A formal approach for RDFizing semi-structured data

Noorani Bakerally, Nathalie Jane Hernandez, Sébastien Bolle, Christelle Ecrepont, Pauline Folz, Thierry Monteil, Fano Ramparany

A formal approach for RDFizing semi-structured data

Noorani Bakerally[1,2], Nathalie Hernandez[1], Sébastien Bolle[3], Christelle Ecrepont[2], Pauline Folz[3], Thierry Monteil[2], and Fano Ramparany[3]

[1] IRIT, Toulouse, France {firstname.lastname}@irit.fr
[2] LAAS-CNRS, National Institut of applied Sciences of Toulouse, Toulouse 31400, {firstname.lastname}@laas.fr
[3] Orange Labs, Meylan, France {firstname.lastname}@orange.com

Abstract. Several approaches to facilitate the publication of data in compliance with RDF exists. However, the majority of them depends on mapping languages that have a steep learning curve and require knowledge of the languages' syntax and semantics in addition to the Semantic Web stack. While the remaining approaches are easier to use, they are not properly described, thus making it difficult to reproduce them. To tackle the latter issue, in this technical report, we thouroughly describe a semi-automatic approach to facilitate the generation of RDF data from semi-structured data. The strength of our approach lies in the ability to automatically generate customizable mappings from several ontologies without the need to have any prior knowledge of the ontologies.

Keywords: RDF· Data transformation · Semi-automatic approach

1 Introduction

To enhance data interoperability on the Web, the adoption of RDF is a necessary condition. Yet, the lack of adoption of RDF is a known fact. Reusing existing vocabularies or defining new ones is a major step in the transition to RDF.

Two main categories of approaches have been developed to facilitate semi-structured data to RDF. *Mapping Languages* (R2RML [1], RML [2], XSPARQL [3], SPARQL-Generate [4] etc.) are full-fledged languages for defining the transformation from a heterogeneous dataset to the RDF model. The main disadvantage of the latter approaches is their steep learning curve. To further facilitate the usage of mapping languages, tools like *RML editor* [5] and *Juma* [6] are available. *RML editor* and *Juma* are graphical languages on top of the mapping languages R2RML and RML respectively. However, they only relieve the user from having to master the syntax of the mapping languages. In situations requiring complex modelings, one may even argue that these tools steepen the learning curve. In short, the transformation of semi-structured data to RDF is still a hard nut to crack.

In this paper, we provide an approach to further facilitate the RDFization of data in semi-structured formats (e.g. CSV, JSON, etc.). The strength of our approach is that users do not need to have knowledge of ontologies prior to RDFizing the data. Users only need to describe their data with keywords that is then used by our approach to

generate holistic customizable mappings that describes objects' characteristics while exploiting the semantics of available ontologies. We formally describe this approach using an illustrating scenario.

The rest of this paper is organized as follows. We formally describe our approach in Section 2. In Section 3, we conclude this report.

2 Our Approach: Semi-automatic Mapping Generation

In this section, we describe our approach that generates mappings for transforming semi-structured data to RDF. We begin by first describing an illustrating scenario that we use to examplify concepts throughout this report. Then, we provide an informal overview of our approach in Section 2.2. We proceed with a formal description of our approach. In Section 2.3, we describe the type of raw data schema that we consider. We provide a model for describing such schemas in Section 2.4. In Section 2.5, we describe a model for representing ontologies we are interested in. Finally, in Section 2.6, we describe the generation of mappings for a schema with respect to a set of ontologies.

2.1 Illustrating Scenario

To explain the objective of our approach, we consider a parking dataset[4] available on Grenoble (a French city) open data portal[5]. Figure 1 is part of a preview of that dataset taken directly from the data portal. Each row corresponds to a specific parking and each column to a specific parking characteristic. Our aim is to RDFize such datasets by providing an exhaustive description of the objects described. By exhaustive description, we mean to look for suitable ontology entities to represent both the type of object (e.g. parking) and the objects' other characteristics (e.g. `lat`, `ADDRESSE`, etc.). Note that some columns may be ignored such as `_id` that contains data for internal use or `CODE` that contains the same information as `id`.

_id	CODE	LIBELLE	ADRESSE	TYPE	TOTAL	type	id	lon	lat
11	SPR_PKG…	CATANE	RUE AMP…	PKG	490	PKG	SPR_PKG…	5.70503	45.181035
9	QPA_PKG…	CHAVANT	17, BD M…	PKG	394	PKG	QPA_PKG…	5.731463	45.185612
15	PVP_PKG…	ENCLOS…	PLACE V…	PKG	130	PKG	PVP_PKG_…	5.728358	45.188401

Fig. 1. Parking data from Grenoble Open Data Portal

Figure 2 is an example of a mapping that may be used to describe the parking data. It is composed of correspondences established between column headers of the raw data and entities from vocabularies that represent the objects' characteristics. The light blue and orange arrows are data and object properties respectively. Also, the green labels refer to column headers in the parking data and above them is the ontology entity IRI to which they have been mapped. The prefixes are shown in Table 1.

[4] http://data.metropolegrenoble.fr/ckan/dataset/parkings-de-grenoble/resource/a6919f90-4c38-4ee0-a4ec-403db77f5a4b, last accessed on 7 December 2019 under the licence odc-odblhttps://opendatacommons.org/licenses/odbl/index.html

[5] http://data.metropolegrenoble.fr/

Prefix	Vocabulary Name	Namespace IRI
xsd	XML Schema Definition	http://www.w3.org/2001/XMLSchema#
rdfs	RDFS	http://www.w3.org/2000/01/rdf-schema#
sc	Schema.org	http://schema.org/
mv	MobiVoc	http://schema.mobivoc.org/
wgs84	WGS84	http://www.w3.org/2003/01/geo/wgs84_pos#
dc	Dublin Core Metadata Terms	http://purl.org/dc/terms/

Table 1. List of Vocabulary Prefixes

As we can see (*cf.* Figure 2), the mapping reuses entities from several ontologies. The mapping is composed of three types of ontology entities. Firstly, there is an entity, i.e. `mv:ParkingFacility`, to type the parking object. Secondly, there are entities that represent a parking object characteristic. For example, `ADDRESSE` is mapped to the data property `sc:address`. In addition to representing an object's characteristic using a data property, expressions involving several related ontology entities may be used. For example, `TOTAL` is mapped to an expression composed of `mv:Capacity` and `mv:maximumValue`. Thirdly, there may be additional entities to link the class representing the object type to entities representing objects' characteristics. For example, the object property `mv:capacity` is added to the mapping to link `mv:ParkingFacility` and `mv:Capacity`. Once defined, the mapping can be used to RDFize the parking data.

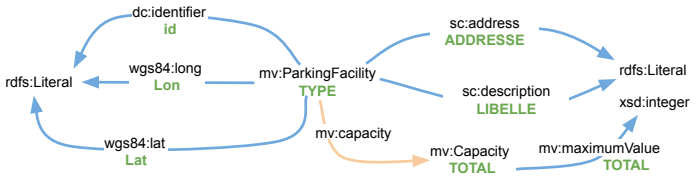

Fig. 2. An example of mapping to describe parking data

2.2 Overview

Our approach to generate mappings consists of four main steps as shown in Figure 3. As we can see, it relies on the existence of an `Ontology repository`. Suppose that this repository contains the vocabulary MobiVoc, Schema.org, WGS84 and Dublin Core Metadata Terms.

Step 1 In order to capture background knowledge provided by the user about the schema used in the raw data, firstly we generate a `Schema description` consisting of `Type description` and `Elements description`. `Type Description` characterizes the type of objects described by the schema and `Elements description` the schema elements (e.g. `lon` in Figure 1). Both consist of keywords manually entered by users. For example, with respect to the CSV file in Figure 1, the `Type Description` can be the keyword 'parking facility', and the `Elements description` of the *schema element* `lon` the keywords 'longitude'.

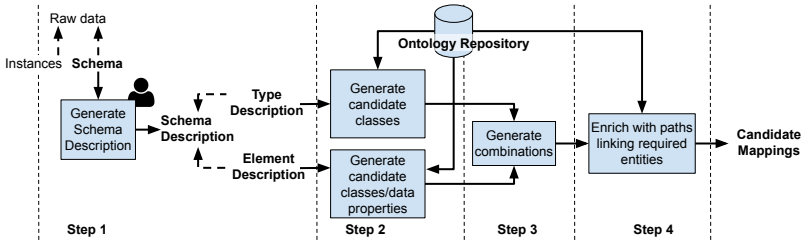

Fig. 3. Mappings Generation Process

Step 2 Secondly, using an `Ontology repository`, the `Type description` and the `Elements description`, a set of candidate classes for typing objects and a set of candidate data properties or classes for modeling schema elements are generated. Table 2 shows a schema description for the schema, i.e. column headers, of Figure 1 and some generated ontology entities for its elements.

Schema Description			Generated Entities	
		Keyword	**Classes**	**Data Properties**
Type Description		'parking facility'	`mv:ParkingFacility`, `sc:Park`	
Elements Description	`id`	'identifier'		`dc:identifier`
	`LIBELLE`	'description'		`sc:description`
	`TOTAL`	'capacity','total'	`mv:Capacity`	`sc:totalTime`
	`lat`	'latitude'		`wgs84:lat`
	`lon`	'longitude'		`wgs84:long`
	`ADDRESSE`	'address'		`sc:address`

Table 2. Example of candidate entities for typing and schema elements

Step 3 Thirdly, possible combinations are generated. A combination consists of a candidate class for typing, that we refer as the *type class*, and a candidate data property or class for each schema element generated in the previous step. Table 3 shows all combinations generated from the candidate entities in Table 2.

	Type Class	**id**	**LIBELLE**	**TOTAL**	**lat**	**lon**	**ADDRESSE**
1.	`mv:ParkingFacility`	`dc:identifier`	`sc:description`	`mv:Capacity`	`wgs84:lat`	`wgs84:lon`	`sc:address`
2.	`mv:ParkingFacility`	`dc:identifier`	`sc:description`	`sc:totalTime`	`wgs84:lat`	`wgs84:lon`	`sc:address`
3.	`sc:Park`	`dc:identifier`	`sc:description`	`mv:Capacity`	`wgs84:lat`	`wgs84:lon`	`sc:address`
4.	`sc:Park`	`dc:identifier`	`sc:description`	`sc:totalTime`	`wgs84:lat`	`wgs84:lon`	`sc:address`

Table 3. Combinations of generated entities for type class and schema elements

Step 4 Finally, we enrich each combination with the required ontology entities that are needed to describe objects with the type class and candidate entities for schema elements.

For example, Figure 4 shows the first combination from Table 3 and the needed paths, illustrated as dotted lines, that will be generated at this step. Figure 2 is a

possible result when the needed paths are generated. A path between a type class and a data property is generated if the domain of the data property includes the type class. On the other hand, a path between a type class and another class is generated by looking for ontology entities that can connect them while respecting entities' semantics.

Note that it is possible that no path is established. For example, in the fourth combination from Table 3, there is no path between the type class `sc:Park` and the candidate class of `TOTAL` that is `sc:totalTime` as the domain of the data property `sc:totalTime` does not include `sc:Park`. In this case, the candidate entity is removed from the mapping and the mapping generated will not have any correspondence for `TOTAL`.

The set of generated mapping are then presented to the user. The user can analyse them and customize one (by editing or removing correspondences). The selected mapping is used to describe objects from the raw data. From here, the possible result may be expressed in existing mapping languages to generate final RDF.

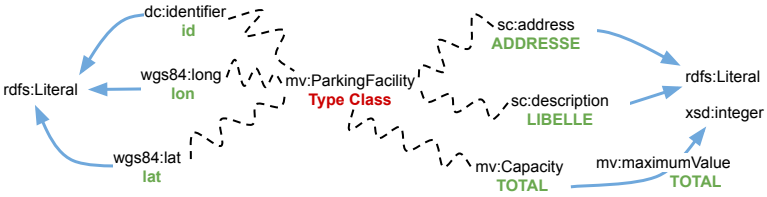

Fig. 4. Paths needed to be generated for a particular combination

2.3 Raw Data

We refer to *raw data* as consisting of both the *schema* and the data given as input to our approach. Such schemas are formally describe in Definition 1.

Definition 1 (*Schema*).
A schema σ is a set of schema elements *that describes only one type of object. We refer to this type as the* schema type*. Every element of σ that we refer to as a* schema element*, is a label for an atomic characteristic of the object.* ***By atomic characteristic, we mean that its values can be typed using an instance of*** `rdfs:datatype` *[7, §2.4] such as* `xsd:integer`*,* `xsd:float` *and* `xsd:string`*, etc.*

The *schema* for the parking data in Figure 1 is{`_id`, `CODE`, `LIBELLE`, `ADDRESSE`, `TYPE`, `TOTAL`, `type`, `id`, `lon`, `lat`}. The type of object described by this schema, that we refer to as *schema type*, is parking facilities. It is made up of several *schema elements* such as `TOTAL`.While a *schema* is independent of data formats, raw data based on such a schema can be naturally serialized in the CSV format. Nevertheless, JSON data containing only primitive data types can also be represented using this model.

2.4 Schema Description

Schemas are made up of only atomic characteristic' labels. Consequently, they may be lightweight and lack background information. To describe them, we provide a *schema*

description model. In short, this model can be used to provide a description about both the *schema type* and *schema elements*. We formally describe the model in Definition 2.

Preliminaries We write **IRI** and **L** as the disjoint sets of *IRIs* and *literals*.

Definition 2 (*schema description*).
For a schema σ, a schema description is a pair $\langle \mathfrak{t}, \mathcal{E} \rangle$ where $\mathfrak{t} \in 2^{\mathbf{L}}$ is set of keywords to provide some description about σ's schema type. $\mathcal{E} \subseteq \mathbf{L} \times 2^{\mathbf{L}}$ is a set of pairs that describe σ's schema elements. $\langle e, K \rangle \in \mathcal{E}$ means that the schema element $e \in \sigma$ is described by the set of keywords K. There cannot be more than one pair in $\mathcal{E}$ describing the same schema element. More formally, $\langle e, K_1 \rangle \in \mathcal{E} \wedge \langle e, K_2 \rangle \in \mathcal{E} \Longrightarrow K_1 = K_2$. Also, we refer to $\mathfrak{t}$ as the schema type description *and an element of $\mathcal{E}$ as a* schema element description*. We write $\mathbb{D}$ as the set of schema descriptions.*

A *schema description* for the parking schema in Figure 1 may be ({'parking facilities'},{(`LIBELLE`,{'label'}), (`ADDRESSE`,{'address'}), (`TOTAL`,{'capacity', 'total'}), (`id`, {'identifier'}), (`lon`,{'longitude'}), (`lat`,{'latitude'})}). Note that the ignored columns (e.g. `_id`) are ommited from the *schema description*.

2.5 Vocabularies

Our approach generates initial mappings between a *schema description* and RDFS/OWL ontology entities. In this section, we define structures related to ontologies that we use. To abstract ontologies from languages and formalisms, we define a *Vocabulary structure* (*cf.* Definition 3). We use this structure to represent an ontology, an ontology repository and a *Description vocabulary* used to describe the raw data.

Definition 3 (*Vocabulary structure*). *A Vocabulary structure v is a tuple $\langle i, \mathcal{C}, \mathcal{P}, \mathcal{D}, \delta, r, s, l \rangle$ where:*
- *i is the IRI of the vocabulary;*
- *$\mathcal{C}$, $\mathcal{P}$, $\mathcal{D}$ are the disjoint set of IRIs for classes, object properties and data properties respectively;*
- *$\delta : \mathcal{P} \cup \mathcal{D} \mapsto 2^{\mathcal{C}}$ maps an object or data property to classes in its domain;*
- *$r : \mathcal{P} \cup \mathcal{D} \mapsto 2^{\mathcal{C} \cup \{\texttt{rdfs:Literal}\}}$ maps object properties only to classes in their range (i.e $\forall p \in \mathcal{P}, r(p) \in 2^{\mathcal{C}}$) and maps data properties only to `rdfs:Literal` (i.e $\forall d \in \mathcal{D}, r(d) = \{\texttt{rdfs:Literal}\}$).*
- *$s : \mathcal{C} \mapsto 2^{\mathcal{C}}$ maps a class to its direct and indirect (inferred) subclasses;*
- *$l : \mathcal{C} \cup \mathcal{P} \cup \mathcal{D} \mapsto 2^{\mathbf{L}}$ maps a class, object property or data property to a set of literals defined by annotation properties such as `rdfs:label`, `dc:label`, etc.*

$\mathbb{V}$ denotes the set of Vocabulary structures. Abusively, given a Vocabulary structure v $\mathcal{C}(v)$,$\mathcal{D}(v)$and $\mathcal{P}(v)$ refer to its set of classes, object properties and data properties.

Modeling RDFS/OWL ontologies using a Vocabulary structure: To obtain a *Vocabulary structure* for an RDFS/OWL ontology, first we use an OWL reasoner considering all OWL constructs to obtain the inferred ontology. Then, from the inferred ontology, we extract information using a subset of the RDFS and OWL constructors. Regarding classes, we consider atomic classes, their subclasses and equivalent classes, and complex classes that are disjunction of atomic classes. For domains and ranges of object properties, and for domains of data properties, we consider only the latter type of classes. Apart from this, we ignore all other constructors (e.g. subproperties, property characteristics, restrictions, etc.).

Modeling an ontology repository using a Vocabulary structure: We also use a *Vocabulary structure* to model an ontology repository as we consider an ontology repository as a merge of *Vocabulary structure* s representing the ontologies.

Modeling paths in a Vocabulary structure: Like paths in RDF graphs, in our approach, we need to model paths between a class and a data property in a *Vocabulary structure*. We refer to these paths as *Data property path* and formally define them in Definition 4.

Definition 4 (*Data property path*). *With respect to a Vocabulary structure* $v = \langle i,\mathcal{C},\mathcal{P},\mathcal{D},\delta,r,s,l\rangle$*, a Data property path* $\mathfrak{p}$ *is a sequence of connected triples where the first triple contains the class* $i_c \in \mathcal{C}$ *and the last triple contains the data property* $i_d \in \mathcal{D}$*. More formally,* $\mathfrak{p} = \langle \mathfrak{t}_0,\mathfrak{t}_1,...,\mathfrak{t}_{n-1},\mathfrak{t}_n\rangle \in \bigcup_{n\geq 1}(\mathcal{C}\times\mathcal{P}\times\mathcal{C})^{n-1}\times(\mathcal{C}\times\mathcal{D}\times\{\texttt{rdfs:Literal}\})$ *is a Data property path if:*

- *the subject or the object of the first triple should be* i_c*, i.e.* $\mathfrak{t}_0(0) = i_c \vee \mathfrak{t}_0(2) = i_c$*;*
- *the predicate in the last triple is* i_d*, i.e.* $\mathfrak{t}_n(1) = i_d$*;*
- *in all triples, the subject and object are in the domain and range of the predicate respectively, i.e.* $\forall \mathfrak{t} \in \{\mathfrak{t}_0,\mathfrak{t}_1,...,\mathfrak{t}_n\}, \big(\mathfrak{t}(0) \in \delta(\mathfrak{t}(1)) \wedge \mathfrak{t}(2) \in r(\mathfrak{t}(1))\big)$
- *the triples in the path are connected, i.e.* $(i_c, \texttt{rdfs:Literal}) \in R^+ = t^+(s^+(R))$ *where* t^+ *and* s^+ *returns the transitive and symmetric closure of a relation and* $R = \big\{(c_1,c_2) \,|\, p \in \mathcal{P} \wedge c_1 \in \delta(p) \wedge c_2 \in r(p)\big\}$ *(connectivity as in* weakly directed graphs*).*

The interpretation of subject, predicate and object are the same as in an RDF triple [8, §3.1]. Abusively, we write $\mathbb{V}(\mathfrak{p})$ *to denote the Vocabulary structure containing the ontology entities used in the Data property path* $\mathfrak{p}$*. Also, for a given Data property path* $\mathfrak{p}$*, we use the notation* $\mathfrak{p}^c$ *and* $\mathfrak{p}^d$ *to refer to* i_c *and* i_d *respectively. We write* $\mathbb{P}$*as the set of Data property path.*

In Figure 2, a *Data property path* can be used to model the path between `mv:ParkingFacility` and `mv:maximumValue`. Entities on this path are found in the vocabulary structure modeling the Schema.org vocabulary. Suppose this *Data property path* is $\mathfrak{p}_1$. $\mathfrak{p}_1$ is defined with respect to the class `mv:ParkingFacility`, the data property `mv:maximumValue` and the *Vocabulary structure* v_{mv}, and is:

$$\big((\texttt{mv:ParkingFacility},\texttt{mv:capacity},\texttt{mv:Capacity}),\ (\texttt{mv:Capacity},\texttt{mv:maximumValue},\texttt{rdfs:Literal})\big) \tag{1}$$

The first and second condition from the definition are satisfied as $\mathfrak{p}_1 = \langle \mathfrak{t}_0,\mathfrak{t}_1\rangle$ starts with the required class (i.e. $\mathfrak{t}_0(0) = $ `mv:ParkingFacility`) and its last triple contains the required data property (i.e. $\mathfrak{t}_1(1) = $ `mv:maximumValue`). Also, in both triples in $\mathfrak{p}_1$, the object and data property has their domain and ranges correctly set. For the last condition, from $\mathfrak{p}_1$, we can derive:
$R = \big\{(\texttt{mv:ParkingFacility},\texttt{mv:Capacity}),(\texttt{mv:Capacity},\texttt{rdfs:Literal})\big\}$. By performing the symmetric closure followed by the transitive closure on R to obtain R^+, we can find (`mv:ParkingFacility`,`rdfs:Literal`) is R^+ meaning the sequence of triples is well connected.

2.6 Mapping Generation System

A mapping generation system contains the required components to generate mappings for schema descriptions. Abusive notations are used: given a *Vocabulary structure* v, we write $\mathcal{C}(v)$,$\mathcal{D}(v)$and $\mathcal{P}(v)$ as its set of classes, object properties and data properties respectively.

Definition 5 (*Mapping generation system*). *A Mapping generation system mp is a tuple* $\langle v,f_s,f_c,f_p,f_d\rangle$ *where:*
- $v=\langle i,\mathcal{C},\mathcal{P},\mathcal{D},\delta,r,s,l\rangle$ *is the* ***Ontology Repository****;*
- $f_s:2^{\mathbf{L}}\times\mathbf{IRI}\mapsto\mathbb{R}$ *is the* ***Similarity Calculator****;*
- $f_c:2^{\mathbf{L}}\mapsto 2^{\mathbf{IRI}\times\mathbb{R}}$ *is the* ***Concept Mapper****;*
- $f_p:\mathbf{IRI}\times\mathbf{IRI}\mapsto 2^{\mathbb{P}}$ *is the* ***Path Generator****;*
- $f_d:\mathbb{D}\mapsto 2^{\mathbf{IRI}\times\mathbb{R}\times 2^{\mathbf{L}\times\mathbf{IRI}\times\mathbb{R}\times 2^{\mathbb{P}}}}$ *is the* ***Schema Description Processor****.*

In a *mapping generation system*, the ontology repository v is in the global state and all other components of a mapping generation system have access to it.

Ontology Repository v is a *Vocabulary structure* representing an ontology repository. It contains the required information from one or more ontologies that we want to use to describe our schemas;

Similarity Calculator $f_s:2^{\mathbf{L}}\times\mathbf{IRI}\mapsto\mathbb{R}$ performs a similarity calculation between a set of keywords and an ontology entity identified by its IRI, and maps them to a real number, that we refer as the *confidence*, in the range [0,1]. We do not further define f_s and leave it as an implementation aspect. We only places a constraint that if the ontology entity is not in the ontology repository, the confidence is 0. More formally,

$$\Big(k\in 2^{\mathbf{L}}\Big)\wedge\Big(i\in\mathbf{IRI}\Big)\wedge\Big(i\notin\mathcal{C}(v)\cup\mathcal{D}(v)\cup\mathcal{P}(v)\Big)\Longrightarrow f_s(k,i)=0$$

Concept Mapper $f_c:2^{\mathbf{L}}\mapsto 2^{\mathbf{IRI}\times\mathbb{R}}$ takes a description of a schema element through a set of keywords and identifies ontology entities that best describe the schema element using their calculated confidences (using f_s). The only ontology entities considered are those that allow specifying a value for a property. They include data properties and classes that are in the domain of a data property. f_c maps a set of keywords to a set of classes or data properties from the ontology repository and their respective confidences are calculated using f_s. More formally, with respect to the ontology repository $v=\langle i,\mathcal{C},\mathcal{P},\mathcal{D},\delta,r,s,l\rangle$ and the set of keywords $k\in 2^{\mathbf{L}}$,

$$f_c(k)=\Big\{\langle i,c\rangle\,|\,i\in\Big(\mathcal{D}(v)\cup\bigcup_{d\in\mathcal{D}(v)}\delta(d)\Big)\wedge c=f_s(k,i)\Big\}$$

Suppose $mp1=\langle v_r,f'_s,f'_c,f'_p,f'_d\rangle$ is *Mapping generation system* where v_r is the ontology Structure representing the schema.org vocabulary and $k1=\{$'capacity','total','places'$\}$. $f'_s(k1)$ may include (`mv:Capacity`,0.9) and (`sc:totalTime`,0.3). As mentioned before, the result should consist of ontology entities that are either data properties or classes in the domain of a data property. `sc:totalTime` is a data property and `mv:Capacity` is in the domain of `mv:capacity`

Path Generator $f_p:\mathbf{IRI}\times\mathbf{IRI}\mapsto 2^{\mathbb{P}}$ takes as input a class i_c and data property i_d from the ontology repository and generate a set of *Data property path* from v_r. We do not further define f_p and leave it as an implementation aspect. We only impose the following constraints on f_p with respect to the ontology repository $v=\langle i,\mathcal{C},\mathcal{P},\mathcal{D},\delta,r,s,l\rangle$:
- f_p maps a pair of ontology entities to $\varnothing$ if at least one of them is not in the ontology repository. More formally, $i_c\notin\mathcal{C}\vee i_d\notin\mathcal{D}\Longrightarrow f_p(i_c,i_d)=\varnothing$
- all *Data property path* (Definition 4) generated are defined with respect to i_c and i_d. More formally, $\forall\epsilon\in f_p(i_c,i_d),\epsilon^c=i_c\wedge\epsilon^d=i_d$;

– the ontology entities in the *Data property path* must be from the ontology repository. More formally, $\forall \epsilon \in f_p(i_c,i_d), \epsilon \in \mathcal{C} \cup \mathcal{D} \cup \mathcal{P}$

The above conditions ensures that the proper set of *Data property path* is generated. The first condition specifies that f_p only maps a pair consisting of a class and data property to a set of *Data property path* if the pair's elements are in the ontology repository. The second and third conditions ensures that *Data property path*s generated are defined with respect to the input pair's elements and that the paths' entities are well from the vocabulary.

Suppose $mp_1 = \langle v_r, f'_s, f'_c, f'_p, f'_d \rangle$ is a *Mapping generation system* where the *Vocabulary structure* v_r is the the vocabulary structure modelling schema.org Section 2.5. Using mp_1, we want to generate the set of *Data property path* between the class and data property `mv:ParkingFacility` and `mv:maximumValue` respectively. Suppose that using f'_p generates a set containing only the following path:

$$\Big((\texttt{mv:ParkingFacility}, \texttt{mv:capacity}, \texttt{mv:Capacity}), \\ (\texttt{mv:Capacity}, \texttt{mv:maximumValue}, \texttt{rdfs:Literal}) \Big) \tag{2}$$

The above path is a valid *Data property path* as explained in the example of Definition 4. Yet, it cannot be used solely as a determinant to ensure that f'_p satisfies the above first condition. Nevertheless, the path satisfy the second and third conditions. The second condition is satisfied as its first and last triple contain `mv:ParkingFacility` and `mv:maximumValue` respectively. Moreover, all the entities in the path are from v_r.

Schema Description Processor $f_d : \mathbb{D} \mapsto 2^{\mathbf{IRI} \times \mathbb{R} \times 2^{\mathbf{L} \times \mathbf{IRI} \times \mathbb{R} \times 2^{\mathbb{P}}}}$ maps a schema descriptions to a set of *Final mapping*. A *Final mapping* consist of entities from the ontology repository for modeling the schema elements in a particular way. We formally define it in Definition 6.

Definition 6 (*Final mapping*). *For a schema description $d = \langle \mathsf{t}, \mathcal{E} \rangle$ and the ontology repository $v = \langle i, \mathcal{C}, \mathcal{P}, \mathcal{D}, \delta, r, s, l \rangle$, a final mapping is a pair $\langle \mu, \Sigma \rangle$ where $\mu \in \mathbf{IRI} \times \mathbb{R}$ and $\Sigma \subseteq \mathbf{L} \times \mathbf{IRI} \times \mathbb{R} \times 2^{\mathbb{P}}$. $\langle i_t, c_t \rangle \in \mu$ means that the class with IRI i_t can model the schema type whose description is given by t with a confidence of c_t. $\langle e, i_e, c_e, \rho \rangle \in \Sigma$ means that the schema element e can be modeled with the entity i_e from v with a confidence of c_e. Also, ρ is a set of Data property paths (cf. Definition 4) with respect to i_t and i_e, i.e. $\forall \varrho \in \rho, \varrho^c = i_t \wedge \varrho^d = i_e$.*

Now that we have define a *Final mapping* (*cf.* Definition 6), we can define the evaluation of a *schema description* $\partial = \langle \mathsf{t}, \mathcal{E} \rangle$ in the *Mapping generation system* $mp = \langle v, f_s, f_c, f_p, f_d \rangle$ in Equation 3.

$$\begin{aligned} f_d(\partial) = & \Big\{ (i_t, c_t) \,|\, (i_t, c_t) \in f_c(\mathsf{t}) \wedge i_t \in \mathcal{C} \Big\} \times \\ & \bigtimes_{(e,\kappa) \in \mathcal{E}}^{\bullet} \Big\{ (e, i_e, c_e, \rho) \,|\, \Big((i_e, c_e) \in f_c(\kappa) \Big) \wedge \\ & \Big((\rho = f_p(i_t, i_e) \wedge i_e \in \mathcal{D}) \vee (\exists i_e^d \in \mathcal{D} \wedge i_e \in \mathcal{C} \wedge i_e \in \delta(i_e^d) \wedge \rho = f_p(i_t, i_e^d)) \Big) \Big\} \\ & \text{where given two sets } A \text{ and } B, A \overset{\bullet}{\times} B = \Big\{ \{a, b\} \,|\, (a, b) \in A \times B \Big\} \end{aligned} \tag{3}$$

The above definition of f_d consists of two parts, *Part (a)* and *Part (b)*.

Part (a) , formalized as $\{(i_t,c_t)\,|\,(i_t,c_t)\in f_c(\mathfrak{t})\wedge i_t\in\mathcal{C}\}$, represents the generation of classes for typing using the *schema type description* (i.e. $\mathfrak{t}$).

Part (b) , formalized as $\{(e,i_e,c_e,\rho)\,|\,...\}$, consists of the right-hand side of the generalized cross-product symbol with the dot. This part represents the generation of an ontology entity (i_e) to model a schema element (e) with a certain confidence (c_e). Additionally, f_p is used to generate the set of *Data property path*s (*cf.* Definition 4) with respect to i_t and i_e. In short, applying the unordered cartesian product on all sets generated from the expression in the second part generates a set of *Final mapping* (Definition 6) (e,i_e,c_e,ρ) for each schema element e described in $\mathcal{E}$. Finally, performing a cartesian product on the set of pairs generated in the first part with the latter set generates a set of *Final mapping*.

To illustrate this, let us consider a *Mapping generation system* $mp_1=\langle v_r,f'_s,f'_c,f'_p,f'_d\rangle$ where the *Vocabulary structure* v_r is as the same as defined in Section 2.5. Also, consider a *schema description* sd=({'parking facilities'},{(`TOTAL`,{'capacity', 'total'}), (`id`, {'identifier'})}) for describing objects in the raw data in Figure 1 consider only the properties `'TOTAL'` and `'id'`.

Let us start by evaluating *Part (a)* of f'_d, that is f'_c ({'parking facilities'}), while respecting the condition $i_t\in\mathcal{C}$, implying that only pairs containing classes are kept. Suppose that the latter evaluation returns:

$$\{(\texttt{mv:ParkingFacilities},0.9),(\texttt{sc:Park},0.3)\} \tag{4}$$

Now, we proceed to evaluate the *Part (b)* of f'_d. Suppose this evaluation for the schema element `TOTAL` generates $\{(\texttt{'TOTAL'},\texttt{mv:Capacity},0.9,\rho_{total}),$ $(\texttt{'TOTAL'},\texttt{sc:totalTime},0.3,\varnothing)\}$. Similarly suppose that the evaluation for the schema element `id` generates $\{(\texttt{'id'},\texttt{sc:identifier},0.9,\rho_{id})\}$. Applying the generalized unordered cartesian product on both latter sets generates:

$$\begin{Bmatrix}\{(\texttt{'TOTAL'},\texttt{mv:Capacity},0.9,\rho_{total}),(\texttt{'id'},\texttt{sc:identifier},0.9,\rho_{id})\},\\ \{(\texttt{'TOTAL'},\texttt{mv:totalTime},0.3,\varnothing),(\texttt{'id'},\texttt{sc:identifier},0.9,\rho_{id})\}\end{Bmatrix} \tag{5}$$

Performing a cartesian product with the sets from Result 4 and Result 5 generates the set of *Final mapping*:

$$\begin{Bmatrix}\big((\texttt{mv:ParkingFacilities},0.9),\{(\texttt{'TOTAL'},\texttt{mv:Capacity},0.9,\rho_{total}),(\texttt{'id'},\texttt{sc:identifier},0.9,\rho_{id})\}\big),\\ \big((\texttt{sc:Park},0.3),\{(\texttt{'TOTAL'},\texttt{mv:totalTime},0.3,\varnothing),(\texttt{'id'},\texttt{sc:identifier},0.9,\rho_{id})\}\big),\\ \big((\texttt{mv:ParkingFacilities},0.9),\{(\texttt{'TOTAL'},\texttt{mv:Capacity},0.9,\rho_{total}),(\texttt{'id'},\texttt{sc:identifier},0.9,\rho_{id})\}\big),\\ \big((\texttt{sc:Park},0.3),\{(\texttt{'TOTAL'},\texttt{mv:totalTime},0.3,\varnothing),(\texttt{'id'},\texttt{sc:identifier},0.9,\rho_{id})\}\big)\end{Bmatrix} \tag{6}$$

The first element in Result 6 means that `mv:ParkingFacilities` can be used as the type with a confidence of 0.9. Moreover, for the *schema element* `'TOTAL'`, the entity `mv:Capacity` can model it with a confidence of 0.9. ρ_{total} contains possible *Data property path*s with respect to `mv:ParkingFacilities` and `mv:Capacity`. Similarly, for the *schema element* `'id'`, the entity `sc:identifier` can model it with a confidence of 0.9. ρ_{id} contains possible *Data property path*s with respect to `mv:ParkingFacilities` and `sc:identifier`. It is the *Data property path*s that makes a *Final mapping* customizable. More explicitly, the user can choose the appropriate *Data property path*.

3 Conclusion

In this technical report, we have presented an approach to facilitate the RDFization of semi-structured data. We have formally described this approach and have used an illustrating scenario to examplify the formal concepts.

References

1. Souripriya Das, Seema Sundara, and Richard Cyganiak. R2RML: RDB to RDF Mapping Language, W3C Recommendation 27 September 2012. Technical report.
2. A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens, and R. Van de Walle. RML: A generic language for integrated RDF mappings of heterogeneous data. In *LDOW*, 2014.
3. W. Akhtar, J. Kopecký, T. Krennwallner, and A. Polleres. XSPARQL: Traveling between the XML and RDF worlds–and avoiding the XSLT pilgrimage. In *ESWC*, 2008.
4. M. Lefrançois, A. Zimmermann, and N. Bakerally. Flexible rdf generation from rdf and heterogeneous data sources with sparql-generate. In *EKAW*. Springer, 2016.
5. Pieter Heyvaert, Anastasia Dimou, Aron-Levi Herregodts, Ruben Verborgh, Dimitri Schuurman, Erik Mannens, and Rik Van de Walle. Rmleditor: a graph-based mapping editor for linked data mappings. In *European Semantic Web Conference*, pages 709–723. Springer, 2016.
6. Ademar Crotti Junior, Christophe Debruyne, and Declan O'Sullivan. Juma: An editor that uses a block metaphor to facilitate the creation and editing of r2rml mappings. In *European Semantic Web Conference*, pages 87–92. Springer, 2017.
7. Dan Brickley and Ramanathan V. Guha. RDF Schema 1.1. W3C Recommendation, World Wide Web Consortium (W3C), February 25 2014.
8. Richard Cyganiak, David Wood, and Markus Lanthaler. RDF 1.1 Concepts and Abstract Syntax, W3C Recommendation 25 February 2014. W3C Recommendation, World Wide Web Consortium (W3C), February 25 2014.