

Pattern-Based Ontology Transformation Service Exploiting OPPL and OWL-API

Ondřej Šváb-Zamazal¹, Vojtěch Svátek¹, and Luigi Iannone²

¹University of Economics, Prague,
{ondrej.zamazal,svatek}@vse.cz

²School of Computer Science, University of Manchester, United Kingdom,
iannone@cs.man.ac.uk

Abstract. Exploitation of OWL ontologies is often difficult due to their modelling style even if the underlying conceptualisation is adequate. We developed a generic framework and collection of services that allow to define and execute ontology transformation (in particular) with respect to modelling style. The definition of transformation is guided by transformation patterns spanning between mutually corresponding patterns in the source and target ontology, the detection of an instance of one leading to construction of an instance of the other. The execution of axiom-level transformations relies on the functionality of the OPPL processor, while entity-level transformations, including sophisticated handling of naming and treatment of annotations, are carried out directly through the OWL API. A scenario of applying the transformation in the specific context of ontology matching is also presented.

1 Introduction

The OWL ontology language, now in its more advanced version, OWL 2,¹ is a de facto standard for designing semantic web ontologies. However, with its relatively high expressivity, it often allows to express the same conceptualisation in different ways. This is an obstacle to using existing ontologies in more advanced semantic web scenarios, in particular:

- Two ontologies using different styles are difficult to *match* or to *import* to one another. Few matching systems support complex matching structures that bridge such heterogeneity, never mind considering schema merging and/or data migration.
- Opting for a style when designing an ontology may have dramatic impact on the usability and performance of *reasoners*, as some features cause performance problems for certain reasoners (for a specific reasoner, this has been investigated e.g. in [9]).

As a simple example of style heterogeneity let's consider the following:

¹ <http://www.w3.org/TR/2009/REC-owl2-overview-20091027/>

Example 1. (In Manchester syntax²) In one ‘conference’ ontology,³ the possibility of accepting or rejecting a paper can be expressed via *classes*:

PaperAcceptanceAct SubClassOf: ReviewerAct.
PaperRejectionAct SubClassOf: ReviewerAct.

In another ontology it can be captured using *object properties*:

accepts Domain: Reviewer. accepts Range: Paper.
rejects Domain: Reviewer. rejects Range: Paper.

A third possibility is the use of *enumerations*:

reviewerDecision Domain: Paper.
reviewerDecision Range: (EquivalentTo {acceptance, rejection}).

Obviously, such modelling choices can be captured using *ontology design patterns* [5], especially the language-specific and domain-neutral ones that are usually called ‘logical patterns’. However, while common catalogues of ontology (design) patterns [1, 2] aim at supplying human designers with best practices, for our purposes we do not distinguish whether the particular occurrence of a pattern in an ontology is an informed modelling choice (possibly based on one of these catalogues) or an unintentional one.

A transformation of an ontology fragment from one modelling style to another has to consider two (occurrences of) patterns: one in the *source* ontology and one in the *target* ontology. The two patterns plus the link between them can then be viewed as a *transformation pattern*. Therefore the first step in our workflow is the *detection* of pattern occurrence in the source ontology; it is followed by generation of *transformation instructions*, and, finally, the actual *transformation*, which is largely based on the OPPL pre-processor [4].

Section 2 briefly surveys OPPL as crucial pre-existing component of the whole approach. Section 3 then describes the workflow of ontology transformation and the RESTful services that implement it. Transformation patterns are presented in Section 4 in terms of general shape of patterns (Section 4.1), inclusion of naming patterns (Section 4.2), entity/axiom transformation operations generated (Section 4.3), and the execution of these operations using OPPL and OWL-API (Section 4.4). Finally, Section 5 illustrates the approach on an example within the ontology matching field. The paper is wrapped up with a brief survey of related work, and a Conclusions and Future Work section.

2 Overview of OPPL 2

OPPL [7] is a macro language, based on Manchester OWL syntax, for manipulating ontologies written in OWL. OPPL was introduced in [4] and applied in [3]. Its initial purpose was to provide a declarative language to enrich lean ontologies with automatically produced axioms. Its new version,⁴ OPPL 2, differs from the

² <http://www.w3.org/TR/2009/NOTE-owl2-manchester-syntax-20091027/>

³ A collection of such ontologies has been used in the OAEI ontology matching contest, see <http://nb.vse.cz/~svabo/oaei2009/>. We also refer to it in Section 5

⁴ <http://www.cs.man.ac.uk/~iannonel/oppl/>

previous one by allowing multiple variables in one script, and by aligning the OPPL syntax to the right level of abstraction.

A generic OPPL 2 script currently consists of three main sections, for variable declarations, queries and actions. *Variables* have types that determine the kind of entity each one may represent, i.e. named classes, data properties, object properties, individuals, or constants. A *query* is a set of axioms containing variables, plus an optional set of further constraints on such variables. An *action* may define the addition or removal of a single axiom containing variables.

In a nutshell, running a script consists of developing it into a set of variable-free changes to be applied to an ontology. This can be summarised in the following steps: resolving the query, and instantiating the actions. *Resolving a query* means identifying those values (OWL objects), which will make all the axioms and constraints in the query hold once they replace a given variable. The result of a query then is a set of bindings (variable assignments) that satisfy the query. Each axiom in the query could be evaluated against the *asserted model* only, or using a *reasoner*. OPPL 2 engines always try to use the current reasoner by default. If the OPPL 2 engine has not been initialised with any reasoner, or if the keyword `ASSERTED` is used before an axiom in the query, the matching will be performed on the asserted set of axioms of the ontology only.

As an example let us take the following OPPL 2 script:

```
?x:CLASS,
?y:OBJECTPROPERTY = MATCH("has(\\w+)"),
?z:CLASS,
?feature:CLASS = create(?y.GROUPS(1))
SELECT ASSERTED ?x subClassOf ?y some ?z
BEGIN
REMOVE ?x subClassOf ?y some ?z,
ADD ?x subClassOf !hasFeature some (?feature and !hasValue some ?z)
END;
```

This script demonstrates most of what we described above. The purpose of the script is a simplified application of the Entity-Feature-Value Ontology Design Pattern.⁵ For each subclass axiom asserting that a named class is the subclass of an existential restriction with a named filler, the script will:

- Create a ‘feature class’ using a portion of the original object property name;
- Link such feature to the original named class by means of a generic property `hasFeature` (created on demand, hence the ‘!’ prefix);
- Specify that in the case of `?x` such a feature has a specific class of fillers, i.e. the filler of the original property.

One of the advantages of employing the target pattern is the possibility to express features of a feature. Let us suppose that our initial ontology has a property `hasPrice` directly attached to the class `StockExchangeTitle`, with generic `MoneyAmount` kind of fillers. If we wanted to specify, for instance, when this price

⁵ http://www.gong.manchester.ac.uk/odp/html/Entity_Feature_Value.html

was last checked, or from what stock exchange index, we would have no choice but to overload our `MoneyAmount` class. However, from the modelling point of view this would not be the cleanest solution, as we would add features to a class that was originally designed to represent money amounts. What we really want is further characterise the feature of having a price. Hence, reifying it allows for adding other information to the mere feature without touching the class `MoneyAmount`, which might incidentally have been imported from a third party ontology and therefore should be better left untouched.

In the approach described in the rest of this paper, OPPL serves both as a baseline approach serving for inspiration (namely, its detection part) and as important computational component (its execution part). Detailed discussion is in Section 4.4.

3 Ontology Transformation Workflow

Figure 1 shows the three-step workflow of ontology transformation as currently implemented. Rectangle-shaped boxes represent the three basic (RESTful) services,⁶ while ellipse-shaped boxes represent input/output data.⁷

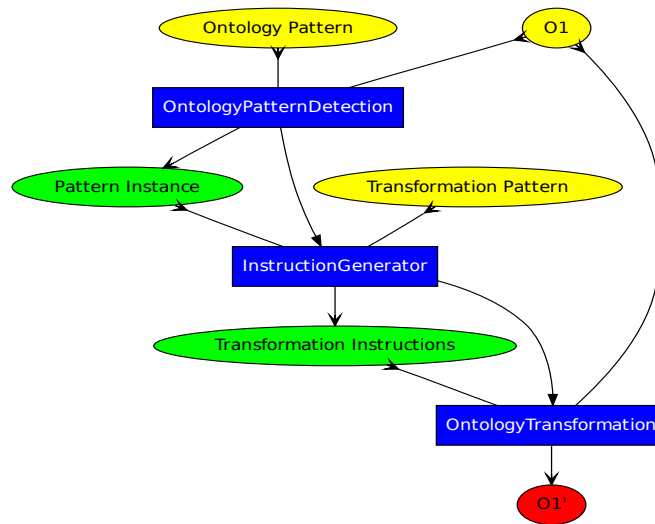


Fig. 1. Ontology transformation workflow; application workflow is depicted using line with normal head and dataflow is depicted using line with vee shape of head

⁶ All accessible via the web interface at <http://owl.vse.cz:8080/>.

⁷ In colours, blue boxes represent RESTful services; yellow ones represent static input data; green ones represent dynamic input/output data; red ones represent output.

The *OntologyPatternDetection* service outputs the binding of entity placeholders⁸ in XML. It takes the transformation pattern (containing the source and target patterns) and a particular original ontology on input. The service internally automatically generates a *SPARQL query* based on 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 service has only been partly implemented by now; its full implementation will leverage on *Terp*, a new syntax for querying OWL ontologies support⁹, which is a combination of Turtle and Manchester syntax.

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.

The *OntologyTransformation* service outputs the transformed ontology. It takes the particular transformation instructions and the particular original ontology on input. This service is based partly on OPPL and partly on our specific implementation over OWL-API.¹⁰

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 further 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 treat 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 other it can be exposed with a unique identifier, as a new *ontology version* pointing to the pre-cursor one using the OWL 2 versioning mechanism.

4 Transformation Patterns and Operational Instructions

4.1 Transformation Pattern Representation

A *transformation pattern* includes two ontology patterns (the source one and the target one) and the schema of transformation of an instance of one to an instance of the other. Transformation patterns are serialized according to an XML schema.¹¹ The representation of *ontology patterns* is based on OWL 2. However, while an OWL ontology refers to particular entities, e.g. to class *Person*, in the

⁸ The detection service is analogous to the first (pattern detection and action instantiation) phase of OPPL pattern application, and placeholders roughly correspond to OPPL variables. For reasons of not using OPPL here see Section 4.4.

⁹ Available in the new release of Pellet, 2.1.

¹⁰ <http://owlapi.sourceforge.net/>

¹¹ <http://nb.vse.cz/~svabo/patomat/tp/tp-schema.xsd>

patterns we generally use *placeholders*. Entities are specified (i.e. placeholders are instantiated) at the time of instantiation of a pattern.

Definition 1 (Ontology Pattern). *Ontology pattern is a triple $\langle E, Ax, NDP^* \rangle$, such that E is a non-empty set of entity declarations, Ax a (possibly empty) set of axioms, and NDP^* a (possibly empty) set¹² of naming detection patterns.*

*Entity declarations*¹³ concern classes, properties and individuals (all at the level of placeholders). Properties can be object, data or annotation ones. Annotation properties enable to capture information about parts of ontology pattern that are not part of the logical meaning of the ontology. *Axioms* are facts about entities included in the transformation; we assume them to be OWL 2 axioms in Manchester syntax. Finally, the *naming detection pattern/s* capture the naming aspect of the ontology pattern for its detection (i.e. it is not used if the pattern is used in the ‘target’ role), see Section 4.2.

Definition 2 (Pattern Transformation). *Let $OP1$ and $OP2$ be ontology patterns. A pattern transformation from $OP1$ (called source pattern) to $OP2$ (called target pattern) is a tuple $\langle LI, NTP^* \rangle$, in which LI is a non-empty set of transformation links, and NTP^* is a (possibly empty) set of naming transformation patterns. Every transformation link $l \in LI$ is a triple $\langle e, e', R \rangle$ where $e \in OP1$, $e' \in OP2$, and R is either a logical equivalence relationship 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*, holding between a ‘logical’ entity and an annotation entity,¹⁴ or, 2) a ‘heterogeneous’ relationships *eqHet*, holding between two ‘logical entities’ of different type. Extralogical relationships correspond to ‘modelling the same real-world notion’ as we saw in the motivating example in Section 1.

Naming transformation patterns capture the way how to name entities in $OP2$ with regard to entities in $OP1$, see Section 4.2.

Definition 3 (Transformation Pattern). *Transformation Pattern TP is a triple $\langle OP1, PT, OP2 \rangle$ such that $OP1$, $OP2$ are ontology patterns and PT is a pattern transformation from $OP1$ to $OP2$.*

4.2 Naming Patterns within Transformation Patterns

The attention paid to naming patterns follows from the finding that untrivial and useful regularities can be observed in ontology entity naming [14]. While

¹² Our current implementation supports at most one naming pattern, in the form described in Section 4.2. However, multiple alternative naming patterns could be employed for detection of ontology pattern occurrence.

¹³ Corresponding to axioms with *rdf:type* property.

¹⁴ OWL 2 annotations may contain various interlinked entities in a separate ‘space’; these are however excluded from the logical interpretation of the ontology.

OPPL supports naming operations at the level of regular expressions (as we saw in Section 2), for modelling style transformation (comprising e.g. part-of-speech alteration) we need a richer inventory of linguistic tools. Naming operations can be divided into *passive* ones, applied for checking purpose, and *active* ones, for naming a new entity.¹⁵ While both can be plugged into naming transformation patterns, only passive operations can be used in naming detection patterns.

Definition 4. A naming detection pattern is a set of passive naming operations, $NDP = \{no_1, no_2, \dots, no_n\}$. All no_i have as operands entities from the ontology pattern to which NDP belongs, and constants.

As an example of NDP with two operations we can take the following:¹⁶

$$\{\text{comparison}(\text{?B}, \text{head_term}(\text{?p})), \text{exists}(\text{verb_form}(\text{?C}))\}$$

For instance, if ?B is ‘Decision’, ?p is ‘hasDecision’ (with ‘Decision’ as head term) and ?C is ‘Acceptance’ (with ‘accept’ as verb form) then the pattern succeeds.

Definition 5. A naming transformation pattern is a set of pairs consisting of an entity and a naming operation, $NTP = \{(e_1, no_1), (e_2, no_2), \dots, (e_n, no_n)\}$. All no_i have as operands entities from the source ontology pattern of the pattern transformation to which NTP belongs, and constants. All e_i are from the target ontology pattern of the pattern transformation to which NTP belongs.

An example of NTP with one compound operation is the following:

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

For instance, if ?A is bound with ‘PresentedPaper’ (with ‘Paper’ as head noun) and ?C with ‘Rejection’ (with ‘Rejected’ as passive verb form), the name of entity ?G in the transformed ontology will become ‘RejectedPaper’.

Naming patterns can be generally defined on any lexical aspect of an ontology: URI of entities, its fragment, 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, e.g. *target="label"*.

The small collection of *implemented naming operations* is being gradually extended as needed for supported transformation patterns. Currently they include (we list together a passive and active variant where relevant):

- *delimiter* detection and change (e.g. underscore or camel-case)
- detection and derivation of *verb form* of a noun (using “derivationally related forms” resource from WordNet and the Stanford part-of-speech tagger¹⁷)
- detection of *head noun* or its complement, for a noun phrase, and of *head term* for verb phrase, typically in a property name (only passive operation)
- construction of *passive form* of verb.

¹⁵ A passive naming operation often has its active variant.

¹⁶ The XML serialisation of this NDP and a superset of the following NTP is in the example in Section 5.

¹⁷ <http://nlp.stanford.edu/software/tagger.shtml>

4.3 Entity and Axiom Transformation Operations

A transformation pattern, $\langle\langle E_1, Ax_1, NDP^*_1 \rangle, \langle LI, NTP^* \rangle, \langle E_2, Ax_2, NDP^*_2 \rangle\rangle$, is converted to transformation instructions for a particular ontology. Building blocks of these instructions are entity and axiom transformation operations.

At the level of *axioms* we consider two operations: *operation of removing of axiom REMOVE(a)* and *operation of adding of axiom ADD(a)*.

At the level of *entities* we consider three operations:

- *operation of adding an entity* where we specify the type and name of the new entity: $ADD(e, t, n)$, where $e \in E_1$, t is an entity type and $n \in NTP^*$.
- *operation of removing an entity: REMOVE(e)*, where $e \in E_1$,
- *operation of renaming an entity*, where we specify the new name of the entity: $RENAME(e, n)$, where $e \in E_1$ and $n \in NTP^*$.

As *removing* is a very sensitive operation with far-reaching effects, we distinguish three different strategies how to cope with this. 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.

When we remove information from the logical content of the ontology, it is still possible to swap it into the *annotations*. For example, when we ‘de-reify’ a property (i.e. change a class expressing a relationship of multiple entities into an object property), we can put information about the third etc. argument of the relationship into annotations of the generated property. Capturing such ‘leaked-out’ information potentially allows reverse transformation. While we already consider annotation as a part of a transformation/ontology pattern, implementation of concrete reverse transformation support is left to future work.

In the following we specify several rules how *entity transformation operations* are derivable from a transformation pattern. For a naming transformation pattern NTP , let $NTP(e)$ denote the function returning the result of a naming operation no such that $(e, no) \in NTP$, and let $TYPE(e)$ denote the function returning the meta-model type of an entity (placeholder) e .

1. If there is an equivalence correspondence between $?A \in E_1$ and $?B \in E_2$ then the instance of $?B$ will be renamed accordingly, i.e. $RENAME(?A, NTP(?B))$
2. If there is an extralogical link *eqAnn* or *eqHet* between $?A \in E_1$ and $?B \in E_2$, then the instance of $?B$ will be named as $NTP(?B)$, typed according to the kind of placeholder of $?B$, and *in the case of radical strategy* $?A$ will be removed, i.e. $ADD(?B, TYPE(?B), NTP(?B))$, $REMOVE(?A)$.
3. All entities from E_2 that are not linked to an entity from E_1 will be ADDED.
4. *In the case of radical strategy*, entities from E_1 that are not linked to any entity from E_2 will be REMOVED.

For Rule 2 and conservative or progressive strategy, there is added an annotation property instance relating the new entity to the original entity. Furthermore, in any strategy we can still refer to the (heterogeneous) transformation link between the original entity and new one at the level of transformation pattern.

For instance, in the transformation pattern for reducing a (reified) *n-ary relation* to binary¹⁸ there is an extralogical link between class ?B and property ?q. According to Rule 2 it would lead to up to two operations:

ADD(?q, ObjectProperty, make_passive_verb(?B)), (*If radical:*) REMOVE(?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 entity.

The *renaming operation* works on the naming aspect (entity URI, *rdfs:label* etc.) of an entity referred with placeholder. By default we process the URI fragment of an entity. Changing the URI fragment is however problematic because it, in principle, means creating a new entity. We can solve this problem by adhering to ontology versioning principles: retain the original entity (with original URI) in the ontology, annotate it as *deprecated*, and 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:

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

While removing of axioms is pretty straightforward, because it works on original entities, adding of axioms must be done in connection with entity operations, because it works on just added or renamed entities.

For instance, in ontology pattern 1 of transformation pattern dealing with restriction class¹⁹ there is axiom '?A equivalentTo (?p value ?a)', e.g. 'PresentedPaper equivalentTo (hasStatus value Acceptance)' which can be swapped to annotations in ontology pattern 2: '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 add two new axioms. The binding of placeholders and entity operations must be considered before.

4.4 Executing Transformation Instructions in OPPL and OWL-API

We base the execution of transformation on OPPL, and add some extensions using OWL-API in order to cover more specific features. We can divide the instructions currently unsupported by OPPL into three groups:

¹⁸ http://nb.vse.cz/~svabo/patomat/tp/tp_1-n-ary-relation.xml

¹⁹ http://nb.vse.cz/~svabo/patomat/tp/tp_ce-hasValue.xml

- Instructions not eligible for putting inside an OPPL script in principle. This regards entity level operations as OPPL is an axiom-level language; an exception is entity addition, which can be understood as axiom addition.
- Instructions that are possibly too specific for the transformation setting. This includes NLP-based operations such as making passive form of a verb.
- Instructions that are in the long-term implementation plan for OPPL, such as handling annotations.

Currently, we use OPPL for the operations on *axioms* and for *adding entities*. Renaming and naming entities according to naming transformation patterns, as well as adding annotations, is done using the OWL-API. As far as detection is concerned, the SELECT part of OPPL could be used to some extent; our naming constraints are however out of the scope of OPPL. Furthermore, in contrast to OPPL, we can take advantage of *decoupling* the process of transformation into parts, which enables user intervention within the whole workflow.

5 Complex Example for Ontology Matching Use Case

For the sake of brevity, we only show one complex example of transformation pattern usage, which addresses the ontology matching use-case: transforming an ontology, *O1*, to a form easier matchable to another one, *O2*. In this experiment we want to match the *cmt* ontology²⁰ to the *ekaw* ontology,²¹ both belonging to the *OntoFarm* collection²² used in the OAEI matching contest.

Transformation Pattern Used. The *cmt* ontology will be transformed using the transformation pattern *tp_hasSome2*, which is based on the matching/detection pattern from [10], see Figure 2. This pattern captures the situation when some concept from *O2* is not explicit in *O1* and should be expressed as restriction. The pattern, containing the NDP and NTP from Section 4.2, looks as follows:²³

- *OP1* : E={Class: ?A, Class: ?B, Class: ?C, ObjectProperty: ?p},
Ax={?p Domain: ?A, ?p Range: ?B, ?C SubClassOf: ?B},
NDP={comparison(?B, head_term(?p)), exists(verb_form(?C))}
- *OP2* : E={Class: ?D, Class: ?E, Class: ?F, Class: ?G, ObjectProperty: ?q},
Ax={?q Domain: ?D, ?q Range: ?E, ?F SubClassOf: ?E, ?G EquivalentTo:
(?q some ?F)}
- *PT* : LI={?A EquivalentTo: ?D, ?B EquivalentTo: ?E, ?C EquivalentTo: ?F,
EquivalentProperties: ?p, ?q},
NTP={(?G, make_passive_verb(?C) + head_noun(?A))}.

Applying the rules from Section 4.3 we would get, at placeholder level, the following *entity operations*:

²⁰ <http://nb.vse.cz/~svabo/oaei2009/data/cmt.owl>

²¹ <http://nb.vse.cz/~svabo/oaei2009/data/ekaw.owl>

²² Each ontology was designed by analysis of either a conference support tool or of the usual procedures of a concrete conference.

²³ XML serialization is at: http://nb.vse.cz/~svabo/patomat/tp/tp_hasSome2.xml

RENAME(?A, NTP(?D)), RENAME(?B, NTP(?E)), RENAME(?C, NTP(?F)),
 RENAME(?p, NTP(?q)), ADD(?G, owl:Class, NTP(?G)).

and *axiom operations*:

REMOVE(?p Domain: ?A), ADD(?q Domain: ?D),
 REMOVE(?p Range: ?B), ADD(?q Range: ?E),
 REMOVE(?C SubClassOf: ?B), ADD(?F SubClassOf: ?E),
 ADD(?G EquivalentTo: ?q some F).

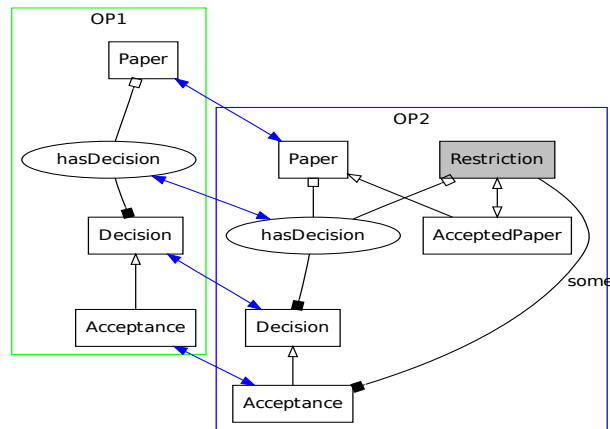


Fig. 2. Instantiated *tp_hasSome2* transformation pattern

Source Pattern Detection. OP1 is serialized as follows:

```

<op1>
  <entity_declarations>
    <placeholder type="ObjectProperty">?p</placeholder>
    <placeholder type="Class">?A</placeholder>
    <placeholder type="Class">?B</placeholder>
    <placeholder type="Class">?C</placeholder>
  </entity_declarations>
  <axioms>
    <axiom>?p domain ?A</axiom>
    <axiom>?p range ?B</axiom>
    <axiom>?C subClassOf ?B</axiom>
  </axioms>
</op1>

```

This is translated into a SPARQL query (omitting declarations of prefixes):

```
SELECT *
WHERE {
  ?p rdf:type owl:ObjectProperty.
  ?A rdf:type owl:Class. ?B rdf:type owl:Class. ?C rdf:type owl:Class.
  ?p rdfs:domain ?A;
    rdfs:range ?B.
  ?C rdfs:subClassOf ?B.
}
```

Furthermore, there are the specific naming constraints that filter out some query results:²⁴

```
<ndp>
  <comparison threshold="1.0" measure="equal">
    <s1>?B</s1>
    <s2>head_term(?p)</s2>
  </comparison>
  <exist>verb_form(?C)</exist>
</ndp>
```

As a result, we have the binding of placeholders, e.g.:

```
<pattern_instance>
  <binding placeholder="?p">hasDecision</binding>
  <binding placeholder="?B">Decision</binding>
  <binding placeholder="?A">Paper</binding>
  <binding placeholder="?C">Acceptance</binding>
</pattern_instance>
```

This would be the output of the *OntologyPatternDetection* RESTful service.

Instruction Generation. In the second step, particular ontology transformation instructions are generated in XML serialization given the specific binding and the transformation pattern, especially its pattern transformation part:

```
<pt>
  <eq op1="?A" op2="?D"/>
  <eq op1="?B" op2="?E"/>
  <eq op1="?C" op2="?F"/>
  <eq op1="?p" op2="?q"/>
  <ntp entity="?G">make_passive_verb(?C)+head_noun(?A)</ntp>
  <ntp entity="?D">?A</ntp>
  <ntp entity="?E">?B</ntp>
  <ntp entity="?q">?p</ntp>
</pt>
```

In this case *entities* are just transferred to the target ontology under the same name. Most *axiom operations* just remove and then add the same axiom given the

²⁴ For explanation we can refer to Section 4.2

equivalence of entities according to LI. An exception is the *addition* of a new axiom, `!AcceptedPaper equivalentTo (hasDecision some Acceptance)`, connected with creation of a new class, `AcceptedPaper`, according to Ax of *OP2*. No operations on *annotations* were needed, as there are no singular removals (without complementary additions).

The resulting *transformation instructions* are serialized as follows:

```
<instructions>
  <oppl_script>
    <remove>hasDecision domain Paper</remove>
    <remove>hasDecision range Decision</remove>
    <remove>Acceptance subClassOf Decision</remove>
    <add>hasDecision domain Paper</add>
    <add>hasDecision range Decision</add>
    <add>Acceptance subClassOf Decision</add>
    <add>!AcceptedPaper equivalentTo (hasDecision some Acceptance)</add>
  </oppl_script>
  <rename>
    <entity type="ObjectProperty" original_name="hasDecision">
      hasDecision
    </entity>
    <entity type="Class" original_name="Paper">Paper</entity>
    <entity type="Class" original_name="Decision">Decision</entity>
  </rename>
  <annotations/>
</instructions>
```

This would be the output of *InstructionGenerator* RESTful service.

Finally, the *cmt* ontology would be transformed, given the transformation instructions, using the *OntologyTransformation* RESTful service. Aside the mentioned enrichment with named entity 'AcceptedPaper', the interface to OWL-API also cares for adding information relating this new entity to the original entity.

Applying *ontology matching* on the *ekaw* ontology and the *transformed cmt* ontology we easily get, among others, the following *simple correspondence*:

`cmt#AcceptedPaper=ekaw#Accepted_Paper`

Although `AcceptedPaper` is not present in the original *cmt*, we can use the simple correspondence for getting a *complex correspondence* for the original *cmt*,

`(cmt#hasDecision some cmt#Acceptance) = ekaw#Accepted_Paper`

corresponding to the 'Class by Attribute Type' alignment pattern from [10, 11].

6 Related Work

Several approaches to *ontology transformation* have recently been published. We refer here to two that look most relevant to our work (aside pure OPPL, to which we made a comparison along the paper). However, their principles and scope are still somehow different from our approach, so direct comparison is hard to make.

In [12] the authors consider *ontology translation* from the Model Driven Engineering perspective. The basic shape of our transformation pattern is very similar to their metamodel. They consider an *input pattern*, i.e. a query, an *output pattern* for creating the output, as well as variables binding the elements. However, the transformation is considered at the *data level* rather than at the *schema level* as (primarily) in our approach.

In comparison with the previous work the authors of [8] leverage 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. There are important differences to 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 one meta-model, the OWL language, and we consider transformation as a way of translating a certain representation into its modelling alternatives.

Our notion of heterogeneous links is also related to *heterogeneous matching* proposed in [6]. The authors propose a logical solution to this problem by extending *Distributed Description Logics* to allow a representation of relationship between classes and properties, for matching purpose. In our approach we use a more generic notion of heterogeneous relationship at extralogical level.

We should also mention prior work of the first two authors of the current paper [13], which was not based on OPPL and viewed ontology transformation primarily in the context of ontology matching, the transformation patterns having been closely associated with the *alignment patterns* from [11].

7 Conclusions and Future Work

We presented pattern-based ontology transformation 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. We formally defined the notions related to transformation patterns and described the rules for generation of transformation instructions. Usefulness of the transformation was shown on a step-by-step example from ontology matching context.

Imminent future work lies in *full implementation* of pattern detection using SPARQL queries automatically generated from ontology patterns, and in enrichment and systematization of the *collection of naming patterns*. We also plan to experiment with detection procedures fine-tuned for the *matching scenario*; for instance, in the example in Section 5 we would instantiate the source ontology pattern so as to achieve a good degree of match to the other ontology. Furthermore, while currently the transformation patterns are designed by end user directly in XML serialization, we envision a *graphical editor* for this purpose. Our approach also definitely needs real *evaluation* in the ontology matching context, which is however difficult due to limited datasets available. Finally, we plan to work out *other use-cases* such as ontology importing and improved reasoning.

This research has been partially supported by the CSF grant no. P202/10/1825, “PatOMat – Automation of Ontology Pattern Detection and Exploitation”.

References

1. Ontology design patterns . org (ODP). Available from: <http://ontologydesignpatterns.org>.
2. Ontology design patterns (ODPs) public catalogue. Available from: <http://www.gong.manchester.ac.uk/odp/html/index.html>.
3. E. Antezana, M. Egaña, W. Blondé, V. Mironov, R. Stevens, B. D. Baets, and M. Kuiper. The Cell Cycle Ontology: a step towards semantic systems biology. In *EKAW-2008*, 2008.
4. M. Egaña, R. Stevens, and E. Antezana. Transforming the axiomisation of ontologies: The Ontology Pre-Processor Language. In *Proceedings of the Fifth OWLED Workshop on OWL: Experiences and Directions (OWLED-2008)*, 2008.
5. A. Gangemi and V. Presutti. *The Handbook on Ontologies*, chapter Ontology Design Patterns. Springer-Verlag, 2009.
6. C. Ghidini and L. Serafini. Reconciling concepts and relations in heterogeneous ontologies. In *Proceedings of the 3rd European Semantic Web Conference (ESWC-2006)*, pages 50–64, 2006.
7. L. Iannone, M. E. Aranguren, A. L. Rector, and R. Stevens. Augmenting the expressivity of the Ontology Pre-Processor Language. In *Proceedings of the Fifth OWLED Workshop on OWL: Experiences and Directions (OWLED-2008)*, 2008.
8. D. Kensché, C. Quix, M. Chatti, and M. Jarke. Gerome: A generic role based metamodel for model management. *Journal on Data Semantics*, 8:82–117, 2007.
9. H. Lin and E. Sirin. Pellint - a performance lint tool for Pellet. In *Proceedings of the Fifth OWLED Workshop on OWL: Experiences and Directions (OWLED-2008)*, 2008.
10. D. Ritze, C. Meilicke, O. Šváb-Zamazal, and H. Stuckenschmidt. A pattern-based ontology matching approach for detecting complex correspondences. In *Ontology Matching workshop (OM-2009)*, 2009.
11. F. Scharffe. *Correspondence Patterns Representation*. PhD thesis, University of Innsbruck, 2009.
12. F. Silva Parreiras, S. Staab, S. Schenk, and A. Winter. Model driven specification of ontology translations. In Q. Lia, S. Spaccapietra, and E. Yu, editors, *27th International Conference on Conceptual Modeling (ER-2008)*, number 5231, pages 484–497. Springer, 2008.
13. O. Šváb-Zamazal, V. Svátek, and F. Scharffe. Pattern-based ontology transformation service. In *International Conference on Knowledge Engineering and Ontology Development (KEOD-2009)*, 2009.
14. V. Svátek, O. Šváb-Zamazal, and V. Presutti. Ontology naming pattern sauce for (human and computer) gourmets. In *Workshop on Ontology Patterns (WOP-2009)*, CEUR. CEUR, 2009.