Computing and Informatics, Vol. 34, 2015, 305–336

PATOMAT – VERSATILE FRAMEWORK FOR PATTERN-BASED ONTOLOGY TRANSFORMATION

Ondřej Zamazal, Vojtěch Svátek

University of Economics, Prague
W. Churchilla 4, 130 67 Prague 3, Czech Republic
e-mail: {ondrej.zamazal, svatek}@vse.cz

Abstract. The purpose of the *PatOMat* transformation framework is to bridge between different modeling styles of web ontologies. We provide a formal model of pattern-based ontology transformation, explain its implementation in *PatOMat*, and manifest the flexibility of the framework on diverse use cases.

Keywords: Ontology engineering, semantic web, ontology transformation, ontology matching, ontology import, ontology profiling, naming refactoring

Mathematics Subject Classification 2010: 68N01, 68T0, 68T3

1 INTRODUCTION

Ontologies have recently become an important part of many information systems and applications, e.g. [7]. Ontological engineering [6] provides methods for covering such an ontology lifecycle, including, aside from ontology authoring proper, other activities such as the reuse of an existing ontology or *pattern-based transformation* of an ontology, as main subject of this paper. The most prominent ontology language nowadays is the Web Ontology Language (OWL) [33]. It contains numerous language constructs, which allows the designers to express the same conceptualization in different ways. This is convenient for the designers but can negatively impact the applications working with such ontologies. In particular, an application might not work at its best performance level, or, it can even crash if the modeling style of the

ontology is different from the anticipated one. To illustrate the possible impact of modeling style, let us consider the following semantic web scenarios:

- Ontologies with different styles are hard to *match* [3] or *import* to one another.
- The modeling style of the ontology has impact on the usability and performance of tools, e.g., *reasoners*, that do not cope with certain language constructs.

As a straightforward example of style heterogeneity we present different formal conceptualizations, in the Manchester syntax of OWL,[1] of accepting or rejecting papers, in ontologies of 'conference organization'.[2] In the first model, the apparatus for distinguishing between accepted and rejected papers can be expressed simply as disjoint *classes* to which the paper can be assigned:

AcceptedPaper SubClassOf: Paper.
RejectedPaper SubClassOf: Paper.
AcceptedPaper DisjointWith: RejectedPaper.

In another ontology the same (or similar) distinction can be captured less directly, using disjoint *object properties*:

accepts Domain: PCChair. accepts Range: Paper.
rejects Domain: PCChair. rejects Range: Paper.
accepts DisjointWith: rejects.

Another option is to express the distinction via an *enumeration* of 'decision' individuals to which the paper can be linked:

hasPCChairDecision Domain: Paper.
hasPCChairDecision Range: (EquivalentTo {acceptance, rejection}).
hasPCChairDecision Characteristics: FunctionalProperty.

As the final option we will introduce the use of disjoint classes to which the (currently unspecified) decisions may belong:

hasPCChairDecision Domain: Paper
hasPCChairDecision Range: Decision.
Acceptance SubClassOf: Decision.
Rejection SubClassOf: Decision.
Acceptance DisjointWith: Rejection.

Since the purpose of the ontology fragment is to allow *separating* of objects of a certain kind (papers) into distinct classes (accepted vs. rejected), we can formally describe such a class via a concept expression in description logic. Table 1 lists the concept expressions for all four modeling options; the last column displays the symbol responsible for making a desired distinction.

[1] `http://www.w3.org/TR/owl2-manchester-syntax/`

[2] A collection of such ontologies has been used in the OAEI ontology matching contest, see `http://oaei.ontologymatching.org/2014/conference/`.

Option	Concept expression	Distinction via
1	C	C
2	$\exists R.\top$	R
3	$\exists R.\{i\}$	i
4	$\exists R.C$	C

Table 1. Concept expressions for classifying 'papers' according to all four modeling options. The C symbol represents a class, R a property and i an individual.

Such modeling choices can possibly be captured at the level of *ontology design patterns*: reusable solutions to a recurrent modeling problems [5], often collected in catalogs.[3] Importantly, each operation of transforming an ontology from one modeling style (defined via the pattern structure) to another can also be captured by a pattern, which we denote as a *transformation pattern*. Figure 1 schematically depicts such an operation: according to a transformation pattern, a source ontology (here rather just a fragment of it) is transformed to an unforeseen target ontology (fragment); the source fragment corresponds to the fourth modeling choice and the target fragment to the second (less complex) modeling choice from the motivating example. The transformation thus has to consider two (occurrences of) ontology patterns: one in the *source* ontology and one in the *target* ontology; the two patterns plus the link between them is what we call *transformation pattern*.

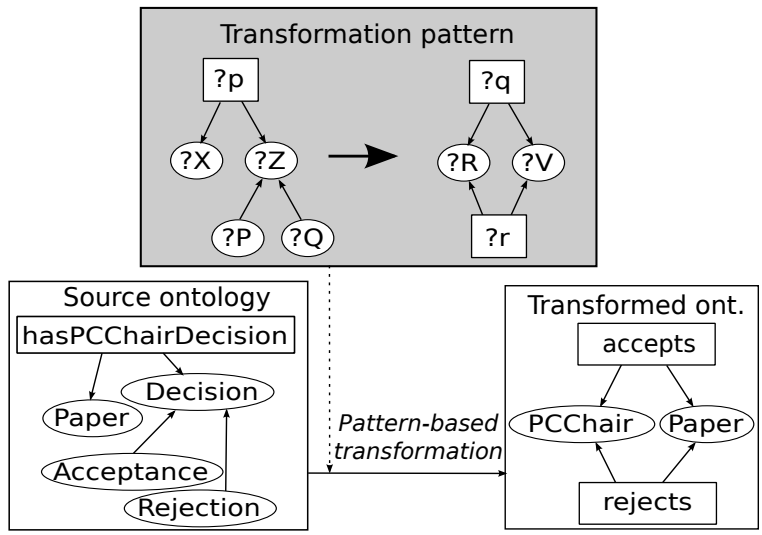


Figure 1. Example of pattern-based ontology transformation

[3] For instance: `http://ontologydesignpatterns.org/`.

In this paper we first describe the formal model of ontology transformation underlying our *PatOMat* transformation framework (Section 2). Next, we present the overall implementation employed (Section 3) along with the user perspective. Further, in Section 4, we provide a thorough description of the use cases tackled by the framework so far. We discuss their motivation, provide their distinguishing characteristics and demonstrate them on examples, as well as elaborate on some of their implementation-related specifics. Section 6 provides an overview of related work. Section 7 wraps up the paper and envisages further directions of this research.

2 FORMAL MODEL OF PATTERN-BASED TRANSFORMATION

Obviously, the basic building block in the approach is the notion of the ontology pattern.

Definition 1 (Ontology Pattern)**.** An *ontology pattern* is a triple $\langle \texttt{E}, \texttt{Ax}, \texttt{NDP}^* \rangle$, such that `E` is a non-empty set of *entity declarations*, `Ax` a (possibly empty) set of *axioms*, and `NDP`* a (possibly empty) set of *naming detection patterns*.

Entity declarations[4] introduce entities, i.e. classes, properties and individuals, either as concrete entities or, more often, at the level of placeholders (we distinguish them with a starting question mark further on). *Axioms* are statements about entities included in the transformation. Finally, the *naming detection patterns* (NDP) capture the naming aspect of the ontology pattern relevant for its detection.

Definition 2. A *naming detection pattern* is a set of passive naming operations, $\texttt{NDP} = \{no_1, no_2, \ldots, no_n\}$. All no_i have as their operands entities from the ontology pattern to which `NDP` belongs, and constants.

For instance, the core pattern for the fourth option from Section 1 could be:

$$
\begin{aligned}
E &= \{\mathit{ObjectProperty} : ?p.\ \mathit{Class} : ?P, ?Q, ?X, ?Z\} \\
Ax &= \{?p\ \mathit{Domain} : ?X.\ ?p\ \mathit{Range} : ?Z.\ ?P\ \mathit{SubClassOf} : ?Z.\ ?Q\mathit{SubClassOf} : ?Z. \\
&\quad\ ?P\ \mathit{DisjointWith} : ?Q\}.
\end{aligned}
$$

NDPs could then restrict the detection of *?P* and *?Q* to the existence of verb forms for their head nouns (e.g., 'accept' for 'acceptance' and 'reject' for 'Rejection').

Next, the relationship between the source and target OP is defined by what we call 'pattern transformation'.

Definition 3 (Pattern Transformation)**.** Let `OP1` and `OP2` be ontology patterns. A *pattern transformation* from `OP1` (source pattern) to `OP2` (target pattern) is a tuple $\langle \texttt{LI}, \texttt{NTP}^* \rangle$, in which `LI` is a non-empty set of *transformation links*, and `NTP`* is a (possibly empty) set of *naming transformation patterns*. Every *transformation*

[4] Corresponding to axioms with the *rdf:type* property.

link `l` ∈ `LI` is a triple ⟨`e`, `e'`, `R`⟩ where `e` ∈ `OP1`, `e'` ∈ `OP2`, and `R` is either a *logical equivalence* relationship between *homogeneous* entities or an *extralogical relationship* between *heterogeneous* entities.

As *logical equivalence relationships* (≡) we consider standard OWL constructs declaring the equivalence/identity of two 'logical entities' of same type: classes, properties or individuals. An *extralogical relationship* (≅) can be

1. a relationship of type *eqAnn*, valid between a 'logical' entity and an annotation entity,[5] or,
2. a 'heterogeneous' relationships *eqHet*, valid between two 'logical entities' of different type.

Extralogical relationships correspond to modeling the same situation using different entity types, as we saw in the motivating example. For instance, for the transformation between the first and fourth modeling options as in Figure 1:

$$\begin{aligned} ?X &\equiv ?V. \\ ?P &\cong ?q \\ ?Q &\cong ?r. \end{aligned}$$

Naming transformation patterns (NTP) provide a way of giving names to newly created entities in OP2 with regard to entities in OP1.

Definition 4. A *naming transformation pattern* is a set of pairs consisting of an entity and a naming operation, $\texttt{NTP} = \{(e_1, no_1), (e_2, no_2), \ldots, (e_n, no_n)\}$. All no_i have as their operands entities from the source OP of the PT to which `NTP` belongs, and constants. All e_i are from the target OP of the PT to which `NTP` belongs.

In our running example we should provide the NTP for *?R* (the new *PCChair* entity, only present implicitly in the source ontology), *?q* and *?r*.

The construction of a proper transformation pattern is now straightforward:

Definition 5 (Transformation Pattern)**.** *Transformation Pattern* is a triple ⟨`OP1`, `PT`, `OP2`⟩ such that `OP1`, `OP2` are ontology patterns and `PT` is a pattern transformation from `OP1` to `OP2`.

3 TRANSFORMATION FRAMEWORK IMPLEMENTATION

In this section we describe the *PatOMat* framework designed over the previously described formal model.

[5] OWL 2 annotations may contain various interlinked entities in a separate 'space'; these are, however, excluded from the logical interpretation of the ontology.

3.1 Naming Patterns within Transformation Patterns

Naming operations can be divided into *passive* ones, applied for checking purposes, and *active* ones, for naming a new entity.[6] While both can be plugged into NTPs,[7] only passive operations can be used in NDPs. Basically, an NDP concerns the detection of an ontology pattern, while an NTP concerns (re)naming an entity from this ontology pattern.

As an example of an NDP with two operations we can take the following:[8]

{comparison(head_noun(?OP1_P), head_noun(?OP1_A), not_hypernym)}

This NDP checks whether there is (no) hypernym relationship between two terms. In this case, those two terms are head nouns of entities (?OP1_P and ?OP1_A). For instance, considering 'ScientificWork' as ?A and 'Paper' as ?P then the pattern succeeds because 'Paper' is not a hypernym of 'Work' (applying WordNet [13]).

The following is an example of NTP with one compound operation:

{(?G, make_passive_verb(?C) + head_noun(?A))}

It names ?G by composing the passive tense of a verb derived from the head noun of ?C with the head noun of ?A. For instance, if ?A is 'PresentedPaper' (with 'Paper' as head noun) and ?C is 'Rejection' (with 'Rejected' as derived verb passive), the name of ?G in the transformed ontology will be 'RejectedPaper'.

Naming patterns can be generally defined on any lexical aspect of an ontology: URI of entities, their fragments, labels, comments etc. By default, we consider naming patterns applied over fragments of URIs, otherwise it is stated in an attribute of the *ndp* or *ntp* element of the XML serialisation, e.g., *target="label"*.

A small collection of *implemented naming operations* was gathered based on the requirements of use cases. They include (we list together both a passive and an active variant where relevant):

- *delimiter* detection and change (e.g., underscore or camel-case)
- detection and derivation of the *verb form* of a noun (using WordNet and the Stanford part-of-speech tagger [30])
- detection of *hypernym* relationship based on WordNet
- detection of *head noun* or its complement, for a noun phrase, and of a *head term* for a verb phrase, typically in a property name (only passive operation)
- construction of the *passive form* of a verb.

[6] A passive naming operation often has its active variant.

[7] Transformation patterns are serialised according to an XML schema: `http://nb.vse.cz/~svabo/patomat/tp/tp-schema.xsd`.

[8] This example is borrowed from the naming use case described in Section 4.1, and its transformation pattern is available at: `http://nb.vse.cz/~svabo/patomat/tp/np/tp_np1c.xml`.

3.2 Annotations within Transformation Patterns

Ontology annotations provide the additional information about the concepts captured in the ontology and about an ontology in general. In order to transform one modeling style into another it often happens that some parts of the transformed fragment must be removed because there is no means for its representation in the logical space of the ontology. However, it is possible to swap the removed parts of the ontology into an *annotation space*, possibly even allowing for reverse transformation in the future.

Let us assume we want to remove a *union construct* from an ontology, i.e., the following axiom is in the source ontology pattern:

$$?OP1\_A equivalentTo(unionOf(\ldots ?OP1\_B))$$

We can swap this disjunction to the annotation space by 1) reification of the to-be-removed axiom using an anonymous individual, $?OP2\_ai1$, and 2) annotating it using the annotation relations *annotation:relation* and *annotation:axiom*.

Considering that $?OP1\_A = ?OP2\_A$, this can be represented as follows:[9]

```
?OP2_A annotation:removed_axiom ?OP2_ai1
?OP2_ai1 annotation:relation ?OP2_n
?OP2_ai1 annotation:axiom ?OP2_m
```

On demand, the framework can also log all removed and added axioms automatically using ontology annotations. In the previous case it would result in:

```
<annotation:removed_axiom>
   A equivalentTo B or C or D
</annotation:removed_axiom>
```

While the first option of using annotations is under control of the transformation pattern designer, the second option is effectuated automatically by the transformation framework (if it is switched on). Either option is sufficient for informing the human user what happened during the transformation.

3.3 Entity and Axiom Transformation Operations

A transformation pattern, $\langle\langle \mathtt{E}_1, \mathtt{Ax}_1, \mathtt{NDP^*}_1\rangle, \langle \mathtt{LI}, \mathtt{NTP^*}\rangle, \langle \mathtt{E}_2, \mathtt{Ax}_2\emptyset\rangle\rangle$, captures relationships between entities and axioms on a higher level of generalisation. In practice, the transformation pattern is converted to *transformation instructions* applicable to a particular ontology. Basically, the instructions consist of entity and axiom transformation operations.

Axiom transformation operations, operating at the level of axioms, can be divided into two classes:

[9] This example is taken from an existing transformation pattern available at `http://nb.vse.cz/~svabo/patomat/tp/lr/tp_unionELbannotations.xml`.

- *axiom removal*, where axiom a is removed from the ontology and
- *axiom addition*, where axiom a is added to the ontology.

Furthermore, we distinguish between two kinds of axioms: logic-based and annotation-based.

Entity transformation operations, operating at the level of entities, can be divided into three classes:

- *entity addition*, where the new entity e is named according to an `NTP`,
- *entity removal*,
- *entity renaming*, where, again, entity e is renamed according to an `NTP`.

We can notice that there are two kinds of *removal* operations: one for an axiom and one for an entity. The removal is derived from a user-designed transformation pattern; however, since a removal can have far-reaching effects, there are three different strategies for coping with them. They differ in the possibility of removing entities and/or axioms:

- *Conservative strategy* does not allow to remove anything. Obviously this is the safest strategy, avoiding undesirable changes in an ontology.
- *Progressive strategy* (used by default) does not allow to remove entities; however, it is possible to remove axioms.
- *Radical strategy* allows to remove both entities and axioms.

We can illustrate the sensitive aspect of the removal operations using an example.[10] Let us assume that application of the transformation pattern leads to the operation of removing class *Author*, but class *Author* is also used in other fragments of the ontology; for instance, the domain of property *writes* is class *Author*. If the removal operation is applied to the class *Author* and the radical strategy is selected, the domain axiom of the property *writes* will be removed. However, if we choose the conservative or progressive strategy then this domain axiom with the *Author* class will be retained because the *Author* class is not removed.

We call such axioms *additional axioms*. They are external to the source pattern but they refer to entities from the pattern. They could not be directly considered at the time of transformation pattern design, and it is usually unfeasible to capture all possible situations in the transformation pattern. Therefore, the radical transformation strategy has additional modalities:

radical-keep modality of radical strategy: entities are kept and additional axioms are annotated using *annotation:remove_warning_by* to link the affected axiom and the transformation pattern, and *annotation:remove_warning_for* to link the

[10] This example is based on application of the transformation pattern `http://nb.vse.cz/~svabo/patomat/tp/tp_agentRoleV4a2.xml` on the *confOf* ontology, `http://oaei.ontologymatching.org/2014/conference/data/confOf.owl`.

affected axiom and the entity which would have been removed, e.g., class *Author* in our example above.

radical-replace modality of radical strategy: entities are replaced either *equivalently*, e.g., *Author* would be replaced by the equivalent concept expression: *Person and (writes some Paper)*, or *approximately*, e.g., if the class *Author* is removed as the domain of the *writes* property, it will be replaced by the superclass of *Author*, which is the class *Person*. In a more general way, the user could provide a set of transformation patterns which could be applied if there are some affected axioms in the ontology. Let us note that in some cases this modality is not applicable.

radical-remove modality of radical strategy: the entities are simply removed.

Some expected additional axioms can be directly included into a transformation pattern. It is possible to mark some axioms as *optional*, which decomposes the transformation patterns into a *mandatory part* (containing the source and target ontology patterns proper and their pattern transformation) and an *optional part* (containing the additional axioms and their pattern transformation). Mandatory axioms from the source ontology are used for generation of SPARQL queries, see Section 3.5.

Additionally, when we remove information from the logical content of the ontology, it is still possible to swap it into the annotations as explained in Section 3.2. While we already consider annotations as part of a transformation/ontology pattern, implementation of a concrete reverse transformation support is left to be addressed in a future work.

3.4 Derivation of Transformation Operations from Transformation Patterns

In the following we specify several rules how *entity transformation operations* are derivable from a transformation pattern:

1. If there is an equivalence relationship between ?A $\in \mathtt{E}_1$ and ?B $\in \mathtt{E}_2$ (where ?A and ?B depict placeholders, $\mathtt{E}_1$ and $\mathtt{E}_2$ depict entities from OP1 and OP2 resp.) then the instance of ?B will be named according to NTP for ?B.

   In other words, if it is stated, in pattern transformation part, that placeholders ?A and ?B are equivalent then the entity mapped to ?A is preserved but renamed according to NTP for ?B.

2. If there is an extralogical link *eqAnn* or *eqHet* between $?A \in \mathtt{E}_1$ and $?B \in \mathtt{E}_2$, then the instance of ?B will be named according to NTP for ?B, typed according to the kind of placeholder of ?B, and *in the case of radical strategy* ?A will be removed.

   In other words, if it is stated, in pattern transformation part, that placeholders of different kinds ?A and ?B are equivalent then the entity mapped to ?A is

copied to ?B and named according to NTP for ?B. Moreover, because ?A and ?B are of different kinds (e.g. class and object property), the entity in ?B must be typed according to type of ?B. Finally, on demand by a user (i.e. in the case of radical strategy selection) entity in ?A is removed.

3. All entities from $\mathtt{E}_2$ that are not linked to an entity from $\mathtt{E}_1$ will be added.

   In other words, entities stemming from OP2 that do not serve as a replacement of original entities are added.

4. *In the case of radical strategy*, entities from $\mathtt{E}_1$ that are not linked to any entity from $\mathtt{E}_2$ will be removed.

   In other words, original entities, related to OP1, that should not be replaced, according to pattern transformation part, are removed on demand by a user (in the case of radical strategy selection) otherwise they are preserved.

As a result of Rule 2 application for conservative or progressive strategy there are original and new entities in one ontology. Thus, it can be useful to mark their relationship by an annotation property instance. We cannot use an annotation property in the case of radical strategy since the original entity is removed. But, in any strategy we can trace roots of new entity from (heterogeneous) transformation link between the original entity and the new one at the level of a transformation pattern.

For instance, in the transformation pattern for reducing a (reified) *n-ary relation* to binary[11] there is an extralogical link between class ?B and property ?q. According to Rule 2 it would lead to up to two operations:

- operation of adding object property ?q named according to NTP for ?B, e.g., make_passive_verb(?B),
- (*If radical:*) operation of removing instance of ?B.

For instance, in the case of ?B = ReviewSubmission, it makes a new object property 'submitted' by verb derivation from the head noun of ?B. Assuming the conservative strategy (hence not removing ?B), an annotation property instance would relate the old and the new entities.

The *renaming operation* works on the naming aspect (entity URI, *rdfs:label*, etc.) of an entity referred to by a placeholder. By default, we process the URI fragment of an entity. Changing the URI fragment is however problematic, since it is, in principle, equivalent to creating a new entity. We can solve this problem by adhering to ontology-versioning principles: to retain the original entity (with its original URI) in the ontology, to annotate it as *deprecated*, and to add an equivalence axiom between these two entities (i.e., between the original and new URI).

For deriving *axiom transformation operations* from a transformation pattern, there are only two simple rules:

[11] `http://nb.vse.cz/~svabo/patomat/tp/tp_1-n-ary-relation.xml`

1. remove all axioms within OP1 *in the case of progressive or radical strategy*,
2. add all axioms within OP2.

While the removal of axioms is pretty straightforward, as it works on the original entities, the addition of axioms must be done in connection with entity operations, since it only works on just-added or renamed entities.

For instance, in ontology pattern 1 of a transformation pattern dealing with a restriction class[12] there is a template axiom '?A equivalentTo (?p value ?a)'. It can match, e.g., the axiom 'PresentedPaper equivalentTo (hasStatus value Acceptance)', which can be swapped with annotations in ontology pattern 2 as follows: 'AcceptedPaper annotation:discr_property 'hasStatus", 'AcceptedPaper annotation:value 'Acceptance". As a result of that rule, there will be an instruction to remove (in the case of progressive or radical strategy) the original axiom and to add two new axioms. The binding of placeholders and entity operations must be considered beforehand.

3.5 Ontology Transformation Workflow

The whole transformation is divided into three steps, which correspond to three services as depicted in Figure 2. Rectangle-shaped boxes represent the three basic services, while ellipse-shaped boxes represent input/output data. This functionality is available as:

- RESTful services providing detection, instruction generation and transformation services through simple HTTP-based access. In the case of detection or instruction generation they return results in XML. They can easily be used in any language, see [31]. They are also accessible via the web interface [32] and as
- *Java library* providing the complete API for all transformation steps. This library can be plugged into an application.

The *OntologyPatternDetection* service outputs the binding of entity placeholders in XML. It takes the transformation pattern (containing the source and target patterns) and a particular original ontology on input. The service automatically generates a *SPARQL query*[13] internally, based on the mandatory part of the ontology pattern (the placeholders becoming SPARQL variables) and executes it. The structural/logical aspect is captured in the query structure, and the possible *naming constraint* is specifically dealt with based on its description within the source pattern. The implementation uses the Jena framework.[14] In more detail, there are two ways to start detection. Using API of transformation framework in Java library one can either first directly execute the generated SPARQL and then the naming aspect is considered, or the other way around. The best option depends on the given case. If there is a very specific structure that should be detected then the SPARQL

[12] `http://nb.vse.cz/~svabo/patomat/tp/tp_ce-hasValue.xml`
[13] `http://www.w3.org/TR/rdf-sparql-query/`
[14] `http://jena.apache.org/`

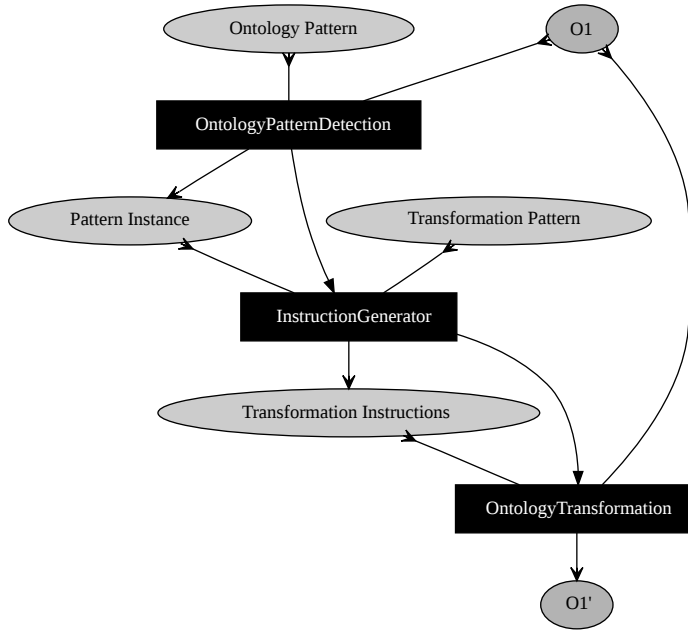


Figure 2. Ontology transformation workflow; application workflow is depicted using a line with a normal head and dataflow is depicted using a line with a V-shaped head

query can run first. However, if the source pattern structure is too generic, such as the plain subClassOf relationship, then the combinatorial complexity of the query is very high. On the other hand, if there is a naming detection pattern present in a transformation pattern, the naming aspect can be applied first and then the SPARQL query can already be limited by the naming detection results. For example, thanks to application of the naming detection pattern, exhaustive search for the Time-Indexed Participation pattern (in the importing use case, see Section 4.5) in an ontology (with recursion enabled), which originally took 16 minutes, was reduced to 40 seconds (which includes naming detection pattern processing, SPARQL query construction, its execution and final SPARQL query processing). In general, detection phase usually takes up to tens of seconds (e.g. all examples in use cases from Sections 4.2, 4.3, 4.4, 4.5). However, detection phase can take much more time (hundreds of seconds) if there is either too generic source pattern structure or too complex naming detection pattern (e.g. in use case from Section 4.1).

The *InstructionGenerator* service outputs particular transformation instructions, also in XML. It takes the particular binding of placeholders and the transformation pattern on input. Transformation instructions are generated according to the transformation pattern and the pattern instance. In general, transformation instructions phase takes up to hundreds of milliseconds.

The *OntologyTransformation* service outputs the transformed ontology. It takes the particular transformation instructions and the particular original ontology on input. Generally, ontology transformation phase takes tens of milliseconds.

The intermediate products, pattern instance and transformation instructions, are assumed to be inspected and possibly edited by the user. In particular, the user can choose which pattern instances (from automatic detection) should be used.

However, there is also an aggregative *one-step Ontology Transformation service* that takes the original ontology, transformation pattern and pattern instance on input and returns the transformed ontology at once.

For the moment, we do not specifically take into account the status of the transformed ontology within the semantic web. In some contexts it can be used *locally*, as in an ontology matching scenario, while in some others it can be exposed with a unique identifier, as a new *ontology version* pointing to its pre-cursor using the OWL 2 versioning mechanism.

4 OVERVIEW OF USE CASES

We already mentioned a few use cases in Section 1. In this section we will describe all five use cases that have been tackled by the framework so far: naming-based ontology repair, ontology simplification ('SKOSification'), ontology profiling, ontology matching, and content pattern import. We first describe three use cases that share the general shape of basic pattern-based transformation schema as depicted in Figure 1. Then, we describe two use cases that have some additional steps before and after the transformation step.

It should be noted that some of the use cases have been presented, to a certain degree, in isolated publications. Namely, a very early version of the 'matching' use case was described in [27]; an extended version of the 'profiling' use case was described in [25], and an extended version of the 'import' use case was described in [23]. However, the remaining use cases as well as the synoptic and systematic presentation, are the completely new contribution of the current paper.

4.1 Ontology Naming Repair Use Case

Ontology Naming. The first use case deals with the naming aspect of ontologies. In general, an ontology has three aspects by which a user can capture the required semantics of concepts. The first one is the *logical* aspect, which is often considered as the only representational means for capturing the meaning of concepts. The logical perspective is of the highest importance if we want to apply automatic reasoning and deduction methods. At the same time, from the logical perspective, the *naming* of concepts is not an important issue since the results of deduction would be the same regardless of the concept names. On the other hand, it has been recognised [28] that the logical aspect is not the only one since machines are not the only consumers of ontologies. There are also their designers and users for whom clear names (naming

aspect) of entities can mitigate the effort to comprehend the meaning of a concept in an ontology. It is even possible to apply machine analysis to naming in order to improve the presentation of ontologies, e.g., by avoiding redundant axioms [29]. This comprehensibility issue is also related to the third, *annotation*, aspect, since it is also highly recommended to provide annotations to concepts in order to properly convey the desired semantics of defined concepts.

Motivation for Ontology Naming Repair. An ontology designer can either consider correct naming from the very beginning or (s)he can initiate naming repair inspection when the ontology is otherwise finished. This naming inspection can be made manually using ontology editors or (semi-)automatically. In the latter case we call this *ontology naming repair*, which aims at helping us improve bad or awkward naming of concepts in an ontology. There are several naming conventions [21] and also warnings about bad naming practices [26]. In the following Section 4.1, one type of naming issue taken from [26] is exemplified.

Example. In this example we detect and repair the naming pattern called "non-matching child" from an ontology O_1. The pattern is connected with the hypothesis that the nature of an underlying entity should not change while subclassing, so a change of the head noun would indicate 'some problem'. The derivation associated with the detection of this pattern is thus an 'alerting one. Possible faults can be of two types: *fault in set-theoretical semantics*: e.g., a part-of relationship mistaken for subclass relationship (e.g., 'Car/Wheel) and *improper style of concept naming*, e.g., omission of the head noun in the child name (e.g. 'Paper/Accepted'); unlike the previous one, this situation often occurs even in ontologies created by relatively skilled designers.

The corresponding transformation pattern – in particular, its detection part – can be designed in different modalities. We can use simple head noun identity, which provides low precision. Or, we can include *thesaurus correspondence*, which improves the precision of detection. Namely, we can avoid alerting in situations when the head noun of the child name is a *hyponym* or *synonym* of the head noun of the parent name, e.g., 'Author' is a hyponym of 'Person'.

As far as the *structure* of the source ontology pattern[15] is concerned, it only contains the subsumption relationship, ?$OP1\_A$ subClassOf ?$OP1\_P$. The naming aspect consists in evaluation of the hypernym relationship between head nouns of the two classes in this relationship:

```
NDP = {comparison(not_hypernym,head_noun(?OP1_A), head_noun(?OP1_P))}
```

If a pair of classes is detected (e.g., *Paper* and *Accepted*), then ?$OP2\_A$ (e.g., *Accepted*) is renamed using a naming transformation pattern:

[15] Transformation patterns for this example are available at `http://nb.vse.cz/~svabo/patomat/tp/np/`.

```
NTP = {( ?OP2_A,  ?OP1_A + head_noun(?OP1_P))}
```

The ontology naming repair functionality has been integrated into the ORE (Ontology Repair and Enrichment) tool [9] developed at the University of Leipzig.

Discussion of Characteristics. The structure of transformation patterns can be quite complex. It is obvious that the naming aspect is of high importance in this use case. We assume that the detection is semi-automatic, since the user should decide which repair plan should eventually be performed. The annotation aspect is not so important, since this kind of transformation is rather one-way (without reverse transformation). As naming repair use case is mainly focused on the naming aspect, a change of logical semantics is usually not expected.

The characteristics of all use cases are summarised in Table 2.

4.2 Ontology Simplification Use Case

Simple Knowledge Organization System (SKOS). SKOS [12] is an RDFS vocabulary for expressing the structure of simple knowledge models – classifications and thesauri. Such models do not exhibit the axiomatic structure of OWL, yet they can model a hierarchical conceptual structure analogous to that of an OWL class taxonomy. Thanks to their simplicity, as well as the possibility to include additional features (such as the notion of 'near match' between concepts) which are incompatible with the classical set-theoretic view of OWL, they are often preferred by practitioners.

Motivation for Ontology Simplification. The simplification of ontologies and conversion of OWL to SKOS have recently been recognised as an important task, since most applications are tuned to such lightweight models. This need has already been reflected by several tools providing such a conversion, see Section 6.

Example. Let us consider an example of transformation[16] from the taxonomic structure of an ontology O_1 to a SKOS taxonomy. While we only need one axiom on the side of the source pattern, i.e., $?OP1\_A\ subClassOf\ ?OP1\_B$, the target pattern is more complex. First, instead of OWL classes in an OWL taxonomy, there are instances of $skos : Concept$ in the SKOS taxonomy:

```
?OP2_a types skos:Concept.
?OP2_b types skos:Concept
```

In order to express a SKOS taxonomy, the $skos : broader$ (or $skos : narrower$) property is used:

[16] The complete transformation pattern `tp_skos3.xml` is available at `http://nb.vse.cz/~svabo/patomat/tp/skos/`.

```
?OP2_a skos:broader ?OP2_b.
```

Furthermore, we can represent this SKOS taxonomy as part of one scheme using the *skos* : *inScheme* property, which has instances of *skos* : *ConceptScheme* in its range:

```
?OP2_scheme types skos:ConceptScheme.
?OP2_a skos:inScheme ?OP2_scheme.
```

Finally, we can add an *rdfs* : *label* annotation to instance ?*OP*2_*scheme* in order to make the scheme name more comprehensible:

```
?OP2_scheme rdfs:label ?label.
```

where ?*label* and ?*OP*2_*scheme* are newly named according to the transformation pattern part:

```
<ntp entity="?OP2_scheme">schema1</ntp>
<ntp entity="?label">skos-scheme</ntp>
```

Due to the fact that concepts in SKOS are modelled as instances of *skos* : *Concept* while they are at the same time classes in OWL, we need a heterogeneous link between the original class in OWL representation and the instance in SKOS representation. We do not use a naming detection pattern in this case; however, we can imagine that the transformation of an OWL taxonomy might be restricted to a certain part of the OWL taxonomy based on the naming aspect.

Discussion of Characteristics. Transformation patterns related to the SKOS use case are generally not very complex. As already discussed, the naming aspect could be useful if we wanted to restrict the transformation based on the naming aspect. However, this does not play an important role here as in the use case of the ontology naming repair. Using annotations is useful, since there are many cases in which we lose information. The detection should be rather automatic, however some user interaction could be considered. Finally, the OWL to SKOS transformation generally includes a change of logical semantics, as classes are meta-modelled at instance level.

We remind the reader once more that the characteristics of all five use cases are summarised in Table 2.

4.3 Ontology Profiling Use Case

Ontology Profiling. Existing tools operating on ontologies normally support a certain well-defined set of logical operators. In many cases this set of operators is not sufficient to completely capture the semantics of the OWL language. As a result, these tools cannot be used on certain ontologies or they provide incomplete reasoning results. In both cases, a transformation of the input ontology can improve the situation. In particular, the ontology can be transformed into a version that only uses the supported operators.

Motivation for Ontology Profiling. Language profiling outside the tools gives a user more flexibility because (s)he can design a transformation that is not directly hard-coded into the tool. Generally, language profiling can be driven either by a need for *replacing a specific language construct* or by an *ontology complexity requirement* of a particular tool. The former case contains a single transformation, while the latter case, *downgrading*, is comprised of more than one transformation pattern in order to achieve the required complexity level. In the next section we will illustrate ontology profiling on a specific construct replacement example, while ontology downgrading is discussed in Section 4.3.

Example. In order to demonstrate the use case of ontology profiling, let us consider that a particular tool (mostly the reasoner) has problems to tackle nominals. The simplest way could be the removal of nominals. However, this would cause a large change in conceptualisation. In the following example we will show how nominals can be replaced rather than removed. Let us assume that we have nominals describing continents and the *Continent* class defined as 'one of' those nominals (implicitly assuming their mutual difference):

```
Continent equivalentTo {Africa, America, Antarctica,
                        Asia, Australia, Europe}.
```

Let us further assume that we have the *AfricanRedSlip* class,[17] defined via the *hasContinentOfOrigin* property:

```
AfricanRedSlip subClassOf Ware.
AfricanRedSlip subClassOf (hasContinentOfOrigin value Africa).
```

Nominals could be simply removed; however, we would then lose, e.g., part of the description of *AfricanRedSlip*. Instead, we can replace a set of nominals using a union of classes where *xxx_nc* depicts the class that was originally the nominal class:

```
OneOfContinent equivalentTo (Africa_nc or America_nc or
        Antarctica_nc or Asia_nc or Australia_nc or Europe_nc).
Africa a Africa_nc. America a America_nc.
Antarctica a Antarctica_nc. Asia a Asia_nc.
Australia a Australia_nc. Europe a Europe_nc.
```

This transformation is approximate (with regard to original representation) because it is no longer assured that, e.g., *Africa_nc* could not have other individuals than Africa. Due to this change, we should also modify the description of *AfricanRedSlip*:

[17] African red slip is a kind of ancient pottery, see `http://open.vocab.org/docs/AfricanRedSlip`.

```
AfricanRedSlip subClassOf Ware.
AfricanRedSlip subClassOf (hasContinentOfOrigin some Africa_nc).
```

This can be done automatically using our framework with a specific TP.[18] It is worth noting that, historically, nominals used to be represented in this ('transformed') way.

Additional off-the-shelf transformation patterns for replacing different OWL constructs are available online at `http://nb.vse.cz/~svabo/patomat/tp/lr/`.

Regarding the *ontology complexity requirement* case, in the past [25] we focused on ontology downgrading to OWL2EL profile [15].

Discussion of Characteristics. Transformation patterns related to the ontology profiling use case are generally simple since they usually capture a single construct. A naming aspect is not employed at all – it is merely about language constructs. On the other hand, an annotation aspect can be an important issue since this transformation could be reversible. Although a transformation tries to be logically equivalent, it naturally features a change of logical semantics.

The transformation of language profiling constructs can generally be done in one of three ways: they can be replaced with an *equivalent different representation*, or they can be replaced with an *approximate different representation*, or they can be *removed*. The first option is obviously the best one. However, it is only rarely possible to find an equivalent representation using other constructs when we need to eliminate a problematic construct during complexity downgrading. The second option is more realistic. However, there is often no (even approximate) alternative means of representation and the problematic construct has to simply be removed.

In comparison with other transformation use cases, the language profiling scenario leads to a fully automatic pipeline (including the detection step). First, a source ontology is pre-processed in order to syntactically decompose the constructs that can hinder querying in a unified way. In our case we decompose ($\rightarrow$) the following constructs:

- *intersection*: A subClassOf (B and C) $\rightarrow$ A subClassOf B. A subClassOf C.
- *disjointness*: DisjointClasses(B, C, D) $\rightarrow$ B disjointWith C. C disjointWith D. B disjointWith D.
- and *disjoint union*: DisjointUnion(B, C, D)[19] $\rightarrow$ B equivalentTo C or D. C disjointWith D.

In the specific situation of the language profiling use case – complexity downgrading, the source ontology is inspected (by using the OWL-API library) in order to specify (based on a given list of forbidden constructs) which transformation patterns

[18] The pattern `tp_nominals-6a.xml` is available at `http://nb.vse.cz/~svabo/patomat/tp/lr/`.

[19] B is the disjoint union of C and D.

should be applied. Consequently, selected transformation patterns are dynamically composed into a sequence.

For each transformation pattern a detection is performed by the *OntologyPatternDetection* service. There are typically more than one pattern instances as a result of the detection step. Furthermore, it is usually precise, because in the case of the language profiling scenario, detection is merely based on structural/logical aspects and naming detection patterns are not generally needed. The following step amounts to generation of transformation instructions by the *Instruction Generator* service. Finally, the application of instructions is carried out by the *OntologyTransformation* service according to the selected transformation strategy. By default, it uses the "progressive" transformation strategy, which enables the removal of axioms but not the removal of entities.

Additionally, there is a *post-processing* step where the remaining forbidden constructs are removed using the OWL-API library. This step ensures completeness of the downgrading process.

4.4 Ontology Matching Use Case

Ontology Matching. Ontology matching is an important part of semantic web vision. Ontologies help to unambiguously describe concepts. However, because of a distributive character of the web there is more than one ontology from a certain domain of discourse. Concepts are consequently described differently in various ontologies. The heterogeneity problem thus was moved one layer above and it is still an open challenge to, automatically or at least semi-automatically, discover proper relationships between concepts from various ontologies. Ontology matching provides *graph-*, *lexical-* and *semantic-* based techniques and methods to achieve such a challenge [3]. Matching is a process of discovering semantic relationships, a *correspondence*, between concepts of different ontologies. A set of semantic relationships between concepts is an *alignment*. Furthermore, we can distinguish a simple correspondence and a complex one where the simple correspondence contains exactly one element on each side of a correspondence. On the contrary, a complex correspondence includes more than one element on at least one side of a correspondence.

For instance, given two ontologies O_1 and O_2, a correspondence is the statement that property *hasName* in O_1 is equal to property *name* in O_2. This is an example of a simple correspondence in which an *equivalence* is the employed relationship. An example of a complex correspondence could extend the previous one in the sense that *hasName* in O_1 is equal to composition of $hasFirstname$ and $hasSurname$ in O_2.

Motivation for Ontology Matching. Ontology matching systems inspect all aspects of ontologies in order to deliver proper matching results. It means that if a matching system does not adequately work with a certain modeling style, the matching process will fail. Therefore, by transforming an ontology to a form that is

more easily matched to another one, the matching results will be improved. Furthermore, a transformation can contribute to a discovery of a complex correspondence.

For this purpose, a transformation can focus on two aspects of ontologies: *logical* (structural) aspect and *naming* aspect. They can be combined or used separately. An obvious application of only the naming transformation is a transformation of all names in O_1 (e.g., local fragments of URI or labels of concepts) according to the naming convention employed in O_2, e.g., a kind of delimiter. A separate logical transformation can be also considered; however, more often both logical and naming aspects will be considered together, as exemplified in Section 4.4, to obtain a complex correspondence.

Example. Let us consider matching of two ontologies O_1 and O_2,[20] which both deal with a conference organisation domain and model the concept that the "paper was accepted". While the structure of O_1 is rather simple and we can only find *Paper* concept but not directly an accepted paper concept, O_2 has a very detailed structure where there is also the concept *Accepted_Paper* (this corresponds to the above-mentioned analysis of the second ontology). However, after more careful inspection of O_1, we can find that *Paper* is in the domain of the property *hasDomain*, which can have *Acceptance* or *Rejection* in its range. Thus, this axiomatisation also concerns an accepted paper, however this cannot be found by traditional matchers since it needs a complex correspondence to come into play. The transformation pattern which can enable discovering the "paper was accepted" concept adds an explicit class *AcceptedPaper* defined as *Paper* for which the decision is acceptance:

```
AcceptedPaper equivalentTo (hasDecision some Acceptance)
```

Designing this transformation pattern, one has to specify a logical (structure) aspect and naming aspect. The logical aspect consists of three axioms:

```
?p domain ?A.
?p range ?B.
?C subClassOf ?B.
```

However, this structure constraint is not enough. Furthermore, there is a naming constraint:

```
NDP = {comparison(?B, head_term(?p), equal), exists(verb_form(?C))}
```

This connects the entity names of $?B$ and $?p$. Particularly, it checks whether the head term of the property named $?p$ is the same as the name of the entity B. Additionally, the name of $?C$ should be one we can verbalise, e.g., a noun "Rejection" can be verbalised as "to reject".

[20] This is a real example where O_1 is *cmt* and O_2 is *ekaw* ontology. Technical details about each step of transformation are available at `http://owl.vse.cz:8080/tutorial/node36.html`.

If those constraints are fulfilled, a new axiom is added:

```
?G equivalentTo (?q some ?F)
```

where $?q = ?p$, $?F = ?C$ and $?G$ form a new entity whose name has to be generated. This is specified by the naming transformation pattern:

```
NTP = {( ?G,  make_passive_verb(?C) + head_noun(?A))}
```

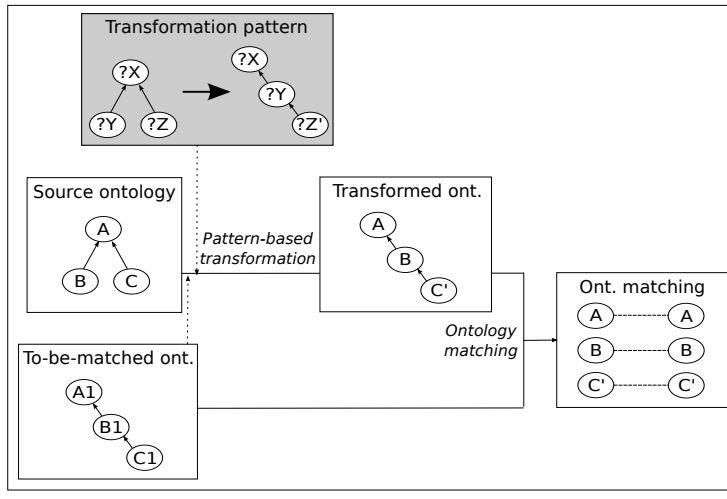
Due to detection, $?C$ and $?A$ are already assigned. E.g., $?C$ is *Acceptance* and $?A$ is *Paper*. Applying the naming instructions above, we get name *AcceptedPaper*.

After such a transformation, a simple matching system will find a simple correspondence $O_1\#AcceptedPaper = O_2\#Accepted\_Paper$. By using the definition of *AcceptedPaper* in O_1 we can get a complex correspondence ($O_1\#hasDecision\ some\ O_1\#Acceptance$) $= O_2\#Accepted\_Paper$ corresponding to the 'Class by Attribute Type' alignment pattern from [18, 20]. This discovery of accepted paper correspondence increases precision of the data translation task, since we can now translate instances of accepted papers among O_1 and O_2 ontologies. It was not possible before because we did not have an equivalent counterpart of $O2\#Accepted\_Paper$ in the O_1 ontology.

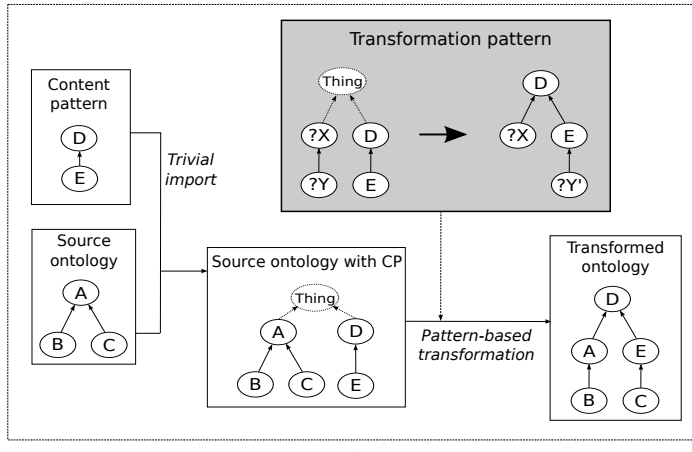
Furthermore, without that discovery it could find some incorrect correspondences such as $O_1\#Acceptance = O_2\#Accepted\_Paper$ in a traditional 1-to-1 matching setting. While the Levenshtein score would be too high (7), Jaro-Winkler would give 84% of similarity which makes it a correspondence candidate, though incorrect.

Discussion of Characteristics. Transformation patterns related to the ontology matching use case can be generally very complex since they can include complex structures of axioms on each side of a pattern. Furthermore, the naming aspect plays an important role, because proper (re)naming of (old) new entities can provide a better clue for ontology matching systems. An annotation aspect provides a useful means for possible both-sided matching. Regarding detection, it is rather semi-automatic since we can consider intervention by the user. Logical semantics could be changed but it will mostly be preserved.

Specific workflow in a situation of matching use case is schematically depicted in Figure 3 a) (the "structural changes" shown are merely illustrative and do not claim to correspond to meaningful transformations). The transformation step here precedes the actual matching step in the overall workflow. A (possibly just a fragment of a) given ontology, which is to be matched to a second ontology, is first matched to the source OP of a TP; the choice of the fragment as well as of the TP is, however, guided by an analysis of the second ontology (such that the target OP of the TP should use the same modeling style as the second ontology). The transformed ontology is built in the style of the target OP of the TP. This makes the subsequent matching to the second ontology easier; e.g., simple and fast string matching methods can be used instead of sophisticated and fragile matching methods.



a)



b)

Figure 3. Schematic depiction of transformation for a) ontology matching b) CP import

4.5 Ontology Design Pattern Import Use Case

Ontology Import. Ontology import is an important part of any ontology design methodology. Generally, it is highly recommended to search for already existing ontologies which could be reused; usually by specialisation of their concepts. This practice enables direct mapping between different ontologies and to take advantage of some wise ontology design decisions.

In this use case we do not consider import of the whole ontology or its arbitrary fragment. We consider here a particular small and widely reusable ontological component – a content pattern (CP). Thus, it is rather a CP import.

Motivation for Ontology Design Pattern Import Use Case. Ontology content patterns [5] are nowadays considered as a central artifact promoting best practices and supporting shareability in ontology design. Although, ideally, CPs should be taken into account from the very start of the design process, it is a common situation that the authors of the ontology are, at the onset, either unaware of the existence of CPs at all or too novice to choose the right one. The CPs then only enter the design process at a later phase, when at least a prototype of the ontology already exists, and their adoption has the nature of ontology reengineering.

Example. Let us consider ontology O_1,[21] where the role authorship is modelled as the *Author* class being a subclass of *Person*. The *Author* class has a pair of existential and universal restrictions over the *writes* property, and also appears in the domain of this property and in the range of its inverse, *writtenBy*. Furthermore, there are two subclasses of *Author*: *PaperAuthor* and *PosterAuthor* respectively. This is a "class-oriented" role pattern. There are various approaches for modeling role-playing aspects, e.g., a sophisticated approach is presented in [22]. However, we demonstrate the CP import use case by importing one of the ontology content design patterns from *OntologyDesignPatterns.org* – the *AgentRole* pattern. It is depicted in a UML-like notation in Figure 4. The *AgentRole* pattern, displayed with its elements in a solid line, specialises the *ObjectRole* pattern (entities from the 'or' namespace), which in turn specialises the *Classification* pattern (entities from the 'class' namespace), both displayed with their elements in a dashed line. Following from the most generic model, the *Classification* pattern merely allows us to state the relationship between an entity and the concept to which this entity is somehow classified; this corresponds to an informal 'reification' of the *subClassOf* relationship. *ObjectRole*, in turn, already deals with role playing, understood as a specific type of classification; entities playing roles can be any objects (i.e., no more arbitrary things such as properties). Let us recall that there is a subproperty relationship between *isRoleOf* and *Classified*, and *hasRole* and *isClassifiedBy*, respectively. Finally, *AgentRole* introduces an even more specific class of such potentially role-playing objects, called Agent, which is declared as disjoint with class Role.

Notice that *content pattern* is already present in the ontology when the ontology is submitted to transformation, and thus has to be a part of the source pattern. However, it is still unconnected to the rest of the ontology (and the source pattern) at this point. A further part of the source pattern is the subsumption relationship between the classes of *Author* and *Person*. The content pattern is transferred to

[21] This is a real example where O_1 is the *confOf* ontology, `http://oaei.ontologymatching.org/2014/conference/data/confOf.owl`.

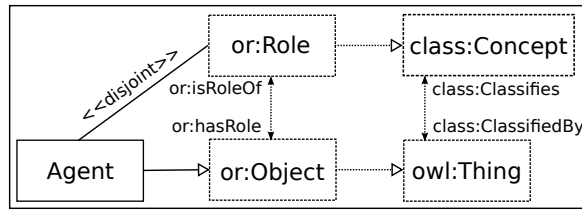


Figure 4. *AgentRole* pattern and its imports

the target pattern without any change. However, there are many alternative ways (discussed in [23]) to smoothly include content pattern into ontology O_1.

Note that, in the particular context of *AgentRole pattern* and the class-oriented role modeling style of O_1, the transformation of the notion of 'Author' from a (seemingly) natural concept to a role amounts to transition from the 'instance of' relationship (being a language primitive in OWL DL) to the 'hasRole' relationship (i.e., object property). In consequence, the fact of a person having the author role, which was previously expressed as, e.g., "John rdf:type Author", now has to connect the individual John to some 'Author' entity through the 'hasRole' property. Assuming we formalise the author role as a class in the target pattern, we arrive at an instance of the "defining classes as property values" problem, treated as a logical pattern in [17].

Let us consider one approach (out of five presented in [23]) which is about "creation of special instances of the class to be used as property values". This may lead, assuming some naming transformations, to the situation depicted in Figure 5 (we only focus on the part of the ontology directly connected to the content pattern, and also omit the namespace prefixes and do not distinguish imports, for better readability). The *Author* class is removed, while there is a new class, with the same name but different meaning, subordinated to *Role*; there is also a distinguished instance of the latter class now, called *AuthorRole* (in a rounded rectangle) as part of the ontology. Let us mention that in the "books-about-animals" example in [17], the class corresponds to an animal species and the instance to the 'topic' of this animal. In both cases, the nature of such instances – 'roles' and 'topics', respectively, entails the semantics of the class; the name of the class (rather appropriate for a natural concept) is then intuitively not very coherent with this semantics.

While instantiating the transformed ontology, an instance of *Person* can be connected by the *hasRole* property with the *AuthorRole* individual.

Let us recall that there are two subclasses of the class *Author*. Due to *recursive application* of transformation pattern along the taxonomy, they will be removed and transformed into subclasses of *AuthorRole* (i.e., *PaperAuthorRole*, *PosterAuthorRole*). Furthermore, each such class will also have a corresponding individual (e.g., *PosterAuthorRole*) as a *direct* instance. An additional axiom issue dealing with this example has already been discussed in 3.3.

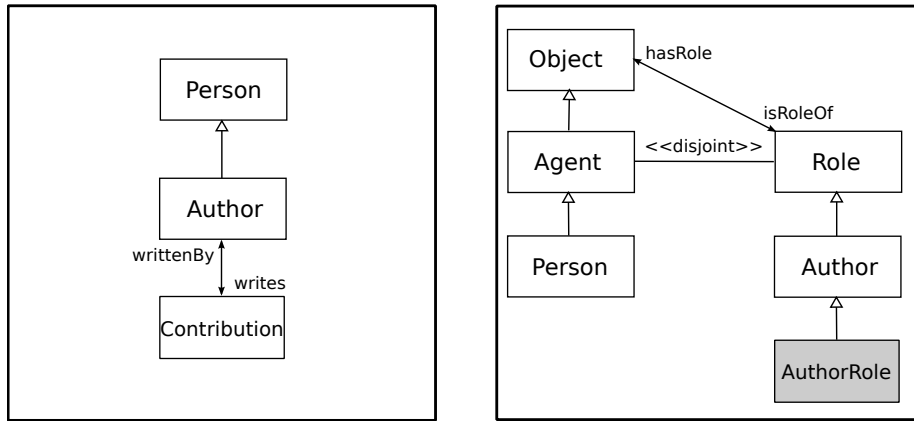


Figure 5. Source and target of import-oriented transformation

Discussion of Characteristics. Transformation patterns related to the ontology import use case can be generally quite complex because they have to reflect structure of content patterns to some extent. On the other hand, the naming aspect will not play so important role since the naming aspect is not part of the content patterns. Annotations can be used but they play minor role in a transformation here. Detection of the source pattern will be mostly automatic. This transformation should not often change logical semantics.

Specific workflow in an ontology import use case is schematically depicted in Figure 3 b). It differs from the matching use case in the overall workflow, in the sense that the import literally precedes the transformation. Namely, in the first step, the CP is included in the ontology as a separate structure, merely subordinated to *owl:Thing*. Then the transformation takes place; however, only TPs specifically tailored to the given CP are considered. In the transformed ontology,[22] shaped to the style of the target OP of the TP, the CP is already integrated into the ontology, as part of its root structure.

Even if a part of the input to the transformation, the content pattern, is fixed, there is potentially a great variability in the ways different ontologies can be adapted to the pattern [23].

5 ON LIMITATIONS OF THE APPROACH

In the past we encountered several tasks on which the transformation framework did not work. The first category of impossible transformations is related to dealing with anonymous classes. Let us say that we want to replace an existential restriction, applied in the position of filler of universal restriction, by a named class, e.g.:

[22] For illustration, Figure 3 b) also contains an entity B, which is not matched by the TP and thus implicitly copied to the transformed ontology.

Use Case	Complexity	Naming	Annotation	Automation	Semantics
Ontology Naming Repair	***	*****	*	**	*
Ontology Simplification (SKOS)	**	*	**	***	*
Ontology Profiling	*	-	***	*****	***
Ontology Matching	*****	***	*****	**	*
Ontology Design Pattern Import	****	**	*	***	*

Table 2. Use cases overview. Semantics means change of the logical semantics.

$$C \sqsubseteq \forall r.(\exists s.Z) \Rightarrow C \sqsubseteq \forall r.Anonymous\_Class.$$

This can be carried out by a transformation pattern where a new named class *Anonymous_Class* is defined. Let us further note that we want to replace any anonymous class expression in the position of filler of universal restriction by a named class, i.e. this should cover the following examples:

$$\begin{aligned} C \sqsubseteq \forall r.(\exists s.Z) &\Rightarrow C \sqsubseteq \forall r.Anonymous\_Class, \\ C \sqsubseteq \forall r.(\leq x\ s.Z) &\Rightarrow C \sqsubseteq \forall r.Anonymous\_Class, \\ C \sqsubseteq \forall r.(X \sqcap Y) &\Rightarrow C \sqsubseteq \forall r.Anonymous\_Class, \\ &\ldots \end{aligned}$$

In order to capture this transformation we need multiple transformation patterns, each specifically tailored to the given restriction. There is no means to refer to *any anonymous expression* in some position and to reference it in other parts of transformation pattern similarly as this can be done with entity placeholders. This hampers the design of a transformation pattern that could fulfil this task all at once.

The second category of impossible transformations is related to an unknown number of components in diverse constructs. Let us consider that we want to split conjunctions statements in position of superclass into their atomic parts, i.e.:

$$C \sqsubseteq \forall r.X_1 \sqcap \exists r.X_2 \sqcap \ldots \Rightarrow C \sqsubseteq \forall r.X_1; C \sqsubseteq \exists r.X_2; \ldots$$

In order to cover this situation we would have to design multiple transformation patterns that would differ in their number of components. Moreover, even in such

a case there would be again the issue described above, i.e. we would have to fix the position of the constructs. All these cases have been encountered during implementation of ontology normalisation detailed in [19]. These limitations stem from the expressiveness of the transformation pattern language.

6 RELATED WORK

The most prominent representative of the transformation across languages is the approach described in [8], which leveraged the ontology translation problem to the generic meta-model. This work has been done from the *model management* perspective, which implies a generality of this approach. From this perspective, meta-models are *languages* for defining models. In general, model management tries to 'support the integration, evolution and matching of (data) models at the conceptual and logical design level'. There are important differences from our approach. Although they consider transformations of ontologies (expressed in OWL DL), these transformations are directed into the generic meta-model or into any other meta-model such as that of UML or XML Schema. In contrast, in our approach we stay within the OWL language (being the meta-model of OWL ontologies) and we consider transformation as a way of translating a certain representation into its modeling alternatives.

There are plenty of different approaches related to ontology transformation within a language. OPPL (Ontology Pre-Processing Language), introduced in [4], is a macro language, based on Manchester OWL syntax, for manipulating ontologies written in OWL. Its initial purpose was to provide a declarative language to enrich lean ontologies with automatically produced axioms. OPPL is based on OWL-API which can be directly used for an ontology manipulation but OPPL makes this easier. Our transformation framework is based on OPPL and we also partly employ OWL-API. Our approach enables the user to define ontology transformation from a pattern-based perspective. Similarly to OPPL, but in the context of Prolog, POPL (Prology Ontology Processing Language) is accessible via POSH (The Prology OWL Shell) [16].

The core of our transformation language are transformation patterns. As for the pattern discovery, traditional data mining techniques could help. In [10] authors introduced data mining algorithm working on ontology-annotated data along with the ontology (represented as an unified RDF bipartite graph) in order to mine *semantic associations* relating diverse entities from ontology-annotated data source. The approach was applied on biomedical domain where they demonstrated not only discovering implicit semantic associations but also detecting flaws in ontologies. For the transformation patterns, we could consider data, i.e. ABox, annotated with the appropriate meta-ontology, e.g. PURO [24] which allows indicating true ontological nature of data entities (e.g. particulars vs. universals etc). Mining modeling style in the form of transformation patterns could be, to some extent, similar as mining semantic associations. Discovering transformation patterns is close to ontology matching considering relationships between ontology and source patterns one by

one (and not in whole as modeling style). However while ontology matching deals with aligning particular entities, patterns are on a higher level of abstraction, i.e. placeholders within ontology patterns. Nowadays, there are plenty of diverse matching techniques providing correspondences between ontology elements [3]. There are also ontology matching approaches which directly base their matching process on association rule mining, e.g. [2].

From the use case perspective, prior research on ontology profiling and downgrading can be divided into generic approaches and those specifically tailored for a certain (popular) reasoner. [14] aimed at elimination of transitivity axioms from an ontology in order to reduce its expressivity. [1] presented an inference service for approximate translation of a concept from one Description Logic to (typically) less expressive Description Logic. In comparison with our approach, both of these approaches focus on logical features, while we follow a more engineering-oriented approach, taking into account the view of the human modeller.

This article is strongly related to the authors' previous papers about this topic. Original idea and first introduction of pattern-based ontology transformation has been in [27]. Afterwards, in [23] the authors presented the ontology design pattern import use case, and in [25] the ontology profiling use case was presented. In this article we concentrate on describing the principles of the whole approach and on revealing its flexibility from the use case perspective, where new use cases are included together with already published ones. All of them are newly presented from the viewpoint of their motivation, inherent distinctive characteristics and actual implementation.

7 CONCLUSIONS AND FUTURE WORK

We presented a pattern-based ontology transformation framework based on OPPL and OWL-API, which includes ontology pattern detection, generation of instructions and finally transformation as such. All steps are implemented as RESTful services; a standalone Java library is available as well. We formally defined the notions related to transformation patterns and described the rules for generating transformation instructions. We demonstrate versatility of the framework on five different use cases. The strong point of the presented approach is that the users can design their own transformation patterns for diverse use cases. This can be done either in XML or in the graphical syntax. Authoring transformation patterns directly in XML syntax allows the user to use all features of the transformation pattern language; however, as we noticed, even an experienced designer sometimes makes an error in terms of syntax or missing transformation pattern parts (e.g., pattern transformation part), therefore we designed and implemented the graphical transformation pattern editor as a proper alternative that helps with those two issues.

We plan to investigate further directions for each use case mentioned above. We also plan to enhance the theory underpinning the transformation framework. This

will be based on the deeper conceptual structure behind the 'surface' RDF/OWL knowledge structures' model of ontology language [24].

We also plan to do more research in a direction of transformation pattern discovery or at least its parts, ontology patterns. So far, we developed transformation patterns manually based on individual examples from those diverse applications/use cases (e.g. ontology import etc). We think that an automatic approach is unfeasible due to the lack of data – examples of an ontology and its transformation variants are very rare and moreover those transformation variants are usually very heterogeneous so that it is hard to generalize a pattern. However, parts of an transformation pattern, i.e. ontology patterns, should be automatically discovered more easily by applying traditional techniques for the pattern discovery, i.e. data mining. This could allow us to detect recurrent patterns (structures) which would be then potentially assembled to a transformation pattern. While an abstraction of discovered recurrent patterns could be a semi-automatic process, linking ontology source and target patterns would be rather manual since automatic linking would be probably too low in precision and recall. From this perspective, another discipline could be useful as a possible inspiration for discovering transformation patterns i.e. ontology matching. Since transformation patterns are, in fact, links between elements of ontology patterns, techniques enabling discovery of links between particular ontologies and their concepts can be considered, i.e. ontology matching. However, we would have to abstract those relationships on higher level of abstraction and figure out meaningful (abstract) structures not only within ontologies but also across ontologies. This brings us to data mining techniques, such as association rule mining, which could be potentially applied on the alignments. Data mining techniques will be considered separately and also in connection with ontology matching in our future work.

Acknowledgement

This research has been partly supported by CSF Grant No. P202/10/1825, "PatOMat – Automation of Ontology Pattern Detection and Exploitation" and by long-term institutional support of research activities by Faculty of Informatics and Statistics, University of Economics, Prague. Ondřej Zamazal has been supported by the CSF Grant No. 14-14076P, "COSOL Categorization of Ontologies in Support of Ontology Life Cycle". Further, we want to thank Marek Dudáš and Ján Černý for their participation on development of graphical tools using and supporting the pattern-based ontology transformation framework.

REFERENCES

[1] Brandt, S.—Kuesters, R.—Turhan, A.-Y.: Approximation and Difference in Description Logics. 8th Conference Principles of Knowledge Representation and Reasoning (KR 2002), Toulouse, 2002.

[2] David, J.: AROMA Results for OAEI 2011. The Sixth International Workshop on Ontology Matching (OM-2011), 2011, pp. 122–125.

[3] Euzenat, J.—Shvaiko, P.: Ontology Matching. Springer-Verlag, 2007, ISBN 3-540-49611-4.

[4] Egaña, M.—Stevens, R.—Antezana, E.: Transforming the Axiomisation of Ontologies: The Ontology Pre-Processor Language. Proceedings of the Fifth OWLED Workshop on OWL: Experiences and Directions (OWLED-2008), 2008.

[5] Gangemi, A.: Ontology Design Patterns for Semantic Web Content. Proceedings of the 4th International Semantic Web Conference (ISWC '05), Galway, Ireland, Springer 2005, pp. 262–276.

[6] Gómez-Pérez, A.—Fernández-López, M.—Corcho, O.: Ontological Engineering: With Examples from the Areas of Knowledge Management, e-Commerce and the Semantic Web. Springer, 2007.

[7] Hanh, H. H.—Jung, J. J.: An Ontological Framework for Context-Aware Collaborative Business Process Formulation. Computing and Informatics, Vol. 33, 2014, No. 3, pp. 553–569.

[8] Kensche, D.—Quix, C.—Chatti, M.—Jarke, M.: GeRoMe: A Generic Role Based Metamodel for Model Management. Journal on Data Semantics, Vol. 8, 2007, pp. 82–117.

[9] Lehmann, J.—Bühmann, L.: ORE – a Tool for Repairing and Enriching Knowledge Bases. The 9th International Semantic Web Conference on The Semantic Web (ISWC 2010), Springer 2010, pp. 177–193.

[10] Liu, H.—Dou, D.—LePendu, P.—Shah, N.—Jin, R.: Mining Biomedical Ontologies and Data Using RDF Hypergraphs. 2013 12th International Conference on Machine Learning and Applications (ICMLA), Miami, FL, IEEE, 2013, pp. 141–146.

[11] Lösch, U.—Sebastian, S.—Vrandečić, D.—Studer, R.: Tempus Fugit – Towards an Ontology Update Language. 6th European Semantic Web Conference (ESWC 2009), Heraklion, LNCS, Vol. 5554, 2009, pp. 278–292.

[12] Miles, A.—Bechhofer, S.: SKOS Simple Knowledge Organization System Reference. W3C Recommendation, August 18, 2009. http://www.w3.org/TR/skos-reference/.

[13] Miller, G. A.: WordNet: A Lexical Database for English. Communications of the ACM, Vol. 38, 1995, No. 11, pp. 39–41.

[14] Motik, B.: Reasoning in Description Logics Using Resolution and Deductive Databases. Ph.D. thesis, Univ. Karlsruhe, 2006.

[15] Motik, B.—Grau, B. C.—Horrocks, I.—Wu, Z.—Fokoue, A.—Lutz, C.: OWL 2 Web Ontology Language Profiles. W3C Recommendation. 2009.

[16] Mungall, C.: Posh – The Prolog OWL Shell. Proceedings of the 8th International Workshop on OWL: Experiences and Directions (OWLED 2011).

[17] Noy, N. (Ed.): Representing Classes as Property Values on the Semantic Web. W3C Working Group Note, 5 April 2005, online at http://www.w3.org/TR/swbp-classes-as-values/.

[18] Ritze, D.—Meilicke, Ch.—Šváb-Zamazal, O.—Stuckenschmidt, H.: A Pattern-Based Ontology Matching Approach for Detecting Complex Correspondences. ISWC Workshop on Ontology Matching (OM-2009), Chanti, USA, 2009, pp. 25–36.

[19] Roussey, C.—Zamazal, O.: Antipattern Detection: How to Debug an Ontology without a Reasoner. The Second International Workshop on Debugging Ontologies and Ontology Mappings (WoDOOM), 2013, pp. 45–56.

[20] Scharffe, F.—Zamazal, O.—Fensel, D.: Ontology Alignment Design Patterns. Knowledge and Information Systems, Vol. 40, 2014, No. 1, pp. 1–28, DOI 10.1007/s10115-013-0633-y.

[21] Schober, D.—Smith, B.—Lewis, S. E.—Kusnierczyk, W.—Lomax, J.—Mungall, C.—Taylor, C. F.—Rocca-Serra, P.—Sansone, S. A.: Survey-Based Naming Conventions for Use in OBO Foundry Ontology Development. BMC Bioinformatics, Vol. 10, 2009, Art. No. 125, DOI 10.1186/1471-2105-10-125.

[22] Sunagawa, E.—Kozaki, K.—Kitamura, Y.—Mizoguchi, R.: Role Organization Model in Hozo. Proceedings of 15th International Conference on Knowledge Engineering and Knowledge Management Managing Knowledge in a World of Networks (EKAW 2006), Podebrady, Czech Republic, 2006. LNCS, Vol. 4248, 2006, pp. 67–81.

[23] Svátek, V.—Šváb-Zamazal O.—Vacura, M.: Adapting Ontologies to Content Patterns Using Transformation Patterns. Workshop on Ontology Patterns (WOP-2010), CEUR, 2010, online `http://ceur-ws.org/Vol-671/pap5.pdf`.

[24] Svátek, V.—Homola, M.—Kľuka, J.—Vacura, M.: Mapping Structural Design Patterns in OWL to Ontological Background Models. Proceedings of the Seventh International Conference on Knowledge Capture (K-CAP '13), ACM, 2013, pp. 117–120.

[25] Šváb-Zamazal, O.—Schlicht, A.—Stuckenschmidt, H.—Svátek, V.: Constructs Replacing and Complexity Downgrading Via a Generic OWL Ontology Transformation Framework. Proceedings of the 39th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM '13), 2013, LNCS, Vol. 7741, 2013, pp. 528–539.

[26] Šváb-Zamazal, O.—Svátek, V.: Analysing Ontological Structures through Name Pattern Tracking. Knowledge Engineering: Practice and Patterns, Proceedings of 16th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2008), LNCS, Springer, Vol. 5268, 2008, pp. 213–228.

[27] Šváb-Zamazal, O.—Svátek, V.—Iannone, L.: Pattern-Based Ontology Transformation Service Exploiting OPPL and OWL-API. Knowledge Engineering and Knowledge Management by the Masses (EKAW-2010), 2010, pp. 105–119.

[28] Svátek, V.—Šváb-Zamazal, O.—Presutti, V.: Ontology Naming Pattern Sauce for (Human and Computer) Gourmets. Workshop on Ontology Patterns (WOP-2009), CEUR, 2009, online `http://ceur-ws.org/Vol-516/pap18.pdf`.

[29] Third, A.: "Hidden Semantics": What Can We Learn from the Names in an Ontology? Proceedings of the 7th International Natural Language Generation Conference (INLG 2012), 2012, Starved Rock, IL, USA, pp. 67–75.

[30] Toutanova, K.—Klein, D.—Manning, Ch.—Singer, Y.: Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network. Proceedings of the Human

Language Technology Conference of the North American Chapter of the Association for Computational Linguistics (HLT-NAACL 2003), 2003, pp. 252–259.

[31] Zamazal, O.—Svátek, V.: Tutorial v3.0.0: Ontology Transformation, Transformation Patterns, Naming Patterns and RESTful Services. Online tutorial, http://owl.vse.cz:8080/tutorial/, 10-05-2013.

[32] The Ontology Transformation Framework Web. Accessed on 03-07-13, http://owl.vse.cz:8080/.

[33] W3C Owl Working Group and Others: OWL 2 Web Ontology Language Document Overview. W3C Recommendation. 2009, http://www.w3.org/TR/owl2-overview/.

Ondřej Zamazal is Researcher and Lecturer at the University of Economics, Prague (UEP), Department of Information and Knowledge Engineering, where he also obtained the Ph.D. degree in 2010. His main research topics are ontological engineering and ontology matching. He participated in EU projects such as K-Space, Knowledge Web, LOD2 and LinkedTV and undertook a research internship at INRIA Rhône-Alps, France and University of Mannheim, Germany. Holder of the Josef Hlávka Award (2006). Author of about 50 refereed publications. Co-organiser of the OAEI initiative, PC member of a number of conferences, including top-class ones such as ISWC or EKAW.

Vojtěch Svátek is Associate Professor (since 2007) and Senior Researcher at the University of Economics, Prague (UEP), Department of Information and Knowledge Engineering. His main research topic is the synergy of ontological engineering, linked data management and data mining. Participant Coordinator of the EU projects K-Space, MedIEQ, LOD2 and LinkedTV, and Coordinator of three projects of the Czech Science Foundation. Author of about 200 refereed publications. Editorial Board member of Elsevier's Journal of Web Semantics and PC member of dozens of top-class conferences such as ISWC, ESWC or EKAW.