OWL provides a machine-actionable and straightforward way of describing a domain using classes, properties, and individuals. In OWL, everything forms a triple: statement with subject, predicate, and object. This triplet is similar to RDF schemas, as OWL itself is based on RDF at a technical level[1]. However, OWL is a higher conceptual languages and provides additional semantics not available in RDF. Also, OWL uniquely identifies every object using international resource identifiers (IRIs).

OWL documents can be captured using various syntaxes such as RDF/XML, Turtle, Manchester, or OWL/XML. Another aspect that is vital for Semantic Web but also other use cases of OWL is the ability to link terms from other ontologies or use ontology mapping and alignment techniques for integration or transformation of knowledge in ontologies. Ontologies, together with OWL and RDF, are very flexible and enable to capture any knowledge to be further used, re-used, or linked to other knowledge. It is crucial in the domain of software engineering where semantics capture in machine-actionable and standard way streamline integration and data processing (Bhatia et al., 2016).

[1] uses RDF as an interchange/persistence mechanism

2.5 Ontology-driven Software Development

Ontologies as domain description are used in the software development cycle similarly to model-driven development already for several years in various ways. First, there are libraries and frameworks, e.g. RDF4J (Eclipse Foundation, 2019), rdflib (RDFLib Team, 2019), or Apache Jena (The Apache Software Foundation, 2019), that allow using OWL and RDF specifications in software. Then there are persistence libraries that allow integrating data classes with OWL ontologies and store instances in various triple-stores. Example of this approach is, for instance, JOPA (Ledvinka et al., 2016). Ontologies can also be used during the design and specification of information systems (Křemen and Kouba, 2011). Such ontology can provide semantics to multiple applications and correctness of data integration.

Related to the transformations of various conceptual and other models to OWL ontologies and back, several attempts for UML and especially its Class Diagrams has been made in (Gasevic et al., 2004), (Zedlitz et al., 2011), and (Sadowska and Huzar, 2019). There is also a working transformation from already-mentioned OntoUML to OWL and SWRL (Barcelos et al., 2013). Even closer to our work is the transformation of Extended ER models into OWL described in (Telnarova, 2018). It will be possible to take advantage of existing to-OWL transformations once we enable bi-directional conversion between NS and OWL.

2.6 Bi-directional Transformations

Bi-directional transformations (often called *BX*) are means of maintaining consistency between two or more representations of information used in various disciplines (Czarnecki et al., 2009). There are already existing languages and other means for the specification of bi-directional transformations between some representations, i.e. between models using certain

metamodels. The most work has been done for XML, for example, biXid (Kawanaka and Hosoya, 2006) based on *programming-by-relation* paradigm for relations over XML documents, or Multifocal (Pacheco and Cunha, 2012) for XML schemas using algebraic rewrite system. In the field of model-driven development, JTL (Cicchetti et al., 2010) is a language designed to support non-bijective transformations and change propagation. BOTL (Braun and Marschall, 2003) uses a different approach with a focus on the transformation of objects.

Although our requirements are very specific and implementation follows certain constraints, existing bi-directional transformations are a valuable source of information for our design. Eventually, our transformation will be rewritten into a model and code for executing transformation will be generated according to the NS approach.

3 OUR APPROACH

This section explains the goals and design of their fulfilment in order to achieve the bi-directional transformation between OWL and NS models. It also briefly describes the background and necessary details of NS metamodel.

3.1 Goals and Resources

First, we need to summarize the problem statement, set the goals, and describe the available resources usable for their fulfilment. The problem is allowing NS models to be transformable from its XML representation into OWL ontologies and back. Ideally, the bi-directional transformation should be lossless, i.e. all domain-specific parts will be transformed in both directions, if present in the model or the ontology, respectively. Moreover, in both NS model and OWL ontology, relations of NS models to NS metamodel are needed to provide the semantics (e.g. capture that as *Aircraft* in NS model is an instance of *Data Element* from NS metamodel, *Aircraft* as an class in ontology is instance of class *Date Element* from NS metamodel ontology).

1. Create a subset of NS metamodel that holds domain-specific and implementation-agnostic knowledge.
2. Allow transformation of NS model from its XML format to OWL with all domain-specific information provided and relations to NS metamodel.
3. Allow transformation of OWL with relations to NS metamodel to the XML format of NS model with all domain-specific information provided.

As for the resources, libraries to work efficiently with both OWL ontologies and NS metamodels using XML import and export are available. NS metamodel itself is supplied as a standard NS model in XML format (Mannaert et al., 2019). By transforming the domain-specific part of the metamodel to OWL, we will get all the terminology to that the models refer.

3.2 Domain-specific Parts

Before devising the transformation architecture, we need to select the domain-specific parts of the NS metamodel, i.e. the entities, relations, and attributes that carry the semantics of the domain and are not implementation-related details. The core selected parts are:

- Application – integrates together Components for some particular purpose and has its metadata such as name or description among implementation details.
- Component – is reusable encapsulation of a model that can be bound to multiple Applications. Among its metadata, it also has Component Dependencies and links to Data Elements.
- Data Element – represents an entity or a concept. In contrast to Task, Flow, and Service Elements who are tightly related to implementation, Data Elements carry the structural domain knowledge.
- Fields – are describing properties of a Data Element and are also the most complex part of domain-related metamodel section. There are Link Fields to form relationships between Data Elements and Value Fields to serve as traditional attributes. Optionally, Value Field can also be Calculated Field, which means that the value is computed from others (e.g. `age` using `birthdate`). This concept is similar to a derived attribute in UML.
- Data Projection – is a specified view on a Data Element, i.e. a subset of its original Fields using Reference Field together with possible other Calculated Fields.
- Options – for Components, Data Elements, and Fields, it is possible to specify options of certain pre-defined type with name-value pair. Although Options serve for implementation details, they can be used to store domain knowledge as well.

Another important part is Value Field Type, that serves to assign data types to value fields. These types can be defined on the level of Components, but there are also several pre-defined types with its counterparts

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Table 1: Concept transformation from NS to OWL.

NS Metamodel	OWL Ontology
Component	Ontology
Component Dependency	Imported Ontology
Data Element	Class
Value Field	Datatype Property
Link Field	Object Property
* (all)	Individual (of NS:*)

in code. We consider this currently as an implementation detail and use default type String. On the other hand, for Link Types, we need to distinguish different types of links (one-to-many, many-to-many, and directions) as it is crucial for domain modelling.

3.3 Architecture

The overall architecture of our proposed solution is depicted in Figure 1. For the direction towards the OWL ontology, after loading the NS Elements model of an Application, we process separate Components and their domain-specific parts which lie in the Data Elements subtree of the metamodel. As the tree of component is processed, individuals are added to the ontology (e.g. `Aircraft` with `rdf:type` of `NS:DataElement`). Those individuals represent the NS Elements model and will provide the transformation back to the XML.

In addition to transformation into individuals, some parts of the model can also have other special meaning in the OWL ontology. For example, `Aircraft` as `NS:DataElement` is also `OWL:Class` and its `callSign` as `NS:ValueField` is `OWL:DatatypeProperty`. It enables to have individuals on the ontological level of this domain-specific model. With this approach it is possible, for instance, to create an individual of `Aircraft` with `callSign` of literal `"BEL812"` as described in Figure 2.

The equivalent concepts for the mapping of NS Elements and OWL metamodels can be found in Table 1. As a Component forms an Ontology, the fields of a Component, such as a name, description, or version, should become annotations of the transformed Ontology. This level of transformation allows instantiation of the model and interrelating with other ontologies by imports or using ontology matching techniques. Produced OWL ontology can be exported to a file.

For the other direction, an Ontology is loaded from OWL a file, and backward re-construction of an NS Elements Component can start. Everything is retrieved only from individuals representing each NS Component. Individual of a Component has its value fields such as name, description, or version encoded using datatype properties. Component's link fields are encoded using object properties. One of the properties of a Component is a link field to Data Elements. These elements, and generally any other elements connected via link fields, are processed in the same way as a Component. Once a Component is completely transformed from the Ontology, it can be again exported to XML for further work in NS tools.

3.4 NS Elements Metamodel Transformation

With the described architecture and the homoiconicity of NS metamodel (Mannaert et al., 2019), some interesting consequences need to be pointed out. In the NS Elements metamodel, Component, Data Element, Field, Link Field, Data Option, Data Option Type, and others are modelled as Data Elements. According to Table 1, we expect the generated ontology to contain Data Element (from NS metamodel) both as an OWL Class `DataElement` and an OWL Individual of type OWL Class `DataElement`, i.e. itself. This principle of treating a Class as an Instance (of a Metaclass) is called *punning* (Hitzler et al., 2009).

NS models do not allow inheritance due to combinatorial effect concerns (Mannaert et al., 2016), therefore models are "flat", i.e. without any hierarchy. However, there are Taxonomy Data Elements in the metamodel that specify certain types of entities, e.g. Data Element Type specifies types of Data Elements. These taxonomy instances are not directly encoded as parts of the Component model. Instead, they are stored separately inside "prime-data" model as depicted in Figure 1.

To make transformations lossless in both directions, it is required to use the ontology of metamodel so individuals are bound by its semantics. To make this possible, we bootstrap by manually declaring a minimum subset of elements necessary to automatically read all identifiers of NS metamodel. After reading of those elements, we can discard the manual bootstrap, and rely only on continuous regeneration.

4 TRANSFORMATION IMPLEMENTATION

In this section, we describe the implementation of the bi-directional transformation between NS models and OWL ontologies. We focus on the key parts and techniques used to achieve extensible transformation according to the design that we discussed above.

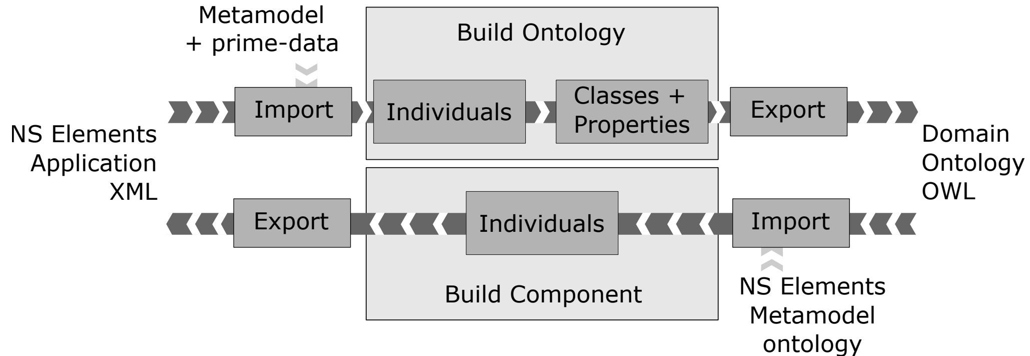

Figure 1: Architecture of NS-OWL bi-directional transformation.

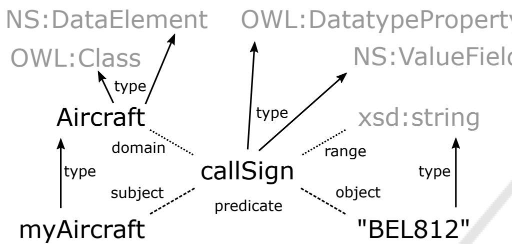

Figure 2: Example of class and related individual.

4.1 Technology Stack

Because of interoperability with the current NS technology stack, we use Java as the core technology. The NS library used is capable of XML import, export, and representation using Tree classes. In terms of NS, both XML and Tree representations are data projections, i.e. different views on a NS model. Tree projections are designed to be Plain Old Java Objects (POJOs) – therefore does not bound any special restrictions. With this library, the boxes *Import* and *Export* on the left side in Figure 1 are trivially covered. Import requires a path to a model folder or XML file and gives Application tree representation object (it links to other objects according to the metamodel). For the other direction, export requires application object and target folder (and optionally additional options) to write the model.

There are multiple options for working with OWL ontologies in Java. We identified a free and open-source framework Apache Jena (The Apache Software Foundation, 2019), which targets Semantic Web and Linked Data applications, as the most suitable for our purpose. Its multiple APIs are well-documented and capable of working with Individuals, all types of Properties, Ontologies, and Models as well as with representation using Graphs, Nodes, and Statements. An ontology model in Jena is an extension of the RDF model, providing extra capabilities for handling ontologies.

For both ontology and RDF models, there is a possibility of import and export using a wide variety of formats including Turtle, RDF/XML, JSON-LD, N-Quads, or RDF binary. This functionality covers the right-hand side export/import in Figure 1. Also, the framework helps significantly with building ontology models and querying the individuals.

4.2 URI Resolution

With the ability to import and export NS models and ontologies, it remains to implement the transformation itself between NS Tree projections and Jena's ontology model. Every single entity in an ontology model needs to have a unique identifier – typically random for anonymous/nameless nodes and Uniform Resource Identifier (URI) for other. For this purpose, our URI resolver creates a URI composed of a configured prefix, Component name, entity name (e.g. data element name), and in some cases also "parent" name(s). An example of such URI for data element Flight in Airlines component with https://normalizedsystems.org prefix is `https://normalizedsystems.org/airlines/Flight` (for further example we leave out the prefix). In the case of fields or similar parent-child related entities where unique naming in the NS model is limited to a parent's scope, we need a parent or eventually more parent names to be included. For instance, a number of the Flight from an example depicted in Figure 3 would have URI `airlines/Flight/flightNumber`. To avoid name clashes with instances, those are separated by hash symbol, e.g. `elements/DataElementType#Primary`.

For relating entities of an NS model to the metamodel, having all URIs of metamodel entities used in transformation is necessary. Although the metamodel is metacircular and transformable into ontology with its URIs, we need to define them manually as an NS

vocabulary (at least) for the initial transformation. A similar approach is used in Apache Jena framework for well-known vocabularies such as OWL, RDFS, DCTerms, SKOS, or FOAF (The Apache Software Foundation, 2019). Generation of this vocabulary is possible due to its simplicity – it is just naming of resources, e.g. resource name DataElement has defined constant URI `elements#DataElement`. It may tempt to use more flexible dynamic URI composing but inconsistencies must be avoided.

Another option would be to generate persistent URIs, e.g. using UUID – `airlines#<UUID>`. In this case, URI remains the same even when the underlying construct is renamed. On the other hand, there is always a need to query the name and any rename in the model must still be solved by three-way merging even when using UUIDs because they are not currently encoded in the NS metamodel. In case of a change in the metamodel, it is only a question of changing the URI resolver component to change the whole logic of URIs.

4.3 NS to OWL Transformation

With Apache Jena, the transformation in the direction from NS tree POJOs to an ontology model composed of Individuals, Classes, and Properties is very straightforward and systematic as shown in Source Code 1. We implemented it using a combined builder and visitor pattern. A transformation class encapsulates the addition of a single entity from the metamodel. Each transformation class is instantiated with some context information (such as uriResolver utility, or mutable target ontology model), and executed with the element that is currently being transformed. Then the method contains a "recipe" of its inclusion to the given ontology model:

1. Create an individual of a specific class from metamodel using URI from the NS vocabulary and with own URI generated by the URI resolver of the context.
2. Convert data and object properties using URIs from the NS vocabulary. For object properties, the targets are also added to the ontology model (propagates transformation through the tree) or looked up in case of backlinks.
3. Add specific transformation based on Table 1, e.g. an ontology class for a data element using still the same URI.
4. Return resulting individual for further processing, e.g. for object properties.

A somewhat different treatment is applied to Fields[2]. Different types of Fields (Value Field, Link Field) are implemented via composition inside the NS Field element. Therefore, the returned individual is of multiple types at once and has properties according to the types. Secondly, the transformation of link fields requires the addressing of data elements. There are numerous possible implementations, but to ensure the existence of target, we first process all data elements and then try to find the adequate using a lookup table. Finally, we also implemented a mechanism to find the inversed object properties, i.e. the matching link field from the target element.

This code may appear verbose, and one could believe that it may be hard to maintain and keep consistent. However, our goal is to generate the code in both directions based on the NS metamodel, and manually intervene only where necessary by adding custom code between `// anchor:custom-*` comments, that will be preserved between code regenerations.

4.4 OWL to NS Transformation

As explained in the design, the direction back from OWL ontology to the NS model is driven by individuals with NS metamodel compliant types and properties. The implementation is using the same patterns as the other direction explained above. A transformation class encapsulates construction of a single tree POJO. Each transformation object takes an ontology model, URI, and optionally context information as inputs of constructor and method call. The method contains the "recipe" of composing the POJO according to NS metamodel:

1. Find an individual based on a given URI and verify its type using the NS vocabulary.
2. Create a corresponding tree POJO.
3. Set all properties that are available in the ontology model. If missing a required one, throw an appropriate exception. In the case of links, transform the linked resource(s). All backlinks are ignored and left empty as those are optional in NS model – computed when necessary.
4. Return the tree POJO for further processing, e.g. for linking.

The transformation process starts on the level of application where it tries to query the model for application individual according to the type; then components are processes, then its data elements and de-

[2]NS Fields and relations between Data Elements is currently being redesigned, but the design have not been finalized before publishing of this paper.

Source Code 1: Example of Data Element to ontology model transformation in Java with Apache Jena.

```
public class DataElementTreeToOwl implements TreeToOwlTransformer<DataElementTree, ComponentTree>
↪  {

    public DataElementTreeToOwl(OwlModel owlModel, UriResolver uriResolver) {
        // ...
    }

    @Override
    public Individual transform(DataElementTree dataElementTree, URI parentURI) {
        URI dataElementURI = uriResolver.uriFor(parentURI, dataElementTree.getName());
        // anchor:custom-uri:start
        // anchor:custom-uri:end

        // Individual
        Individual individual = owlModel.createIndividual(dataElementURI, NS.DataElement);
        individual.addProperty(NS.DataElement_name, dataElementTree.getName());

        // DataElement/dataOptions link field
        DataOptionTreeToOwl dataOptionTreeToOwl = new DataOptionTreeToOwl(owlModel, uriResolver);
        ArrayList<RDFNode> dataOptionNodes = new ArrayList<>();
        for (DataOptionTree dataOption : dataElementTree.getDataOptions()) {
            dataOptionNodes.add(dataOptionTreeToOwl.transform(dataOption, dataElementURI));
        }
        individual.addProperty(NS.DataElement_dataOptions, owlModel.createList(dataOptionNodes));

        // Class
        OwlClass owlClass = ontModel.createClass(dataElementURI);
        owlClass.setSuperClass(OWL2.Thing);
        owlClass.addProperty(RDF.type, NS.DataElement);

        // anchor:custom-transformation:start
        // custom code that will be preserved by code generator
        // anchor:custom-transformation:end
        return individual;
    }
}
```

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

pendencies, etcetera. Currently, other additional information stored in the given ontology model is ignored. If there is, for example, an individual of the data element type that is not linked from any component, it will not be transformed into a data element POJO. Similarly, all additional information such as classes, properties, inversed properties, or extra annotations, are ignored as well. It produces the same NS model (more specifically, the same part that we identified as domain-specific) when transformed into OWL and then back again.

Although transformation from OWL to NS is lossy with respect to OWL-specific information, we are interested in keeping both representations. One of our aims is to use OWL to contain additional knowledge which has no execution semantics and thus is not directly applicable to NS code generation. Therefore this limitation is not applicable, as we can transform the NS-compatible ontology subset, whilst preserving additional OWL knowledge.

5 TRANSFORMATION EXAMPLE

In this section, we provide an example to illustrate the functionality of the transformation. The example explains the usefulness by showing the resulting application and use of existing tools for both NS models and OWL ontologies. The summary of the presented example is in Section 6.

5.1 Flight Booking Model

For this example, we will start with a simple model of *Flight Booking* domain. In terms of the NS model, it is a single component with six data elements and several value/link fields. The diagram of this model is shown in Figure 3. The model includes both many-to-many and one-to-many relationships. Also, various types of value fields are being used, including Date, Double, and Boolean. With all this, the example model covers the most commonly used structural constructs of the NS model.

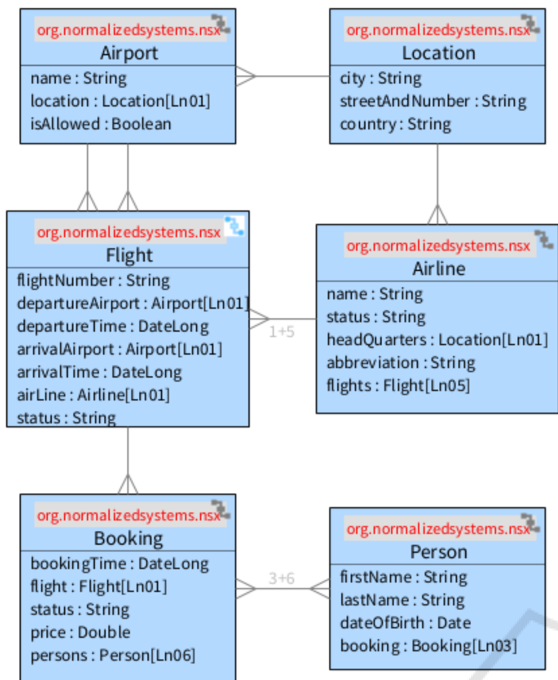

Figure 3: The example NS model for flight booking domain.

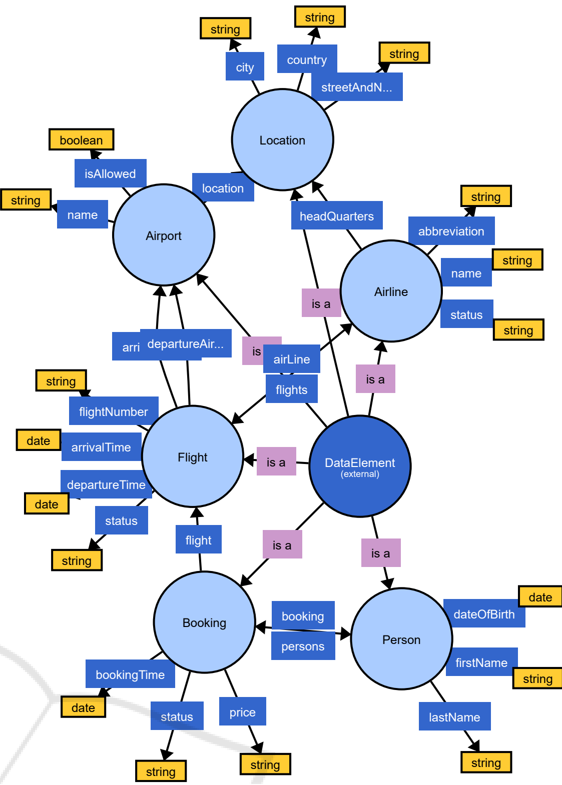

Figure 4: The generated OWL ontology for flight booking domain.

5.2 Transformed Ontology

Figure 4 shows the result of NS-to-OWL transformation in WebVOWL tool. All six data elements became classes (light blue circles) with is-a relationship to external *DataElement* from the metamodel ontology (dark blue circle). The link fields (i.e. relationships) are also present in the ontology as object properties with navigation between classes, e.g. *headQuarters* between *Airline* and *Location*. Both bidirectional relationships (*Person-Booking* and *Flight-Airline*) are present and connected as inverse object properties. Finally, matching datatypes (yellow boxes) were used for generated datatype properties based on value fields.

5.3 Adjustments in Ontology

With OWL ontology for our model, we can take advantage of tooling for analysing and editing ontologies. To demonstrate this, we can split *streetAndNumber* property to two using Protégé. First, we rename the existing property to *street*. Then, we create a new datatype property *number* with domain *Location* and range *xsd:int*. We additionally need to create a new individual of type *ValueField* to capture other necessary details according to the metamodel ontology such as if it is an info field. Similarly, we could edit classes and object properties to improve our model. Finally, we can transform back to NS model – it queries the individuals according to the metamodel ontology.

5.4 Information System from Ontology

Our example model can then serve for expanding an enterprise information system for flight booking management. NS tooling takes care of generating the application, injecting harvested code fragments, building, and then maintaining the application instance. The fragment of changed entity *Location* where *street* and *number* are separate arguments is shown in Figure 5. With the use of Normalized Systems, we can continually update and improve the model (or ontology) as we need and re-generate the application.

Data in the application instances will be migrated as long as the naming remains. For example, with our change of street, all records of *Location* would remain with its *city* and *country* but divided *street* and *number* would be empty as those are new attributes without any connection to the previous *streetAndNumber*. If we need some more complicated data migration, it has to be done using a separate OWL-OWL transformation, or in case of migration of a running system using e.g. ETL workflows.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Figure 5: Screenshot from generated NS application.

6 RESULTS DISCUSSION AND FUTURE WORK

The features of the designed and implemented transformation are discussed in this section. We focus mainly on its evolvability and outcomes. We also provide thoughts about future development.

6.1 Model Openness

OWL representation of NS models increases its openness and enables the use of a wide variety of existing tools for OWL analysis, design, transformation, documentation, and others. The entities from OWL based on an NS model can be easily related to other domain-specific ontology or to describe data. Annotation tools provide developers and analysts with a way of capturing additional knowledge that is not possible to achieve in current NS tools. A considerable advantage is that those annotations can be based on standard ontologies for metadata such as Dublin Core[3], but also a specific one can be created. Tools such as Widoco (Garijo, 2017), Protégé (Musen, 2015), or WebVOWL (Lohmann et al., 2014) allow different views on the model (as shown in Figure 6) and its documentation which should improve the efficiency of the development process. However, it must be verified in practice on real-world and large-scale NS applications.

6.2 NS Metamodel Ontology

The transformation tool has been used for various tests of NS models in order to verify the functionality of different transformation rules and mainly for the NS metacircular metamodel. For the metamodel, the resulting ontology contains 1493 individuals, 132 classes, 468 datatype properties, and 484 object properties. The generated documentation using Widoco tool is now experimentally used together with custom annotations added to the ontology for its analysis and serves as another developers documentation of the metamodel. Longer-term use of the documentation is needed prior to making conclusions about its advantages. With this ontology, it is possible to compose an NS model purely in OWL as individuals and then turn it into a domain-specific part of an NS application. For future development, metacircularity can be reflected in the transformation where static NS vocabulary can be checked using generated ontology or derived from it directly.

6.3 Ontology Evolvability

A resulting ontology of an NS model can be re-generated via XML representation when there is a change in an NS model and vice versa. The version number of an NS component is used for versioning its OWL counterpart. Although there is no mechanism of merging changes done in both representations, if annotations to the ontology are linking from a different file using URIs, the re-generation can still use them. Other verification procedures may be implemented in the future to check the consistency of models and even merge the changes when loading both an NS model and related OWL ontology using comparison and user interaction.

Concerning the development of NS metamodel itself, there might be a need for edits in the transformation tool. The library used for import, export, and tree objects can be then upgraded and recipes of transformations changed. The only important aspect is that transformation recipes need to be consistent in both directions. It is desirable to have a NS expander for OWL that would ensure the evolvability as OWL ontology is practically just a different view on the same model, i.e. data projection. Still, we need the other direction to be consistent.

6.4 Ontology Enrichment and Mapping

Using the ontology of an NS model, it is possible to enrich it and capture aspects that are not allowed with current NS metamodel. For example, it is possible to use inheritance between classes in ontology, or link other ontologies such as FOAF[4]. Also, ontology mapping methods can be used to find correspondences between an NS model represented as ontology and other domain ontologies. All of this could result in proposals of NS metamodel enhancements related to its ex-

[3] https://dublincore.org/specifications/

[4] http://xmlns.com/foaf/spec/

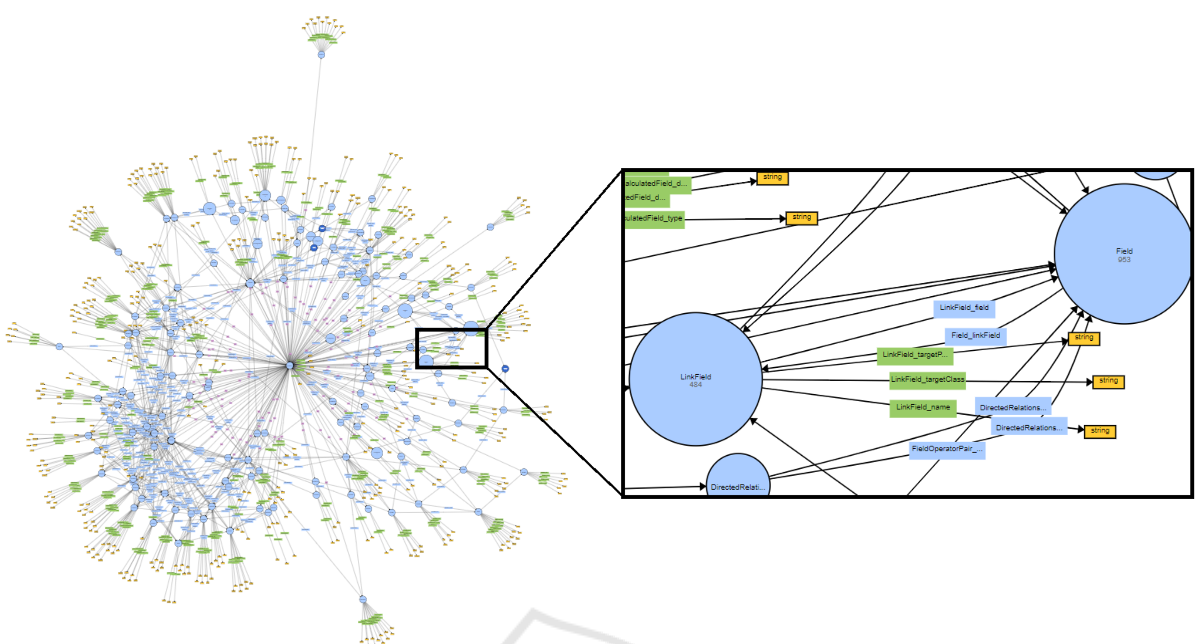

Figure 6: Example of WebVOWL visualization of generated ontology for NS metamodel.

pressiveness. Another possible future improvement is the transformation of other parts of NS models related to behaviour or even application settings.

7 CONCLUSIONS

In this paper, we first discussed the need for a way how to represent Normalized Systems domain models in OWL ontologies. We briefly explained the core terminology and methods related to the topic. As Normalized Systems provide a way of building evolvable information systems based on an NS model using expanders. As OWL ontologies allow flexible knowledge representation with various existing tools, we designed and implemented the bi-directional transformation between those model representations. It can be used to transform any domain-specific part of NS application to OWL and vice versa. Owing to the homoiconicity of the NS metamodel, we were able to transform it into an OWL ontology as well and use documentation tool together with other OWL techniques to create and maintain a flexible developers documentation. We demonstrated the use of bi-directional transformation on a simple example that concludes by generating an NS application. Finally, we also discussed other advantages brought by the transformation, its evolvability, and suggested possible future development. Nevertheless, the real usage with large-scale NS application will verify the value of the transformation and reveal opportunities for improvement.

ACKNOWLEDGEMENTS

This research was done thanks to the collaboration between Czech Technical University in Prague, University of Antwerp, and NSX bvba. The research was also supported by the grant of Czech Technical University in Prague No. SGS17/211/OHK3/3T/18.

REFERENCES

Barcelos, P. P. F., dos Santos, V. A., Silva, F. B., Monteiro, M. E., and Garcia, A. S. (2013). An Automated Transformation from OntoUML to OWL and SWRL. *Ontobras*, 1041:130–141.

Bhatia, M., Kumar, A., and Beniwal, R. (2016). Ontologies for Software Engineering: Past, Present and Future. *Indian Journal of Science and Technology*, 9(9):1–16.

Braun, P. and Marschall, F. (2003). Botl the bidirectional object oriented transformation language.

Cicchetti, A., Di Ruscio, D., Eramo, R., and Pierantonio, A. (2010). Jtl: a bidirectional and change propagating transformation language. In *International Conference on Software Language Engineering*, pages 183–202. Springer.

Czarnecki, K., Foster, J. N., Hu, Z., Lämmel, R., Schürr, A., and Terwilliger, J. F. (2009). Bidirectional transformations: A cross-discipline perspective. In *International*

Conference on Theory and Practice of Model Transformations, pages 260–283. Springer.

De Bruyn, P. (2011). Towards Designing Enterprises for Evolvability Based on Fundamental Engineering Concepts. In *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*, pages 11–20. Springer.

Eclipse Foundation (2019). rdf4j.

Embley, D. W. and Thalheim, B. (2012). *Handbook of Conceptual Modeling: Theory, Practice, and Research Challenges*. Springer.

Garijo, D. (2017). WIDOCO: A Wizard for Documenting Ontologies. In *International Semantic Web Conference*, pages 94–102. Springer.

Gasevic, D., Djuric, D., Devedzic, V., and Damjanovi, V. (2004). Converting uml to owl ontologies. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 488–489. ACM.

Guizzardi, G. (2005). Ontological Foundations for Structural Conceptual Models.

Hitzler, P., Gangemi, A., and Janowicz, K. (2016). *Ontology Engineering with Ontology Design Patterns: Foundations and Applications*, volume 25. IOS Press.

Hitzler, P., Krötzsch, M., Parsia, B., Patel-Schneider, P. F., Rudolph, S., et al. (2009). OWL 2 Web Ontology Language Primer. *W3C recommendation*, 27(1):123.

Huysmans, P. and Verelst, J. (2013). Towards an Engineering-Based Research Approach for Enterprise Architecture: Lessons Learned from Normalized Systems Theory. In *International Conference on Advanced Information Systems Engineering*, pages 58–72. Springer.

Isotani, S., Bittencourt, I. I., Barbosa, E. F., Dermeval, D., and Paiva, R. O. A. (2015). Ontology Driven Software Engineering: A Review of Challenges and Opportunities. *IEEE Latin America Transactions*, 13(3):863–869.

Kawanaka, S. and Hosoya, H. (2006). bixid: a bidirectional transformation language for xml. *ACM SIGPLAN Notices*, 41(9):201–214.

Křemen, P. and Kouba, Z. (2011). Ontology-Driven Information System Design. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(3):334–344.

Ledvinka, M., Kostov, B., and Křemen, P. (2016). JOPA: Efficient Ontology-Based Information System Design. In *European Semantic Web Conference*, pages 156–160. Springer.

Lohmann, S., Link, V., Marbach, E., and Negru, S. (2014). WebVOWL: Web-based visualization of ontologies. In *International Conference on Knowledge Engineering and Knowledge Management*, pages 154–158. Springer.

Mannaert, H., De Cock, K., and Uhnak, P. (2019). On the Realization of Meta-Circular Code Generation: The Case of the Normalized Systems Expanders. In *ICSEA 2019, The Fourteenth International Conference on Software Engineering Advances*. IARIA.

Mannaert, H., Verelst, J., and De Bruyn, P. (2016). *Normalized Systems Theory: From Foundations for Evolvable Software Toward a General Theory for Evolvable Design*. Koppa, Kermt (Belgium).

Musen, M. A. (2015). The Protégé Project: A Look Back and a Look Forward. *AI Matters*, 1(4):4–12.

NSX bvba (2019). NS Foundation.

Oorts, G., Huysmans, P., De Bruyn, P., Mannaert, H., Verelst, J., and Oost, A. (2014). Building Evolvable Software Using Normalized Systems Theory: A Case Study. In *2014 47th Hawaii International Conference on System Sciences*, pages 4760–4769. IEEE.

Pacheco, H. and Cunha, A. (2012). Multifocal: A strategic bidirectional transformation language for xml schemas. In *International Conference on Theory and Practice of Model Transformations*, pages 89–104. Springer.

RDFLib Team (2019). RDFLib.

Sadowska, M. and Huzar, Z. (2019). Representation of UML Class Diagrams in OWL 2 on the Background of Domain Ontologies. *e-Informatica Software Engineering Journal*, 13(1):63–103.

Telnarova, Z. (2018). Transformation of Extended Entity Relationship Model into Ontology. In *Asian Conference on Intelligent Information and Database Systems*, pages 256–264. Springer.

The Apache Software Foundation (2019). Apache Jena: A free and open source Java framework for building Semantic Web and Linked Data applications.

Tolvanen, J.-P. and Kelly, S. (2016). Model-Driven Development Challenges and Solutions: Experiences with Domain-Specific Modelling in Industry. In *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 711–719. IEEE.

Verdonck, M., Gailly, F., de Cesare, S., and Poels, G. (2015). Ontology-Driven Conceptual Modeling: A Systematic Literature Mapping and Review. *Applied Ontology*, 10(3-4):197–227.

Zedlitz, J., Jörke, J., and Luttenberger, N. (2011). From UML to OWL 2. In *Knowledge Technology Week*, pages 154–163. Springer.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS