Pattern-Based Ontological Transformations for RDF Data using SPARQL

The Twelfth International Conference on Pervasive Patterns and Applications
PATTERNS 2021
April 18, 2021 to April 22, 2021 - Porto, Portugal

Marek Suchánek (FIT CTU in Prague, FBE UA)
marek.suchanek@fit.cvut.cz

Robert Pergl (FIT CTU in Prague)

Presenter

- **Marek Suchánek**, PhD student

- Faculty of Information Technology, Czech Technical University in Prague
- Faculty of Business and Economics, University of Antwerp (double PhD)

- Teaching: conceptual modelling and advanced programming courses
- Other projects: Data Stewardship Wizard, Smart City Compass, FAIR Data Point

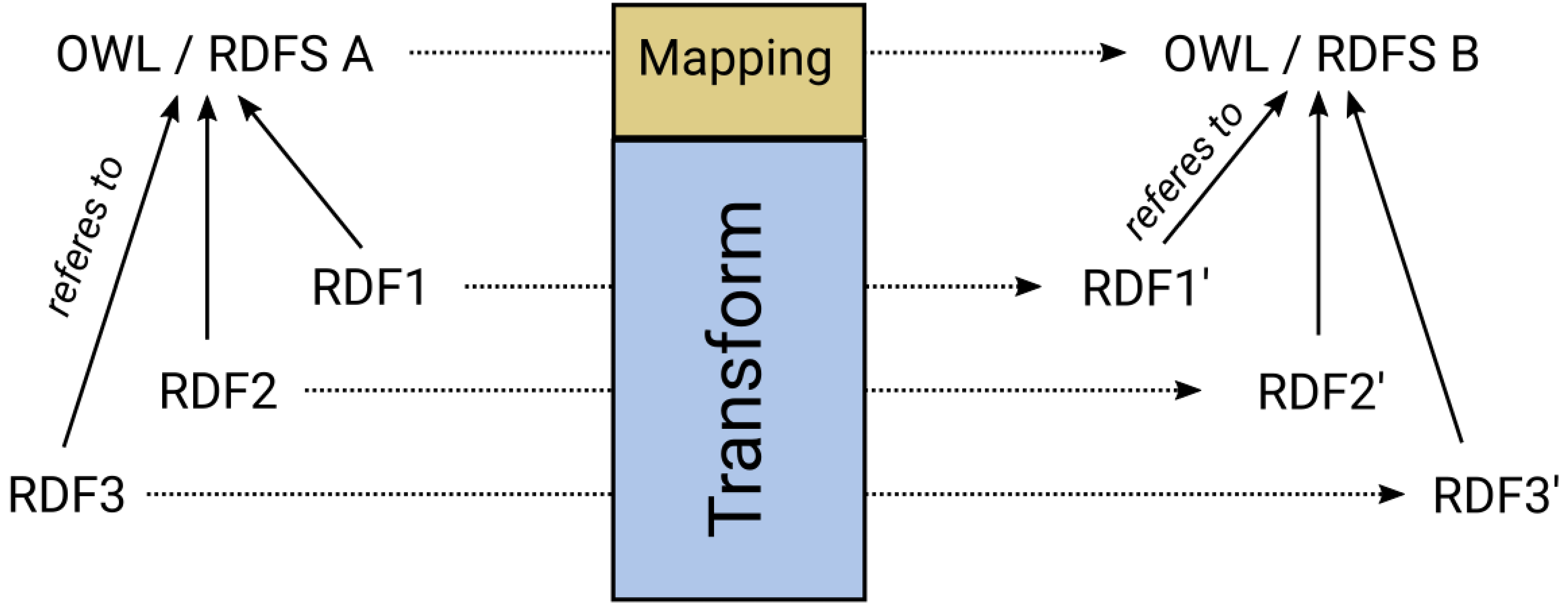
Motivation

- RDF data are more and more used for enabling machine-readability, machine-understandability, and machine-actionability
- Various domains: Semantic Web, Linked Open Data, Bioinformatics, AI & Machine Learning, Software Development, etc.
- RDFs are *just* triples, to describe its structure RDF schemas or OWL ontologies can be used
- The same RDF data can be described using multiple ontologies
- How to define **mapping between ontologies** and **execute corresponding transformation on RDF data**

Related Work and Problem Statement

- There are several methods of defining ontology mapping
- There are also several methods for transforming RDF data (however, mainly focus on RDB, XML, or JSON to RDF transformations)
- It is possible to transform RDF data with SPARQL CONSTRUCT queries
- But those are hard to maintain, i.e., cannot form modules
- How to make the mapping definition with focus on transformation evolvable?

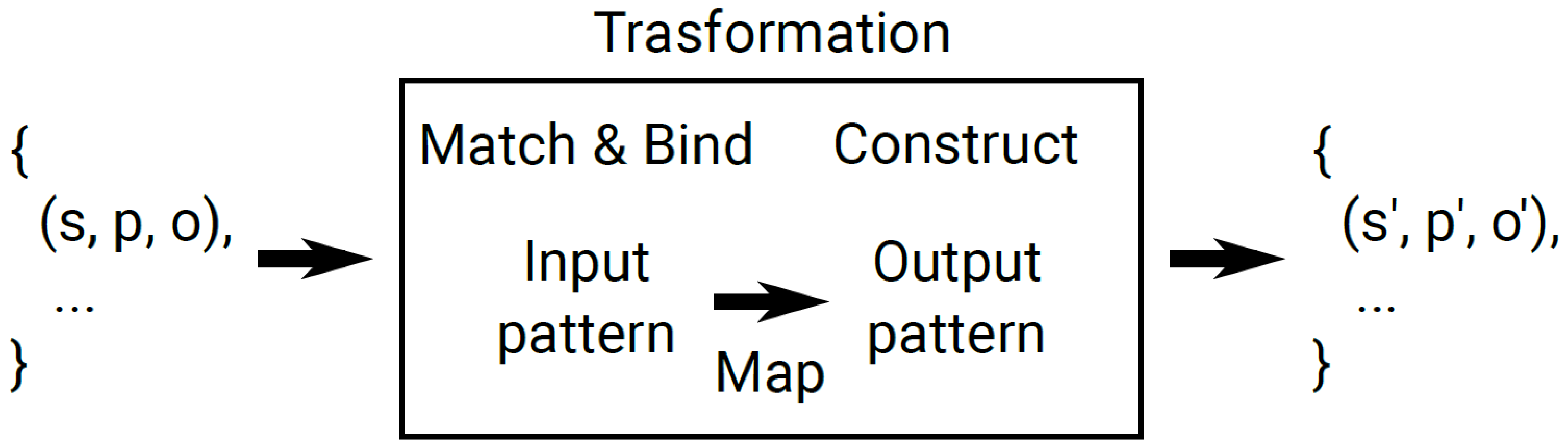
Problem Statement



Our Approach: General Idea

- We propose to define the mapping using **RDF input and output patterns**
- The patterns use the same syntax as SPARQL
- The patterns can be easily re-used and maintained
- For transformation execution, SPARQL CONSTUCT query is compiled from specific patterns with resolved dependencies (includes/imports)
- SPARQL CONSTRUCT query can be executed using standard tools over RDF file(s) or triple store (SPARQL endpoint)

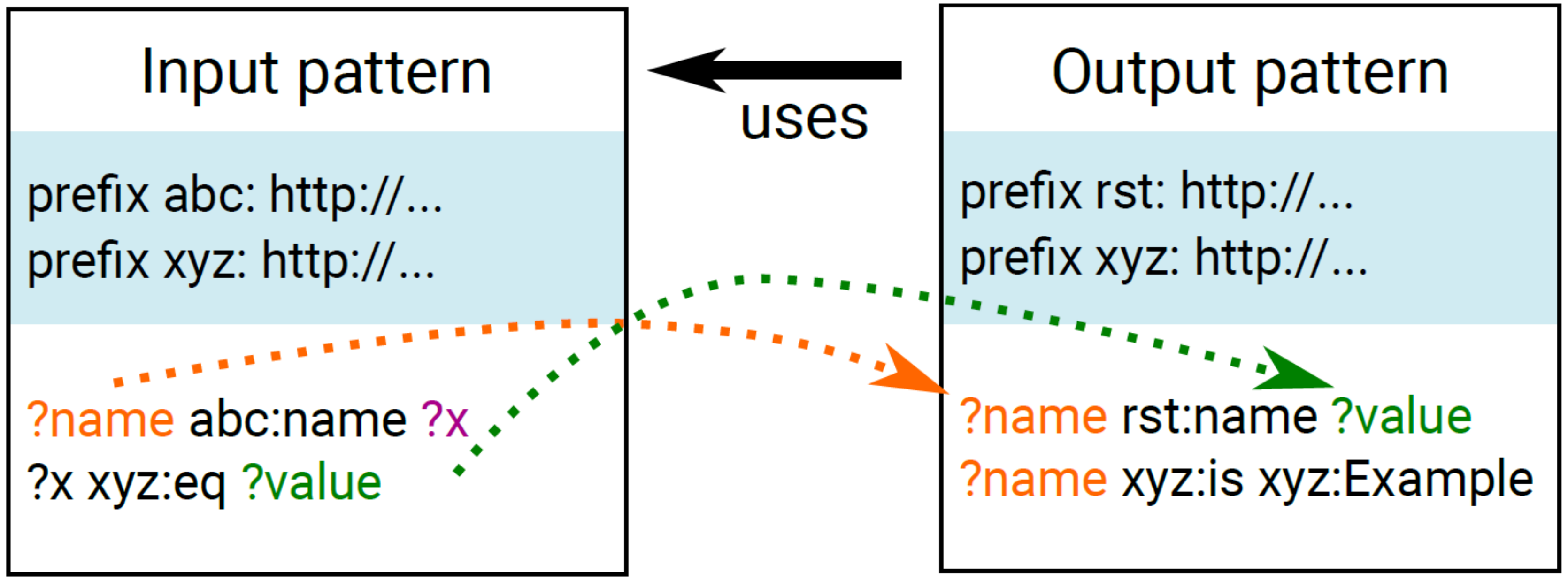
Our Approach: Modular Architecture



Our Approach: Input and Output Pattern

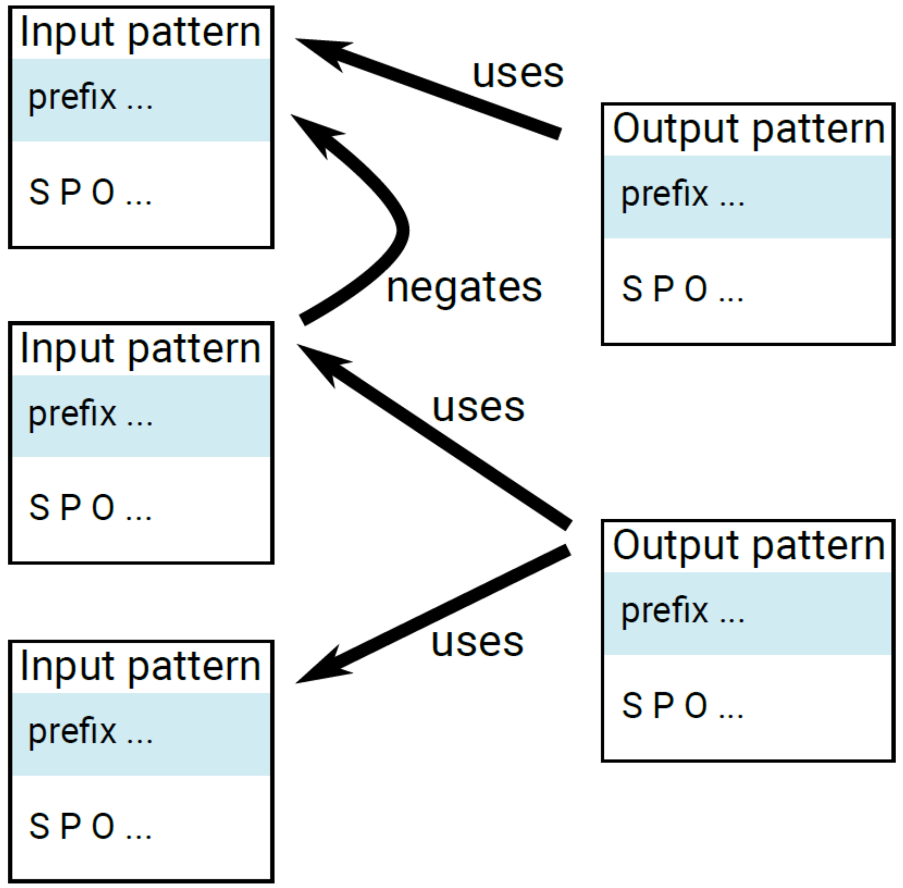
- Input Pattern
  - RDF/Turtle syntax, prefix imports
  - Match data using triples (subject, predicate, object)
  - Bound variables for transformation
- Output Pattern
  - RDF/Turtle syntax, prefix imports
  - Use bound variables from input pattern(s)
  - Construct new triples (subject, predicate, object)

Our Approach: Input and Output Pattern



Our Approach: Pattern Re-Use

- Output pattern uses one or more input patterns
- Input pattern can also use other input pattern, e.g., extend it or negate it
- One input pattern can be used by multiple other patterns
- Naming conflicts must be resolved when compiling SPARQL query



Our Approach: Query Compilation

1. Resolve imports in all input definitions, including variable renaming.
2. For each output definition, import input definition(s) including variable renaming.
3. Merge used prefixes and use renaming mechanism for conflicts (when name and URI do not match).
4. Generate SPARQL CONSTRUCT query with input part in WHERE clause.
5. Execute the query over the input dataset and add result into output dataset.

Example: FOAF -> vCard

```
# Input pattern: input_foaf_person
@prefix rdf:
<http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix foaf:
<http://xmlns.com/foaf/0.1/> .

?person rdf:type foaf:Person .
?person foaf:name ?name .

# Input pattern: input_foaf_organization
@prefix rdf:
<http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix foaf:
<http://xmlns.com/foaf/0.1/> .
?organization rdf:type foaf:Organization .
?organization foaf:name ?name .
```

```
# Output pattern: output_complex
@input: input_foaf_person .
@input: input_foaf_organization
  {organization: org, name: orgName} .

@prefix rdf:
<http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix vcard:
<http://www.w3.org/2006/vcard/ns#> .

?person rdf:type vcard:Individual .
?person vcard:fn ?name .
?org rdf:type vcard:Organization .
?org vcard:title ?orgName .
```

Conclusions and Future Work

- We revisited and prototyped the inheritance implementation patterns
- Focused on generation from model and maintainability
- Avoid order-related combinatorial effects by solving it upon transformation
- Other change-related combinatorial effect are partially avoided by delegation

- Future work:
  - Use to define mapping between conceptual modelling metamodels (and enhance)
  - Create a user-friendly application for defining, testing, and compiling the patterns

Questions & Discussion