# Propagating Ontology Changes to Declarative Mappings in Construction of Knowledge Graphs

Diego Conde-Herreros[1,*], Lise Stork[2], Romana Pernisch[2,3], María Poveda-Villalón[1], Oscar Corcho[1] and David Chaves-Fraga[4]

[1]Universidad Politécnica de Madrid, Ontology Engineering Group, Boadilla del Monte, Spain
[2]Vrije Universiteit Amsterdam, Department of Computer Science, Amsterdam, Netherlands
[3]Elsevier, Discovery Lab, Amsterdam, Netherlands
[4]Universidade de Santiago de Compostela, Departamento de Electrónica e Computación, Santiago de Compostela, Spain

**Abstract**

Knowledge Graphs (KGs) are usually constructed through a set of data transformation pipelines that turn heterogeneous sources into triples following a set of rules. These rules, usually in the form of mapping rules (e.g., RML, R2RML, etc.), are a key resource for the construction of the KG as they describe the relationship between the input data sources and the ontology terms. Several efforts have been made to manage and describe the evolution of the ontology; however, its propagation over interrelated assets (e.g., mapping rules) is commonly done in manual processes. In this paper, we present a preliminary approach to automatically project the evolution of the ontology on the mapping rules used to construct the KG. For each potential change, we analyse the impact on the mappings and the required steps to ensure that the KG is up-to-date w.r.t. the ontology. We implement our solution in fully declarative workflows and demonstrate its benefits in a real-world project in the public procurement domain.

**Keywords**

Knowledge Graphs, Ontology Evolution, Mapping Rules, Impact Assessment

## 1. Introduction

Knowledge Graphs (KGs) have emerged as a powerful mechanism for representing and integrating data on the web. KGs are often constructed from data sources in diverse formats (e.g., CSV, JSON, etc) using a set of mappings that describe the relationship between the data and terms (i.e. classes and properties) from a target ontology. Mapping rules can be described declaratively using languages such as R2RML [1], RML [2, 3] and SPARQL-Anything [4]. When the ontology is changed, mappings usually have to be manually modified. This is a very knowledge and time-intensive task, and is thus not sustainable.

The construction of a knowledge graph is formally defined as a data integration system $DIS = (O, S, M)$, where $O$ is the ontology or vocabulary that defines the global view, $S$ are

*KGCW'24: 5th International Workshop on Knowledge Graph Construction, May 26th, 2024, Crete, Greece*
*Corresponding author.
diego.conde.herreros@upm.es (D. Conde-Herreros); l.stork@vu.nl (L. Stork); r.pernisch@vu.nl (R. Pernisch); m.poveda@upm.es (M. Poveda-Villalón); oscar.corcho@upm.es (O. Corcho); david.chaves@usc.es (D. Chaves-Fraga)
0000-0002-4788-1509 (D. Conde-Herreros); 0000-0002-2146-4803 (L. Stork); 0000-0001-8590-1817 (R. Pernisch); 0000-0003-3587-0367 (M. Poveda-Villalón); 0000-0002-9260-0753 (O. Corcho); 0000-0003-3236-2789 (D. Chaves-Fraga)

a set of input sources to be integrated w.r.t. $O$, and $M$ are a set of rules that describe the relationship between $O$ and $S$ [5]. In this context, ontologies can evolve by incorporating new knowledge or changing the representation of the domain [6]. Ontology evolution has been widely investigated in previous works [7], for instance defining possible change operations [8], developing ontologies to describe these changes [9] or creating new ontology engineering methodologies that consider ontology evolution [10]. There have been theoretical studies of how ontology evolution impacts KG construction and mappings, such as [11], where mappings are updated so that they answer the competency questions for DL-Lite$_R$ ontologies, however, there have not been studies that have resulted in engines that semi-autonomously propagate ontology changes to RML mappings.

During the construction and maintenance of a KG, updates to the underlying ontology impact associated assets, such as mapping rules. For example, if a new class is incorporated into the ontology, mapping rules need to describe the relationship between that class and the input sources. Currently, there is no standard methodology to describe the evolution of an ontology [12]. Moreover, changes to the mapping rules are commonly implemented manually, since there are no tools that keep track of ontology changes and incorporate them into the mappings automatically.

In this paper, we present Ontology Changes Propagation to KG (OCP2KG), a novel framework to describe ontology changes, and propagate them over mapping rules, used for knowledge graph construction from diverse input sources. Following a fully declarative pipeline, we represent the changes of the ontology between two consecutive versions with an extension of previous work by Palma et al. [9]. Taking into account these changes, we study their impact on the mapping rules used for constructing the knowledge graph. Finally, changes to the ontology are propagated to the mapping rules, producing a new version of the mappings that is based on the new version of the ontology. The latter allows knowledge engineers to construct a KG that is compliant to the new version of the ontology.

This paper is divided as follows: Section 2 describes a motivating scenario from the public procurement domain. Section 3 presents the related work and the background concepts. Section 4 describes the impact of each ontology change on the mapping rules, and Section 5 shows the implementation of our tool as a fully declarative pipeline. Section 6 presents a case study of our approach in a real project, the EU Public Procurement Data Space. Finally, Section 7 outlines the main conclusions of the paper and future lines of work.

## 2. Motivating example

The EU Public Procurement Data Space (PPDS)[1] aims to provide a semantic layer of public and private procurement data across Europe. The main objective is to calculate a set of standard performance indicators for each EU member state following a systematic approach. Technically, PPDS aims to construct a decentralised KG, by declaratively mapping any procurement data from each member state to the e-Procurement Ontology (ePO)[2]. However, the ontology's development remains ongoing, resulting in continuous updates with the integration of new

[1] https://europa.eu/!qx9WxQ
[2] https://docs.ted.europa.eu/EPO/latest/

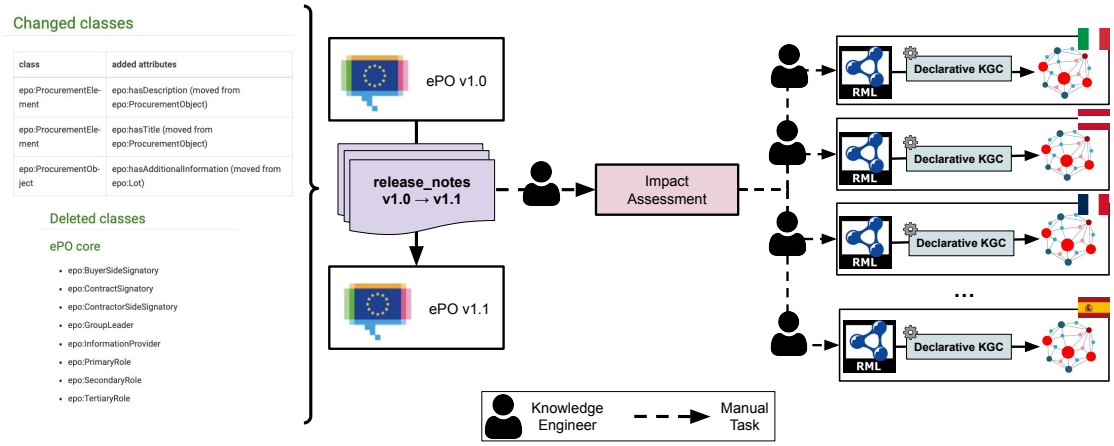


**Figure 1: Motivating Scenario in the EU Public Procurement Data Space.** Release notes are provided in HTML documents and processed by knowledge engineers, to understand and implement changes over RML mappings which generate a KG for each member state.

features and knowledge. The ontology code is publicly available in a GitHub repository[3] and changes between consecutive versions are documented in the form of HTML files[4].

The PPDS relies on the e-Procurement Ontology to map data of each member state into RDF. The mappings are specified in RML [3], and most of the data are provided as XML files. Currently, each time a new version of ePO is released, PPDS knowledge engineers process the release notes and manually accommodate the associated mappings to produce the desirable RDF. In the case of PPDS, these steps need to be repeated for each member state, as the input sources differ. Figure 1 exemplifies this scenario: the left side shows an excerpt of the ontology changes between two consecutive versions of the ontology, and the right side showing how knowledge engineers process the changes to update the dependent RML mappings and KG. It is clear that this produces a non-maintainable KG lifecyle procedure, as each time a new version of the ontology is released, knowledge engineers have to manually implement these changes over the mappings, a time-intensive process that heavily relies on domain and technical expertise. This is a common situation for many KG-driven projects, where there are no standard frameworks or guidelines to manage the evolution of interrelated assets used to construct the KG.

Taking into account the benefits of describing all these assets in RDF, being interoperable among them, it is possible to develop a framework that pursues the automation of the impact of ontology changes on mappings, facilitating the construction of the knowledge graph. The improvements on the knowledge graph construction process are the following: First, the amount of manual work done by the knowledge engineers will be drastically reduced, only required when new knowledge needs to be integrated (e.g., adding a new class to the ontology). Second, the KG construction pipeline can be informed about new rules to be processed, avoiding the necessity of re-generating the complete knowledge graph from scratch, which is currently the common practice. The latter will also reduce execution time and memory consumption.

[3]https://github.com/OP-TED/ePO
[4]https://docs.ted.europa.eu/EPO/latest/release-notes

## 3. Background & Related Work

To the best of our knowledge, no work as of yet provides tools for the (semi)automated propagation of OWL ontology changes to dependent artifacts, such as mapping rules. We, thus, describe related work with respect to ontology change management and the automated creation or refinement of ontology-dependent artifacts, specifically RML mappings.

### 3.1. Related Work

**Ontology Change Management.** Ontology change management or ontology evolution management addresses the need for change in ontologies underlying ontology-based data management systems. When application requirements change, often the underlying ontology needs to change as well [13]. For a detailed work on knowledge graph evolution, we refer the readers to Polleres et al. [7].

The authors of [14] describe requirements for ontology change management, stating that: users should be able to resolve ontology changes, ensuring consistency of all dependent artifacts; users should be helped in managing changes easily; and advice should be given on refinement in the ontology lifecycle. The authors define four phases for ontology change management: representation, change semantics, implementation, and change propagation (to dependent artifacts). Since then, many ontology evolution frameworks have been published [10, 15, 16]. Khattak et al. [17], for instance, define a a change history management framework for evolving ontologies, that enables backtracking changes via change history logs, allowing users to revert changess. It manages issues related to change management in evolving ontologies, such as versioning, provenance, consistency, recovery, change representation and visualization.

The important part of change propagation is the availability of change documentation. In the case that an ontology is updated by a third party, such as in our ePO use case, changes need to be detected after the fact and stored in an interoperable format. Many approaches for classifying change operations between ontology versions have been proposed in the past. One example is COnto-Diff [8], a tool that compares two versions of an ontology and detects and extracts a set of basic change operations that have occurred between the two versions. Then, using a rulebased approach, the tool can detect more complex change operations. TDDonto2 [18] or DynDiff [19] are other examples. Most of these approaches, however, all follow their own change classification and export format, hampering interoperability. Moreover, many implementations are over ten years old [8], of a limited implementation, plugin [18] or not available at all [19]. There are additional downsides of using ontology diffs rather than tracked changes. The after-the-fact change detection is a heuristic approach as a mapping between entities needs to be established first. Further, the order of changes can only be established to some extend and the intend behind edits can also not be identified [20]. However, given the lack of standard in tracking and sharing changes, we need to rely on these approaches at this point in time.

Whereas some of the above-mentioned works indicate the importance of propagating ontology changes to dependent artifacts, and use cases such as that of section 2, no works, provide tools for the (semi)automated propagation of changes to dependent artifacts. Moreover, most approaches for producing change documentation lack a standardised vocabulary.

**Automated Mapping Extraction and Refinement.** Creating RML mapping rules is a time-consuming task that heavily depends on domain expertise. Whereas no methods as of yet exist that mitigate the (semi)automated *update* of mapping rules from ontologies, there are some works that automatically *extract* or *refine* mapping rules from relational databases (RDBs), ontologies or RDF data. We mention the most relevant ones below.

Some works, such as MIRROR [21] or BootOX [22], bootstrap the creation of mappings by automatically extracting them from schemas of RDBs [23]. In doing so, some works use features from the target ontology together with those from the RDB [24]. OWL2YARRRML[5], used as a tool in [25], automatically generates YARRRML templates from ontologies. Lastly, [26] semi-automatically refine existing RML mappings based on the results of linked data quality assessments. There has been theoretical studies in the propagation of ontology changes to mappings, [11] studies the evolution of ontology based data access (OBDA) specifications and modifications to mappings fow DL-Lite$_R$ expressed ontologies.

In our work, however, we do not create mappings from scratch. Instead, we define declarative rules that dictate how updates to an ontology change the dependent mappings. This mitigates the work of ontology and knowledge graph engineers in semi(automatically) propagating ontology changes to dependent artifacts.

## 3.2. Background: The RDF Mapping Language

We rely on the RDF Mapping Language [2, 3] (RML), a mapping language that defines how to relate heterogeneous data and ontology terms to generate RDF. It is a generic language that extends R2RML [1], the W3C Recommendation for mapping relational databases to RDF. RML allows the definition of mappings from multiple heterogeneous sources, leading to higher integrity and richer interlinking among resources. The main components of an RML mapping are the following:

1. **rml:TriplesMap**: It specifies the rules for translating each row of a database or a CSV, element of XML, etc. to RDF triples. It usually generates instances of one or several classes of the ontology. It contains one `rml:logicalSource`, one subject map, and zero to multiple `rml:predicateObject Maps`.
2. **rml:logicalSource**: It describes the input source to be mapped to RDF. Within it, the `rml:source` locates the input data source, `rml:iterator` defines the iteration loop for mapping the data, and the `rml:referenceFormulation` term specifies the way the mapping engine has to parse the input data (e.g. in CSV the rows, in XML the XPath, in JSON the JSONPath, etc.).
3. **rml:subjectMap**: It is a term map (set of rules that generate an RDF term) used to generate the subjects of the RDF triples. Within it, `rml:class` is used to create a `rdf:type` triple for that subject. The relationship with the input source can be defined by `rml:template`, `rml:reference`, or `rml:constant`.
4. **rml:predicateObjectMap**: Defines the predicate and object of the generated triples, it is split into `rml:predicateMap` & `rml:objectMap`, which are also term maps. Within these term maps, it can be defined as a constant value (`rml:constant`), a reference to

[5] https://github.com/oeg-upm/owl2yarrrml

the column, record, element, etc. from the source (`rml:reference`), and a template value (`rml:template`). To describe a join between sources, it is possible to replace `rml:objectMap` by `rml:refObjectMap`. Within it, `rml:parentTriplesMap` specifies the subjects to be used in the object that will be generated under a set of conditions specified by `rml:joinCondition`.

The correspondences between the ontological terms and the elements of the RML mappings for generating triples are the following: a class from an OWL ontology corresponds to a TriplesMap that contains a SubjectMap with `rr:class` indicating its type. A data property corresponds to a PredicateObjectMap within a TriplesMap that generates the triples that have the domain of the property as a subject. An object property from OWL corresponds to a join PredicateObjectMap in the TriplesMap that generates the triples that have the domain of the property as a subject, and where the ParentTriplesMap is the TriplesMap corresponding to the range of the property.

## 4. Propagating Ontology Changes over Mapping Rules

The following section explains the approach followed for the definition of the change operations and the impact of those changes on the mapping rules used for constructing the knowledge graph. We describe in detail how each change operation performed over the ontology impacts them.

Among the different tools that exist in the literature on the evolution of an ontology, there is COnto-Diff [8], a tool that compares two versions of an ontology and detects and extracts a set of basic change operations that have occurred between the two versions. Then, using a rule-based approach, the tool can detect more complex change operations. A limitation, however, is that the change operations are not one-to-one compatible with the RDF(S) and OWL syntax. There is the need to have separate operations for object-type properties and data-type properties, to have a renameTerm operation that is more specific for the changes than the more abstract map(c1,c2) and the substitute(c1,c2) operations. For OCP2KG the operations without an influence on the mappings can be ignored, e.g. those related to annotation properties. Other operations have been added, such as the SubClass related ones, the rename term operation, and the property operations have been split into datatype and object type properties. With this in mind, COnto-Diff tool has not actually been used but its change operations. It has have been inspiration alongside the work from [9], that has been extended. A complete list of the change operations can be found in Table 1.

**AddClass(C):** When a new class C is added to an ontology the main effect it has on the mappings is the addition of a new `rml:TriplesMap`. It requires a `rml:logicalSource` with its mandatory properties to comply with the RML specification: `rml:source`, `rml:iterator`, and `rml:referenceFormulation`. In the `rml:SubjectMap`, the added class C will be specified in the `rml:class` property, and a `rml:template` is also added. **This operation requires a knowledge engineer to define the relationship with the input source for the `rml:logicalSource` and the `rml:SubjectMap` template**.

**RemoveClass(C):** When a class C is removed from an ontology, there are two different situations that lead to different changes. On the one hand, if the class is not involved in any `rdfs:subClassOf` relationship, then all `rml:TriplesMap`(s) that instantiate entities of class C are removed. Also, all predicate object maps where their reference object map includes one of the removed `rml:TriplesMap` in `rml:parentTriplesMap` are deleted as well.

On the other hand, if C is either subject, or object, of a subClassOf property it does a CONSTRUCT query that returns a graph made of the triples that will be deleted from the mappings that will be output as a separate file. Then it removes the `rml:class` triple from the `rml:TriplesMap`, then all `rml:TriplesMap`(s) that instantiate entities of class C are removed. Also, all predicate object maps where their reference object map includes one of the removed `rml:TriplesMap` in `rml:parentTriplesMap` are deleted as well. This is done to avoid losing the instantiation of the entities and that the user can review those triples.

**RenameTerm(T):** The rename operation could be understood as a combination of deletion and the addition of a class or property T. However, in this approach, it is considered an independent operation. It describes an operation in which a term T (class or property) is renamed so that it has a different URI (e.g., `epo:Contract` evolves to `epo:Document` in ePO v4.0.0).

**AddSubClass(C,D):** The operation refers to the addition of the `rdfs:subClassOf` property between classes C and D. The way this changes the mappings is by affecting the `rml:subjectMap` of the `rml:TriplesMap`(s) where C is instantiated. The child class C is also an instance of the parent class D; for this to be represented, it would need to add to the subject map of child C an additional `rml:class` property with the object being the parent class. The child C would also inherit the properties of parent class D, and thus the `rml:predicateObjectMap` of the parent class are also added.

**RemoveSubClass(C,D):** The operation refers to the removal of the `rdfs:subClassOf` property between classes C and D. When removing the property between two classes C and D, the `rml:class` from the parent class D that is present in the `rml:subjectMap` of the `rml:TriplesMap` that generate instances of the child class C is removed. The `rml:predicateObjectMaps` where the value of the predicate has as the domain the class D class are removed.

**AddObjectProperty(C1,P,C2):** Adding an object property is defining a property in the ontology and defining domain C1 and range C2 for it, where the range is a class of the ontology. The changes in the mappings that come from this are the addition of a `rml:predicateObjectMap` in the `rml:TriplesMap` that has the instance of the class C in the subject map as an `rml:class` with P as a `rml:predicate`. Being an object type property, the object of the triple is stated as a reference object map, composed of a `rml:parentTriplesMap` and a `rml:joinCondition`. **It requires knowledge engineer intervention to define the join conditions**.

**RemoveObjectProperty(C1,P,C2):** Removing an object property P, with domain C1, and range C2 results in the corresponding `rml:predicateObjectMaps` with P as `rml:predicate`

**Table 1**
Change operations, effect on mappings, the required knowledge engineer intervention, and if it was present in [9]

| Operations | Changes | KE Intervention | From OWL change |
|---|---|---|---|
| AddClass(C) | Adds TriplesMap | YES | NO |
| RemoveClass(C) | Removes TriplesMap and POM | NO | NO |
| RenameTerm(T) | Replaces URI | NO | NO |
| AddSubClass(C,D) | Adds Class to child & POM | NO | NO |
| RemoveSubClass(C,D) | Removes Class from child & POM | NO | NO |
| AddObjectProperty(C1,P,C2) | Adds POM | YES | YES |
| RemoveObjectProperty(C1,P,C2) | Removes POM | NO | YES |
| AddDataProperty(C,P) | Adds POM | YES | YES |
| RemoveDataProperty(C,P) | Removes POM | NO | YES |
| AddSubProperty(P,Q) | Adds rml:predicate | NO | YES |
| RemoveSubProperty(P,Q) | Removes rml:predicate | NO | YES |
| DeprecateTerm(T) | Removes instances of term | NO | NO |
| RevokeDeprecate(T) | Puts back instances of term | NO | NO |

from `rml:TriplesMap` that has C on the `rml:subjectMap` as `rml:class` being removed.

**AddDataProperty(C,P):** Adding a data property is to define a property in the ontology and define the domain and range for it. where the range is a literal value. The changes in the mappings that come from this are the addition of a `rml:predicateObjectMap` to `rml:TriplesMap` that has C in the `rml:subjectMap` as `rml:class` with P as `rml:predicate`. Being a data type property, the object of the triple is recommended as a `rml:reference`, but it could be an `rml:constant`, or a `rml:template`. **It requires human intervention to define the relationship with the input source**.

**RemoveDataProperty(C,P):** Removing a data property P, with a domain C, results in the corresponding `rml:propertyObjectMap` with P as a `rml:property` from the `rml:TriplesMap` that has C on the `rml:subjectMap` as and object of the `rml:class` property being removed.

**AddSubProperty(P,Q):** The *AddSubProperty* operation consists on the addition of the `rdfs:subPropertyOf` relation between two properties P and Q. The way this affects the mappings is by adding another `rml:predicate` with the super-property Q as an object within those POMs that contain P as `rml:predicate`.

**RemoveSubProperty(P,Q):** The *RemoveSubProperty* operation consists on the removal of the `rdfs:subPropertyOf` relation between two properties P and Q. The way this affects the mappings is by removing the `rml:predicate` with the super-property Q as object in those `rml:predicateObjectMap` that contain P as `rml:predicate`.

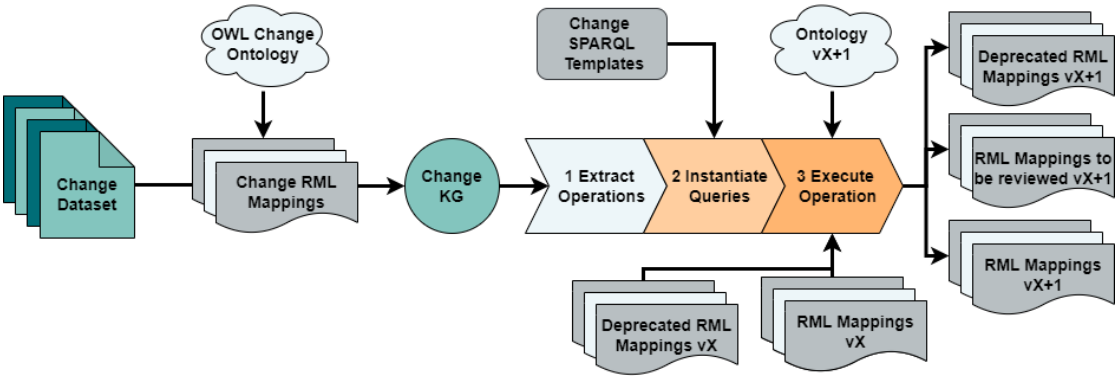


**Figure 2:** Diagram of the OCP2KG tool, its steps, inputs, and outputs.

**DeprecateTerm(T):** Deprecation in software engineering refers to functionality that still exists in software whose use is not recommended. Since it has not yet been removed, it must be kept within the ontology (with the corresponding annotation property). If an ontological term is deprecated, those parts of the mappings must be taken into a different document of the deprecated terms. In the case of a class T, the `rml:TriplesMap` it corresponds to those `rml:PredicateObjectMap` where it is present, and the corresponding joins must be removed from the mappings and placed in a deprecated document. For properties, the `rml:predicateObjectMap` corresponding to them must be moved to the document describing the term deprecated. It functions in a similar fashion to the *RemoveClass* and *RemoveProperty* operations, but instead of erasing, they are erased and added to a new file.

**RevokeDeprecate(T):** The inverse operation of the *DeprecateTerm(T)* operation consists of removing the instances of that ontological term T from the deprecate terms document and placing them back into the mappings file.

## 5. OCP2KG: Ontology Changes Propagation to KG

In this section, we present OCP2KG[6], the design and implementation of our approach. We describe all input and output using RDF data, and all operations within the graphs stored in the program are done via SPARQL queries, so it is implemented in a fully declarative pipeline.

The ontology followed for the representation of changes in RDF data is an extension of the OWL change ontology presented in [9]. Our proposal, which is publicly available, aims to make this ontology more general and suitable for our work. In detail, we add several classes to the ontology to represent our desired changes, which are: a generic *addClass* for the classes without specifying if they are disjoint or not, a *RemoveClass* operation, the *RenameClass*, and *RenameProperty* classes, the *SubclassChange* class, and its children, *AddSubClass*, and *RemoveSubClass*.[7]

[6]https://github.com/oeg-upm/ocp2kg
[7]https://https://w3id.org/OWLChangeOntolog

```
PREFIX epo: <http://data.europa.eu/a4g/ontology#>
PREFIX omv: <http://omv.ontoware.org/2009/09/OWLChanges#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX epo-change: <http://epo-changes.org/1.0-1.1/RemoveClasslass/>

epo-change:AwardDecision rdf:type omv:RemoveClass.
epo-change:AwardDecision omv:deletedClass epo:AwardDecision.
```

Listing 1: **Example Of Class Removal in the e-Procurement Ontology**

First, the changes are defined as RDF data, and a URI is assigned to each of the changes. Each change must be instantiated as a type of change, and then the parameters of that operation must be indicated by using the properties of the class. This set of changes is used as input for OCP2KG. The new version of the ontology should also be provided so that the tool can consult additional information such as `rdfs:subClassOf` relations when needed. The outdated mappings also have to be provided so that they serve as a template for the new ones. The output is the updated mappings that have been changed according to the change propagation defined in Section 4. A complete overview of the workflow implemented in OCP2KG is provided in Figure 2. We provide a detailed description of all steps implemented in the pipeline along with a set of representative examples.

The input change RDF data represents the different changes between two consecutive versions of the ontology, being modeled by the OWL change ontology extension. Each change is an entity that specifies the kind of change through its class. Then there is a triple with each of the ontological terms involved in that particular change operation. Each operation has its own properties defined for that, for instance: *och:addedClass* for the class being added in the *addClass* operation. A design decision that has been made for simplicity sake is that any operations that modify elements of a property or class is the same as a Remove and Add operation, it may not be efficient, but for an early approach is simple and manageable. For instance if a predicateObjectMap has multiple object maps for the same property and that property was to be changed, then the predicateObjectMap would be removed and then added again with the updated values. We present a small example in Listing 1 that represents the removal of a class in the ePO.

OCP2KG first loads the change data, outdated mappings, the current version of the ontology as separate graphs. Then, for each operation, the engine applies those changes to the mappings graph with SPARQL queries. These SPARQL queries are provided to OCP2KG as a set of predefined templates, one per operation. Said templates take into account the different ways that RML mappings can be described, taking into account constant predicates and objects, for instance. The variables of each SPARQL query are then substituted by the actual values of the changes. Regarding the implementation of changes that add or remove ontological terms from the ontology, the following conventions have been followed. First, for those operations that do not require any knowledge engineer intervention, the operation is performed automatically on the graph that will be returned as output and no special treatment is given. An example of this can be seen in Listing 2 for the *RemoveObjectProperty* operation.

In operations where ontological terms are added, the relationship of the term with the input

```
DELETE WHERE {
   ?tm rml:predicateObjectMap ?pom.
   ?pom rml:predicate %REMOVED_PROPERTY% .
   ?pom rml:objectMap ?object.
   ?object rml:parentTriplesMap ?parent_tm
   ?object rml:joinCondition ?join_condition .
   ?join_condition ?conditions ?object_conditions .
}
```

Listing 2: **Remove Object Property query**

```
PREFIX rml: <http://www.w3.org/ns/r2rml#>
PREFIX rml: <http://w3id.org/rml#>
PREFIX epo: <http://data.europa.eu/a4g/ontology#>
INSERT DATA {
    %TM_class_ID% a rml:TriplesMap;
    rml:logicalSource [
        rml:source "XXXX";
        rml:referenceFormulation "XXXX"];
    rml:subjectMap [
        rml:template "XXXX";
        rml:class %CLASS% ].
}
```

Listing 3: **Add class query**

source must be established by the knowledge engineer, as it still cannot be obtained automatically. This happens in operations where a `rml:subjectMap` or a `rml:predicateObjectMap` is being added and the relationship with the source must be asserted via a `rml:reference` or `rml:template` property, or when creating a new `rml:TriplesMap` a `rml:LogicalSource` must be stated with a `rml:logicalSource`, and a `rml:referenceFormulation`. For this situation we are using a token made of four X's to indicate that it is a value that has to be replaced by the knowledge engineer. An example of this can be seen in Listing 3.

Another instance where an engineer intervention being required is when there is doubt as to whether something should be deleted. This happens, for instance, when deleting a class, since if it has a super class then those instances where it appears in the mappings could be substituted by the super class. However, this cannot be done automatically and requires intervention. The way we have handled this is so that it removes those triples from the graph and inserts them into another graph for the engineer to review. This example can be seen in Listing 4.

Subclass and superclass relationships are not needed to be explicitly stated in the RML mappings, those can be inferred by the RDF engine, that have different ways of handling it. For the tool the authors have materialized the inferred triples from the sub-super class relationships to avoid time-consuming inference at query time.

```
PREFIX rml: <http://www.w3.org/ns/r2rml#>
PREFIX epo: <http://data.europa.eu/a4g/ontology#>
DELETE WHERE{
    ?triplesmap rml:subjectMap ?subject.
    ?subject rml:class %REMOVED_CLASS%.
    ?triplesmap rml:predicateObjectMap ?pom.
    ?pom rml:predicate ?predicate.
    ?pom rml:object ?object.

    OPTIONAL{
        ?parent_tm rml:predicateObjectMap ?parent_pom.
        ?parent_pom rml:predicate ?parent_predicate .
        ?parent_pom rml:objectMap ?parent_object.
        ?parent_object rml:parentTriplesMap ?triplesmap.
        ?parent_object rml:joinCondition ?join_condition .
        ?join_condition ?conditions ?object_conditions
    }}
```

Listing 4: **Remove class query**.

## 6. Case Study: The EU Public Procurement Data Space

In this section, we describe the experiment of propagating the changes from a change RDF data file into an outdated version of the mappings from the e-Procurement ontology use case. We describe the steps followed, the expected results from the experiment. The code and data are openly available[8]. The mappings we are using are those from the v3.0.0 version of the ontology version, which we will update with the v3.0.1 ontology and its corresponding change log. This comes from the need to be automatically able to update the mappings to conform to the newer versions of the ontology. Currently, the e-Procurement ontology is in its version 4.0.2, whereas the most up-to-date PPDS mappings are from the 3.0.0 version and have not been updated since.

First, there is the input change data, which comes from the 3.0.0-3.0.1 change log[9], it contains changes for a section of the mappings corresponding to the `tedm:SubmissionTerm` and the `tedm:Organization` `rml:TriplesMap`. The list of operations is shown in Table 2.

The following are the mappings for the evaluation. Since the ontology is quite large, this is a representative portion of the total mappings and comprises `rml:TriplesMap` of `tedm:SubmissionTerm` and the `tedm:Organization`, containing the necessary components of the mappings to perform the operations of Table 2.

In the resulting mappings from the execution of the tool, we get updated mappings where the operations described in Table 2 have taken effect. As expected, the *AddClass* operation has resulted in the addition of five lines that create a `rml:TriplesMap`, its `rml:logicalSource`, and its `rml:subjectMap` with the token values. The *AddSubClass* operation results in one line being added within the Submission term `rml:subjectMap`, adding the super class. The *AddObjectProp-*

[8] https://github.com/oeg-dataintegration/ocp2kg
[9] https://docs.ted.europa.eu/EPO/3.0.1/release-notes.html

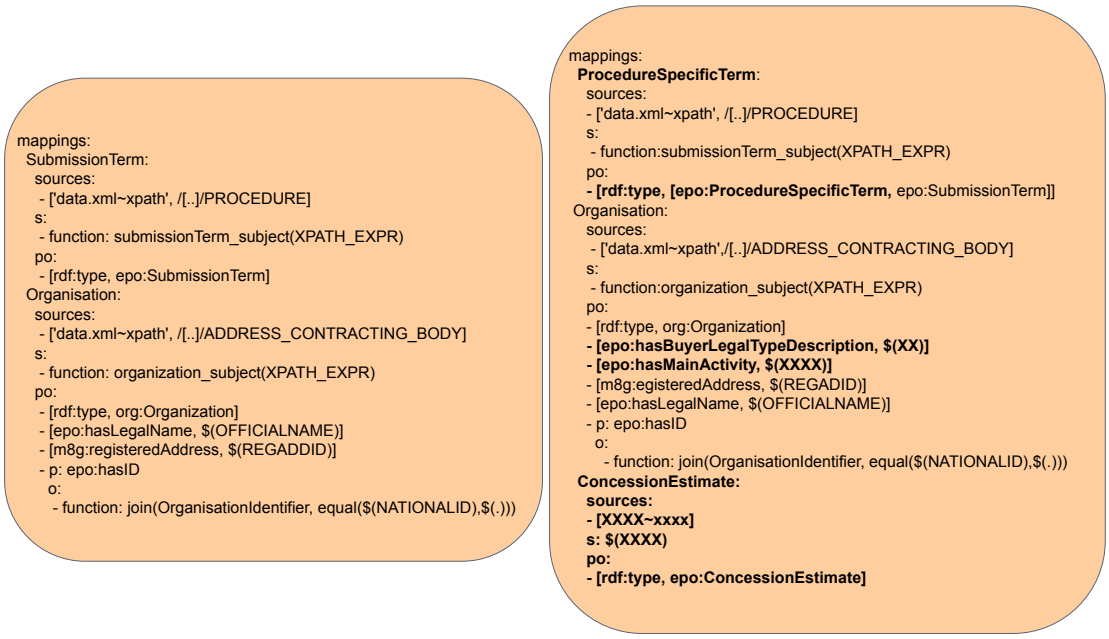


**Figure 3:** Excerpt of the automatic evolution from v3.0.0 to v3.0.1 of the PPDS mapping rules.

**Table 2**
Operations performed in evaluation

| Operations | Parameters |
|---|---|
| *AddClass* | *epo:ConcessionEstimate* |
| *AddSubClass* | *epo:SubmissionTerm,epo:ProcedureSpecificTerm* |
| *AddObjectProperty* | *org:Organization,epo:hasMainActivity,at-voc:main-activity* |
| *RemoveObjectProperty* | *org:Organization,epo:hasMainActivityType,at-voc:main-activity* |
| *AddDataProperty* | *org:Organization,epo:hasBuyerLegalTypeDescription* |
| *RemoveDataProperty* | *org:Organization,epo:hasBuyerTypeDescription* |

*erty* operation results in the addition of two lines that form the `rml: predicateObjectMap`, and the *RemoveObjectProperty* operation does not have an effect since that property is not present in the outdated mappings, and therefore nothing had to be removed. The *AddDataProperty* operation results in the addition of two lines, adding its `rml:predicateObjectMap` to the mappings. The *RemoveDataProperty* has no effect, since that property has no presence in the outdated mappings, and thus nothing had to be removed. In Table 3 we can see the differences between the old mappings and the updated mappings. In the table, we compare the number of `rml:TriplesMap`, `rml:logicalSource`, `rml:subjectMap`, & `rml:predicateObjectMap` to check whether the change operations are taking effect as intended.

As limitations, our approach is not fully automatized, as it still requires the intervention of a knowledge engineer to define the relationship between the new terms of the ontology and the data to be transformed into RDF data [27]. Much work has to go into ontology engineering practices to have a more extensive list of change operations that contemplate different ways to define classes, properties, and restrictions and create queries that contemplate those cases.

**Table 3**
Differences in input mappings, and output mappings, in terms of the amount of `rml:TriplesMap`

| | Old | Updated |
|---|---|---|
| Number of lines | 58 | 40 |
| #TriplesMap | 2 | 3 |
| #LogicalSource | 2 | 3 |
| #SubjectMap | 2 | 3 |
| #PredicateObjectMap | 3 | 5 |

## 7. Conclusions and Future Work

In this paper, we present an approach that enables propagating ontology evolution in the construction of the knowledge graph. For the first time, knowledge evolution is considered as a parameter in this process, analyzing its impact on the mapping rules, while providing a semi-automatic tool for updating RML mappings with ontology changes. As seen in section 6 the tool correctly propagates changes to the RML, requiring minimal knowledge engineer intervention, mainly just define the relationship between the input data and the classes and properties being added. We implement a fully declarative pipeline as a proof of concept of the contribution, namely OCP2KG, and test it on a real use case in the public procurement domain.

The main limitation of the work is that full automation has not been achieved, as the tool still requires KE intervention for new knowledge being added. Since this is an early work the tool only takes into account major ontology changes, not fully contemplating the expressivity of both OWL and RML.

For future work, we will perform an experimental study to understand the benefits of our contribution w.r.t. the current approach. We want to study the impact on the execution time and memory consumption of KG construction engines dealing with ontology changes in a control environment or benchmark [28], and the effects it can have usage on different mapping engines. Work has to go to defining an order and priority for the change operations since some changes can neutralize each other and some changes require other changes to take place before. For example: AddClass(C) has to take place before AddSubClass(C,D) can be executed. Additionally, we will extend our approach to include not only the mapping rules used to construct the knowledge graph but other associated assets such as shapes for the validation or queries for the exploitation.

## Acknowledgements

David Chaves-Fraga is funded by the Galician Ministry of Education, University and Professional Training and the European Regional Development Fund (ERDF/FEDER program) through grants ED431C2018/29 and ED431G2019/04. María Poveda-Villalón is funded by the European Union's Horizon 2020 research and innovation programme under the grant agreement no. 101016854 (AURORAL). Diego Conde-Herreros is supported by the project *Knowledge Spaces* (Grant PID2020-118274RB-I00 funded by MCIN/AEI/10.13039/50110 0011033) & by Spanish Statistical Office (INE).

## References


[1] C. Sundara, S. Das, R. Cyganiak, R2RML: RDB to RDF Mapping Language, W3C Recommendation, W3C, 2012. Http://www.w3.org/TR/2012/REC-r2rml-20120927/.

[2] A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens, R. Van de Walle, RML: A generic language for integrated RDF mappings of heterogeneous data., Ldow 1184 (2014).

[3] A. Iglesias-Molina, D. Van Assche, J. Arenas-Guerrero, B. De Meester, C. Debruyne, S. Jozashoori, P. Maria, F. Michel, D. Chaves-Fraga, A. Dimou, The RML Ontology: A Community-Driven Modular Redesign After a Decade of Experience in Mapping Heterogeneous Data to RDF, in: International Semantic Web Conference, Springer, 2023, pp. 152–175.

[4] E. Daga, L. Asprino, P. Mulholland, A. Gangemi, et al., Facade-X: an opinionated approach to SPARQL anything, Studies on the Semantic Web 53 (2021) 58–73.

[5] M. Lenzerini, Data integration: A theoretical perspective, in: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, 2002, pp. 233–246. doi:10.1145/543613.543644.

[6] N. F. Noy, M. Klein, Ontology evolution: Not the same as schema evolution, Knowledge and information systems 6 (2004) 428–440.

[7] A. Polleres, R. Pernisch, A. Bonifati, D. Dell'Aglio, D. Dobriy, S. Dumbrava, L. Etcheverry, N. Ferranti, K. Hose, E. Jiménez-Ruiz, et al., How does knowledge evolve in open knowledge graphs?, Transactions on Graph Data and Knowledge 1 (2023) 11–1.

[8] M. Hartung, A. Groß, E. Rahm, COnto–Diff: generation of complex evolution mappings for life science ontologies, Journal of biomedical informatics 46 (2013) 15–32.

[9] R. Palma, P. Haase, Ó. Corcho, A. Gómez-Pérez, Change Representation For OWL 2 Ontologies, in: R. Hoekstra, P. F. Patel-Schneider (Eds.), Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED 2009), Chantilly, VA, United States, October 23-24, 2009, volume 529 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2009. URL: https://ceur-ws.org/Vol-529/owled2009_submission_14.pdf.

[10] F. Zablith, G. Antoniou, M. d'Aquin, G. Flouris, H. Kondylakis, E. Motta, D. Plexousakis, M. Sabou, Ontology evolution: a process-centric survey, The knowledge engineering review 30 (2015) 45–75.

[11] D. Lembo, R. Rosati, V. Santarelli, D. F. Savo, E. Thorstensen, Mapping repair in ontology-based data access evolving systems, in: IJCAI International Joint Conference on Artificial Intelligence, International Joint Conference on Artificial Intelligence, 2017, pp. 1160–1166. doi:10.24963/ijcai.2017/161.

[12] R. Pernisch, M. Poveda-Villalón, D. Conde-Herreros, D. Chaves-Fraga, L. Stork, When ontologies met knowledge graphs: Tale of a methodology (2024).

[13] L. Stojanovic, Methods and tools for ontology evolution, Ph.D. thesis, Karlsruhe Institute of Technology, Germany, 2004. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000003270.

[14] L. Stojanovic, A. Maedche, B. Motik, N. Stojanovic, User-driven ontology evolution management, in: Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web: 13th International Conference, EKAW 2002 Sigüenza, Spain, October 1–4, 2002 Proceedings 13, Springer, 2002, pp. 285–300.

[15] H. Wardhana, A. Ashari, Review of ontology evolution process, International Journal of

Computer Applications 45 (2018) 26–33.
[16] M. Javed, Y. M. Abgaz, C. Pahl, Ontology change management and identification of change patterns, Journal on Data Semantics 2 (2013) 119–143.
[17] A. M. Khattak, K. Latif, S. Lee, Change management in evolving web ontologies, Knowl. Based Syst. 37 (2013) 1–18. doi:10.1016/J.KNOSYS.2012.05.005.
[18] K. Davies, C. M. Keet, A. Lawrynowicz, More Effective Ontology Authoring with Test-Driven Development and the TDDonto2 Tool, Int. J. Artif. Intell. Tools 28 (2019) 1950023:1–1950023:25. doi:10.1142/S0218213019500234.
[19] S. D. Benavides, S. D. Cardoso, M. D. Silveira, C. Pruski, Analysis and implementation of the DynDiff tool when comparing versions of ontology, J. Biomed. Semant. 14 (2023) 15. doi:10.1186/S13326-023-00295-7.
[20] M. Tury, M. Bieliková, An approach to detection ontology changes, in: Workshop Proceedings of the Sixth International Conference on Web Engineering, ICWE '06, Association for Computing Machinery, 2006. doi:10.1145/1149993.1150009.
[21] L. F. de Medeiros, F. Priyatna, O. Corcho, MIRROR: Automatic R2RML mapping generation from relational databases, in: Engineering the Web in the Big Data Era: 15th International Conference, ICWE 2015, Rotterdam, The Netherlands, June 23-26, 2015, Proceedings 15, Springer, 2015, pp. 326–343.
[22] E. Jiménez-Ruiz, E. Kharlamov, D. Zheleznyakov, I. Horrocks, C. Pinkel, M. G. Skjæveland, E. Thorstensen, J. Mora, BootOX: Practical mapping of RDBs to OWL 2, in: The Semantic Web-ISWC 2015: 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II 14, Springer, 2015, pp. 113–132.
[23] M. A. Hazber, R. Li, G. Xu, K. M. Alalayah, An approach for automatically generating R2RML-based direct mapping from relational databases, in: Social Computing: Second International Conference of Young Computer Scientists, Engineers and Educators, ICYCSEE 2016, Harbin, China, August 20-22, 2016, Proceedings, Part I 2, Springer, 2016, pp. 151–169.
[24] Á. Sicilia, G. Nemirovski, AutoMap4OBDA: Automated generation of R2RML mappings for OBDA, in: Knowledge Engineering and Knowledge Management: 20th International Conference, EKAW 2016, Bologna, Italy, November 19-23, 2016, Proceedings 20, Springer, 2016, pp. 577–592.
[25] D. Chaves-Fraga, O. Corcho, F. Yedro, R. Moreno, J. Olías, A. De La Azuela, Systematic construction of knowledge graphs for research-performing organizations, Information 13 (2022) 562.
[26] A. Dimou, D. Kontokostas, M. Freudenberg, R. Verborgh, J. Lehmann, E. Mannens, S. Hellmann, R. Van de Walle, Assessing and refining mappings to rdf to improve dataset quality, in: The Semantic Web-ISWC 2015: 14th International Semantic Web Conference, Bethlehem, PA, USA, October 11-15, 2015, Proceedings, Part II 14, Springer, 2015, pp. 133–149.
[27] A. Dimou, D. Chaves-Fraga, Declarative description of knowledge graphs construction automation: Status & challenges, in: Proceedings of the 3rd International Workshop on Knowledge Graph Construction (KGCW 2022) co-located with 19th Extended Semantic Web Conference (ESWC 2022), volume 3141, 2022.
[28] D. Chaves-Fraga, F. Priyatna, A. Cimmino, J. Toledo, E. Ruckhaus, O. Corcho, GTFS-Madrid-Bench: A benchmark for virtual knowledge graph access in the transport domain, Journal of Web Semantics 65 (2020) 100596.