Exchanging Data amongst Linked Data applications

**Carlos R. Rivero, Inma Hernández,
David Ruiz, Rafael Corchuelo**

Abstract

The goal of data exchange is to populate the data model of a target application using data that come from one or more source applications. It is common to address data exchange building on correspondences that are transformed into executable mappings. The problem that we address in this article is how to generate executable mappings in the con-text of Linked Data applications, that is, applications whose data models are semantic-web ontologies. In the literature, there are many proposals to generate executable mappings. Most of them focus on relational or nested-relational data models, which cannot be applied to our context; unfortunately, the few proposals that focus on ontologies have important drawbacks, namely: they solely work on a subset of taxonomies, they require the target data model to be pre-populated or they interpret correspondences in isolation, not to mention the propos-als that actually require the user to handcraft the executable mappings. In this article, we present MostoDE, a new automated proposal to generate SPARQL executable mappings in the context of Linked Data applications. Its salient features are that it does not have any of the previous drawbacks, it is computationally tractable and it has been validated using a series of experiments that prove that it is very efficient and effective in practice.

Keywords

Knowledge and data engineering · Data exchange · Linked Data · Executable mappings · SPARQL

C. R. Rivero (✉) · I. Hernández · D. Ruiz · R. Corchuelo
University of Sevilla, ETSI Informática, Avda. Reina Mercedes, s/n, 41012 Seville, Spain
e-mail: carlosrivero@us.es

I. Hernández
e-mail: inmahernandez@us.es

D. Ruiz
e-mail: druiz@us.es

R. Corchuelo
e-mail: corchu@us.es

1 Introduction

As data become pervasive, there is an increasing need to integrate them [5,7,9,37]. Integration is a common term by means of which researchers refer to a variety of problems, including: data integration, which deals with answering queries posed over a target data model using a number of source data models only [24,31]; model matching, aka model mapping, whose goal is to unveil correspondences amongst the entities of two data models [17,49]; model evolution, which focuses on the modifications of an existing data model in response to a need for change [19,20,38]; or data exchange, which aims to populate the data model of a target application using data that come from one or more source applications [4,16,18,33,34,41,45,47].

This article focuses on data exchange, which has been paid much attention in the fields of relational and nested-relational data models [4,18,45]. Recently, with the emergence of the Semantic Web [57] and Linked Data applications [9,22], the problem is motivating many authors to work on data exchange in the context of ontologies [16,33,34,41,47], that is, data models that are represented using RDF, RDFS or OWL and queried using SPARQL [3,26,57].

Without an exception, data exchange has been addressed using mediators, which, in turn, rely on so-called mappings [17,36]. In the literature, it is common to distinguish between two kinds of mappings: correspondences and executable mappings [17,45,49,51]. A correspondence is a hint that specifies which entities in the source and target data models correspond to each other, that is, are somewhat related [17,49]; an executable mapping, aka operational mapping, is an executable artefact that encodes how the correspondences must be interpreted, that is, how to perform data exchange [45,51]. (By executable artefact, we mean a SPARQL query, a Datalog rule, an XSLT script or other means to read, transform and output data). Note that correspondences are inherently ambiguous since there can be many different executable mappings that satisfy them, but generate different target data [2,8,16,45].

Creating mappings automatically is appealing because this relieves users from the burden of handcrafting them [7,43,48,49]. Regarding correspondences, the literature provides a number of proposals that help generate them (semi-) automatically [15,49], even in the context of ontologies [12,13,17]. Generating executable mappings automatically in the context of relational and nested-relational data models has been studied extensively [2,23,45]. Unfortunately, the proposals in this context are not applicable to ontologies due to the inherent differences amongst these data models [35,38,51]. This has motivated several authors to work on proposals that are specifically tailored towards ontologies, namely: Mergen and Heuser [33] presented a proposal that can deal with only a subset of taxonomies; Qin et al. [47] devised a proposal that requires the target model to be pre-populated; the proposal by Mocan and Cimpian [34] may lead to incoherent target data since it interprets correspondences in isolation, without taking their interrelationships into account; Dou et al. [16] and Parreiras et al. [41] presented two proposals that require the user to handcraft the executable mappings.

In this article, we present a proposal called MostoDE to automatically generate SPARQL executable mappings in the context of ontologies that rely on quite a complete subset of the OWL 2 Lite profile. The salient features of our proposal are as follows: it does not suffer from any of the problems we mentioned before regarding other proposals in the literature; it is computationally tractable, since $O(e_s^4\, e_t + e_s\, e_t^4 + e_s^2\, e_t^2(e_s + e_t))$ is an upper bound to its worst-time complexity, where e_s and e_t denote the number of entities in the source and target ontologies, respectively; furthermore, it has been validated using 3,783 non-trivial experiments in which the time to execute our algorithms never exceeded one second, and the exchanged data were as expected by experts in every case. These results suggest that it is

very efficient in practice and that the interpretation of correspondences that our executable mappings encode is appropriate. We presented a preliminary 14-page version of these results in [51]; in that version, we did not take incomplete data exchange problems into account (cf. Sect. 4.4), which, together with our formalisation, analysis of complexity and validation, constitute the major differences.

The rest of the article is organised as follows: in Sect. 2, we report on several related proposals and compare them with ours; Sect. 3 introduces the conceptual framework on which our proposal relies; Sect. 4 presents our algorithms; in Sect. 5, we first prove that they are correct and then prove that they are computationally tractable; in Sect. 6, we present the results of our validation; finally, we present our conclusions in Sect. 7. Appendix 8 characterises the subset of the OWL 2 Lite profile with which our proposal can deal.

2 Related work

The initial work on data exchange in the context of ontologies put the emphasis on defining the problem and the strategies to address it, which mainly consisted in devising handcrafted correspondences and performing data exchange using ad hoc techniques [32,39]. Later proposals performed data exchange using reasoners [16,34,56] or SPARQL query engines [41,44,51] instead of ad hoc techniques.

The majority of current proposals in the literature rely on correspondences and executable mappings that interpret them; there are a few proposals, however, that require the user to provide executable mappings using ad hoc languages. Correspondences can be handcrafted using visual tools [1,48,50], or discovered automatically using model matching proposals [13, 12,15,17,30,40,49]. In either case, finding them is orthogonal to the problem of generating executable mappings, which is the reason why we do not discuss them further; in the sequel, we assume that a set of correspondences is available. The most closely related proposals in the field of ontologies were presented by Mergen and Heuser [33], Qin et al. [47], Mocan and Cimpian [34], Dou et al. [16] and Parreiras et al. [41]; Popa et al. [45] presented the state-of-the-art proposal in the context of nested-relational data models. Below, we report on these proposals and compare them to ours.

Mergen and Heuser [33] devised an automated proposal that works with a subset of taxonomies in which there are only classes, data properties and single specialisations amongst classes. Their algorithm analyses every class correspondence independently and tries to find the set of correspondences that are involved in its properties and superclasses; this helps identify data that must be exchanged together and the many possible exchanges that can be performed. These subsets of correspondences are then translated into an ad hoc script language that was devised by the authors. Contrarily, our proposal can deal with quite a complete subset of ontologies that adhere to the OWL 2 Lite profile, which allows for object properties and multiple specialisations between classes and properties. Furthermore, our proposal generates executable mappings that are represented in standard SPARQL, which allows to perform data exchange using a SPARQL engine.

Qin et al. [47] devised a semi-automatic proposal that relies on data mining. They first require the user to select a subset of source and target data for each data property; these are used to feed a mining algorithm that attempts to discover a set of queries that can exchange these data; these queries are then sorted according to an ad hoc metric, and the top ones are selected and transformed into Datalog rules, SWRL rules or Web-PDDL, an ad hoc language that was designed by the authors. The most important problem with this proposal is that it requires the target data model to be pre-populated so that the data mining algorithm can work,

which is not the case in common data exchange problems [2,7,18,45]; if the target cannot be pre-populated with actual data, then the user must provide representative-enough synthetic data. Furthermore, it requires the user to select appropriate subsets of data from both the source and target data models, that is, subsets of data that capture variability well enough; this is not a trivial task since the data have to be selected very carefully to avoid over-fitting, that is, generating executable mappings that can deal with only the selected training data. Contrarily, our proposal does not require the target data model to be pre-populated, and it does not require the user to provide any additional data to compute the resulting executable mappings.

Mocan and Cimpian [34] studied the problem of data exchange in the context of semantic-web services. They presented a formal framework to describe correspondences in terms of first-order logic formulae that can be mapped onto WSML rules very easily. Their proposal is similar in spirit to the one by Omelayenko [39], whose focus was on B2B applications, and the one by Maedche et al. [32], whose focus was on modelling correspondences in a general-purpose setting. The main difference with the previous proposals is that Mocan and Cimpian went a step beyond formalising correspondences and devised a mediator that executes them using a WSML reasoner. Note that no attempt is made to identify groups of correspondences that must be taken into account together, which may easily lead to incoherent target data, that is, data that do not satisfy the constraints in the target ontology [2,8,45]. Contrarily, our proposal identifies groups of correspondences that must be analysed together, which impedes the resulting executable mappings from producing incoherent data.

Parreiras et al. [41] presented a proposal within the framework of model-driven engineering. They extended the active template library (ATL) metamodel to support OWL 1 ontologies, which allows to express constraints on them using the object constraint language (OCL). They devised a mapping language called MBOLT by means of which users can express executable mappings that are later transformed into SPARQL and Java by means of a library of ATL transformations. Their proposal does not build on correspondences, but requires users to handcraft and maintain their executable mappings using MBOLT. This is similar in spirit to the proposal by Dou et al. [16]; the difference is the language used to represent the executable mappings. Contrarily to these proposals, ours is able to compute the executable mappings automatically, which saves the user from the burden of designing them.

The previous proposals have problems that hinder their applicability in practice: Mergen and Heuser [33] deal with only a subset of taxonomies, Qin et al. [47] require the target model to be pre-populated, Mocan and Cimpian [34] interpret correspondences in isolation, and Dou et al. [16] and Parreiras et al. [41] require the user to handcraft executable mappings. This motivated us to work on a new proposal to overcome these limitations. Our first step was to study the literature on data exchange in the context of relational, nested-relational and XML data models. Relational data models are a special case of nested-relational data models, which is the reason why we do not provide any further details [45]. A nested-relational data model is defined by means of a tree that comprises a number of nodes, which may be nested and have a number of attributes. Furthermore, it is also possible to specify referential constraints that relate attributes in this model. Fagin et al. [18] and Arenas and Libkin [4] presented the theoretical foundations for performing data exchange using executable mappings in the context of relational and XML data models, respectively.

Popa et al. [45] devised the state-of-the-art proposal regarding data exchange in nested-relational data models. Although it seems efficient and effective enough to be used in practical applications, it cannot be applied in the context of ontologies. The main reason is that it builds on computing primary paths from the root node of a data model to every attribute

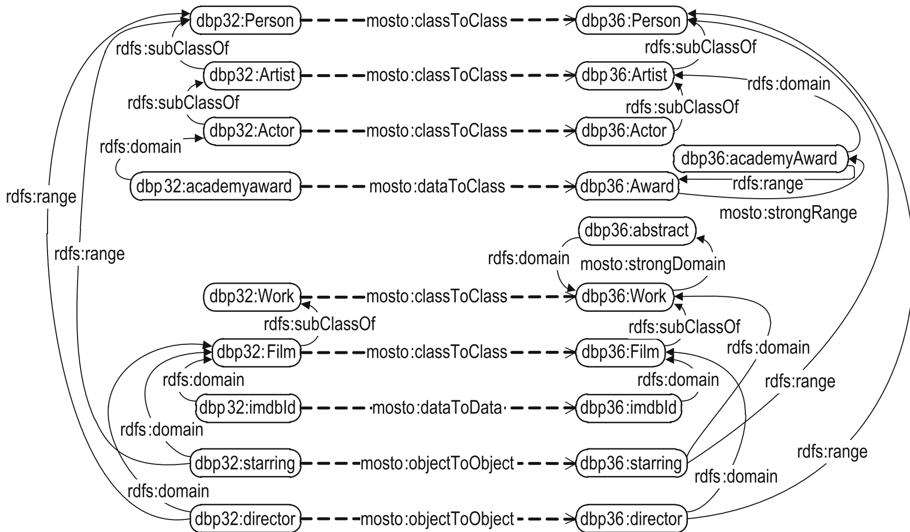

Fig. 1 A running example: exchanging data from DBpedia 3.2 to DBpedia 3.6

in this model and then applying a variation of the well-known Chase algorithm that takes referential constraints into account [14]. In general, an ontology is not a tree, but a graph in which there is not a root node, which prevents us from computing primary paths, and it can contain cycles, which prevents us from using the Chase algorithm. There are more differences between nested-relational data models and ontologies [35,38,51]. One of the most important is that executable mappings are encoded using XQuery or XSLT in the nested-relational context; these languages build on the structure of the XML documents on which they are executed; contrarily, in an ontology, these mappings must be encoded in a language that is independent from the structure of the documents used to represent it, since the same ontology may be serialised to multiple languages, e.g., XML, N3 or Turtle.

3 Conceptual framework

In this section, we present the conceptual framework that we use to describe our proposal. We first define its foundations, triples, ontologies, executable mappings and data exchange problems; then, we define precisely what satisfying a constraint or a correspondence means. To illustrate our proposal, we use a non-trivial, real-world data exchange problem that is based on the evolution from DBpedia 3.2 to DBpedia 3.6 [10], which involved many major structural changes (cf. Fig. 1).

3.1 Foundations

Entities lay at the heart of every ontology, and they are denoted by means of URIs. Entities can be classified into classes and properties; the latter can be further classified into data properties and object properties. We denote the sets of classes, data properties and object properties as follows:

$$[Class, DataProperty, ObjectProperty]$$

and define the set of entities and properties as follows:

$$Entity == Class \cup Property$$
$$Property == DataProperty \cup ObjectProperty$$

Ontologies also build on resources, which refer to everything that can be identified by means of a URI, literals, which denote values of simple data types, and blank nodes, which denote anonymous data. We denote these sets as follows:

$$[Resource,\ Literal,\ BlankNode]$$

$Entity$ is a subset of $Resource$, and $Resource$, $Literal$, and $BlankNode$ are pairwise disjoint sets [28]:

$$\begin{array}{|l} Entity \subseteq Resource \\ (Resource \cap Literal = \emptyset) \wedge (Resource \cap BlankNode = \emptyset) \wedge \\ \quad (Literal \cap BlankNode = \emptyset) \end{array}$$

Example 1 In Fig. 1, we denote entities as rounded boxes, for example, $dbp32{:}Person$ is a class, $dbp32{:}academyaward$ is a data property, $dbp32{:}starring$ is an object property, and they all are represented by means of URIs. We use the W3C notation to represent literals and blank nodes; for instance, '$Best\ Actor$'^^$xsd{:}string$ is a sample literal, and _:$BestActorAward$ is a sample blank node.

3.2 Triples

Triples are three-tuples in which the first element is called subject, the second predicate and the third object. They help describe both the structure and data of ontologies. We define the set of all triples as follows:

$$Triple == Subject \times Predicate \times Object$$
$$Subject == Resource \cup Literal \cup BlankNode$$
$$Predicate == Property \cup BuiltInConstruct$$
$$Object == Resource \cup Literal \cup BlankNode$$

Note that there is a contradiction between the RDF and the SPARQL recommendations. On the one hand, the RDF recommendation does not allow to use literals in the subject of a triple [28]; on the other hand, the SPARQL recommendation actually allows to use literals in the subject of a triple [46]. Therefore, we can build triples in which the subject is a literal using SPARQL. Since our research focuses on SPARQL queries, we have decided to include literals in the subject of triples.

Note, too, that a predicate can be a property or a built-in construct. By built-in construct, we mean a URI that denotes one of the predefined RDF, RDFS and Mosto constructs with which we deal. (In the sequel, we use prefix *mosto*: to refer to the constructs that we have defined in our proposal). We define these sets as follows:

$$\begin{aligned} BuiltInConstruct == {} & \{rdf{:}type\} \cup ConstraintConstruct \cup \\ & CorrespondenceConstruct \\ ConstraintConstruct == {} & \{rdfs{:}subClassOf, rdfs{:}subPropertyOf, \\ & rdfs{:}domain, rdfs{:}range, \end{aligned}$$

$$mosto{:}strongDomain, mosto{:}strongRange\}$$

$$CorrespondenceConstruct == \{mosto{:}classToClass, mosto{:}dataToData, mosto{:}objectToObject, mosto{:}dataToClass\}$$

ConstraintConstruct does not actually include every possible construct in the OWL 2 Lite profile, but the minimal subset of constraints with which we deal. Section 3.6 provides additional details regarding the semantics of this subset of constraints, and 'Appendix 8' provides additional details on how to translate other constructs into this minimal subset.

Set *CorrespondenceConstruct* includes the correspondence constructs with which we can deal. They allow to establish correspondences from classes to classes, data properties to data properties, object properties to object properties and data properties to classes. Section 3.7 provides additional details regarding the semantics of these constructs.

For the sake of convenience, we also define the following subsets of triples:

$$Constraint == Subject \times ConstraintConstruct \times Object$$
$$Correspondence == Subject \times CorrespondenceConstruct \times Object$$

Example 2 The following are examples of triples:

$$\begin{array}{lll}(dbp32{:}starring, & rdfs{:}domain, & dbp32{:}Film) \\ (dbp32{:}Unforgiven, & rdf{:}type, & dbp32{:}Film) \\ (dbp32{:}Unforgiven, & dbp32{:}starring, & dbp32{:}ClintEastwood) \\ (dbp32{:}starring, & mosto{:}objectToObject, & dbp36{:}starring)\end{array}$$

The first triple asserts that $dbp32{:}starring$ is an object property whose domain is class $dbp32{:}Film$; the second triple asserts that $dbp32{:}Unforgiven$ is of type $dbp32{:}Film$; the third triple asserts that $dbp32{:}Unforgiven$ is related to $dbp32{:}ClintEastwood$ by means of object property $dbp32{:}starring$; the fourth triple asserts that object property $dbp32{:}starring$ in DBPedia 3.2 is related to object property $dbp36{:}starring$ in DBPedia 3.6.

In Fig. 1, we represent constraint constructs by means of labelled arrows and correspondences by means of labelled dashed arrows. Note that data properties like $dbp32{:}academyaward$ do not have an explicit range since the OWL 2 Lite profile states that the object of a triple with the $rdfs{:}range$ construct must be a class [6]; data properties are then implicitly assumed to range over the set of literals.

3.3 Ontologies

An ontology is a representation of a data model, which comprises a description of the structure of the data and the data themselves. For the purpose of this article, we just need the description of the structure. An ontology can thus be defined by means of a two tuple that consists of a set of entities and a set of constraints. We then define the set of all ontologies as follows, where $\mathbb{P}$ denotes powerset:

$$Ontology == \{E : \mathbb{P}Entity; C : \mathbb{P}Constraint \mid \forall s : Subject; p : ConstraintConstruct; o : Object \cdot (s, p, o) \in C \Rightarrow \{s, o\} \subseteq E\}$$

For the sake of convenience, we define the following projection functions:

$$
\begin{array}{|l}
entities : Ontology \rightarrow \mathbb{P}\, Entity \\
constraints : Ontology \rightarrow \mathbb{P}\, Constraint \\
\hline
\forall E : \mathbb{P}\, Entity; C : \mathbb{P}\, Constraint; o : Ontology \mid o = (E, C) \cdot \\
\quad entities(o) = E \land \\
\quad constraints(o) = C
\end{array}
$$

Furthermore, we introduce the concept of path in an ontology, which comprises a sequence of one or more entities:

$$Path == \mathrm{seq}_1\ Entity$$

Given an ontology, we may wish to compute the set of all paths in that ontology, which is formally defined by the following function:

$$
\begin{array}{|l}
paths : Ontology \rightarrow \mathbb{P}\, Path \\
\hline
\forall o : Ontology \cdot \\
\quad paths(o) = \{p : Path \mid \\
\quad\quad (\forall i : \{1..\#p\} \cdot p(i) \in entities(o)) \land \\
\quad\quad (\forall i : \{1..\#p - 1\} \cdot \exists c : Constraint \mid c \in constraints(o) \cdot \\
\quad\quad\quad subject(r) = p(i) \land object(r) = p(i + 1))
\end{array}
$$

An ontology is connected if there is a path between any two entities, or there exists an intermediate entity that is connected to them by means of two paths. This is formally defined by the following predicate:

$$
\begin{array}{|l}
connected : Ontology \\
\hline
\forall o : Ontology \cdot \\
\quad connected(o) \Leftrightarrow \forall e_1, e_2 : Entity \mid \{e_1, e_2\} \subseteq entities(o) \cdot \\
\quad\quad (\exists p : Path \mid p \in paths(o) \cdot \\
\quad\quad\quad first(p) = e_1 \land last(p) = e_2 \lor first(p) = e_2 \land last(p) = e_1) \lor \\
\quad\quad (\exists p_1, p_2 : Path \mid p_1 \neq p_2 \land \{p_1, p_2\} \subseteq paths(o) \cdot \\
\quad\quad\quad last(p_1) = e_1 \land last(p_2) = e_2 \land first(p_1) = first(p_2))
\end{array}
$$

Example 3 In Fig. 1, the ontology on the left is not connected, for example, there are not any paths that connect entities $dbp32{:}academyawards$ and $dbp32{:}imdbId$, and there does not exist another entity that has a path to them both. Note, however, that the ontology formed by entities $dbp32{:}academyawards$ and $dbp32{:}Actor$, together with their $rdfs{:}domain$ constraint is connected.

3.4 Executable mappings

In our proposal, executable mappings are SPARQL conjunctive queries of the construct type. Such queries consist of a construct clause that specifies which triples have to be constructed in the target ontology, and a where clause that specifies which triples should be retrieved from the source ontology. These clauses can then be represented as sets of triple patterns that are implicitly connected by means of a logical. A triple pattern generalises the concept of triple by allowing the subject, the predicate and/or the object to be variables. In the rest of this article, we refer to triple patterns as patterns for the sake of brevity. We denote the set of all variables as follows:

$$[Variable]$$

```
CONSTRUCT {
  ?x rdf:type dbp36:Work .
  ?x rdf:type dbp36:Film .
  ?x dbp36:imdbId ?y .
  ?x dbp36:abstract _:a .
}
WHERE {
  ?x rdf:type dbp32:Work .
  ?x rdf:type dbp32:Film .
  ?x dbp32:imdbId ?y .
}
```

Fig. 2 A sample executable mapping

and define the set of all patterns as follows:

$$\begin{aligned} Pattern &== Subject^? \times Predicate^? \times Object^? \\ Subject^? &== Subject \cup Variable \\ Predicate^? &== Predicate \cup Variable \\ Object^? &== Object \cup Variable \end{aligned}$$

An executable mapping can thus be represented as a two tuple in which the first component corresponds to the set of patterns in the construct clause, and the second component corresponds to the set of patterns in the where clause. We then define the set of all executable mappings as follows:

$$ExecutableMapping == \mathbb{P}Pattern \times \mathbb{P}Pattern$$

For the sake of convenience, we define an instance of a class as a pattern of the form: $(s, rdf{:}type, c)$, in which $s \in Subject^?$; $c \in Class$; and an instance of a property as a pattern of the form: (s, p, o), in which $s \in Subject^?$; $p \in Property$; $o \in Object^?$. Furthermore, we define the following projection functions:

$$\begin{array}{|l} subject : Pattern \rightarrow Subject^? \\ predicate : Pattern \rightarrow Predicate^? \\ object : Pattern \rightarrow Object^? \\ \hline \forall s : Subject^?; p : Predicate^?; o : Object^?; t : Pattern \mid t = (s, p, o) \cdot \\ \quad subject(t) = s \wedge \\ \quad predicate(t) = p \wedge \\ \quad object(t) = o \end{array}$$

Note that $Triple \subseteq Pattern$, which implies that instances may refer to both triples and patterns, and that the previous projection functions can be applied to both triples and patterns.

Example 4 Figure 2 shows a sample executable mapping that comprises four patterns in the construct clause and three patterns in the where clause. In this example, $?x$ and $?y$ are variables, and _:a is a blank node. This executable mapping retrieves source instances whose type is $dbp32{:}Work$ and $dbp32{:}Film$, which are related to the instances of $dbp32{:}imdbId$; it builds target instances of type $dbp36{:}Work$ and $dbp36{:}Film$, which are related to the instances of $dbp36{:}imdbId$ and $dbp36{:}abstract$. Note that the object of the instance of property $dbp32{:}imdbId$ is copied to the object of the instance of property $dbp36{:}imdbId$;

however, the object of the instance of property $dbp36{:}abstract$ does not exist in the source; therefore, we use a blank node to generate anonymous data.

3.5 Data exchange problems and kernels

A data exchange problem consists of a source ontology, a target ontology and a set of correspondences between some of their entities. We define the set of all data exchange problems as follows:

$$\begin{aligned} DataExchangeProblem == \{ & o_1, o_2 : Ontology; V : \mathbb{P}\ Correspondence \mid \\ & \forall v : Correspondence \cdot v \in V \Rightarrow \\ & \quad subject(v) \in entities(o_1) \wedge \\ & \quad object(v) \in entities(o_2)\} \end{aligned}$$

For the sake of convenience, we define the following projection functions:

$$\begin{array}{|l} source : DataExchangeProblem \to Ontology \\ target : DataExchangeProblem \to Ontology \\ correspondences : DataExchangeProblem \to \mathbb{P}\ Correspondence \\ \hline \forall o_1, o_2 : Ontology; V : \mathbb{P}\ Correspondence; d : DataExchangeProblem \mid \\ \quad d = (o_1, o_2, V) \cdot \\ \quad\quad source(d) = o_1 \wedge \\ \quad\quad target(d) = o_2 \wedge \\ \quad\quad correspondences(d) = V \end{array}$$

Given two data exchange problems, it may be necessary to determine if the former is contained into the latter, that is, if the entities, constraints and correspondences of the former are included in the entities, constraints and correspondences of the latter. We formally define the containment relation as follows:

$$\begin{array}{|l} _ \sqsubseteq _ : DataExchangeProblem \times DataExchangeProblem \\ \hline \forall d_1, d_2 : DataExchangeProblem \mid \\ \quad d1 \sqsubseteq d_2 \Leftrightarrow \\ \quad\quad entities(source(d_1)) \subseteq entities(source(d_2)) \wedge \\ \quad\quad entities(target(d_1)) \subseteq entities(target(d_2)) \wedge \\ \quad\quad constraints(source(d_1)) \subseteq constraints(source(d_2)) \wedge \\ \quad\quad constraints(target(d_1)) \subseteq constraints(target(d_2)) \wedge \\ \quad\quad correspondences(d_1) \subseteq correspondences(d_2) \end{array}$$

A kernel describes the structure of a subset of data in the source ontology that needs to be exchanged as a whole, and the structure of a subset of data in the target ontology that needs to be created as a whole; in other words, if more or less data are considered, then the exchanged data would not satisfy target constraints and/or correspondences. Intuitively, a kernel is contained in a given data exchange problem, its source and target ontologies are connected, and there are not any source constraints, or target constraints, or correspondences in the data exchange problem that can be added to it.

At a first glance, it might be surprising that we take source constraints into account, since source data should ideally satisfy them; however, in the decentralised environment of Linked

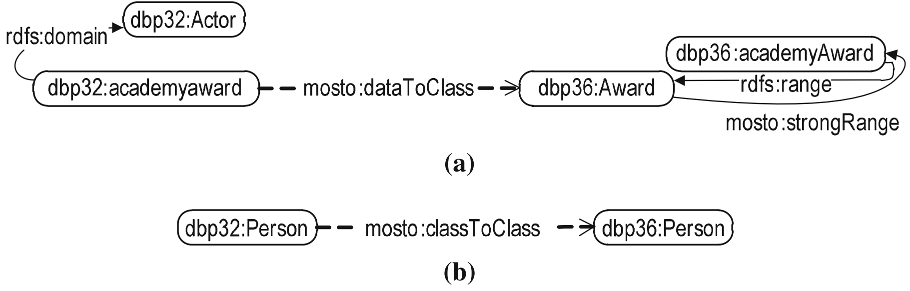

Fig. 3 Sample data exchange problems. **a** A sample data exchange problem that is not a kernel. **b** A sample kernel

Data applications, it is common that real-world ontologies have data that do not satisfy their constraints [11].

We define the function that identifies the kernels of a given data exchange problem as follows:

Example 5 Figure 1 shows a sample data exchange problem that is not a kernel, since its source and target ontologies are not connected, for example, there are not any paths that connect entities $dbp32{:}academyawards$ and $dbp32{:}imdbId$. Figure 3a shows a data exchange problem that is contained in the data exchange problem in Fig. 1; it is connected, but it is not a kernel since it includes entity $dbp32{:}Actor$ but does not include constraint $(dbp32{:}Actor, rdfs{:}subClassOf, dbp32{:}Artist)$. Furthermore, Fig. 3b shows a sample kernel since it is contained in the data exchange problem in Fig. 1, both the source and target ontologies are connected, and there are not any further entities, constraints or correspondences that can be added to this problem.

$$
\begin{array}{l}
\mathit{kernels} : \mathit{DataExchangeProblem} \rightarrow \mathbb{P}\, \mathit{DataExchangeProblem} \\
\hline
\forall d : \mathit{DataExchangeProblem} \cdot \\
\quad \mathit{kernels}(d) = \{k : \mathit{DataExchangeProblem} \mid \\
\quad\quad k \sqsubseteq d \wedge \mathit{connected}(\mathit{source}(k)) \wedge \mathit{connected}(\mathit{target}(k)) \wedge \\
\quad\quad (\nexists c_1 : \mathit{Constraint} \mid c_1 \in \mathit{constraints}(\mathit{source}(d)) \cdot \\
\quad\quad\quad c_1 \notin \mathit{constraints}(\mathit{source}(k)) \wedge \mathit{subject}(c_1) \in \mathit{entities}(\mathit{source}(k))) \wedge \\
\quad\quad (\nexists c_2 : \mathit{Constraint} \mid c_2 \in \mathit{constraints}(\mathit{target}(d)) \cdot \\
\quad\quad\quad c_2 \notin \mathit{constraints}(\mathit{target}(k)) \wedge \mathit{subject}(c_2) \in \mathit{entities}(\mathit{target}(k))) \wedge \\
\quad\quad (\nexists v : \mathit{Correspondence} \mid v \in \mathit{correspondences}(d) \cdot \\
\quad\quad\quad v \notin \mathit{correspondences}(k) \wedge \mathit{subject}(v) \in \mathit{entities}(\mathit{source}(k))) \wedge \\
\quad\quad\quad \mathit{object}(v) \in \mathit{entities}(\mathit{target}(k)))\}
\end{array}
$$

3.6 Satisfaction of constraints

Given an executable mapping, it is important to know whether it satisfies a given constraint or not. This prevents us from retrieving or constructing data that does not satisfy the constraints in the source or the target of a data exchange problem, respectively. In the following paragraphs, we describe how to infer whether a set of patterns in the construct or the where clause of an executable mapping satisfies a given constraint or not. We use symbol $\models$ to denote satisfaction, T to denote a set of patterns, c, c_1, and c_2 to denote classes, and p, p_1 and p_2 to denote properties.

Subclassification constraints: A set of patterns satisfies a subclass or a subproperty constraint if, for every instance of a given class or property, we may find another instance of the corresponding subclass or subproperty, respectively. The following inference rules formalise this idea:

$$[\text{SC}]\quad \frac{\forall s : Subject^? \cdot (s, rdf{:}type, c_1) \in T \Rightarrow (s, rdf{:}type, c_2) \in T}{T \models (c_1, rdfs{:}subClassOf, c_2)}$$

$$[\text{SP}]\quad \frac{\forall s : Subject^?; o : Object^? \cdot (s, p_1, o) \in T \Rightarrow (s, p_2, o) \in T}{T \models (p_1, rdfs{:}subPropertyOf, p_2)}$$

Domain and range constraints: A set of patterns satisfies a domain or a range constraint regarding a property if, for every property instance that states that a subject is related to an object, there are additional class instances that state that the type of the subject or the object are the appropriate classes, respectively. The following inference rules formalise this idea:

$$[\text{D}]\quad \frac{\forall s : Subject^?; o : Object^? \cdot (s, p, o) \in T \Rightarrow (s, rdf{:}type, c) \in T}{T \models (p, rdfs{:}domain, c)}$$

$$[\text{R}]\quad \frac{\forall s : Subject^?; o : Object^? \cdot (s, p, o) \in T \Rightarrow (o, rdf{:}type, c) \in T}{T \models (p, rdfs{:}range, c)}$$

Strong domain and strong range constraints: We deal with two additional constraints to which we refer to as strong domain ($mosto{:}strongDomain$) and strong range ($mosto{:}strongRange$). Intuitively, if a class is the strong domain of a property, that means that this property has a minimum cardinality of one regarding that class; similarly, if a class is the strong range of a property, that means that for every instance of that class, there must be a subject that is related to that instance by means of the property. Note that these semantics are expressed in OWL 2 as a combination of several triples. For the sake of simplicity, we introduced $mosto{:}strongDomain$ and $mosto{:}strongRange$ as shorthands, cf. 'Appendix 8' for further details. The following inference rules formalise the satisfaction of these constraints:

$$[\text{SD}]\quad \frac{\forall s : Subject^? \cdot (s, rdf{:}type, c) \in T \Rightarrow \exists o : Object^? \cdot (s, p, o) \in T}{T \models (c, mosto{:}strongDomain, p)}$$

$$[\text{SR}]\quad \frac{\forall o : Object^? \cdot (o, rdf{:}type, c) \in T \Rightarrow \exists s : Subject^? \cdot (s, p, o) \in T}{T \models (c, mosto{:}strongRange, p)}$$

Example 6 To illustrate the satisfaction of constraints, we use the following sample triples:

$$\begin{array}{lll}
(dbp32{:}ClintEastwood, & rdf{:}type, & dbp36{:}Artist) \\
(dbp32{:}ClintEastwood, & rdf{:}type, & dbp36{:}Actor) \\
(dbp32{:}Unforgiven, & rdf{:}type, & dbp36{:}Work) \\
(_{:}BestActorAward, & rdf{:}type, & dbp36{:}Award) \\
(dbp32{:}Unforgiven, & dbp36{:}starring, & dbp32{:}ClintEastwood)
\end{array}$$

Every instance of type $dbp36{:}Actor$ is also of type $dbp36{:}Artist$; every instance of property $dbp36{:}starring$ has an instance of type $dbp36{:}Work$ as subject. Therefore, the following constraints are satisfied by the previous set of triples:

$$\begin{array}{lll}
(dbp36{:}Actor, & rdfs{:}subClassOf, & dbp36{:}Artist) \\
(dbp36{:}starring, & rdfs{:}domain, & dbp36{:}Work)
\end{array}$$

Additionally, there are not any instances of type $dbp36{:}Person$; the object of the instance of property $dbp36{:}starring$ is not of type $dbp36{:}Person$; furthermore, blank

node $_:BestActorAward$ is not related to any instance of property $dbp36{:}academyAward$. As a conclusion, the following constraints are not satisfied in the previous set of triples:

$$
\begin{array}{lll}
(dbp36{:}Artist, & rdfs{:}subClassOf, & dbp36{:}Person) \\
(dbp36{:}starring, & rdfs{:}range, & dbp36{:}Person) \\
(dbp36{:}Award, & mosto{:}strongRange, & dbp36{:}academyAward)
\end{array}
$$

3.7 Satisfaction of correspondences

Given an executable mapping, it is also important to know whether it satisfies a given correspondence or not. This prevents the executable mapping from constructing data that do not satisfy the correspondences in a data exchange problem.

In the next paragraphs, we describe how to infer whether an executable mapping satisfies a correspondence or not. We use symbol $\models$ to denote satisfaction, T_C to denote a set of patterns in a construct clause, T_W to denote a set of patterns in a where clause, c, c_1 and c_2 to denote classes, p, p_1 and p_2 to denote data properties, and q_1 and q_2 to denote object properties.

Class-to-class correspondences: These correspondences specify that an instance of a class in the source has to be reclassified in the target. We formalise this idea as follows:

$$
[\text{C2C}] \quad \frac{\forall s : Subject^? \cdot (s, rdf{:}type, c_1) \in T_W \Rightarrow (s, rdf{:}type, c_2) \in T_C}{(T_C, T_W) \models (c_1, mosto{:}classToClass, c_2)}
$$

Data property to data property correspondences: Such a correspondence specifies that, if there is an instance of a data property in the source, then there must be another instance of the corresponding data property in the target that must have the same object. Note that the correspondence itself cannot specify what the subject is. This idea is formalised as follows:

$$
[\text{D2D}] \quad \frac{\forall s : Subject^?; o : Object^? \cdot (s, p_1, o) \in T_W \Rightarrow \exists s' : Subject^? \cdot (s', p_2, o) \in T_C}{(T_C, T_W) \models (p_1, mosto{:}dataToData, p_2)}
$$

Object property to object property correspondences: These correspondences specify that, for every instance of an object property in the source, there must exist another instance of the corresponding object property in the target. Note that the correspondence itself cannot specify which the subjects and the objects are. This idea is described formally as follows:

$$
[\text{O2O}] \quad \frac{\forall s : Subject^?; o : Object^? \cdot (s, q_1, o) \in T_W \Rightarrow \exists s' : Subject^?; o' : Object^? \cdot (s', q_2, o') \in T_C}{(T_C, T_W) \models (q_1, mosto{:}objectToObject, q_2)}
$$

Data property to class correspondences: A correspondence of this kind specifies that the object of every instance of a data property in the source must be reclassified in the target. The following inference rule formalises this idea:

$$
[\text{D2C}] \quad \frac{\forall s : Subject^?; o : Object^? \cdot (s, p, o) \in T_W \Rightarrow (o, rdf{:}type, c) \in T_C}{(T_C, T_W) \models (p, mosto{:}dataToClass, c)}
$$

Example 7 To illustrate the satisfaction of correspondences, we use the following sample triples in the where clause:

$$
\begin{array}{lll}
(dbp32{:}ClintEastwood, & rdf{:}type, & dbp32{:}Person) \\
(dbp32{:}ClintEastwood, & rdf{:}type, & dbp32{:}Artist) \\
(dbp32{:}Unforgiven, & dbp32{:}imdbId, & \text{`0105695'}\hat{}\hat{}xsd{:}string) \\
(dbp32{:}PaleRider, & dbp32{:}imdbId, & \text{`0089767'}\hat{}\hat{}xsd{:}string) \\
(dbp32{:}Unforgiven, & dbp32{:}starring, & dbp32{:}ClintEastwood) \\
(dbp32{:}Unforgiven, & dbp32{:}director, & dbp32{:}ClintEastwood) \\
(dbp32{:}ClintEastwood, & dbp32{:}academyawards, & dbp32{:}BestActorAward)
\end{array}
$$

And the following sample triples in the construct clause:

$$\begin{array}{lll}
(dbp32{:}ClintEastwood, & rdf{:}type, & dbp36{:}Artist) \\
(dbp32{:}PaleRider, & dbp36{:}imdbId, & \text{`0105695'\^{}\^{}}xsd{:}string) \\
(dbp32{:}PaleRider, & dbp36{:}imdbId, & \text{`0089767'\^{}\^{}}xsd{:}string) \\
(dbp32{:}Unforgiven, & dbp36{:}starring, & dbp32{:}ClintEastwood) \\
(dbp32{:}BestActorAward, & rdf{:}type, & dbp36{:}Award)
\end{array}$$

Every instance of type $dbp32{:}Artist$ in the where clause is reclassified as an instance of type $dbp36{:}Artist$ in the construct clause; for every instance of property $dbp32{:}imdbId$ in the where clause, there exists an instance of property $dbp36{:}imdbId$ in the construct clause, and both instances have the same object in common; for every instance of property $dbp32{:}starring$ in the where clause, there exists an instance of property $dbp36{:}starring$ in the construct clause; furthermore, for every instance of property $dbp32{:}academyawards$ in the where clause, there exists an instance of type $dbp36{:}Award$, and the object of the former is the same as the object of the latter. Therefore, the following correspondences are satisfied in the previous sets of triples:

$$\begin{array}{lll}
(dbp32{:}Artist, & mosto{:}classToClass, & dbp36{:}Artist) \\
(dbp32{:}imdbId, & mosto{:}dataToData, & dbp36{:}imdbId) \\
(dbp32{:}starring, & mosto{:}objectToObject, & dbp36{:}starring) \\
(dbp32{:}academyawards, & mosto{:}dataToClass, & dbp36{:}Award)
\end{array}$$

Additionally, every instance of type $dbp32{:}Person$ in the where clause is not reclassified as $dbp36{:}Person$ in the construct clause; for every instance of property $dbp32{:}director$ in the where clause, there does not exist an instance of property $dbp36{:}director$ in the construct clause. As a conclusion, the following correspondences are not satisfied in the previous sets of triples:

$$\begin{array}{lll}
(dbp32{:}Person, & mosto{:}classToClass, & dbp36{:}Person) \\
(dbp32{:}director, & mosto{:}objectToObject, & dbp36{:}director)
\end{array}$$

4 Description of our proposal

Our proposal relies on the algorithm in Fig. 4. It takes a data exchange problem as input and outputs a set of executable mappings. The algorithm loops through the set of correspondences; the first step allows to compute every kernel of the input data exchange problem; later, these kernels are transformed into executable mappings by means of the following steps: computing initial executable mappings, computing variable links in the previous mappings, computing and applying substitutions. In the following subsections, we provide additional details on each step.

4.1 Step 1: computing kernels

For every correspondence in the data exchange problem being analysed, our algorithm constructs a kernel; for instance, Fig. 5 presents the kernel that is associated with the correspondence between property $dbp32{:}academyaward$ and class $dbp36{:}Award$ in our running example, which specifies that, for every instance of property $dbp32{:}academyaward$ that is found in the source ontology, the object of this instance must be classified as $dbp36{:}Award$ in the target ontology. But property $dbp32{:}academyaward$ has a domain constraint that relates it to class $dbp32{:}Actor$, which has a subclassing constraint that

```
 1: algorithm  generateExecutableMappings
 2: input      d : DataExchangeProblem
 3: output     M : ℙ ExecutableMapping
 4: variables  o1, o2 : Ontology; TC, TW : ℙ TriplePattern;
 5:            V : ℙ Correspondence; LW, LC, LV : ℙ Link; S : Substitution
 6:
 7: M := ∅
 8: for each v : Correspondence | v ∈ correspondences(d) do
 9:     — Step 1: Compute the kernel (o1, o2, V) of v
10:     o1 := findSubOntology(subject(v), source(d))
11:     o2 := findSubOntology(object(v), target(d))
12:     V := findCorrespondences(o1, o2, correspondences(d))
13:     — Step 2: Compute the initial executable mapping (TC, TW)
14:     TW := initialisePatterns(entities(o1))
15:     TC := initialisePatterns(entities(o2))
16:     — Step 3: Compute variable links in (TC, TW)
17:     LW := findConstraintLinks(TW, constraints(o1))
18:     LC := findConstraintLinks(TC, constraints(o2))
19:     LV := findCorrespondenceLinks(TW, TC, V)
20:     — Step 4: Compute substitutions and apply them to (TC, TW)
21:     S := findSubstitution(LW ∪ LC ∪ LV, TW)
22:     TC := applySubstitution(TC, S)
23:     TW := applySubstitution(TW, S)
24:     — Step 5: Update the resulting set of executable mappings
25:     M := M ∪ {(TC, TW)}
26: end for
```

Fig. 4 Algorithm to generate executable mappings

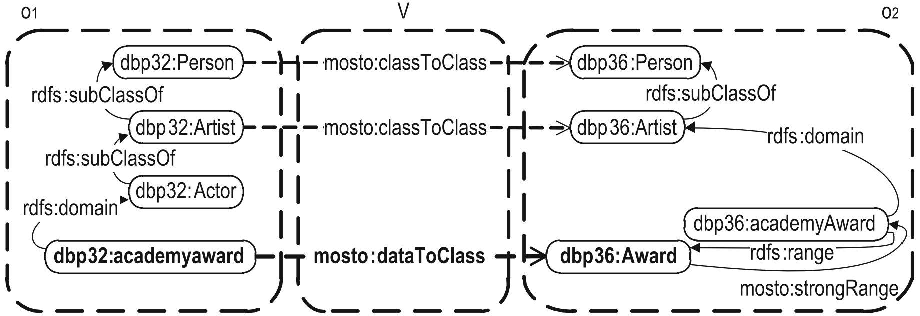

Fig. 5 A sample kernel

relates it to class *dbp*32:*Artist*, which has an additional subclassing constraint that relates it to class *dbp*32:*Person*. As a conclusion, the object of an instance of property *dbp*32:*academyaward* cannot be exchanged in isolation, but in the context of an instance of class *dbp*32:*Actor* that acts as the subject, which is also an instance of classes *dbp*32:*Artist* and *dbp*32:*Person*. Similarly, we cannot simply classify the object of property *dbp*32:*academyaward* as an instance of class *dbp*36:*Award* in the target data model, since this class is the strong range of property *dbp*36:*academyAward*, which in turn is related to class *dbp*36:*Artist* by means of domain constraints, and to *dbp*36:*Person* by means of a subclassing constraint. As a conclusion, the instance of *dbp*36:*Award* cannot be

```
1: algorithm findSubOntology
2: input      e : Entity; o : Ontology
3: output     o′ : Ontology
4: variables  E, Q : ℙ Entity; C, Z : ℙ Constraint; f : Entity
5:
6: E, C := ∅
7: Q := {e}
8: while Q ≠ ∅ do
9:     f := pick an entity from Q
10:    Q := Q \ {f}
11:    E := E ∪ {f}
12:    Z := {z : Constraint | z ∈ constraints(o) ∧ subject(z) = f}
13:    C := C ∪ Z
14:    Q := Q ∪ {g : Entity | g ∉ E ∧ ∃z : Constraint · z ∈ Z ∧ object(z) = g}
15: end while
16: o′ := (E, C)
```

Fig. 6 Algorithm to find subontologies

```
1: algorithm findCorrespondences
2: input      o1, o2 : Ontology; V : ℙ Correspondence
3: output     V′ : ℙ Correspondence
4:
5: V′ = {v : Correspondence | v ∈ V ∧ subject(v) ∈ entities(o1) ∧
6:                            object(v) ∈ entities(o2)}
```

Fig. 7 Algorithm to find the correspondences between two subontologies

constructed in isolation, but in the context of an instance of property *dbp*36:*academyAward* that has an instance of classes *dbp*36:*Artist* and *dbp*36:*Person* as subject.

In the algorithm in Fig. 4, the computation of the kernel that is associated with a correspondence is performed at lines 10–12. Given correspondence v, we first find the subontologies that are associated with the subject and the object of v and then find the correspondences between them.

We present the algorithm to compute subontologies in Fig. 6. It works on an input entity e and an input ontology o, and returns an ontology (E, C) that results from exploring e in depth. The algorithm iterates over a set of entities Q that is initialised to $\{e\}$; intuitively, Q stores the entities that remain to be explored. In each iteration of the main loop, the algorithm removes an entity f from set Q and then computes the subset of constraints whose subject is f, which is immediately added to the resulting set of constraints C; note, however, that we only add to Q the objects that have not been explored so far to prevent the algorithm from looping forever in the many common cases in which the ontology has cycles; set E keeps record of the entities that have been explored so far.

We present the algorithm to compute the correspondences between two subontologies in Fig. 7. It takes two ontologies o_1 and o_2 and a set of correspondences V as input. The output is computed as the subset of correspondences from V whose subject belongs to the entities of ontology o_1, and the object belongs to the set of entities of ontology o_2.

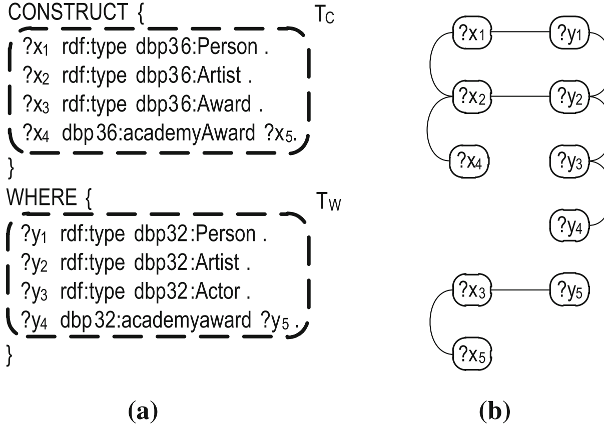

Fig. 8 Sample initial executable mapping and variable links

4.2 Step 2: computing initial executable mappings

A kernel is just a starting point; it needs to be transformed into an executable mapping. This requires to transform its source and its target subontologies into two sets of initial patterns. The set that corresponds to the where clause must include a pattern to retrieve every instance of an entity in the source ontology of the kernel, whereas the set that corresponds to the construct clause must include a pattern to construct an instance of every entity in the target ontology of the kernel.

Figure 8a presents the initial executable mapping of the kernel in Fig. 5. For example, to retrieve the instances of class $dbp32{:}Actor$, we need a pattern of the form $(?y_3, rdf{:}type, dbp32{:}Actor)$, where $?y_3$ denotes a fresh variable. Similarly, there is a property called $dbp32{:}academyaward$, which requires a pattern of the form $(?y_4, dbp32{:}academyaward, ?y_5)$, where $?y_4$ and $?y_5$ denote fresh variables, as well. Note that, intuitively, we should link variables $?y_3$ and $?y_4$ since the subject of an instance of property $dbp32{:}academyaward$ must be an instance of class $dbp32{:}Actor$; the initial executable mapping does not take these links into account, since computing them is the goal of the next step.

In the algorithm in Fig. 4, the computation of the initial executable mapping is performed at lines 14 and 15. We present the algorithm to initialise the patterns in Fig. 9. It takes a set of entities E as input and returns a set of patterns T. The resulting set is initialised to the empty set, and then the algorithm iterates over the set of entities. In each iteration, it adds a new pattern to the result set according to the type of entity being analysed, namely: for every class c, it adds a pattern of the form $(?x, rdf{:}type, c)$, where $?x$ denotes a fresh variable; and for every property p, it adds a pattern of the form $(?y, p, ?z)$, where $?y$ and $?z$ denote two fresh variables.

4.3 Step 3: computing variable links

The variables in the patterns generated by the previous step are pairwise distinct. This step is responsible for finding links amongst these variables. To find them, we need to analyse the constraints and the correspondences in the kernel that is associated with the correspondence being analysed.

```
1: algorithm initialisePatterns
2: input      E : ℙ Entity
3: output     T : ℙ Pattern
4:
5: T := ∅
6: for each e : Entity | e ∈ E do
7:     if e ∈ Class then
8:         T := T ∪ {(freshVariable(), rdf:type, e)}
9:     else if e ∈ Property then
10:        T := T ∪ {(freshVariable(), e, freshVariable())}
11:    end if
12: end for
```

Fig. 9 Algorithm to initialise patterns

Figure 8b presents the variable links regarding the initial executable mapping in Fig. 8a. Note that links can be naturally represented as an undirected graph in which the nodes are variables, and the edges indicate which variables are linked. For example, property $dbp32{:}academyaward$ in our running example has a constraint of type $rdfs{:}domain$ that indicates that its domain is class $dbp32{:}Actor$; this means that we need to locate the pattern that we generated for these entities and link their corresponding subjects, i.e., $?y_3$ and $?y_4$. Similarly, there is a correspondence between property $dbp32{:}academyaward$ and class $dbp36{:}Award$; this means that we need to locate the pattern that we generated for these entities and link $?y_5$ and $?x_3$.

We formally define the sets of links as follows:

$$Link == Variable \times Variable$$

In the algorithm in Fig. 4, the computation of variable links is performed at lines 17–19. We first find the links that are due to the constraints in the source ontology, then the links that are due to the constraints in the target ontology, and finish the process by finding the links that are due to the correspondences.

We present the algorithms to find the variable links due to constraints and correspondences in Figs. 10 and 11, respectively. They both operate very similarly: they iterate over the set of input constraints or correspondences, find the patterns associated with their subjects and objects and create the appropriate links. The only feature that requires a little explanation is that correspondences between object properties do not result in any links. The reason is that, according to inference rule O2O (cf. Sect. 3.7), an object property to object property correspondence does not specify how to link the subject or the object of the target instance; it only specifies that an instance of the source property must exist in the source patterns, and an instance of the target property must exist in the target patterns.

The algorithm to find the pattern that is associated with an entity is presented in Fig. 12. Note that, due to the way we initialise patterns, their subject is always a variable; only the predicate or the object can be entities. This is the reason why this algorithm does not check the subject of the patterns it examines.

4.4 Step 4: computing and applying substitutions

The result of the previous step is a graph in which every connected component includes a group of variables that are linked, that is, a group of variables that should actually be the

```
1: algorithm findConstraintLinks
2: input      T : ℙ Pattern; C : ℙ Constraint
3: output     L : ℙ Link
4: variables  t1, t2 : Pattern
5:
6: L := ∅
7: for each z : Constraint | z ∈ C do
8:     t1 := findPattern(subject(z), T)
9:     t2 := findPattern(object(z), T)
10:    if predicate(z) ∈ {rdfs:subClassOf, rdfs:domain, mosto:strongDomain} then
11:        L := L ∪ {(subject(t1), subject(t2))}
12:    else if predicate(z) = rdfs:subPropertyOf then
13:        L := L ∪ {(subject(t1), subject(t2)), (object(t1), object(t2))}
14:    else if predicate(z) = rdfs:range then
15:        L := L ∪ {(object(t1), subject(t2))}
16:    else if predicate(z) = mosto:strongRange then
17:        L := L ∪ {(subject(t1), object(t2))}
18:    end if
19: end for
```

Fig. 10 Algorithm to find variable links using constraints

```
1: algorithm findCorrespondenceLinks
2: input      T_W, T_C : ℙ Pattern; V : ℙ Correspondence
3: output     L : ℙ Link
4: variables  t1, t2 : Pattern
5:
6: L := ∅
7: for each v : Correspondence | v ∈ V do
8:     t1 := findPattern(subject(v), T_W)
9:     t2 := findPattern(object(v), T_C)
10:    if predicate(v) = mosto:classToClass then
11:        L := L ∪ {(subject(t1), subject(t2))}
12:    else if predicate(v) = mosto:dataToData then
13:        L := L ∪ {(object(t1), object(t2))}
14:    else if predicate(v) = mosto:dataToClass then
15:        L := L ∪ {(object(t1), subject(t2))}
16:    end if
17: end for
```

Fig. 11 Algorithm to find variable links using correspondences

```
1: algorithm findPattern
2: input      e : Entity; T : ℙ Pattern
3: output     t : Pattern
4:
5: t := nil
6: for each u : Pattern | u ∈ T while t = nil do
7:     if predicate(t) = e ∨ object(t) = e then
8:         t := u
9:     end if
10: end for
```

Fig. 12 Algorithm to find a pattern of an entity

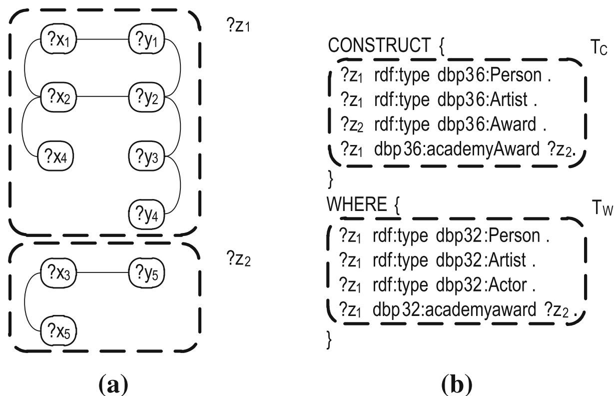

Fig. 13 Resulting substitution and executable mapping

same. Consequently, the next step is to transform this graph into a substitution in which every variable in every connected component is replaced for the same fresh variable and then apply it to the initial executable mapping that we computed in the second step.

Figure 13a highlights the two connected components in the variable links in Fig. 8b; the substitution maps the variables in each connected component to fresh variables $?z_1$ and $?z_2$, respectively. Figure 13b shows the executable mapping that results from applying the previous substitution to the initial executable mapping in Fig. 8a.

In this example, every variable in the construct clause is linked to a variable in the where clause. There can be, however, cases in which there exists a variable in the construct clause that is not linked to any variables in the where clause. Whenever this happens, the interpretation is that the set of correspondences is not complete enough to describe the data exchange problem that is being analysed. In some situations, the set of correspondences can be completed to solve the problem, but there are others in which this is not possible because the target ontology provides more information than the source ontology; for instance, DBPedia 3.6 provides information about work abstracts (property $dbp36{:}abstract$), which is not present in DBpedia 3.2. In these cases, it makes sense to generate a blank node that acts as a placeholder; in other words, instead of failing to exchange any data due to this problem, we can exchange as much data as possible and highlight special cases using blank nodes. These placeholders are known as labelled nulls in the context of nested-relational data models [18].

We formally define a substitution as a finite map from variables onto variables and blank nodes, namely:

$$Substitution == Variable \;\mapsto\; Variable \cup BlankNode$$

In the algorithm in Fig. 4, the computation and the application of substitutions are performed at lines 21–23. We first find the substitution that corresponds to the variable links that we have found in the third step and then apply it to both the initial set of patterns in the construct and the where clauses that we computed in the second step.

We present the algorithm to find substitutions in Fig. 14. It takes a set of links L and a set of patterns T_W as input and returns a substitution S; we implicitly assume that T_W is the set of triples in the where clause of an executable mapping. The algorithm first invokes $findConnectedComponents$ to find the set of connected components CC in the input

```
1: algorithm findSubstitution
2: input      L : ℙ Link; T_W : ℙ Pattern
3: output     S : Substitution
4: variables  CC : ℙ ℙ Link; x : Variable ∪ BlankNode
5:
6: CC := findConnectedComponents(L)
7: S := ∅
8: for each K : ℙ Link | K ∈ CC do
9:     if variables(K) ∩ variables(T_W) ≠ ∅ then
10:        x := freshVariable()
11:    else
12:        x := freshBlankNode()
13:    end if
14:    S := S ∪ {y, z : Variable | (y, z) ∈ K · y ↦ x} ∪
15:              {y, z : Variable | (y, z) ∈ K · z ↦ x}
16: end for
```

Fig. 14 Algorithm to find substitutions

variable links; we do not provide any additional details on this algorithm since it is well known in the literature [25]. It then initialises S to the empty set and iterates through the set of connected components CC. In each iteration, it checks whether a component K includes variables from both the construct and the where clause, in which case a fresh variable is created; otherwise, we have found a group of variables for which there is not a correspondence that assigns values to them, which justifies the creation of a fresh blank node. Immediately after, it updates the resulting substitution S by mapping every pair of variables to the fresh variable or blank node that was created previously.

We present the algorithm to apply a substitution in Fig. 15. It takes a substitution S and a set of patterns T as input and returns a new set of patterns T' that results from applying substitution S to every subject and object in T.

```
1: algorithm applySubstitution
2: input      T : ℙ Pattern; S : Substitution
3: output     T' : ℙ Pattern
4: variables  s : Subject?; o : Object?
5:
6: T' := ∅
7: for each t : Pattern | t ∈ T do
8:     s := subject(t)
9:     if s ∈ dom S then
10:        s := S(s)
11:    end if
12:    o := object(t)
13:    if o ∈ dom S then
14:        o := S(o)
15:    end if
16:    T' := T' ∪ {(s, predicate(t), o)}
17: end for
```

Fig. 15 Algorithm to apply substitutions

5 Analysis of our proposal

In this section, we analyse our proposal to prove that it is correct and we also prove that its worst-case complexity is computationally tractable.

5.1 Analysis of correctness

In the following theorem and propositions, we prove that our algorithm is correct. This requires us to prove the following: (1) that the kernels our algorithm computes are actually kernels; (2) that the executable mappings it generates retrieve source data that satisfy the source constraints and exchanges them into target data that satisfy the target constraints and the correspondences.

Theorem 1 (Kernel satisfaction) *Each data exchange problem created at lines 10–12 in Algorithm* $generateExecutableMappings$ *is a kernel.*

Proof Let k be the data exchange problem created at lines 10–12 in Algorithm $generateExecutableMappings$. Note that o_1 is the source ontology, o_2 is the target ontology and V is the set of correspondences of k. According to the definition of kernel (cf. Sect. 3.5), k must be contained in d, that is, the entities, constraints and correspondences of k must be present in d. As it can be checked, algorithms $findSubOntology$ and $findCorrespondences$ always iterate over the entities, constraints and correspondences of d; therefore, k is contained in d since we do not use any external entity, constraint or correspondence. Furthermore, the source (target) ontology of k must be connected; for every two entities e_1 and e_2 in the source (target) ontology, we can discern the following cases:

- There exists a path that has e_1 as the first entity and e_2 as the last entity: Algorithm $findSubOntology$ would find this path since, starting from e_1, it adds a new entity if there exists a constraint that relates the previous entity with the next entity in the path, ending in e_2.
- There exists a path that has e_2 as the first entity and e_1 as the last entity: Algorithm $findSubOntology$ would find this path as we have analysed in the previous case.
- There exists an entity e_3 and two paths p_1 and p_2, such that p_1 has e_3 as the first entity and e_1 as the last entity, and p_2 has e_3 as the first entity and e_2 as the last entity: Algorithm $findSubOntology$ would find paths p_1 and p_2 in isolation as we have analysed in the first case.

Another implication of the definition of kernel is that d must not contain a source (target) constraint c that does not belong to k, but the subject of c belongs to the entities of k; if this constraint c exists, algorithm $findSubOntology$ would add it to k (cf. line 12 in Fig. 6). The final implication of the definition of kernel is that d must not contain a correspondence v that does not belong to k, but the subject and object of v belong to the source and target entities of k, respectively; if this correspondence v exists, algorithm $findCorrespondences$ would add it to k.

Theorem 2 (Generating executable mappings) *Let d be a data exchange problem. The executable mappings generated by* $generateExecutableMappings(d)$ *satisfy the following properties: i) every source triple it retrieves satisfies the constraints of* $source(d)$*; ii) every target triple it generates satisfies the constraints of* $target(d)$*; and iii) they satisfy the correspondences in the kernels from which they originate.*

Proof The proof builds on Propositions 1, 2 and 3, in which we prove each property of the resulting executable mappings independently.

Proposition 1 (Source constraint satisfaction) *Let* d *be a data exchange problem and* $m = (T_C, T_W)$ *any of the executable mappings that are returned by* $generateExecutableMappings(d)$. m *retrieves data that satisfy the constraints of the source ontology of* d.

Proof The proof follows from reductio ad absurdum: assume that z is a constraint in the source of d, and that T_W does not satisfy it. Depending on the predicate of z, we may distinguish the following cases:

- If constraint z is of the form $(c_1, rdfs{:}subClassOf, c_2)$, where c_1 and c_2 denote two classes, then the initial patterns are of the form $(?x_1, rdf{:}type, c_1)$ and $(?x_2, rdf{:}type, c_2)$; thus, the algorithm to find constraint links must return a link between variables $?x_1$ and $?x_2$, which, after computing the corresponding substitution and applying it to the initial patterns, results in two patterns of the form $(?x, rdf{:}type, c_1)$ and $(?x, rdf{:}type, c_2)$. According to inference rule SC (cf. Sect. 3.6), T_W satisfies constraint z.
- If z is of the form $(p_1, rdfs{:}subPropertyOf, p_2)$, where p_1 and p_2 denote two properties, then the initial patterns are of the form $(?x_1, p_1, ?y_1)$ and $(?x_2, p_2, ?y_2)$; thus, the algorithm to find constraint links must return a link between variables $?x_1$ and $?x_2$, and another link between variables $?y_1$ and $?y_2$, which, after computing the corresponding substitution and applying it to the initial patterns, result in two patterns of the form $(?x, p_1, ?y)$ and $(?x, p_2, ?y)$. According to inference rule SP (cf. Sect. 3.6), T_W satisfies constraint z.
- If z is of the form $(p, rdfs{:}domain, c)$, where p denotes a property and c denotes a class, then the initial patterns are of the form $(?x_1, p, ?y_1)$ and $(?x_2, rdf{:}type, c)$; thus, the algorithm to find constraint links must return a link between variables $?x_1$ and $?x_2$, which, after computing the corresponding substitution and applying it to the initial patterns, results in two patterns of the form $(?x, p, ?y_1)$ and $(?x, rdf{:}type, c)$. According to inference rule D (cf. Sect. 3.6), T_W satisfies constraint z.
- If z is of the form $(p, rdfs{:}range, c)$, where p denotes a property and c a class, then the initial patterns are of the form $(?x_1, p, ?y_1)$ and $(?x_2, rdf{:}type, c)$; thus, the algorithm to find constraint links must return a link between variables $?y_1$ and $?x_2$, which, after computing the corresponding substitution and applying it to the initial patterns, results in two patterns of the form $(?x_1, p, ?y)$ and $(?y, rdf{:}type, c)$. Therefore, according to inference rule R (cf. Sect. 3.6), T_W satisfies constraint z.
- If z is of the form $(c, mosto{:}strongDomain, p)$, where c denotes a class and p a property, then the initial patterns are of the form $(?x_1, rdf{:}type, c)$ and $(?x_2, p, ?y_2)$; thus, the algorithm to find constraint links must return a link between variables $?x_1$ and $?x_2$, which, after computing the corresponding substitution and applying it to the initial patterns, results in two patterns of the form $(?x, rdf{:}type, c)$ and $(?x, p, ?y_2)$. According to inference rule SD (cf. Sect. 3.6), T_W satisfies constraint z.
- If z is of the form $(c, mosto{:}strongRange, p)$, where c denotes a class and p a property, then the initial patterns are of the form $(?x_1, rdf{:}type, c)$ and $(?x_2, p, ?y_2)$; thus, the algorithm to find constraint links must return a link between variables $?x_1$ and $?y_2$, which, after computing the corresponding substitution and applying it to the initial patterns, results in two patterns of the form $(?y, rdf{:}type, c)$ and $(?x_2, p, ?y)$. According to inference rule SR (cf. Sect. 3.6), T_W satisfies constraint z.

Since we have found a contradiction in every case, we can conclude that the initial hypothesis is wrong. As a conclusion, m retrieves data that satisfies the constraints of the source ontology of d.

Proposition 2 (Target constraint satisfaction) *Let d be a data exchange problem and $m = (T_C, T_W)$ any of the executable mappings that are returned by $generateExecutableMappings(d)$. m constructs data that satisfy the constraints of the target ontology of d.*

Proof The proof follows straightforwardly using the same reasoning as in the previous proposition.

Proposition 3 (Correspondence satisfaction) *Let d be a data exchange problem and $m = (T_C, T_W)$ any of the executable mappings that are returned by $generateExecutableMappings(d)$. Assume that v is the correspondence from which $generateExecutableMappings$ generated m, and that k is the kernel associated with v. m satisfies the subset of correspondences in k.*

Proof The proof follows from reductio ad absurdum: assume that v is a correspondence in $correspondences(k)$, and that m does not satisfy it. Depending on the predicate of v, we may distinguish the following cases:

- If v is of the form $(c_1, mosto{:}classToClass, c_2)$, where c_1 and c_2 denote two classes, the initial patterns of the where and the construct clauses contain two patterns of the form $(?x_1, rdf{:}type, c_1)$ and $(?y_1, rdf{:}type, c_2)$, respectively; the algorithm to find correspondence links must then find a link between variables $?x_1$ and $?y_1$, which, after computing the corresponding substitution and applying it, results in patterns $(?x, rdf{:}type, c_1)$ and $(?x, rdf{:}type, c_2)$. According to inference rule $C2C$ (cf. Sect. 3.7), m satisfies this correspondence.
- If v is of the form $(p_1, mosto{:}dataToData, p_2)$, where p_1 and p_2 denote two data properties, the initial patterns of the where and the construct clauses contain two patterns of the form $(?x_1, p_1, ?y_1)$ and $(?x_1, p_2, ?y_2)$, respectively; the algorithm to find correspondence links must then find a link between variables $?y_1$ and $?y_2$, which, after computing the corresponding substitution and applying it, results in patterns $(?x_1, p_1, ?y)$ and $(?x_2, p_2, ?y)$. According to inference rule $D2D$ (cf. Sect. 3.7), m satisfies this correspondence.
- If v is of the form $(p_1, mosto{:}objectToObject, p_2)$, where p_1 and p_2 denote two object properties, the initial patterns of the where and the construct clauses contain two patterns of the form $(?x_1, p_1, ?y_1)$ and $(?x_1, p_2, ?y_2)$, respectively; note that the algorithm to find correspondence links ignores this kind of correspondences and that, according to inference rule $O2O$ (cf. Sect. 3.7), the initial patterns satisfy this correspondence.
- If v is of the form $(p, mosto{:}dataToClass, c)$, where p denotes a data property and c denotes a class, the initial patterns of the where and the construct clauses contain two patterns of the form $(?x_1, p, ?y_1)$ and $(?x_2, rdf{:}type, c)$, respectively; the algorithm to find correspondence links must then find a link between variables $?y_1$ and $?x_2$, which, after computing the corresponding substitution and applying it, results in patterns $(?x_1, p, ?y)$ and $(?y, rdf{:}type, c)$. According to inference rule $D2C$ (cf. Sect. 3.7), m satisfies this correspondence.

Since we have found a contradiction in every case, we can conclude that the initial hypothesis is wrong. As a conclusion, m satisfies the correspondences in the kernel that is associated with the correspondence from which it originated.

5.2 Analysis of complexity

In the following theorem and propositions, we analyse the worst-case complexity of our algorithm and its subalgorithms. The worst case is a data exchange problem in which every entity of the source ontology has a correspondence with every entity of the target ontology. Furthermore, the source and target ontologies are complete graphs, that is, every pair of entities are connected by a constraint. As a conclusion, in the worst-case problem, $v = e_s\, e_t$, $c_s = (e_s^2 - e_s)/2$, and $c_t = (e_t^2 - e_t)/2$, where e_s and e_t denote the number of entities in the source and target ontologies, v denotes the number of correspondences, and c_s and c_t denote the number of source and target constraints, respectively.

In our proofs, we assume that simple set operations like invoking a projection function, checking for membership, merging two sets, or constructing a tuple can be implemented in $O(1)$ time with regard to the other operations. We also implicitly assume that data exchange problems must be finite, that is, the sets of entities, constraints and correspondences involved are finite.

Theorem 3 (Generating executable mappings, cf. Fig. 4) *Let d be a data exchange problem.* $O(e_s^4\, e_t + e_s\, e_t^4 + e_s^2\, e_t^2 (e_s + e_t))$ *is an upper bound for the worst-time complexity of algorithm* $generateExecutableMappings(d)$*, where* e_s *and* e_t *denote the number of entities in the source and target of d, respectively.*

Proof Algorithm $generateExecutableMappings(d)$ has to iterate through the whole set of correspondences in d. It calls $findSubOntology$ two times for each correspondence (lines 10 and 11): the first time is to compute the source subontology and the second time to compute the target subontology, which, according to Proposition 4, terminates in $O(e_s\, c_s)$ and $O(e_t\, c_t)$ time, respectively, where $c_s = (e_s^2 - e_s)/2$ is the number of source constraints and $c_t = (e_t^2 - e_t)/2$ is the number of target constraints. In the next step, the algorithm calls $findCorrespondences$ (line 12), which, according to Proposition 5, terminates in $O(v)$ time, where $v = e_s\, e_t$ denotes the number of correspondences of d. Furthermore, the algorithm calls $initialisePatterns$ two times (lines 14 and 15): the first time is to compute the initial source patterns and the second time to compute the initial target patterns, which, according to Proposition 6, terminates in $O(e_s)$ and $O(e_t)$ time, respectively. In the following steps, the algorithm calls $findConstraintLinks$ two times (lines 17 and 18): the first time is to compute the links that are related to the source constraints and the second time to compute the links that are related to the target constraints, which, according to Proposition 7, terminate in $O(c_s\, t_w)$ and $O(c_t\, t_c)$ time, where $c_s = (e_s^2 - e_s)/2$ is the number of source constraints, $t_w = e_s$ is the number of patterns in the where clause, which is equal to the whole set of source entities in the worst case, $c_t = (e_t^2 - e_t)/2$ is the number of target constraints and $t_c = e_t$ is the number of patterns in the construct clause, which is equal to the whole set of target entities in the worst case. In the next step, the algorithm calls $findCorrespondenceLinks$ (line 19), which, according to Proposition 8, terminates in $O(v\, (t_w + t_c))$ time, where $v = e_s\, e_t$ denotes the number of correspondences of d, $t_w = e_s$ is the number of patterns in the where clause, and $t_c = e_t$ is the number of patterns in the construct clause. In the next step, the algorithm calls $findSubstitution$ (line 21), which, according to Proposition 10, terminates in $O(l)$ time, where $l = 2\, c_s + 2\, c_t + v = e_s^2 - e_s + e_t^2 - e_t + e_s\, e_t$ is the total number of links which, in the worst case, involves two links for each source and target constraint (subproperty constraint), and a link for each correspondence. Finally, the algorithm calls $applySubstitution$ two times (lines 22 and 23): the first time is to compute the substitution of the where clause and the second time to compute the substitution of the construct clause, which, according

to Proposition 11, terminates in $O(t_w)$ and $O(t_c)$ time, respectively, where $t_w = e_s$ is the number of patterns in the where clause and $t_c = e_t$ is the number of patterns in the construct clause.

Therefore, we get the following expression: $O(e_s\, e_t (e_s/2\,(e_s^2 - e_s) + e_t/2\,(e_t^2 - e_t) + e_s\, e_t + e_s + e_t + e_s/2\,(e_s^2 - e_s) + e_t/2\,(e_t^2 - e_t) + e_s\, e_t (e_s + e_t) + e_s^2 - e_s + e_t^2 - e_t + e_s\, e_t + e_s + e_t)) = O(e_s^4\, e_t + e_s\, e_t^4 + 2\, e_s^2\, e_t^2 + e_s^3\, e_t^2 + e_s^2\, e_t^3 + e_s^2\, e_t + e_s\, e_t^2)$, in which $e_s^4\, e_t > e_s^2\, e_t$, $e_s\, e_t^4 > e_s\, e_t^2$, and $e_s^2\, e_t^3 > 2\, e_s^2\, e_t^2$. As a conclusion, $O(e_s^4\, e_t + e_s\, e_t^4 + e_s^2\, e_t^2 (e_s + e_t))$ is an upper bound for the worst-time complexity of algorithm $generateExecutableMappings(d)$.

Proposition 4 (Finding subontologies, cf. Fig. 6) *Let f be an entity and o an ontology. $findSubOntology(f, o)$ terminates in $O(e\, c)$ time in the worst case, where e and c denote the number of entities and constraints in ontology o, respectively.*

Proof In the worst case, $findSubOntology(f, o)$ has to iterate through the whole set of entities in o; additionally, in each iteration, it has to iterate through the whole set of constraints. As a conclusion, $findSubOntology(f, o)$ terminates in $O(e\, c)$ time in the worst case.

Proposition 5 (Finding correspondences, cf. Fig. 7) *Let o_1 and o_2 be two ontologies, and V a set of correspondences. $findCorrespondences(o_1, o_2, V)$ terminates in $O(v)$ time in the worst case, where v denotes the number of correspondences in set V.*

Proof Algorithm $findCorrespondences$ has to iterate through the whole set of correspondences V to find the ones whose subject belongs in o_1 and whose object belongs in o_2. As a conclusion, $findCorrespondences(o_1, o_2, V)$ terminates in $O(v)$ time in the worst case.

Proposition 6 (Initialising patterns, cf. Fig. 9) *Let E be a set of entities. $initialise Patterns(E)$ terminates in $O(e)$ time in the worst case, where e denotes the number of entities in set E.*

Proof Algorithm $initialisePatterns$ has to iterate through the whole set of entities E. As a conclusion, $initialisePatterns(E)$ terminates in $O(e)$ time in the worst case.

Proposition 7 (Finding constraint links, cf. Fig. 10) *Let T be a set of patterns, and C a set of constraints, $findConstraintLinks(T, C)$ terminates in $O(c\, t)$ time in the worst case, where t denotes the number of patterns in T and c denotes the number of constraints in C.*

Proof Algorithm $findConstraint\ Links$ iterates over the whole set of input constraints C. In each iteration, it invokes $find\ Pattern$ on T twice, and each invocation terminates in $O(t)$ time in the worst case according to Proposition 9. As a conclusion, $findConstraints(T, C)$ terminates in $O(c\, t)$ time in the worst case. **Proposition 8** (Finding correspondence links, cf. Fig. 11) *Let T_W and T_C be two sets of pat-terns, and V an arbitrary set of correspondences. $findCorrespondenceLinks(T_W, T_C, V)$ terminates in $O(v\,(t_w + t_c))$ time in the worst case, where v denotes the number of corre-spondences in V, and t_w and t_c denote the number of patterns in T_W and T_C, respectively.*

Proof Algorithm $findCorrespondenceLinks$ iterates through the whole set of input correspondences V. In each iteration, it calls algorithm $findPattern$ on both T_W and T_C; these invocations terminate in $O(t_w + t_c)$ time in the worst case according to Proposition 9. As a conclusion, algorithm $findCorrespondenceLinks$ terminates in $O(v\,(t_w + t_c))$ time in the worst case.

Proposition 9 (Finding patterns, cf. Fig. 12) *Let e be an entity, and T a set of patterns.* $findPattern(e, T)$ *terminates in* $O(t)$ *time in the worst case, where t denotes the number of patterns in* T*.*

Proof Algorithm $findPattern$ iterates through the whole set of patterns T, as long as it does not find the pattern that refers to e in its predicate or its object. In the worst case, this triple is the last one. As a conclusion, $findPattern$, terminates in $O(t)$ time in the worst case.

Proposition 10 (Finding substitutions, cf. Fig. 14) *Let L be a set of variable links and* T_W *a set of patterns.* $O(l)$ *is an upper bound for the worst-time complexity of* $findSubstitution(L, T_W)$*, where l denotes the number of links in L.*

Proof Algorithm $findConnectedComponents$ terminates in $O(\max\{a, l\})$ time in the worst case [25], where a denotes the number of variables in L. The algorithm then iterates through the set of connected components in L; note that it is not easy to characterise the worst case, but it is safe to assume that l must be an upper bound to the number of connected components that is returned by $findConnectedComponents$ since a graph with l edges may have a maximum of l connected components. Therefore, $O(\max\{a, l\} + l)$ is an upper bound to the worst-case time complexity of algorithm $findSubstitution$. If $\max\{a, l\} = a$, then the upper bound is $O(a + l)$; contrarily, if $\max\{a, l\} = l$, then the upper bound is $O(l)$. Therefore, $O(a + l)$ is also an upper bound for the worst-case time complexity of this algorithm. Note that, in the worst case, $a = 2l$ since all variables related by the links are pairwise distinct. As a conclusion, $O(l)$ is an upper bound to the worst-case time complexity of algorithm $findSubstitution$.

Proposition 11 (Applying substitutions, cf. Fig. 15) *Let T be a set of patterns and S a substitution.* $applySubstitution(T, S)$ *terminates in* $O(t)$ *time in the worst case, where t denotes the number of patterns in* T*.*

Proof Algorithm $applySubstitution$ has to iterate through the whole set of patterns T. As a conclusion, $applySubstitution$ terminates in $O(t)$ time in the worst case.

6 Validation of our proposal

Recall that an executable mapping encodes an interpretation of the correspondences in a data exchange problem. It is then necessary to check whether our interpretation of correspondences agrees with the interpretation of domain experts by means of experiments. Note that we cannot compare our proposal to others in the literature from an empirical point of view: the proposal by Popa et al. [45] cannot be applied since it focuses on nested-relational models, which is not our case; the proposal by Mergen and Heuser [33] cannot be applied because it focuses on a subset of taxonomies, and the data exchange problems on which we have carried out our experimentation are far from such taxonomies; the proposal by Qin et al. [47] requires the target ontology to be pre-populated, which is not the case of our repository, and they require the ontologies to be expressed using Web-PDDL; unfortunately, there is not an automatic translator from OWL into Web-PDDL. Mocan and Cimpian [34] automatically generate WSML rules, so we cannot compare our SPARQL executable mappings with these

rules. Last, but not least, Dou et al. [16] and Parreiras et al. [41] require the user to handcraft the executable mappings, whereas in our proposal, executable mappings are automatically generated.

We report on the details of our repository of data exchange problems in Sect. 6.1, then report on the validation process in Sect. 6.2 and the results of the validation are presented in Sect. 6.3; finally, we report on the only limitation we have found in Sect. 6.4.

6.1 Repository

Unfortunately, there is not a standard repository of data exchange problems on which different proposals can be tested and compared in the context of Linked Data applications. To address this problem, we have set-up a repository of representative real-world data exchange problems that consist of three real-world data exchange problems and 3,780 synthetic problems that were produced using MostoBM [53,54]. This tool allows to fine tune the generation of data exchange problems by means of seven parameters; it also provides executable mappings to perform data exchange in each problem it generates; these executable mappings are automatically generated by instantiating parameterised templates that were devised by human experts.

The real-world problems included the following: (DBP) exchanging data from DBpedia 3.2 to DBPedia 3.6 [10], which is a typical ontology evolution problem; (O2M) exchanging data from OWL-S 1.1 [27] to MSM 1.0 [42], which is a typical ontology adaptation problem; (MO) exchanging data from both DBpedia 3.6 and Review 2.0 into a fictitious Movies Online ontology, which is a typical problem of ontology integration; (BBC) exchanging data from Programmes Ontology 2009 [29] to DBPedia 3.7. The synthetic problems included 540 problems of each of the following categories [54]: (LP) Lift Properties, that is, moving data properties from some source subclasses to a common target superclass; (SP) Sink Properties, that is, moving some data properties of a source class to target subclasses; (ESB) Extract Subclasses, that is, splitting a source class into several target subclasses and its data properties amongst them; (ESP) Extract Superclasses, that is, splitting a source class into several target superclasses and its data properties amongst them; (ERC) Extract Related Classes, that is, spreading the data properties of a source class into several target classes, which are related to the original by means of object properties; (SS) Simplify Specialisations, that is, flattening a number of subclasses into a single target class; and (SRC) Simplify Related Classes, that is, transforming several source classes that are related by means of object properties into a single target class.

Our repository and an implementation of our proposal are publicly available at http://www.tdg-seville.info/carlosrivero/MostoDE; a demo that provides more details about our implementation was published elsewhere [52]; a detailed description of the real-world problems is presented in reference [55].

6.2 Validation process

We used our implementation to create a set of executable mappings for each data exchange problem in the repository and then run these mappings and compared the results with the expected ones.

Figure 16 shows our validation process: for each problem, we performed data exchange using, on the one hand, the executable mappings output by MostoDE, and on the other hand, the executable mappings output by MostoBM; finally, we compared whether or not the resulting data are equal or not.

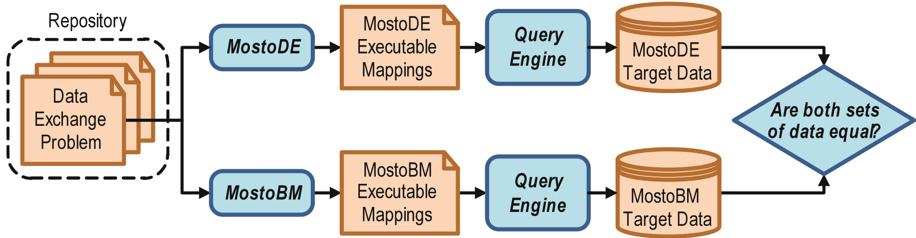

Fig. 16 Validation process

6.3 Experimental results

Table 1 summarises our results. The columns represent the data exchange problems and the rows a number of measures; the first group of measures provides an overall idea of the 'size' of each data exchange problem, whereas the second group provides information about the number of executable mappings, the time to generate them (Gen. time) and the time they took to execute, that is, the time spent at performing the data exchange (Data ex.). In the case of synthetic problems, we provide intervals that indicate the minimum and maximum value of each measure.

The target data generated by the executable mappings of MostoBM were exactly the same as the target data generated by executable mappings of MostoDE in every experiment expect one: the BBC problem that is explained below (cf. Sect. 6.4). This reveals that the interpretation of the correspondences that we encode in our executable mappings captures the intuition behind them. In the DBP problem, it is worth noting that BDpedia 3.6 provides more data than DBpedia 3.2; in this case, whether source and target data are equal or not were measured on the subset of DBpedia 3.6 that can be exchanged from DBpedia 3.2, since the remaining data resulted, obviously, in blank nodes.

The time our proposal took to generate the mappings was less than one second in all cases; since timings are imprecise in nature, we repeated each experiment 25 times and averaged the results after discarding roughly 0.01 % outliers using the well-known Chevischev's inequality. The experiments were run on a computer that was equipped with a single 2.66 GHz Core 2 Duo CPU and 4 GB RAM, Windows XP Professional (SP3), JRE 1.6.0, Jena 2.6.4, ARQ 2.8.8, and Oracle 11g.

We also measured the time our executable mappings took to exchange data. Although these timings depend largely on the technology being used, that is, the database used to persist triples and the SPARQL engine used to query them, we think that presenting them is appealing insofar they prove that the queries we generate can be executed on reasonably large ontologies in a sensible time.

6.4 Limitations

The previous data exchange problems are representative of the many real-world problems we face daily. We have found only a real-world problem in which the results of our technique are not satisfactory. We found this limitation in the BBC data exchange problem, in which property $po{:}series$ relates a television serial ($po{:}Brand$) with its seasons ($po{:}Series$), and both classes have the same superclass: $po{:}Programme$; unfortunately, the executable mappings that we generate state that a television serial is equal to a season, which is usually not true.

Table 1 Summary of our validation

	DBP	O2M	MO	BBC	LP (540)	SP (540)	ESB (540)	ESP (540)	ERC (540)	SS (540)	SRC (540)
Classes	12	72	9	11	[4–84]	[4–84]	[3–45]	[3–45]	[3–45]	[3–45]	[3–45]
Data properties	4	41	8	15	[50–154]	[50–154]	[50–154]	[50–154]	[50–154]	[50–154]	[50–154]
Object properties	5	90	8	15	0	0	0	0	[1–43]	0	[1–43]
Correspondences	9	11	10	24	[27–115]	[27–115]	[26–76]	[26–76]	[27–115]	[26–76]	[26–76]
Src. constraints	12	696	54	36	[26–489]	[26–489]	[25–309]	[25–309]	[25–309]	[26–489]	[29–660]
Tgt. constraints	49	118	58	32	[26–489]	[26–489]	[26–489]	[51–564]	[29–660]	[25–309]	[25–309]
Triples	2,107K	2,536K	1,093K	7,275K	[871–18K]	[769–14K]	[776–8K]	[776–8K]	[776–8K]	[872–18K]	[1355–17K]
Exec. mappings	9	11	10	–	[27–115]	[27–115]	[26–76]	[26–76]	[27–115]	[26–76]	[26–76]
Gen. time (secs)	0.06	0.25	0.08	–	[0.02–0.11]	[0.03–0.1]	[0.02–0.06]	[0.02–0.15]	[0.02–0.34]	[0.02–0.09]	[0.02–0.34]
Data ex. (secs)	2.95	55.2	20.62	–	[0.14–0.69]	[0.14–0.72]	[0.14–0.63]	[0.14–0.63]	[0.17–2.42]	[0.14–0.44]	[0.16–8.31]

As a conclusion, our proposal cannot deal well with data exchange problems in which we must deal with more than one instance of the same class or superclass (except $rdfs{:}Resource$ and $owl{:}Thing$). The reason is that we assume that it is only necessary a single instance of a hierarchy of classes to exchange source into target data. Although this problem is interesting from a theoretical point of view, our experience suggests that it is not common at all in practice.

7 Conclusions

In this article, we present MostoDE, a proposal that aims to automatically generate SPARQL executable mappings in the context of ontologies that are represented in quite a complete subset of the OWL 2 Lite profile. These mappings are executed over a source ontology using a SPARQL query engine, and the source data are exchanged into data of a target ontology.

Our proposal takes a data exchange problem as input, which comprises a source ontology, a target ontology and a number of correspondences between them, and it outputs a SPARQL executable mapping for each correspondence of the data exchange problem. These SPARQL executable mappings are generated by means of kernels, each of which describes the structure of a subset of data in the source ontology that needs to be exchanged as a whole, and the structure of a subset of data in the target ontology that needs to be created as a whole: if more or less data are considered, then the exchange would be incoherent.

Correspondences are inherently ambiguous since there can be many different executable mappings that satisfy them, but generate different target data. Building on our validation, our technique seems promising enough for real-world data exchange problems: we have validated it on three real-world and 3,780 synthetic data exchange problems. We checked that the interpretation of correspondences that our technique encodes is coherent with the expected results. Furthermore, the time taken to generate SPARQL executable mappings by our technique did not exceed one second in any of the data exchange problems of our validation. The only limitation that we have found to our proposal is that it cannot deal well with data exchange problems in which it is necessary to handle more than one instance of the same class or superclass (except $rdfs{:}Resource$ and $owl{:}Thing$).

Acknowledgments We would like to thank Dr. Alberto Pan, Dr. Paolo Papotti and Dr. Carlos Pedrinaci for their helpful suggestions on an earlier draft of this article. We also thank our reviewers for their insightful and valuable comments, which helped us improve the paper significantly. The work on which we report was supported by the European Commission (FEDER), the Spanish and the Andalusian R&D&I programmes (grants TIN2007-64119, P07-TIC-2602, P08-TIC-4100, TIN2008-04718-E, TIN2010-21744, TIN2010-09809-E, TIN2010-10811-E, TIN2010-09988-E, and TIN2011-15497-E).

8 Subset of the OWL 2 Lite profile

The OWL 2 Lite profile specification provides a number of constructs that are classified into the following groups [6]: RDFS features, equality, restricted cardinality, property characteristics, general property restrictions, class intersection and meta-information. We analyse them in the following paragraphs. The conclusion is that out of the 43 constructs, the only ones with which our proposal cannot deal are zero-cardinality restrictions, general property restrictions and intersection of classes and restrictions. Our experience proves that this does

not hinder its practical applicability since the constructs with which we cannot deal are not so common in real-world applications [21].

RDFS features: This group includes $owl{:}Class$, $rdfs{:}subClassOf$, $rdf{:}Property$, $rdfs{:}subPropertyOf$, $rdfs{:}domain$, $rdfs{:}range$, and $owl{:}Individual$. The only one that our proposal ignores is $owl{:}Individual$, since individuals are retrieved or constructed by means of the executable mappings we generate, but need not be dealt with explicitly in our proposal.

Equality and inequality: The constructs of this group are $owl{:}equivalentClass$, $owl{:}equivalentProperty$, $owl{:}sameAs$, $owl{:}differentFrom$, $owl{:}AllDifferent$, and $owl{:}distinctMembers$. The first two constructs need not be dealt with explicitly, since they are actually abbreviations. If an ontology contains a triple of the form $(c_1, owl{:}equivalentClass, c_2)$, it can be replaced by the following triples, with which we can deal natively:

$$(c_1,\ rdfs{:}subClassOf,\ c_2)$$
$$(c_2,\ rdfs{:}subClassOf,\ c_1)$$

Similarly, if it contains a triple of the form $(p_1, owl{:}equivalentProperty, p_2)$, we may replace it by the following triples, with which we can also deal natively:

$$(p_1,\ rdfs{:}subPropertyOf,\ p_2)$$
$$(p_2,\ rdfs{:}subPropertyOf,\ p_1)$$

The remaining constructs in this category deal with individuals; thus, we may ignore them for data exchange purposes.

Restricted cardinality: The constructs in this group allow to restrict the cardinality of a property, namely: $owl{:}minCardinality$, $owl{:}maxCardinality$ and $owl{:}cardinality$.

The $owl{:}minCardinality$ construct can restrict the minimum cardinality of a property p with respect to a class c to be zero as follows:

$$(c, rdfs{:}subClassOf, _{:}x)$$
$$(_{:}x, rdf{:}type, owl{:}Restriction)$$
$$(_{:}x, owl{:}minCardinality, \text{`0'}\hat{}\hat{}xsd{:}int)$$
$$(_{:}x, owl{:}onProperty, p)$$

Note that, by default, all data and object properties have a minimum cardinality of zero. Therefore, it does not require any special treatment. Similarly, construct $owl{:}minCardinality$ can restrict the minimum cardinality of a property p with respect to a class c to be one as follows:

$$(c, rdfs{:}subClassOf, _{:}x)$$
$$(_{:}x, rdf{:}type, owl{:}Restriction)$$
$$(_{:}x, owl{:}minCardinality, \text{`1'}\hat{}\hat{}xsd{:}int)$$
$$(_{:}x, owl{:}onProperty, p)$$

The previous set of triples can be replaced by $(c, mosto{:}strongDomain, p)$, if property p has class c as domain, or by $(c, mosto{:}strongRange, p)$, if property p has class c as range.

Construct $owl{:}maxCardinality$ can be used to restrict the maximum cardinality of a property to zero or one. In the former case, the property is annulated, which is a case with which our proposal cannot deal; we, however, have not found this to be a practical limitation since it is not common at all to annulate a property. The later case is the default in the OWL 2 Lite profile; thus, it does not require a special treatment.

Construct $owl{:}cardinality$ is a shorthand to combine the previous constructs; thus, neither does it require special treatment.

Property characteristics: This group includes constructs $owl{:}ObjectProperty$ and $owl{:}DatatypeProperty$, with which we deal natively; $owl{:}TransitiveProperty$ must actually be dealt with by a semantic-web reasoner, that is, we can assume that their semantics have been made explicit before using our technique; the rest of constructs are shorthands, namely: $owl{:}inverseOf$, $owl{:}SymmetricProperty$, $owl{:}FunctionalProperty$, and $owl{:}InverseFunctionalProperty$.

If an ontology contains a subset of triples of the form:

$$
\begin{array}{l}
(p_1, rdfs{:}domain, c_1) \\
(p_1, rdfs{:}range, c_2) \\
(p_1, owl{:}inverseOf, p_2)
\end{array}
$$

we can transform them into the following triples, with which we can deal natively:

$$
\begin{array}{l}
(p_2, rdfs{:}domain, c_2) \\
(p_2, rdfs{:}range, c_1)
\end{array}
$$

Furthermore, we can transform $(p, rdf{:}type, owl{:}SymmetricProperty)$ into a triple of the form $(p, owl{:}inverseOf, p)$. Similarly, we can transform a triple of the form $(p, rdf{:}type, owl{:}FunctionalProperty)$ into the following triples before using our technique:

$$
\begin{array}{l}
(_{:}x, rdf{:}type, owl{:}Restriction) \\
(_{:}x, owl{:}minCardinality, \text{‘0’}\hat{}\hat{}xsd{:}int) \\
(_{:}x, owl{:}maxCardinality, \text{‘1’}\hat{}\hat{}xsd{:}int) \\
(_{:}x, owl{:}onProperty, p)
\end{array}
$$

Finally, if we find a subset of triples of the form:

$$
\begin{array}{l}
(p_1, owl{:}inverseOf, p_2) \\
(p_2, rdf{:}type, owl{:}InverseFunctionalProperty)
\end{array}
$$

we can transform it into the following triples before applying our technique:

$$
\begin{array}{l}
(p_1,\ rdf{:}type,\ owl{:}FunctionalProperty) \\
(p_2,\ rdf{:}type,\ owl{:}FunctionalProperty)
\end{array}
$$

General property restrictions: This group includes a number of constructors that allow to express general constraints on properties, namely: $owl{:}Restriction$, $owl{:}onProperty$, $owl{:}allValuesFrom$ and $owl{:}someValuesFrom$. We cannot deal with these constructs in a general problem, but only in the cases that we have mentioned in the previous paragraphs, that is, functional properties and cardinality restrictions.

Class intersection: This group includes a single construct: $owl{:}intersectionOf$. We cannot deal with these constructs.

Meta-information: This category comprises three groups of constructs in the OWL 2 Lite profile, namely: header information, versioning and annotation types. They include constructs like $owl{:}Ontology$, $owl{:}imports$, $owl{:}versionInfo$ or $owl{:}backwardCompatibleWith$, to mention a few. They provide meta-information about an ontology, which make them irrelevant for data exchange purposes.

References

1. Alexe B, Chiticariu L, Miller RJ, Pepper D, Tan WC (2008a) Muse: A system for understanding and designing mappings. In: SIGMOD, pp 1281–1284
2. Alexe B, Tan WC, Velegrakis Y (2008b) STBenchmark: towards a benchmark for mapping systems. PVLDB 1(1):230–244
3. Antoniou G, van Harmelen F (2008) A semantic web primer, 2nd edn. The MIT Press, Cambridge
4. Arenas M, Libkin L (2008) XML data exchange: consistency and query answering. J ACM 55(2):1–72
5. Arjona JL, Corchuelo R, Ruiz D, Toro M (2007) From wrapping to knowledge. IEEE Trans Knowl Data Eng 19(2):310–323
6. Bechhofer S, van Harmelen F, Hendler J, Horrocks I, McGuinness DL, Patel-Schneider PF, Stein LA (2004) OWL web ontology language reference. Technical report, W3C. http://www.w3.org/TR/owl-ref/
7. Bernstein PA, Haas LM (2008) Information integration in the enterprise. Commun ACM 51(9):72–79
8. Bernstein PA, Melnik S (2007) Model management 2.0: manipulating richer mappings. In: SIGMOD, pp 1–12
9. Bizer C (2009) The emerging Web of Linked Data. IEEE intelligent systems 5(3):87–92
10. Bizer C, Lehmann J, Kobilarov G, Auer S, Becker C, Cyganiak R, Hellmann S (2009) DBpedia: a crystallization point for the Web of Data. J Web Semant 7(3):154–165
11. Bouquet P, Giunchiglia F, van Harmelen F, Serafini L, Stuckenschmidt H (2004) Contextualizing ontologies. J Web Semant 1(4):325–343
12. Candan KS, Kim JW, Liu H, Suvarna R (2006) Discovering mappings in hierarchical data from multiple sources using the inherent structure. Knowl Inf Syst 10(2):185–210
13. Choi N, Song I-Y, Han H (2006) A survey on ontology mapping. SIGMOD Rec 35(3):34–41
14. Deutsch A, Nash A, Remmel JB (2008) The chase revisited. In: PODS, pp 149–158
15. Dorneles CF, Gonçalves R (2011) Approximate data instance matching: a survey. Knowl Inf Syst 27(1): 1–21
16. Dou D, McDermott DV, Qi P (2005) Ontology translation on the semantic web. J Data Semant 2:35–57
17. Euzenat J, Shvaiko P (2007) Ontology matching. Springer, Berlin
18. Fagin R, Kolaitis PG, Miller RJ, Popa L (2005) Data exchange: semantics and query answering. Theor Comput Sci 336(1):89–124
19. Flouris G, Konstantinidis G, Antoniou G, Christophides V (2012) Formal foundations for RDF/S KB evolution. Knowl Inf Syst 1–39. doi:10.1007/s10115-012-0500-2
20. Flouris G, Manakanatas D, Kondylakis H, Plexousakis D, Antoniou G (2008) Ontology change: classification and survey. Knowl Eng Rev 23(2):117–152
21. Glimm B, Hogan A, Krötzsch M, Polleres A (2012) OWL: yet to arrive on the web of data? In: LDOW
22. Groza T, Grimnes G, Handschuh S, Decker S (2011) From raw publications to Linked Data. Knowl Inf Syst 1–21. doi:10.1007/s10115-011-0473-6
23. Haas LM, Hernández MA, Ho H, Popa L, Roth M (2005) Clio grows up: From research prototype to industrial tool. In: SIGMOD, pp 805–810
24. Halevy AY (2001) Answering queries using views: a survey. VLDB J 10(4):270–294
25. Hopcroft JE, Tarjan RE (1973) Efficient algorithms for graph manipulation [H] (Algorithm 447). Commun ACM 16(6):372–378
26. Jing Y, Jeong D, Baik D-K (2009) SPARQL graph pattern rewriting for OWL-DL inference queries. Knowl Inf Syst 20(2):243–262
27. Klusch M, Fries B, Sycara KP (2009) OWLS-MX: a hybrid semantic web service matchmaker for OWL-S services. J Web Semant 7(2):121–133
28. Klyne G, Carroll JJ (2004) Resource description framework (RDF): concepts and abstract syntax. Technical report, W3C. http://www.w3.org/TR/rdf-concepts/
29. Kobilarov G, Scott T, Raimond Y, Oliver S, Sizemore C, Smethurst M, Bizer C, Lee R (2009) Media meets semantic web: how the BBC uses DBpedia and Linked Data to make connections. In: ESWC, pp 723–737
30. Leite M, Ricarte I (2012) Relating ontologies with a fuzzy information model. Knowl Inf Syst 1–33. doi:10.1007/s10115-012-0482-0
31. Lenzerini M (2002) Data integration: A theoretical perspective. In: PODS, pp 233–246
32. Maedche A, Motik B, Silva N, Volz R (2002) MAFRA: A MApping FRAmework for distributed ontologies. In: EKAW, pp 235–250
33. Mergen SLS, Heuser CA (2006) Data translation between taxonomies. In: CAiSE, pp 111–124
34. Mocan A, Cimpian E (2007) An ontology-based data mediation framework for semantic environments. Int. J Semant Web Inf Syst 3(2):69–98

35. Motik B, Horrocks I, Sattler U (2009) Bridging the gap between OWL and relational databases. J Web Semant 7(2):74–89
36. Mrissa M, Ghedira C, Benslimane D, Maamar Z, Rosenberg F, Dustdar S (2007) A context-based mediation approach to compose semantic web services. ACM Trans Internet Tech 8(1):4
37. Noy NF (2004) Semantic integration: a survey of ontology-based approaches. SIGMOD Rec 33(4):65–70
38. Noy NF, Klein MCA (2004) Ontology evolution: not the same as schema evolution. Knowl Inf Syst 6(4):428–440
39. Omelayenko B (2002) Integrating vocabularies: discovering and representing vocabulary maps. In: ISWC, pp 206–220
40. Palopoli L, Rosaci D, Terracina G, Ursino D (2005) A graph-based approach for extracting terminological properties from information sources with heterogeneous formats. Knowl Inf Syst 8(4):462–497
41. Parreiras FS, Staab S, Schenk S, Winter A (2008) Model driven specification of ontology translations. In: ER, pp 484–497
42. Pedrinaci C, Domingue J (2010) Toward the next wave of services: linked services for the web of data. J UCS 16(13):1694–1719
43. Petropoulos M, Deutsch A, Papakonstantinou Y, Katsis Y (2007) Exporting and interactively querying web service-accessed sources: the CLIDE system. ACM Trans Database Syst 32(4):22
44. Polleres A, Scharffe F, Schindlauer R (2007) SPARQL++ for mapping between RDF vocabularies. In: OTM, pp 878–896
45. Popa L, Velegrakis Y, Miller RJ, Hernández MA, Fagin R (2002) Translating web data. In: VLDB, pp 598–609
46. Prud'hommeaux E, Seaborne A (2008) SPARQL query language for RDF. Technical report, W3C. http://www.w3.org/TR/rdf-sparql-query/
47. Qin H, Dou D, LePendu P (2007) Discovering executable semantic mappings between ontologies. In: ODBASE, pp 832–849
48. Raffio A, Braga D, Ceri S, Papotti P, Hernández MA (2008) Clip: A visual language for explicit schema mappings. In: ICDE, pp 30–39
49. Rahm E, Bernstein PA (2001) A survey of approaches to automatic schema matching. VLDB J 10(4): 334–350
50. Ressler J, Dean M, Benson E, Dorner E, Morris C (2007) Application of ontology translation. In: ISWC, pp 830–842
51. Rivero CR, Hernández I, Ruiz D, Corchuelo R (2011a) Generating SPARQL executable mappings to integrate ontologies. In: ER, pp 118–131
52. Rivero CR, Hernández I, Ruiz D, Corchuelo R (2011b) Mosto: Generating SPARQL executable mappings between ontologies. In: ER, pp 345–348
53. Rivero CR, Hernández I, Ruiz D, Corchuelo R (2011c) On benchmarking data translation systems for semantic-web ontologies. In: CIKM, pp 1613–1618
54. Rivero CR, Hernández I, Ruiz D, Corchuelo R (2012) Benchmarking data exchange amongst semantic-web ontologies. IEEE Trans Knowl Data Eng PP(99). doi:10.1109/TKDE.2012.175
55. Rivero CR, Ruiz D, Corchuelo R (2011d) Automatic generation of executable mappings: a semantic-web technologies approach. Technical report TDG-247, University of Sevilla. http://www.tdg-seville.info/Download.ashx?id=247
56. Serafini L, Tamilin A (2007) Instance migration in heterogeneous ontology environments. In: ISWC, pp 452–465
57. Shadbolt N, Berners-Lee T, Hall W (2006) The semantic web revisited. IEEE Intell Syst 21(3):96–101