Datalift

Un ascenseur pour les données

ANR Contint – ANR-10-CORD-009

D4.2.2 Dataset interlinking module

Coordinator: Zhengjie Fan
With contributions from: Ngoc Nguyen Thinh Dong (INRIA), Jérôme Euzenat (INRIA), Fayçal Hamdi (IGN), François Scharffe (LIRMM)

Quality reviewer:	Serena Villata (INRIA-Wimmics)
Reference:	Datalift/2011/D4.2.2/v18
Project:	Datalift ANR Contint ANR-10-CORD-009
Date:	July 18, 2013
Version:	18
State:	final
Destination:	public

Executive Summary

This document updates the description of the implementation of the Datalift data interlinking module.

We first characterize the function of the data interlinking modules as providing a set of links from two RDF data sets. This function is further detailed with respect to the modalities that can be used in the platform implementation. In particular, we identify the need for translating alignments into Silk scripts that can be evaluated within the platform. We shows a minimalistic graphical user interface for generating links for an EDOAL alignment or an existing Silk script.

Then the EDOAL language for expressive alignments and the Silk scripting language for data interlinking are described. EDOAL allows for expressing expressive alignments between ontologies. Silk scripts instruct Silk where and how to find links in the datasets. We also describe a Silk extension for matching geographical data discussed in Deliverable 4.2.1.

The generation of Silk scripts from EDOAL alignments is then comprehensively presented. Based on the description of EDOAL alignments, the inductive generation of data sources, blocking factors and linkage rules are provided. Then their integration within a full Silk script is described.

We conclude by suggesting further evolutions of the data interlinking module.

Document Information

ANR Project Number	ANR Contint – ANR-10-CORD-009	**Acronym**	Datalift
Full Title	Un ascenseur pour les données		
Project URL	`http://www.datalift.org/`		
Document URL			

Deliverable	**Number**	4.2.2	**Title**	Dataset interlinking module
Work Package	**Number**	4	**Title**	Data interlinking

Date of Delivery	**Contractual**	M24	**Actual**	28-09-2012
Status	final		final ☐	
Nature	prototype ☒ report ☐ dissemination ☐			
Dissemination level	public ☒ consortium ☐			

Authors (Partner)	Zhengjie Fan, Ngoc Nguyen Thinh Dong (INRIA), Jérôme Euzenat (INRIA), Fayçal Hamdi (IGN), François Scharffe (LIRMM)			
Resp. Author	**Name**	Zhengjie Fan	**E-mail**	Zhengjie.Fan@inria.fr
	Partner	INRIA		

Abstract (for dissemination)	This report presents the second version of the interlinking module for the Datalift platform as well as strategies for future developments.
Keywords	data interlinking, linked data, instance matching

Version Log			
Issue Date	**Rev No.**	**Author**	**Change**
05/09/2012	1	J. Euzenat	Set up file and outline
11/09/2012	2	Z. Fan	Added EDOAL to SILK script section
13/09/2012	3	J. Euzenat	Added material from Thinh's master
15/09/2012	4	J. Euzenat	Improved translation chapter
08/11/2012	5	Z. Fan	Improved Generating Silk script Chapter
06/12/2012	6	Z. Fan	Improved Future work Chapter
09/12/2012	7	J. Euzenat	Reorganised deliverable; improved Generating Silk script Chapter
12/12/2012	8	Z. Fan	Improved Linkage Rule definition and the reference
16/12/2012	9	Z. Fan	Improved Future work Chapter
03/01/2013	10	F. Hamdi	Added description of WGS84 plug-in
09/01/2013	11	J. Euzenat	Written executive summary
28/02/2013	12	J. Euzenat	Revised all documents (reorganisation)
03/03/2013	13	J. Euzenat	Rewritten executive summary
04/03/2013	14	J. Euzenat	Rewritten EDOAL to SILK part
11/03/2013	15	Z. Fan	Revising according to François' comments
12/03/2013	16	J. Euzenat	Finished François' comments
18/03/2013	17	J. Euzenat	Added Zhengjie's appendix B
18/07/2013	18	J. Euzenat	Taken review remarks onto account

Table of Contents

1. Introduction

There are lots of isolated RDF data sets being published everyday. A web of linked RDF data sets is highly required for web users to share information with each other. For that purpose, the dataset published from the Datalift platform must be linked to other, likely external, datasets. This is the purpose of the interlinking module.

In Deliverable 4.2.1, we decided to provide this function through the embedding of the Silk tool within the Datalift platform. In the present deliverable we detail further this issue by:

- Better describing the function of the data interlinking module and the way it is implemented (§2);
- Describing Silk plug-ins developed for the purpose of the platform (§3);
- Providing full details of the *EDOAL2Silk* transformation allowing to generate a workable Silk script from an EDOAL alignment (§4).

2. Implementation choices

We first describes the functional interface of a data interlinking module (§2.1) before providing the principles of our implementation in Datalift (§2.2) and its minimal user interface (§2.3).

2.1 Functional description of the module

The interlinking module should have the functional signature:

$$f : RDF \times RDF \rightarrow linkset$$

From two RDF data sets, it should produce a linkset, i.e., a set of triples using the `owl:sameAs` predicate between resources of each data sets, a.k.a., `owl:sameAs` links. In principle, the first data set is the data set to be lifted, stored in the platform, while the second data set is a data set to be linked with.

There are many ways to implement this functional description. They should all be seen from this perspective (from the easiest to the more elaborate):

1. Fetch links from a link server, e.g., sameas.org:

$$f(dataset_1, dataset_2) = fetchFromLinkServer(dataset_1, dataset_2)$$

2. Find links from the URI fragments:

$$f(dataset_1, dataset_2) = StringToUri(dataset_1, dataset_2)$$

3. Generate links from a Silk script between the two datasets:

$$f(dataset_1, dataset_2) = Silk(dataset_1, dataset_2, SilkScript_{1,2})$$

3′. Generate links from a Silk script generated from an alignments (given by user) between the ontologies of the datasets:

$$f(dataset_1, dataset_2) = Silk(dataset_1, dataset_2, EDOAL2Silk(EDOALAlignment_{1,2})$$

3″. Generate links from a Silk script generated from vocabulary alignments (taken from server) between the ontologies of the datasets:

$$f(dataset_1, dataset_2) = Silk(dataset_1, dataset_2, EDOAL2Silk(FetchFromAlignServer(voc(dataset_1), voc(dataset_2)))$$

4. Generate links from a Silk script generated from vocabulary alignments (generated on the fly):

$$f(dataset_1, dataset_2) = Silk(dataset_1, dataset_2, EDOAL2Silk(match(voc(dataset_1), voc(dataset_2)))$$

4′. Generate links from a Silk script generated from vocabulary alignments (input through interface):

$$f(dataset_1, dataset_2) = Silk(dataset_1, dataset_2, EDOAL2Silk(Interface(voc(dataset_1), voc(dataset_2)))$$

5. Generate links from a Silk script generated from key inference:

$$f(dataset_1, dataset_2) = Silk(dataset_1, dataset_2, \\ Keys2Silk(findKeys(dataset_1, dataset_2)))$$

6. Generate links through learning from feedback:

$$f(dataset_1, dataset_2) = LearnerFromSilk(dataset_1, dataset_2)$$

2.2 Principles

As discussed in Deliverable 4.2.1, we have developed the module using Silk as a linking engine instead of writing a new one from scratch. The current prototype implements option 2 and integrates Silk to implement options 3-5. It produces `owl:sameAs` links from the data in the platform or elsewhere and store them in the public repository.

In Silk, the user has to write a script telling the system how to find links: between which classes, with which methods and which thresholds. On the other side, in Datalift, we want to find links without such an input. Hence we have developed a module for generating Silk scripts from EDOAL Alignments (*EDOAL2Silk* above used in options 3′, 3″, 4 and 4′). This has been implemented by as a renderer for EDOAL alignments which generates Silk scripts that can be run by the Silk processor (see Figure 2.1).

The current interlinking module is able to:

– Load its (RDF) data from the platform triple store;
– Load an EDOAL alignment from anywhere on the web;
– Generate a corresponding Silk script;
– Run the Silk script;
– Store the (RDF) results in the platform triple store.

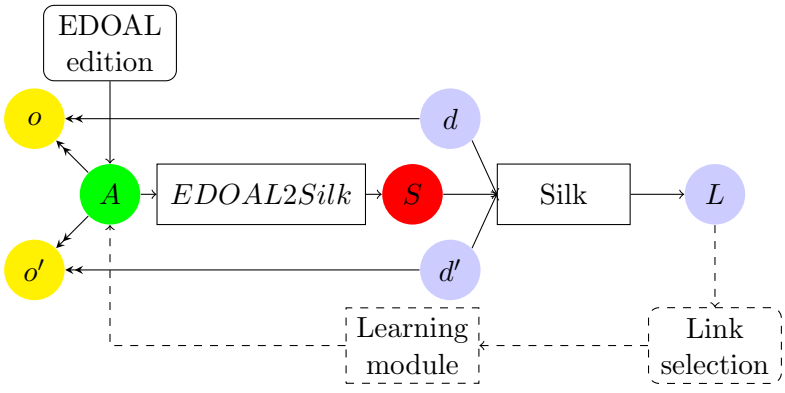

Figure 2.1: Link generation process: from an EDOAL alignment (A) between the ontologies (o and o') of two data sources expressed in these ontologies (d and d'), we generate a Silk script (S) which when passed to Silk generates a link set (L). It would be further useful to add an interface for generating EDOAL alignments from the ontologies and one for providing feedback about generated links so that the system can improve the alignment for generating better links.

The implementation of this approach is the object of this deliverable.

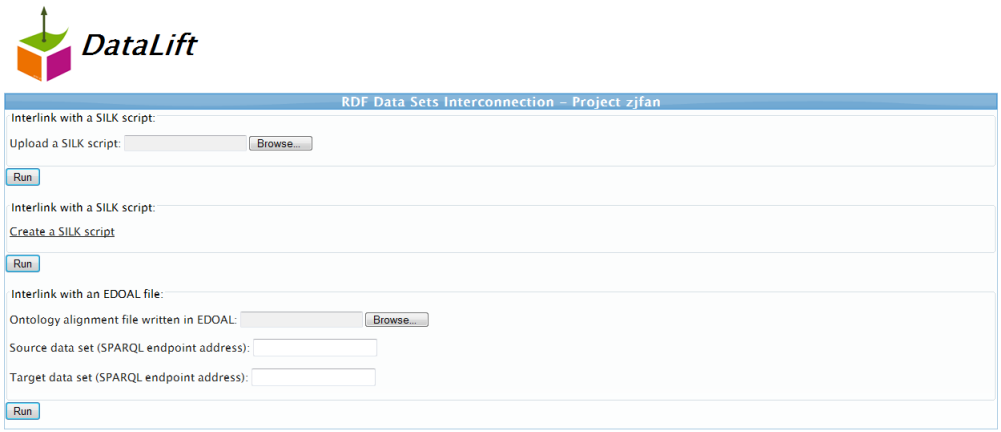

Figure 2.2: The user can either directly run a Silk script on the data or select an EDOAL alignment. In the second event, the system will generate the Silk script and run it.

The Datalift code repository uses Silk (version 2.5.3) since its version 0.6.5. We use Silk as a standing alone program but it can also be used as a web service or deployed to the cloud. It is even possible that the Silk developers offer online services that could be used by the Datalift platform.

2.3 User interface

The user interface for running the Datalift interlinking module offers these various options to users (see Figure 2.2):

- selecting a Silk script (option 3);
- creating a Silk script through the interface (option 3′);
- using an EDOAL alignment (option 3′).

Once the user has selected the source of alignment, it is presented with a SPARQL query that can retrieve the obtained links (see Figure 2.3).

The interlinking module may be used in fully autonomous mode as described in this deliverable. In this case, once the user has provided an EDOAL file and pointed to the two datasets, the module can generate links between the two data sets.

There may also be more control from the user side in providing the Silk scripts either by editing it within the Datalift platform or by designing it independently and providing it to the platform.

2.4 Conclusion

We have presented the principles under which the Datalift interlinking module has been designed.

The next chapter is a short reminder about the technologies which are used in its development (EDOAL, Silk, SPIN) as well as Silk extension specifically defined for Datalift.

Chapter 4 will present the implementation of *EDOAL2Silk*.

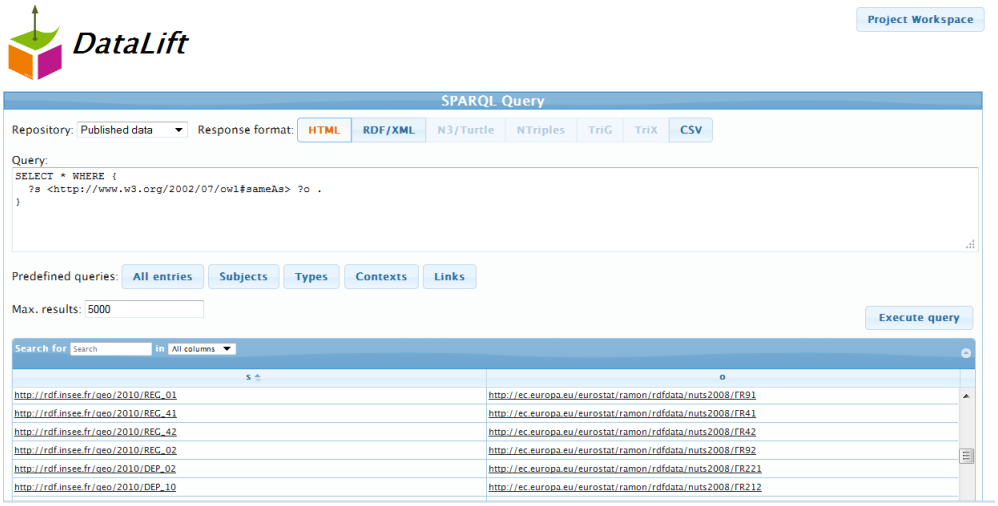

Figure 2.3: Resulting sameAs links as visible through the SPARQL query interface of the Datalift platform.

3. EDOAL and Silk Link Scripting Language

We consider here the use of alignments as input of the interlinking process. More precisely, as specified as *EDOAL2Silk*, we will transform EDOAL alignments, that can be manipulated by the Alignment API, into Silk scripts, that can be processed by Silk for generating link sets.

Hence we first present briefly EDOAL (§3.1) and Silk (§3.2) for the sake of completeness. We then discuss the problem of comparing concrete values: EDOAL promoted the use of XPath functions, Silk has its own functions and SPARQL itself can be extended by the SPIN. SPIN is presented in §3.3. Finally we also present briefly the extension of Silk that we made in order to compare geographic areas (§3.4).

In the next chapter, we present in detail the *EDOAL2Silk* algorithm for generating Silk scripts from EDOAL expressions (§4).

3.1 EDOAL

The type of alignments required to express interlinking with the precision of Silk scripts goes beyond simple alignments. For expressing expressive alignments, we have designed the EDOAL language [2][1].

For dealing with linked data, we need to retrieve all instances of EDOAL expressions appearing in both sides of a correspondence in order to see if they can be matched. For instance, the following EDOAL correspondence, corresponds to Figure 3.1:

```
<Cell>
  <entity1>
    <edoal:Class rdf:about="&insee;Region" />
  </entity1>
  <entity2>
    <edoal:Class>
      <edoal:and rdf:parseType="Collection">
        <edoal:Class rdf:about="&nults;NUTSRegion"/>
        <edoal:AttributeValueRestriction>
          <edoal:onAttribute>
            <edoal:Property rdf:about="&nuts;level"/>
          </edoal:onAttribute>
          <edoal:comparator rdf:resource="&xsd;equals"/>
          <edoal:value>2</edoal:value>
        </edoal:AttributeValueRestriction>
        <edoal:AttributeValueRestriction>
          <edoal:onAttribute>
            <edoal:Relation rdf:about="&nuts;hasParentRegion"/>
          </edoal:onAttribute>
          <edoal:comparator rdf:resource="&xsd;equals"/>
          <edoal:value><edoal:Instance rdf:about="&nuts;FR1"/></edoal:value>
        </edoal:AttributeValueRestriction>
      </edoal:and>
    </edoal:Class>
  </entity2>
  <measure rdf:datatype='&xsd;float'>1.</measure>
  <relation>=</relation>
</Cell>
```

it expresses the equivalence between the class `Region` in the insee data set with `NUTSRegion` instances whose `level` is equal to 2 and whose parent region is `FR1` in the NUTS data set.

[1] `http://alignapi.gforge.inria.fr/edoal.html`

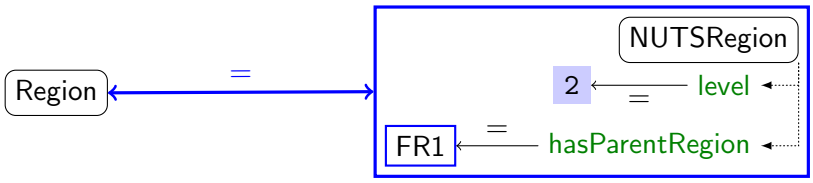

Figure 3.1: The EDOAL correspondence between INSEE regions and NUTS French region of level 2 corresponding to the example given in the Silk script of §3.1.

For the purpose of the transformation presentation, we summarize the EDOAL grammar:
For class expressions:
$C ::= u$
$\quad | \neg C \mid C \cap C \mid C \cup C$
$\quad | \; occ(A, cp, n)$
$\quad | \; dom(R, C)$
$\quad | \; type(P, t)$
$\quad | \; val(A, cp, v)$
For property expressions:
$P ::= u$
$\quad | \neg P \mid P \cap P \mid P \cup P \mid R \circ P$
$\quad | \; dom(C)$
$\quad | \; type(t)$
$\quad | \; val(cp, v)$
For relation expressions:
$R ::= u$
$\quad | \neg R \mid R \cap R \mid R \cup R \mid R \circ R$
$\quad | \; reflex(R) \mid sym(R) \mid \overline{R} \mid R^{-1}$
$\quad | \; dom(C)$
$\quad | \; coDom(C)$
For instance & value expressions:
$I ::= u$
$v ::= I \mid literal$

C denotes a class. u denotes a URI. A denotes a property or a relation. P denotes a property. R denotes a relation (object property). $\cup, \cap, \neg, \circ, dom$ and $coDom$ denote union, intersection, complement, composition, domain, range restriction respectively. t denotes a data type. v denotes a value expression. cp denotes a comparator, n denotes an integer. R^{-1}, $sym(R)$, $\overline{R}$ and self denotes the inverse, the symmetric closure, the transitive closure and the identity relation respectively.

3.2 LSL

The Silk framework[2] provides a declarative language for specifying which types of RDF links should be discovered between data sources as well as which conditions data items must fulfill in order to be interlinked [5]. In other terms, it is an interlinking software that helps to

[2] `http://www4.wiwiss.fu-berlin.de/bizer/silk/`

find out similar items in RDF, and it accepts RDF data sets both in file and in SPARQL endpoint.

LSL is a script specification language. Silk scripts fulfill two roles, as illustrated by the script to interlink persons in the two data sets example.org/humans.rdfs and inria.fr/humans.rdfs. [3]:

- It is an alignment: it specifies the classes in which entities to link can be found. Restrictions to *humans1:Adult* and *humans2:Person* whose *age* is greater than 17 are in fact an alignment between these two concepts. Similarly, the compared properties *humans1:name* and *humans2:name* provide the correspondences between properties.
- It specifies how to link entities. Indeed, what Silk brings in addition to an alignment is the specification of how to decide if two entities should be linked: when the average (*AVG*) of their respective distances (*Compare*) is over a threshold (*Threshold*, there are two thresholds, one for accepting automatically the equivalence and one for drawing the attention of a user).

3.3 SPIN

In the EDOAL language, data transformations can be used to transform data between property values. Transformations do not express constraints on the classes or properties, but constraints on instances that should match. Using operators from the XQuery 1.0 and XPath 2.0 Functions and Operators or from other service (specified by URIs), transformations can be invoked, for example for dynamic transformations like currency conversions:

```
<align:Cell>
  <align:entity1><Property rdf:about="&O1;price" /></align:entity1>
  <align:entity2><Property rdf:about="&O2;hasPrice" /></align:entity2>
  <transformation>
    <Transformation edoal:direction="o-">
      <entity1>
        <Property><compose rdf:parseType="Collection" /></Property>
      </entity1>
      <entity2>
        <Apply edoal:operator="http://www.google.com/finance/converter">
          <arguments rdf:parseType="Collection">
            <Property><compose rdf:parseType="Collection" /></Property>
            <Literal edoal:string="EUR" />
            <Literal edoal:string="CNY" />
          </arguments>
        </Apply>
      </entity2>
    </Transformation>
  </transformation>
</align:Cell>
```

The problem is to translate operators in the EDOAL expressions into the Silk scripts. This may occur in two places in the generated Silk script:

- Graph patterns used in dataset descriptions (see §4.1), and
- Silk linkage rules constraints (see §4.3).

Since graph patterns, are part of SPARQL, we consider how SPARQL can be extended with operators. In spite of supports about SPARQL extensions in many implementations, SPARQL provides no way to use directly the operators above in SPARQL queries. So that it may be wise to change the way operators are used in transformations in EDOAL.

Instead of using operators from the XQuery 1.0 and XPath 2.0 Functions and Operators or from other unknown services, we can define user owned functions by using SPARQL Rules, which are a collection of RDF vocabularies, such as SPARQL Inferencing Notation (SPIN)[3]. SPIN is a way to define rules and constraints for Semantic Web data. It is an open specification that has been submitted to the World Wide Web Consortium (W3C) [6]. Using SPIN, one can express business rules in SPARQL and execute them directly on any RDF data.

SPIN may be used to compute the value of a property based on other properties, for example, the area of a rectangle as the product of its height and width, the age of a person as the difference between today's date and person's birthday, the display name as the concatenation of the first and last names.

The SPIN framework is made of three layers:

1. **An RDF vocabulary for SPARQL** A SPIN implementation stores SPARQL queries as RDF triples, like any other data or metadata in a Semantic Web model.
2. **SPIN Vocabulary** This vocabulary includes terms such as `spin:constraint` and `spin:constructor`, which allow for defining business rules and attach them to classes.
3. **Module Library** The library holds frequently needed modeling patterns, with functions and templates to constrain cardinalities and value ranges. SPIN allows for defining and storing user owned re-usable query templates; it also provides several predefined ones based on modeling patterns that are common in real-world applications.

SPIN can be used to define new SPARQL functions so that these new function can be used in expressions such as `FILTER` or `BIND` clauses. Each SPIN function is an instance of the meta-class `spin:Function`, a system class that groups together all available functions in the class hierarchy. A new function may be created by creating a subclass of `spin:Functions`. The name of the function defines a URI.

The following example declares a new function `ex:cardinality` that gets the number of values of a given property at the current subject (?*this*), using a SPARQL `COUNT` query.

```
ex:cardinality a spin:Function;
  rdfs:subClassOf spin:Functions ;
  rdf:comment "Get the number of values for a given property of the current subject (?this)"
  rdf:label "cardinality"^^xsd:string ;
  spin:constraint
    [ a spl:Argument;
      rdf:comment "The property to get the cardinality of"
      spl:predicate sp:arg1 ];
  spin:body
    [ a sp:Select;
      sp:resultVariables ([ a sp:Count ;
                                          sp:expression sp:_object ]);
      sp:where ([ sp:object sp:_object ;
                            sp:predicate spin:_arg1;
                            sp:subject spin:_this ]) ].
```

Once the function is defined, it can be used in SPARQL queries such as:

```
FILTER ( ex:cardinality( ex:child ) > 2 )
```

Because SPIN functions are formalized using an RDF vocabulary, for describing a SPARQL function body in SPARQL and arguments passed to the function, they can be shared on the

[3] `http://www.topquadrant.com/spin/tutorial/`

web using their URIs. This means that we can import them for using, for the above example of unit conversion, once the unit conversion functions have been defined and formalized in an ontology, we can publish this SPIN library and use it.

`func:convert` is a user-defined function that can be shared together with the data models. The translated SPARQL query for the correspondence above would be as follows:

```
CONSTRUCT { ?x O1:hasPrice ?y. }
WHERE {
  ?x O2:price ?z .
  LET ( ?y := func:convert( ?z, "EUR", "CNY" ) )
}
```

3.4 Silk extensions

We present a Silk extention developed for interlinking geographic data on a more precise basis than comparing coordinates as provided by Silk.

Such an extension is very important for precise interlinking of geographic data as provided by IGN.

3.4.1 Computing the distance between two WGS84 coordinates

The WGS84 coordinates are called "geographical coordinates". A point is located at the surface of the earth using two measures of angles: its longitude and latitude. To compute the distance between two points, it is necessary to take into account the curvature of the Earth (see Figure 3.2).

In first instance, we can consider that the Earth is modeled by a sphere with radius R, and calculate this distance on the sphere, as is the case for the geographical measure used by Silk. However, in this type of coordinate system, the Earth is modeled by an oblate ellipsoid of revolution that is the mathematical form that best approximates its "real" shape.

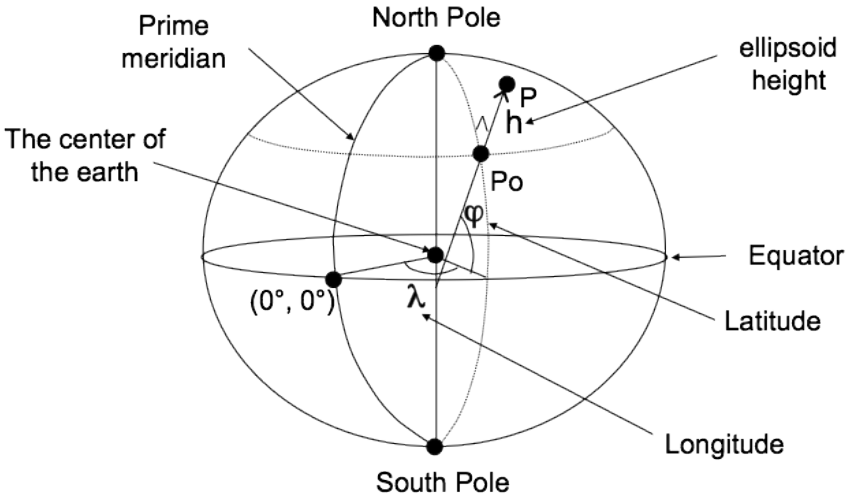

Figure 3.2: Reference model for geographic coordinates computed on the Earth surface.

To get an accurate distance measurement, it is better to calculate the distance by taking into account the ellipsoid coordinate system, in our case, the WGS84 ellipsoid (which is also called WGS84).

JTSPointDistance measure

In our geographical measure `JTSPointDistance`, we used GIS libraries which takes the ellipsoid coordinate system in consideration. More specifically, we used the JTS library, which allows for configuring the ellipsoid parameters, and thus improving, as explained below, the accuracy of the distance measure calculated between two coordinates. Here is the java implementation of this method:

```
public class JTSPointsDistance {

    private double lat1;
    private double long1;
    private double lat2;
    private double long2;

    public JTSPointsDistance(double lat1, double long1, double lat2, double long2){
        this.lat1 = lat1;
        this.long1 = long1;
        this.lat2 = lat2;
        this.long2 = long2;
    }

    public double getPointsDistance() throws NoSuchAuthorityCodeException, ↩
        FactoryException, TransformException {

        Coordinate c1 = new Coordinate(lat1,long1);
        Coordinate c2 = new Coordinate(lat2,long2);

        CRSAuthorityFactory crsFactory = ReferencingFactoryFinder.↩
            getCRSAuthorityFactory("EPSG", null);

        // The parameters of the ellipsoid coordinate system (4326 corresponds to ↩
            the WGS84 ellipsoid)
        GeographicCRS crs = crsFactory.createGeographicCRS("4326");

        return JTS.orthodromicDistance(c1, c2, crs);
    }
}
```

Example use of JTSGeographicMetric

The Silk script below, generates links between two locations by comparing their addresses and their coordinates (using `JTSGeographicMetric`).

```
<?xml version="1.0" encoding="utf-8" ?>
<Silk>
  <Prefixes>...</Prefixes>

  <DataSources>...</DataSources>

  <Interlinks>
    <Interlink id="datalift">...

    <LinkageRule>
      <Aggregate type="average">
        <Compare metric="equality" required="true">
          <Input path="?a/bdadr:numero"/>
          <TransformInput function="numReduce">
            <Input path="?b/sp:adresse/sp:ligneAdresse"/>
          </TransformInput>
        </Compare>
```

```
        ...
        <Compare metric="jtswgs84" threshold="10" required="true">
          <TransformInput function="concat">
            <Input path="?a/bdadr:y" />
            <Input path="?a/bdadr:x" />
            <Param name="glue" value=" " />
          </TransformInput>
          <TransformInput function="concat">
            <Input path="?b/geo-pos:lat" />
            <Input path="?b/geo-pos:long" />
            <Param name="glue" value=" " />
          </TransformInput>
        <Param name="unit" value="m" />
      </Compare>
    </Aggregate>
  </LinkageRule>

    ...
    </Interlink>
  </Interlinks>
</Silk>
```

3.5 Conclusion

We have introduced the two main languages on which the current interlinking module is based: Silk and EDOAL. The goal of the module is to transform an EDOAL alignment into a Silk script. This is presented in the next chapter.

4. From EDOAL to Silk scripts

The goal of this chapter is to specify how Silk scripts are generated from EDOAL alignments.

We first present how each component of a Silk script is generated from an alignment:

1. Generating graph patterns from EDOAL expressions allows to fill the Source and Target datasets part of Silk scripts, which selects instances to compare (§4.1). It does not actually compare resources.
2. Generating the blocking part, which divides the entities in the data sets into several sets to compare (§4.2).
3. Generating linkage rules which actually compare instances (§4.3). The linkage rule specifies how to compare the entities selected before.

Then we consider how to assemble a Silk script from these components (§4.4).

Relating expressive alignments and SPARQL has already been the topic of previous papers [4, 7]. However, these papers are concerned by translating data from alignments, not interlinking them.

4.1 Generation of Source and Target datasets

From the correspondence of Figure 3.1, we want to generate automatically the following graph patterns which respectively extract the two corresponding set of entities:

```
?r rdf:type insee:Region .
```

and

```
?n rdf:type nuts:NUTSRegion .
?n nuts:level 2^^xsd:int .
?n nuts:hasParentRegion nuts:FR1 .
```

These graph patterns will be used in the Silk script as DataSource and DataTarget selectors.

In the data set part, the pair of classes to be compared are defined. The graph patterns of classes are specified here.

In EDOAL, the expressions can be constructed by using the operators or defined through restrictions. Based on the specification of EDOAL expressions with the grammars of Section 3.1, we translate into graph patterns all of the concepts. This transformation depends on the type of entity (Class, Property, Relation), the operations (conjunctions, union, negation, compose, etc.) and the restrictions (domains, values, types, occurrences) they are built from. Each expression (entity) will be translated into a graph pattern.

Because Silk does not support SPARQL 1.1 and dealing with negation would lead to too many comparisons, correspondences with negations ($\neg C$) are ignored.

We designed a EDOAL translation to graph pattern T which is described in the following sections. There is one section for each entity types according to the EDOAL grammar.

4.1.1 Translation of class expressions

For class expressions, the function $T_s(C)$ takes as argument C the expression to convert and s the entity typed by this class. When invoking T, s is usually a variable.

A class C can be constructed through one of the three operators and, or, not. A class can also be identified by using its URI or defined through a restriction:

- *AttributeDomainRestriction* which restricts the values of relations to be in a particular class;
- *AttributeTypeRestriction* which restricts the values of properties to be in a particular type;
- *AttributeValueRestriction* which restricts the value of properties or relations to a particular value;
- *AttributeOccurrenceRestriction* which constrains the cardinality of a property or relation.

In the following we identify operators which require SPARQL 1.1. We also use an operation, $!x$ which may generate a new variable ($?x$) or a blank (_:x) depending of the context. Here it will always generate a new variable.

$$
\begin{aligned}
T_s(u) &= s \texttt{ rdf:type } \langle u \rangle. \\
T_s(\neg C) &= MINUS\ \{T_s(C)\} && (SPARQL\ 1.1) \\
T_s(C \cup C') &= T_s(C)\ UNION\ T_s(C') \\
T_s(C \cap C') &= T_s(C)\ T_s(C') \\
T_s(dom(R, C)) &= T_{s,!o}(R)\ T_{!o}(C) \\
T_s(occ(A, cp, n)) &= T_{s,!o}(A)\ FILTER(COUNT(!o)\ cp\ n) && (SPARQL\ 1.1) \\
T_s(type(P, t)) &= T_{s,!o}(P)\ FILTER(datatype(!o) = t) \\
T_s(val(A, =, v)) &= T_{s,v}(A) \\
T_s(val(A, cp, v)) &= T_{s,!o}(A)\ FILTER(!o\ cp\ v)
\end{aligned}
$$

Example: value restriction class

Entity (EDOAL)	**Graph Pattern (SPARQL)** $T_s(u)$ = *?s rdf:type <u> .* $T_s(val(A, cp, v))$ = $T_{s,o}(A)$ *FILTER(?o cp v)* $T_{s,o}(u)$ = *?s <u> ?o .*

`<Class>`
`  <and rdf:parseType="Collection">`
`  <Class rdf:about="#Person" />`
`  <AttributeValueRestriction>`
`  <onAttribute>`
`  <Property rdf:about="#age">`
`  </onAttribute>`
`  <comparator rdf:resource=`
`  "&edoal;#greater-than" />`
`  <value>`
`  <Literal edoal:type="xsd;integer"`
`  edoal:string="17" />`
`  </value>`
`  </AttributeValueRestriction>`
`  </and>`
`</Class>`
	`?x rdf:type <humans:Person> .`
	`?x <humans:age> ?age .`
	`FILTER ( <xsd:integer>(?age) > 17 )`

4.1.2 Translation of property expressions

For the property expressions, we will use the translation function $T_{s,o}(P)$ such that s is the subject and o is the object in a SPARQL triple pattern, and the predicate P establishes a relationship between the subject s and the object o.

Properties correspond to data properties in OWL. Properties entities can be constructed using one of the operators *and, or, not* and *compose*. They can also be identified by using its URI or defined through a restriction:

- *PropertyDomainRestriction* which restricts the subjects of properties to be in a particular class;
- *PropertyTypeRestriction* which restricts the values of properties to be in a particular type;
- *PropertyValueRestriction* which restricts the value of properties to a particular value.

$$
\begin{aligned}
T_{s,o}(u) &= s\ \langle u \rangle\ o. \\
T_{s,o}(\neg P) &= MINUS\ \{T_{s,o}(P)\} \\
T_{s,o}(P \cup P') &= T_{s,o}(P)\ UNION\ T_{s,o}(P') \\
T_{s,o}(P \cap P') &= T_{s,o}(P)\ T_{s,o}(P') \\
T_{s,o}(R \circ P) &= T_{s,!x}(R)\ T_{!x,o}(P) \\
T_{s,o}(dom(C)) &= T_s(C) \\
T_{s,o}(type(t)) &= FILTER(datatype(o) = t) \\
T_{s,o}(val(cp, v)) &= FILTER(o\ cp\ v)
\end{aligned}
\qquad (SPARQL\ 1.1)
$$

Example: type restriction property

Entity (EDOAL)	**Graph Pattern (SPARQL)** $T_{s,o}(type(t))$ = *FILTER(datatype(?o) = t)*

`<Property>`
`  <PropertyTypeRestriction>`
`  <datatype>`
`  <Datatype rdf:about="&xsd;integer"/>`
`  </datatype>`
`  </PropertyTypeRestriction>`
`</Property>`
	FILTER(datatype(?o) = xsd:integer)

4.1.3 Translation of relation expressions

For the relation expressions, we will use the translation function $T_{s,o}(R)$ such that s is the subject and o is the object in a SPARQL triple pattern, and the predicate R establishes a relationship between the subject s and the object o.

Relations correspond to object properties in OWL. Relation expressions can be constructed using one of the operators and, or, not and compose as well as closure operators *inverse, symmetric, transitive* and *reflexive*. It can also be identified by using its URI or defined through a restriction. There are two classes of relation restrictions (*RelationDomainRestriction* and *RelationCoDomainRestriction*).

$$
\begin{aligned}
T_{s,o}(u) &= s\ \langle u \rangle\ o. \\
T_{s,o}(\neg R) &= MINUS\ \{T_{s,o}(R)\} && (SPARQL\ 1.1) \\
T_{s,o}(R \cup R') &= T_{s,o}(R)\ UNION\ T_{s,o}(R') \\
T_{s,o}(R \cap R') &= T_{s,o}(R)\ T_{s,o}(R') \\
T_{s,o}(R \circ R') &= T_{s,!x}(R)\ T_{!x,o}(R') \\
T_{s,o}(R^{-1}) &= T_{o,s}(R) \\
T_{s,o}(sym(R)) &= T_{s,o}(R)\ UNION\ T_{o,s}(R) \\
T_{s,o}(\overline{R}) &= s\ R*\ o. && (SPARQL\ 1.1) \\
T_{s,o}(reflex(R)) &= T_{s,o}(R)\ UNION\ FILTER(s = o) \\
T_{s,o}(self) &= FILTER(s = o) \\
T_{s,o}(dom(C)) &= T_s(C) \\
T_{s,o}(coDom(C)) &= T_o(C)
\end{aligned}
$$

Example: Transitive relation using the path property (+, *) of SPARQL 1.1

Entity (EDOAL)	**Graph Pattern (SPARQL)** $T_{s,o}(\overline{R})$ = *Triple(s, R*, o)*

`<Relation>`
`  <transitive>`
`  <Relation rdf:about="&wine;loc" />`
`  </transitive>`
`</Relation>`
	`?s  <wine:loc>*  ?o .`

4.1.4 Translation of data transformations

Data comparators (*cp*) are used within the FILTER clause of the transformations. Some of these operators are defined in a standard way through SPARQL (<=, >=, <, >, =, ! =). For other operators, the question is whether, SPIN extensions (see §3.3) may be used within Silk or if it is preferable to use Silk extensions (§3.4).

4.2 Generation of blocking

Blocking is a way to reduce the size of the comparisons across two data sets. It divides the instances into several groups by comparing specific property pairs, so that only the instances in corresponding groups are compared with each other. Suppose there are M instances in the source data set and N instances in the target data set. Theoretically, there should be $M * N$ comparisons. If blocking is set to divide the instances into P groups, there should be $P * \frac{M}{P} * \frac{N}{P}$ comparisons on average.

The formal definition of setting the blocking property pair is the same as the one in the linkage rules introduced below (§4.3). However, usually the property that can represent the feature of the data set context, and also with a few property values are set here. For example, given two data sets of music to interlink, the property related to *country* can be used in the block part to divide the datasets into several parts. So the instances within the

same country of the source data set will only be compared with the instances within the same country of the target data set.

Various methods may be used for blocking, starting by using class correspondences in order to compare only the instances of corresponding classes. The *discriminability* criteria defined in [8, 1] shows to what extent a group of properties can identify the records in the data base. If it is equal to 1, then the property set is a key: it identifies only one record per value combinations. If it is different from 1, a value combination will group closely similar instances that may then be compared. With discriminability equal to 1., sets of properties may be used directly as linkage rules, otherwise, they may be used as blocking factors. If there are M instances, and we wish to divide them into N blocks. The threshold of the discriminability should be set to $\frac{M}{N}$.

The key inference technique for interlinking has been implemented but is not integrated in the interlinking module yet.

4.3 Generation of linkage rules

The strategy followed by Silk is made of four steps, each one corresponding to a different operation:

navigating to the parts of the entities to compare (`Input`);

transforming them to make them comparable (`TransformInput`);

comparing them (`Compare`); and

aggregating the results of their comparison into a single similarity between the two entities (`Aggregate`).

These steps are first presented through an example before presenting the systematic generation of linkage rules.

4.3.1 Example

These steps are present in the following LinkageRule generated for our example:

```
<LinkageRule>
    <Aggregate type="max">
        <Compare metric="levenshteinDistance" weight="1" threshold="0">
            <TransformInput function="lowerCase">
                <Input path="?a/insee:nom[@lang='fr']" />
            </TransformInput>
            <TransformInput function="lowerCase">
                <Input path="?b/eurostat:name" />
            </TransformInput>
        </Compare>
    </Aggregate>
</LinkageRule>
```

Our example starts from the two classes to compare: insee:Regions to eurostat:NUTSRegions. For that purpose, we identify the parts of these entities to compare. This is achieved by finding correspondences in the alignment which match these parts. This can be given by the following:

```
<map>
    <Cell>
        <entity1>
            <edoal:Property rdf:about="&insee;nom"/>
        </entity1>
        <entity2>
            <edoal:Property rdf:about="&eurostat;name"/>
        </entity2>
        <relation>equivalence</relation>
        <measure>1.0</measure>
    </Cell>
</map>
```

If for a class correspondence, there is at least one applicable correspondence, then an interlink rule can be generated for the Silk script. In our case, *Nom* is a property applying to class Region and *name* is a property applying to class NUTSRegion, so the values of these properties can be used to compare the instances. From there, the four steps of the LinkageRule can be generated:

navigation the *nom* property should be inserted into the input path with the same variable that appears in the source data set section. When there is value restriction, it is translated as a constraint, such as $?a/insee : nom[@lang =' fr']$. When there is no value restriction, it is simply translated as a path, such as $?b/eurostat : name$.

transformation when two values of the LinkageRule cannot be compared immediately because of coding or format difference, they may be transformed. A transformation function should be set for changing the values into a uniform format. In EDOAL, the transformation may be described in each correspondence. For example, "replace" can change certain letters in a string to other letters, "lowerCase" can change all letters into lower case. In Silk, the transformation can be defined recursively[1].

comparison uses a particular metrics to compare the values extracted by the paths. If values are strings, they will use a string comparison method, "levenshteinDistance"; if they are geometric points, the method could compare coordinates (as the "jtswgs84" method). The choice of metrics is made according to the range of the datatype properties found in the ontologies. Other optional parameters can be set such as "required", "weight" or "threshold". It may be possible, for instance, to assign weights depending on the coverage degree of properties.

aggregation if finally performed for issuing one value between two instances from the values obtained by comparing their properties. It can be the "AverageAggregator" which computes the weighted average result of comparing results. It also can be "max" to set the final aggregate value to be the maximum similarity of returned comparing results. Other optional parameters are "required" and "weight". Weights could be set based on the confidence in the property correspondence.

Silk offers a whole battery of operations:

- navigation operators, inspired from the XPath language are provided in Table 4.1;
- transformation functions (lowerCase, etc.);
- comparison functions (levenshteinDistance, etc.);
- aggregation operators (max, min, average, etc.).

[1] https://www.assembla.com/spaces/silk/wiki/Transformation

4.3.2 Aggregation generation

For each class related by a correspondence in the alignment $\mathbb{A}$, a linkage rule will be generated aggregating the comparison of all the properties. It may be possible to select the relevant properties to compare and their weights through two methods:

- key: a key is a minimal set of properties for identifying instances. It shows the importance of the property within a data set. Thus, it can be used to mark the importance of comparing pairs.
- learning pattern: it shows which property appears commonly in links. It can be used to mark the importance of comparing pairs as the key also.

Currently, given two class URIs u and u', we generate the comparison of all attributes and average their comparison results:

$$R_{s,s'}(u,u',\mathbb{A}) = Average_{\langle a,a',=,tr\rangle \in \mathcal{K}(u,u',\mathbb{A})} C_{s,range(a),s',range(a')}(a,a',tr)$$

$\mathcal{K}(u,u',\mathbb{A})$ is the set of property or relation correspondences related to the class u and u' in $\mathbb{A}$, i.e., those whose domain is u and u' respectively. s, s' are two variables that will be bound to instances of classes u and u'. The datatype associated to the properties are passed to the transformation. The transformation tr if available in the EDOAL correspondence is also passed as argument.

Because classes descriptions in EDOAL may be non atomic, we have to specify how the compound expressions are compared:

$$\begin{aligned}
R_{s,s'}(C \cap C', C'', \mathbb{A}) &= Min(R_{s,s'}(C, C'', \mathbb{A}), R_{s,s'}(C', C'', \mathbb{A})) \\
R_{s,s'}(C \cup C', C'', \mathbb{A}) &= Max(R_{s,s'}(C, C'', \mathbb{A}), R_{s,s'}(C', C'', \mathbb{A})) \\
R_{s,s'}(C, C' \cap C'', \mathbb{A}) &= Min(R_{s,s'}(C, C', \mathbb{A}), R_{s,s'}(C, C'', \mathbb{A})) \\
R_{s,s'}(C, C' \cup C'', \mathbb{A}) &= Max(R_{s,s'}(C, C', \mathbb{A}), R_{s,s'}(C, C'', \mathbb{A})) \\
R_{s,s'}(\neg C, C', \mathbb{A}) &= \varnothing \\
R_{s,s'}(C, \neg C', \mathbb{A}) &= \varnothing \\
R_{s,s'}(C, occ(A, cp, n), \mathbb{A}) &= \varnothing \\
R_{s,s'}(C, dom(R, C'), \mathbb{A}) &= \varnothing \\
R_{s,s'}(C, type(P, t), \mathbb{A}) &= \varnothing \\
R_{s,s'}(C, val(A, cp, v), \mathbb{A}) &= \varnothing \\
R_{s,s'}(occ(A, cp, n), C', \mathbb{A}) &= \varnothing \\
R_{s,s'}(dom(R, C), C', \mathbb{A}) &= \varnothing \\
R_{s,s'}(type(P, t), C', \mathbb{A}) &= \varnothing \\
R_{s,s'}(val(A, cp, v), C', \mathbb{A}) &= \varnothing
\end{aligned}$$

here $\varnothing$ values have no influence on operators Min, Max or $Average$. It only means that no comparison is generated in this case. These constraints have usually be used in the expression of the data sources and thus are not necessary anymore in the comparison.

4.3.3 Comparison generation

The comparison between two attributes is generated by $C_{s,t,s',t'}(a,a',tr)$ which literally means: generate the comparison of instances s and s' attributes a and a' of types t and t', knowing that a transformation tr may be useful for converting a units into a''s.

This comparison is generated in two distinct ways depending if the attribute expressions are structural, i.e., built from constructors $\neg$, $\cap$ $\cup$ or sym, or navigational, i.e., built from constructor $\circ$, $^{-1}$, $\bar{\cdot}$, $reflex$ or constraints dom, $coDom$, $type$ or val. In the former case, C will aggregate the comparison of subcomponents, while in the latter it will generate paths for applying the comparison.

The comparisons are applied to attributes based on their correspondences. We assume that they are in a form $P\hat{\cap}X$ in which P is a path in inverted normal form, i.e., in which all converse are set at the lower level, and X is the set of constraints on the extremities of attributes. This can be achieved by the normalization function N:

$$\begin{aligned}
N(u) &= u \\
N(u^{-1}) &= u^{-1} \\
N((A^{-1})^{-1}) &= N(A) \\
N((\neg A)^{-1}) &= \neg N((A^{-1})) \\
N((A \cap A')^{-1}) &= N(A^{-1}) \cap N(A'^{-1}) \\
N((A \cup A')^{-1}) &= N(A^{-1}) \cup N(A'^{-1}) \\
N((R \circ R')^{-1}) &= N(R'^{-1}) \circ N(R^{-1}) \\
N(sym(R)^{-1}) &= sym(N(R)) \\
N(reflex(R)^{-1}) &= reflex(N(R^{-1})) \\
N(\overline{R}^{-1}) &= \overline{N(R^{-1})} \\
N(dom(C)^{-1}) &= coDom(N(C)) \\
N(coDom(C)^{-1}) &= dom(N(C))
\end{aligned}$$

and separating at each nesting levels, the constraints from the paths.

We consider the main navigational part:

$$\begin{aligned}
C_{s,t,s',t'}(\neg a, a', tr) &= 1 - C_{s,t,s',t'}(a, a', tr) && (NoSilk) \\
C_{s,t,s',t'}(a, \neg a', tr) &= 1 - C_{s,t,s',t'}(a, a', tr) && (NoSilk) \\
C_{s,t,s',t'}(a' \cap a'', a, tr) &= Min(C_{s,t,s',t'}(a', a, tr), C_{s,t,s',t'}(a'', a, tr)) \\
C_{s,t,s',t'}(a' \cup a'', a, tr) &= Max(C_{s,t,s',t'}(a', a, tr), C_{s,t,s',t'}(a'', a, tr)) \\
C_{s,t,s',t'}(a, a' \cap a'', tr) &= Min(C_{s,t,s',t'}(a, a', tr), C_{s,t,s',t'}(a, a'', tr)) \\
C_{s,t,s',t'}(a, a' \cup a'', tr) &= Max(C_{s,t,s',t'}(a, a', tr), C_{s,t,s',t'}(a, a'', tr)) \\
C_{s,t,s',t'}(a, sym(a'), tr) &= Max(C_{s,t,s',t'}(a, a', tr), C_{s,t,s',t'}(a, a'^{-1}, tr)) \\
C_{s,t,s',t'}(sym(a), a', tr) &= Max(C_{s,t,s',t'}(a, a', tr), C_{s,t,s',t'}(a^{-1}, a', tr))
\end{aligned}$$

This works with $\neg$, $\cup$, $\cap$ and sym at the topmost level. Min is used to compare values when there is a $\cap$ constructor, because the comparison should satisfy both components. Max is used to compare values when there is a $\cup$ constructor, because it has to only satisfy one of the components. In case of symmetric closure, Max is used as well because it is sufficient that one of the component be satisfied.

When none of the above equations can be made to work, the expression is expanded in a comparison. The way it is expanded depends on the value at the extremity of the path, i.e., if the correspondence is between properties or relations.

In case of relations, i.e., if t and t' are classes, then the extremities of the paths are, in turn, compared as instances:

$$C_{s,t,s',t'}(a, a', tr) = R_{tr(s/P(a)),s'/P(a')}(t, t', \mathbb{A})$$

In case of properties, if t and t' are datatypes, then the paths are directly compared:

$$C_{s,t,s',t'}(a, a', tr) = M_{t,t'}(tr(s/P(a)), s'/P(a'))$$

These operations require:

– reducing each member of the comparison to a path (P);
– replacing the comparison by a concrete operator ($M_{t,t}$);
– using the transformation (tr) if necessary.

The transformation operation tr, originating from the correspondence, is here only applied to the first element. It is possible, in EDOAL, to define transformations of the second element as well. In such a case, it should be applied to the second element.

Some transformations, such as `concat` are available in both EDOAL[2] and Silk, some other have to be converted, such as `&fn;safe-divide` which is converted into `divide`, some other are external such as the EDOAL web service calls, e.g.,"`http://www.google.com/finance/converter`".

The comparison operator ($M_{t,t'}$) is chosen depending on the datatype of the values to be compared:

$$\begin{array}{lll}
M_{t,t'} & = & \text{“}levenshteinDistance\text{”}|\text{“}levenshtein\text{”}|\text{“}jaro\text{”}|\text{“}jaroWinkler\text{”}|\text{“}equality\text{”}|\text{“}inequality\text{”} \\
& \text{if} & t = t' = string \\
& = & \text{“}jaccard\text{”}|\text{“}dice\text{”}|\text{“}softjaccar\text{”} \\
& \text{if} & t = t' = token|string, where\{s\ a\ v.\ s'\ a'\ v'.\} \\
& = & \text{“}num\text{”}(float\ minValue, float\ maxValue) \\
& \text{if} & t = t' = integer|float|other\ number\ types, \\
& & where\ minValue = min(v \cup v'), maxValue = max(v \cup v'), \{s\ a\ v.\ s'\ a'\ v'.\} \\
& = & \text{“}date\text{”} \\
& \text{if} & t = t' = date \\
& = & \text{“}dateTime\text{”} \\
& \text{if} & t = t' = dateTime \\
& = & \text{“}wgs84\text{”}(string\ unit, string\ curveStyle) \\
& \text{if} & t = t' = coordinate, unit = \text{“}m\text{”}|\text{“}km\text{”}
\end{array}$$

It may even be possible to attach specific comparisons to classes by passing the recursion.

4.3.4 Navigation generation

Silk offers XPath-inspired navigation operators as presented in Table 4.1. These are used to access the property values in instances.

[2] http://alignapi.gforge.inria.fr/edoal.html

Operator	Syntax	Description
/	$< path > / < prop >$	Leads the query passing from the entity variable as the subject property to its object value
\	$< path > \setminus < prop >$	Leads the query passing from the entity variable as the object to its subject entity
[ ]	$< path > [< prop >< comp >< val >]$ $< path > [@lang =< val >]$	Acts as a filter which only pass the values which satisfy the expression defined inside. comp can be one of $>, <, >=, <=, =, != $

Table 4.1: Silk navigation operators: forward navigation, backward navigation and condition.

The function P generates a path corresponding to a property or relation definition from a node s.

$$\begin{aligned} P(u) &= /u \\ P(u^{-1}) &= \setminus u \\ P(R \circ A) &= P(R)\ P(A) \\ P(P \dot{\cap} X) &= P(P)[Q(X)] \\ P(reflex(R)) &= s/(P_s(R) - s/|\setminus) \\ P(\bar{R}) &= P(R)* \end{aligned}$$

Silk cannot express the restrictions of $dom(C)$, $type(t)$ and $val(cp, v)$ (except numeric operator and language tag operator), so the XPath syntax is used here to express them.

$$\begin{aligned} Q(dom(C)) &= parent :: C && (NoSilk) \\ Q(type(t)) &= child :: t && (NoSilk) \\ Q(coDom(C)) &= child :: C && (NoSilk) \\ Q(val(cp, v)) &= child :: node()\ cp\ v \end{aligned}$$

4.4 Generation of Silk scripts

After setting the source and target data sets and the linkage rules, there are a few other information to be provided for obtaining a complete SILK script.

- All used prefixes should be specified, especially the prefixes of two data sets. This can be directly obtained from the prefixes in EDOAL.
- The data set source and target endpoints could be a local file or a SPARQL endpoint. They are set from the information inputed by the user (see Figure 2.2).
- The output of the link set could also be a local file or a SPARQL endpoint. In Datalift, it is set to "http://localhost:8080/openrdf-sesame/repositories/lifted/statements".
- A parameter named "limit" may restrain how many links are going to be returned after the comparison. By default, if it is not set, so all links are returned[3].

A SILK script can thus be created formally as in Algorithm 1.

[3]https://www.assembla.com/spaces/silk/wiki/Link_Specification_Language

Input: $\mathbb{A}$
Output: Silk script
for *all* $< e, e', r > \in \mathbb{A}$ **do**
  if $\exists < a, a', r > \in \mathbb{A}$,
  such that {*?s a ?o. ?s rdf:type e.*} *and* {*?s a' ?o. ?s rdf:type e'.*} **then**
    generate Interlink section with: SourceDataset($T_{?s}(e)$);
    TargetDataset($T_{?s'}(e')$);
    LinkageRule($R_{?s,?s'}(e, e', \mathbb{A})$);
  end
end
return the LinkageRule;

Algorithm 1: The Silk script building algorithm.

So, the final SILK script for the example is:

```
<?xml version="1.0" encoding="utf-8" ?>
<Silk>
    <Prefixes>
        <Prefix id="rdf" namespace="http://www.w3.org/1999/02/22-rdf-syntax-ns#" />
        <Prefix id="rdfs" namespace="http://www.w3.org/2000/01/rdf-schema#" />
        <Prefix id="xsd" namespace="http://www.w3.org/2001/XMLSchema#" />
        <Prefix id="dc" namespace="http://purl.org/dc/elements/1.1/" />
        <Prefix id="cc" namespace="http://creativecommons.org/ns#" />
        <Prefix id="owl" namespace="http://www.w3.org/2002/07/owl#"/>
        <Prefix id="dcterms" namespace="http://purl.org/dc/terms/" />
        <Prefix id="xmlns" namespace="http://ec.europa.eu/eurostat/ramon/ontologies/geographic.rdf#" />
        <Prefix id="insee" namespace="http://rdf.insee.fr/geo/" />
        <Prefix id="eurostat" namespace="http://ec.europa.eu/eurostat/ramon/ontologies/geographic.rdf#" />
    </Prefixes>

    <DataSources>
        <DataSource id="insee" type="sparqlEndpoint">
            <Param name="endpointURI" value="http://localhost:8080/datalift/sparql" />
        </DataSource>
        <DataSource id="eurostat" type="sparqlEndpoint">
            <Param name="endpointURI" value="http://localhost:8080/datalift/sparql" />
        </DataSource>
    </DataSources>

    <Interlinks>
        <Interlink id="region">
            <LinkType>owl:sameAs</LinkType>

            <SourceDataset dataSource="insee" var="a">
                <RestrictTo>
                    ?a rdf:type insee:Region .
                </RestrictTo>
            </SourceDataset>
            <TargetDataset dataSource="eurostat" var="b">
                <RestrictTo>
                     ?b rdf:type eurostat:NUTSRegion .
                      ?b eurostat:level 2^^xsd:int .
                      ?b eurostat:hasParentRegion eurostat:FR1 .
                </RestrictTo>
            </TargetDataset>
```

```
            <LinkageRule>
                <Aggregate type="max">
                    <Compare metric="levenshteinDistance" threshold="0">
                        <TransformInput function="lowerCase">
                            <Input path="?a/insee:nom[@lang='fr']" />
                        </TransformInput>
                        <TransformInput function="lowerCase">
                            <Input path="?b/eurostat:name" />
                        </TransformInput>
                    </Compare>
                </Aggregate>
            </LinkageRule>

           <Filter />

            <Outputs>
                <Output type="sparul" >
                    <Param name="uri" value=
                    "http://localhost:8080/openrdf-sesame/repositories/lifted/statements"/>
                    <Param name="parameter" value="update"/>
                </Output>
            </Outputs>
        </Interlink>
    </Interlinks>
</Silk>
```

4.5 Implementation

These translations have been implemented as an alignment renderer following the `AlignmentVisitor` class of the Alignment API. The API provides the notion of a visitor of the alignment cells. These visitors are used in the implementation for rendering the alignments by traversing inductively EDOAL expressions.

The implementation follows the presentation of this chapter: a visitor is able to generate graph patterns which may be used in a variety of contexts, such as generating SPARQL queries, and furthers visitors generate specific uses of the graph patterns. We have implemented the following visitors:

- The *GraphPatternRendererVisitor* class traverses EDOAL expressions and translate each EDOAL correspondence into graph patterns. It implements the T functions (see §4.1).
- The *EDOALRendererVisitor* class finds out the class' properties and relations which have correspondences.
- The *SILKRendererVisitor* class extends the *EDOALRendererVisitor* class and generates SILK script for linking the data.

4.6 Limitation of the proposed approach

The proposed approach has been implemented. However, the implemented code departs largely from the presentation of Sections 4.1-4.4 above. In particular:

- The presentation above is not complete as it does not deal correctly with the inductive definition of EDOAL property and relation expressions;
- The presentation does not mention techniques implemented for avoiding looping in the case, very frequent, of two classes depending on each others;

- Instances are not directly compared by values, first instance equality is checked;
- It is not complete since it does not take into account multiple values for a property, or not very well;
- The implementation generates scripts of a very large size.

4.7 Conclusion

In this chapter, we presented the translation of EDOAL alignments into Silk scripts. This translation generates both blocking selectors (Dataset descriptions) and linkage rules. Dataset descriptions may be translated successfully from the class expressions, property expressions and relation expressions (constructed by URI, operators, or restrictions). Linkage rules are obtained by combining correspondences involving properties of the concerned classes within the same Interlink clause.

5. Conclusion

The transformation from EDOAL alignment file to Silk script has been theoretical defined and implemented. It is integrated in the Datalift platform since the February 2013 version. This allows for generating links.

In the next period, we plan to implement two independent aspect in the data interlinking module:

Retrieving EDOAL alignments from an alignment server One of the development perspectives is the extension of the interface provided in the beginning so that users can choose alignments from the alignment server and pass them to the current program to generate Silk scripts from them. This will require supporting EDOAL Alignment within the Alignment server. This is not yet available in the Alignment API 4.5. The Alignment server will remain independent from the platform – as opposed to embedded – in order to avoid cluttering the platform. Instead, having a public server offering alignments between the LOD cloud ontologies is retained.

Learning better EDOAL alignment descriptions from user feedback This will require for users to be able to provide feedback on the generated links. From this feedback, we plan to implement a learning method able to pinpoint which parts of the EDOAL alignment are prone to have generated mistakes in links. This will allow us to modify the alignment, to generate a new Silk script and to generate new links.

REFERENCES

[1] Manuel Atencia, Jérôme David, and François Scharffe. Keys and pseudo-keys detection for web datasets cleansing and interlinking. In *Proc. 18th international conference on knowledge engineering and knowledge management (EKAW), Galway (IE)*, pages 144–153, 2012.

[2] Jérôme David, Jérôme Euzenat, François Scharffe, and Cássia Trojahn dos Santos. The Alignment API 4.0. *Semantic web journal*, 2(1):3–10, 2011.

[3] Jérôme Euzenat, Nathalie Abadie, Bénédicte Bucher, Zhengjie Fan, Houda Khrouf, Michael Luger, François Scharffe, and Raphaël Troncy. Dataset interlinking module. Deliverable D4.2.1, Datalift, 2011.

[4] Jérôme Euzenat, François Scharffe, and Axel Polleres. Processing ontology alignments with sparql (position paper). In *Proc. IEEE international workshop on Ontology alignment and visualization (OAaV), Barcelona (ES)*, pages 913–917, 2008.

[5] Anja Jentzsch. Link specification language, 2012.

[6] Holger Knublauch, James Hendler, and Kingsley Idehen. SPIN: Overview and motivation. Member submission, W3C, 2011.

[7] Carlos Rivero, Inma Hernández, David Ruiz, and Rafael Corchuelo. Generating SPARQL executable mappings to integrate ontologies. In *Proc. 30th International Conference on Conceptual Modeling (ER)*, volume 6998 of *Lecture Notes in Computer Science*, pages 118–131, Brussels (BE), 2011.

[8] François Scharffe, Jérôme David, and Manuel Atencia. Keys and pseudo-keys detection for web datasets cleansing and interlinking. Deliverable D4.1.2, Datalift, 2012.

A. Creating a new SILK Plugin : JTSGeographicMetric

We summarize how to introduce new plug-in metrics in Silk:

1. First, create a new Silk plugin (described bellow) called `JTSGeographicMetric` that uses the `JTSPointDistance` function of the library.

```
package de.fuberlin.wiwiss.silk.plugins.metric
...
import metrics.JTSPointsDistance
...

/**
 * This metric takes geographical coordinates of two points,
 * given in degrees of longitude and latitude, and creates a score measuring
 * The default metric is the distance of the two points in meters but its behaviour is configurable
 * via the following parameters:
 * unit = "meter" or "m" , "kilometer" or "km" (default) - the unit in which the distance is measure
 * @author Faycal Hamdi (IGN France)
 */

@Plugin(
        id = "jtswgs84",
        label = "JTS Geographical distance",
        description = "Computes the geographical distance between two points. Author: Faycal Hamdi (IGN France)")

case class JTSGeographicMetric(unit: String = "km") extends SimpleDistanceMeasure {
    require(Set("m", "meter", "km", "kilometer").contains(unit), "Invalid unit: '" + unit + "'. Allowed units: \"m\", \"meter\", \"km\", \"kilometer\"")
    ...
    private val multipliers = Map("km" -> 0.001, "kilometer" -> 0.001, "meter" -> 1.0, "m" -> 1.0)
    private val unitMultiplier: Double = multipliers.get(unit).getOrElse(1)

    override def evaluate(str1: String, str2: String, limit: Double): Double = {
        //Parse the coordinates and return a similarity value if both coordinates could be extracted.
        (getCoordinates(str1), getCoordinates(str2)) match {
        case (Some(loc1), Some(loc2)) => getGeometricJTSDistance(loc1, loc2)
        case _ => Double.PositiveInfinity
        }
    }


    /**
     * Computes the distance between two geo points using JTSPointDistance.
     */
    private def getGeometricJTSDistance(loc1: GeoPoint, loc2: GeoPoint): Double = {
        val jts = new JTSPointsDistance(loc1.lat, loc1.long, loc2.lat, loc2.long)
        var dist = jts.getPointsDistance()
        dist = dist * unitMultiplier
        dist
    }

}
```

2. Register the new plugin:

```
package de.fuberlin.wiwiss.silk.plugins
...
/**
 * Registers all default plugins as well as external plugins found in the ↩
     provided directory.
 */

object Plugins {
  /**
   * Registers all default plugins.
   * For performance reasons, this is done manually instead of using automatic↩
       classpath lookup.
   */
  private def registerDefaultPlugins() {
    ...
    DistanceMeasure.register(classOf[JTSGeographicMetric])
    ...
  }

  /**
   * Registers external plugins.
   */
  }
}
```

B. How to run the interconnection module

In order to run the Datalift interconnection module, follow the steps:

1. create a project;
2. go to "Sources" to add two RDF data sets that you are going to interlink, e.g., INSEE data set "regions-2010.rdf" and Eurostat data set "nuts2008_complete.rdf" in the example folder of the interconnection module;
3. go back to "Description", press button "Import of RDF source(file or SPARQL)" to load the RDF data sets that you've added at the previous step;
4. go back to "Description", press button "Data publishing to public RDF store" to publish the RDF data sets that you've loaded at the previous step;
5. go back to "Description", press button "Interconnection", then you have three choices to generate links:
   (a) upload the SILK script file, e.g., "script.xml" for INSEE and Eurostat data sets in the example folder of the interconnection module, and press button "Run" (if you are using the wrapper version of Datalift, please load the file "script_forWrapper.xml"), or
   (b) create a SILK script and press "Run", or
   (c) upload an ontology alignment file written in EDOAL, e.g., "insee_nuts.xml" for INSEE and Eurostat ontologies in the example folder. In this case, you should load, import and publish the ontologies of the two data sets before uploading the EDOAL file, e.g., the files of INSEE and Eurostat data set ontologies "onto1_file.rdf" and "onto2_file.rdf" in the example folder. After that, specify the source and target data set addresses, in our example, it is Datalift SPARQL endpoint, which is "http://localhost:8080/datalift/sparql" (if you are using wrapper version of datalift, please type "http://localhost:9091/datalift/sparql" instead). Finally, press button "Run"
6. if a page showing "OK Data linking completes! " comes, go to the SPARQL endpoint of Datalift and press the button "Links", then you can see the produced links.