Mapping Diverse Data to RDF in Practice

Alexandros Chortaras(✉) and Giorgos Stamou

National Technical University of Athens, Athens, Greece
{achort,gstam}@cs.ntua.gr

Abstract. Converting data from diverse data sources to custom RDF datasets often faces several practical challenges related with the need to restructure and transform the source data. Existing RDF mapping languages assume that the resulting datasets mostly preserve the structure of the original data. In this paper, we present real cases that highlight the limitations of existing languages, and describe D2RML, a transformation-oriented RDF mapping language which addresses such practical needs by incorporating a programming flavor in the mapping process.

Keywords: RDF mapping · Data integration · Service integration

1 Introduction

An RDF graph is a set of triples, each one of which consists of a subject, predicate, and object. Thus, despite the powerful semantic interpretation of RDF graphs, their machine representation is very simple: a table with a subject, predicate and object column; actually, several relationally-backed RDF stores use such tabular representations. So, when studying how mappings for diverse data formats to RDF graphs can be defined, essentially we have to define how the underlying data can be transformed to a logical tabular representation.

In this framework, much work has been done on transforming data from relational databases (summarized in [8]). Relational data are pretty simple, because they are kept in tables, each row contains a single value for each column, and each row has usually a unique key that can be used to generate unique identifiers. Moreover, SQL is a powerful language that allows the generation of complex custom view by joining tables, selecting data that meet certain conditions, and performing simple data transformations. R2RML, the W3C language for mapping relational databases to RDF [5] is a powerful language, but owes its power mostly to the inherent tabular nature of the source data and the power of SQL which provides almost all needed data manipulation and restructuring.

Beyond relational data, closer to the tabular model are CSV documents and spreadsheets [11,14]. Other formats, such as XML, differ considerably from tabular data owing to their hierarchical structure, and the mapping systems rely

D. Vrandečić et al. (Eds.): ISWC 2018, LNCS 11136, pp. 441–457, 2018.
https://doi.org/10.1007/978-3-030-00671-6_26

on XSLT transformations, XPath and XQuery (e.g. [1,4]). To resolve the polymorphy of tools and define a uniform way to perform Data-to-RDF mapping, xR2RML [13] and RML [7] extend R2RML to support other data formats.

All such approaches are practical as long as there is no need to alter the structure of the source data, and as long as the underlying source data manipulation languages provide support for data transformations. In a multi-source supporting language, like RML, this is harder to achieve given that not all sources are backed by powerful languages, such as SQL. In this paper, we propose D2RML, a generic Data-to-RDF Mapping Language, which aims at facilitating the generation of custom RDF data stores by selectively collecting and integrating data from diverse data sources and web services into high quality RDF data stores. D2RML is based on a tabular data representation, on which restructuring, transformation and filtering may be applied. D2RML pushes the limits of a mapping language by incorporating 'programming' features. Although a mapping language cannot substitute a programming language, the real world cases that we discuss demonstrate that such features are essential if such languages aspire to gain acceptance in practice. This paper is an extended version of [3], which refines the restructuring features of D2RML and focuses on real word scenarios.

The rest of the paper has as follows: Sect. 2 gives an overview of R2RML and RML on which our work is mostly based. Section 3 discusses real examples where existing mapping languages turn out to be insufficient. In Sect. 4 we present the simple data model underlying D2RML. In Sect. 5 we describe how several widely used information sources can be cast onto that model, and in Sect. 6 we present the definition of D2RML. Section 7, in place of an evaluation, demonstrates the power of D2RML by describing how it can solve the practical needs outlined in Sect. 3. Section 8 concludes the paper.

2 Related Work, R2RML and RML

RML and xR2RML are two R2RML-based RDF mappings languages that support both relational and non-relational data. As such they share several common features, but differ in some of their focus points. E.g. xR2ML supports mapping from mixed formats (e.g. relational tables with JSON values), and also RDF lists. On the other hand, RML extended with FnO [12] supports interaction with data sources using established vocabularies [6] and interaction with abstract data processing functions. For both, R2RML is the starting point.

R2RML works with logical tables (`rr:LogicalTable`), which may be base tables, views, or result sets obtained by an SQL query. Each logical table is mapped to RDF triples using one or more triples maps. A triples map is a rule that maps each logical table row to several RDF triples. The rule consists of a subject map that generates the subject of all RDF triples for each row, and several predicate-object maps, that consist of predicate and object or referencing object maps. A predicate map determines predicates for the RDF triples, and an object map their objects. A subject or predicate-object map may include one or more graph maps, which specify a named graph for the resulting triples. Referring object maps allow joining of triples maps. A referring object map specifies

a parent triples map (`rr:parentTriplesMap`), the subjects of which will act as objects for the current triples map. RDF terms (i.e. concrete IRIs and literals for the triples) are either declared constants (`rr:constant`), or obtained from the underlying table, view or result set by specifying a column name (`rr:column`) that will supply the values, or generated by a string template (`rr:template`) that includes reference to columns. String templates offer very rudimentary options to manipulate actual database values and generate custom IRIs and literals.

RML extends R2RML by allowing other sources (`rml:LogicalSource`, e.g. JSON or XML files), by defining data iterators (`rml:iterator`) to split the data from such sources into base elements (the equivalent of rows), and by allowing particular references (`rml:reference`), in the form of subelement selectors within the base element, to define the value sources for RDF terms. The iterators and the references depend on the underlying data source, and may be XPath or JSONPath expressions, CSV column names or SPARQL return variable names. Their type is declared using `rml:referenceFormulation`. To describe access to diverse data sources, RML suggests the use of vocabularies, such as DCAT, CSVW, Hydra, and SPARQL-SD. However, these vocabularies in general do not prescribe a way to formulate actual requests (e.g. to a web API that paginates the results using next page access keys).

3 Motivating Examples

Here we present some examples that highlight the need for additional flexibility from an RDF mapping language. All are adapted (to save space) real examples.

Example 1. Consider the following except from a database containing a timeline of modern Greek history events. The database was modeled as a single table.

ID	*date*	*summary*	*senderA*	*receiverB*	*sourceA*	*senderB*	*receiverB*	*sourceB*	*keywords*
304	21/11/1940	Palairet calls F.O. saying that ...	Palairet	F.O.	F.O.371/24907/R8517	Palairet	Halifax	F.O.371/24907/R8879	J. Metaxas, Greece - Economy

Each row contains a date, summary and some keywords. Since each event turned out to have in practice at most two references, the modeler of the database included two sets of sender, receiver and source columns. Moreover, keywords are included in a single column, separated by commas, but there may be multi-term keywords in which terms are separated by a dash. xR2RML provides a solution if the keywords entry were in a structured format (eg. JSON). An RDF graph for the above, not inheriting the modeling problems, could be the following:

```
cge:304 [ a cge-t:Event ; cge-t:date "1940-11-21"^^xsd:date ; cge-t:summary "Palairet ..." ;
          cge-t:reference [ a cge-t:Letter ; cge-t:source "F.O.371/24907/R8817" ;
                            cge-t:sender "Palairet" ; cge-t:receiver "F.O." ] ;
          cge-t:reference [ a cge-t:Letter ; cge-t:source "F.O.371/24907/R8879" ;
                            cge-t:sender "Palairet" ; cge-t:receiver "Halifax" ] ;
          cge-t:keyword [ a cge-t:Term ; kvoc-t:text "J. Metaxas" ] ;
          cge-t:keyword [ a cge-t:Term ; kvoc-t:text "Greece", "Economy" ] .
```

The transformation of the keywords, which splits the entry at the commas and then at the dashes to generate a nested structure, is problematic even using

the SQL power of R2RML. For other data sources (e.g. CSV files) it would be impossible to do anything more than copy the original data structure. Certainly, this is a not an optimally designed database, but such cases do occur in practice.

Example 2. Consider the following excerpt from the PeriodO (http://perio.do/) gazetteer of historic periods, which is available as a JSON document:

```
{ "periodCollections": {
    "p0339m9": {
      "id": "p0339m9", "type": "PeriodCollection",
      "definitions": { ... ,
        "p0339m9f72b": {
          "id": "p0339m9f72b", "type": "PeriodDefinition", "start": "1204", "stop": "1453",
          "spatialCoverage": [ {"id": "http://dbpedia.org/resource/Greece" } ],
          "localizedLabels": { "eng": [ "Late Byzantine" ] } },
        "p0339m9jq2m": {
          "id": "p0339m9jq2m",  ... }, ... } },
    "p08nrfc": { ... }, ...  } }
```

Using the SKOS model, we would like to generate the following RDF graph:

```
ark:p0339m9 [ a ark-t:PeriodCollection ] .
ark:p0339m9f72b [ a ark-t:PeriodDefinition ; rdfs:label "Late Byzantine"@en ;
                  ark-t:earliestYear "1204"^^xsd:gYear ; ark-t:latestYear "1453"^^xsd:gYear ;
                  skos:inScheme ark:p0339m9 ; dcterms:spatial dbpedia:Greece ] .
```

Using RML and JSONPath, we could specify a triples map to iterate over the period collections, and then a triples map to iterate over the period definitions to generate the respective triples; but inside a period definition we do not know the enclosing period collection, to generate the `skos:inScheme` triple. Thus we cannot use a referring object map (for period definitions inside period collections). We could possibly specify a third triples map to iterate over the collections and generate triples with object the collection id and subjects the included period ids, but this violates a basic assumption in both R2RML and RML that each iteration over the data should produce a unique subject.

Example 3. Geonames provides its gazetter data as a set of tab-delimited files. Among them, file `admin1Codes.txt` contains top-level administrative regions for all countries, `XX.txt`, where `XX` is a country code, a country's locations, and `alternateNames/XX.txt` alternate location names and links to other resources. Consider line `(GR.ESYE31; Attica; 6692632)` from `admin1Codes.txt`, and lines

```
256601;  Athens; Athinai,Athina; P; PPLC; GR; ESYE31; 445408
445408;  Athens Prefecture; Athena,Athina; A; ADM2; GR; ESYE31; 445408
6692632; Attica; Attica,Attiki; A; ADM1; GR; ESYE31

177543;  264371; el; Athina; 1
1593954; 264371; en; Athens;
2919841; 264371; link; http://en.wikipedia.org/wiki/Athens;
```

from `GR.txt` and `alternatenames/GR.txt`, for Greece, respectively.

The column names are (admin1code, name, geonameid), (geonameid, name, alternate names, feature class, feature code, country code, admin1 code, admin2 code) and (alternateNameid, geonameid, language, alternate name, is PreferredName), respectively. From the above, we want to generate the following triples:

```
geo:256601 [ a gn:Feature ; gn:name "Athens" ; gn:featureCode gn:P.PPLC ;
             gn:officialName "Athina"@el ; gn:alternateName "Athens"@en ;
             gn:wikipediaArticle <http://en.wikipedia.org/wiki/Athens> ;
             gn:parentADM1 geo:6692632 ; gn:parentADM2 geo:445408 ] .
geo:445408 [ a gn:Feature ; gn:name "Athens Prefecture" ; gn:featureCode gn:A.ADM2 ;
             gn:parentADM1 geo:6692632 ] .
geo:6692632 [ a gn:Feature ; gn:name "Attica" ; gn:featureCode geo-ont:A.ADM1 ] .
```

To achieve this we need some conditions (e.g. do not include a `gn:parentADM2`) if in a line of `GR.txt` the geonameid and admin2 code coincide). Moreover, in the triples map iterating over `GR.txt`, we need to include a referring object map to perform a join with `admin1Codes.txt`, so as to know that `GR.ESYE31` has geonameid `6692632`. But to do the join we need to concatenate the country code `GR` with the admin1 code `ESYE31`, which in `GR.txt` are provided in distinct columns. In a relational database we could possibly formulate such queries, but with CSV files we have much less flexibility. Even if we overcome somehow the problem of generating a combined key for the join, iteratively joining large CSV files can be inefficient. Instead, we could probably start building the RDF graph by first mapping `admin1Codes.txt`, and then execute the mapping for `GR.txt`, exploiting the contents of the up to then generated RDF graph. Furthermore, in each line `alternateNames/GR.txt`, the value of the language column determines how to interpret the alternate name. If we see `link` we should use `gn:wikipediaArticle`. If we see a language code we should further check the last column: if it is `1` we use `gn:officialName` otherwise `gn:alternateName`. To do this we need conditions and case statements. If the data was relational, we could exploit SQL and define three triples maps, one for each predicate; but with a CSV file this is not possible. Even with relational data, a single triples map, with a conditional statement selecting each time the right predicate, is probably a clearer, more concise, and possibly more efficient, modeling since it requires a unique iteration over the data.

Example 4. Consider the row `(4821; 1431-1433 AD; Samothrace; <inscription>)` of a CSV database of Christian and Byzantine inscriptions provided by the University of Athens that contains an id, a chronology, a location and the actual inscription text. We would like to generate the following RDF graph:

```
bci:4821 [ bci-t:Inscription ; bci-t:chronology "1431-1433 AD" ; bci-t:location "Samothrace" ;
           kvoc-t:date [ a time:DateTimeInterval ;
                         time:hasBeginning tl-t:Y.1431 ; time:hasEnd tl-t:Y.1433 ] ;
           kvoc-t:location geo:734358 ; kvoc-ont:period ark:p0339m9f72b ] .
```

In this case the mapping involves some processing using data analysis services. A geonames linking service that links `Samothrace` to `geo:734358`, the geonames resource for Samothrace, a date recognizer that transforms `1431-1433 AD` into a time description using OWL Time vocabulary, and finally a periodO linking service that uses the geonames location and the OWL time date range to classify the inscription to the Late Byzantine period `ark:p0339m9f72b`. The first two services can be seen as data sources that require parameters taking values from the original data; this is supported in RML. But for the third service we need to specify parameters values that are results of the previous two services.

Moreover, we might want to use the results of the two first services only as intermediate results for the first service and not produce any triples for them.

4 Model

The above examples demonstrate that a practical RDF mapping language should include provisions for complex mapping capabilities, that may result from the actual structure or the data, from peculiarities of the data representation choices or models, or from the need for structure altering transformations. To create a general framework for such a language, we define first an abstract tabular data model. Essentially, we assume that data coming from a data source give rise to a tabular structure, which is extensible by the mapping language: new columns may be added by transforming existing ones to generate input data for further transformations. Each cell of the tabular structure may contain a set of values.

Definition 1. *A* set row *of arity k is a tuple $\langle D_1, \ldots, D_k \rangle$, where $D_1, \ldots, D_k$ are sets of values. A* name row *of arity k is a tuple $\langle n_1, \ldots, n_k \rangle$, where $n_1, \ldots, n_k$ are names. A* set table *of arity k with m rows is a tuple $\mathcal{S} = \langle N, \mathcal{T} \rangle$, where N is a name row and $\mathcal{T} = [\mathcal{D}_1, \ldots, \mathcal{D}_m]$ a list of set rows, all of arity k, such that the i-th elements of $\mathcal{D}_1, \ldots, \mathcal{D}_m$, for $1 \leq i \leq k$, share all the same domain.*

The names allow us to refer to particular elements of set rows and tables. We denote the set of values that corresponds to name n_i in a set row $\mathcal{D}$ by $\mathcal{D}[n_i]$ and by $\mathcal{S}[n_k]$ (a *column* of $\mathcal{S}$) the list $[\mathcal{D}_1[n_k], \ldots, \mathcal{D}_m[n_k]]$ of value sets that are obtained from the several set rows of $\mathcal{S}$. For a particular set row $\mathcal{D}$ and the several n_i, the sets $\mathcal{D}[n_i]$ may have different numbers of values and in general there is no alignment between the individual values among the several sets, and all individual values are equivalent with respect to their relation to the values of the other sets in the same set row.

Definition 2. *A* filter $\mathcal{F}$ *over a set table $\mathcal{S}$ is a tuple $\langle n, f \rangle$, where n is a column name and f a function, such that $f(\mathcal{D}[n]) \subseteq \mathcal{D}[n]$ for all set rows $\mathcal{D}$ of $\mathcal{S}$.*

We denote the set value $f(\mathcal{D}[n])$, obtained by applying $\mathcal{F}$ on a set row $\mathcal{D}$ by $\mathcal{F}(\mathcal{D})$. A filter may be seen as the implementation of a condition.

Data for set tables are acquired from *information sources*. To accommodate several possible information sources in our model, we consider, as in RML, that the information source upon a *request* provides in a reply an *effective data source*, a structure that groups data in several autonomous elements. The division of the effective data source to these autonomous elements is achieved by an *iterator*, which specifies a *logical array*, through whose items the iterator iterates. Each item of a logical array may be a complex structure (another effective data source), so in order to extract from it lists of values to construct set rows values, we need some *selectors*. The selectors transform a logical array into a set table.

Definition 3. *The triple $\mathcal{A} = \langle \mathcal{I}, t, \mathcal{L} \rangle$, where $\mathcal{I}$ is an effective source specification, t an iterator, and $\mathcal{L}$ a set of selectors, is a* data acquisition pipeline.

Each data acquisition pipeline $\mathcal{A}$ gives rise to a unique set table $\mathcal{S}_{\mathcal{A}}$. A data acquisition pipeline may be parametric. A parametric data acquisition pipeline $\mathcal{A}'$ that depends on $\mathcal{A}$ is a data acquisition pipeline whose parameters take values from one or more columns of $\mathcal{S}_{\mathcal{A}}$ and is called a *transformation* of $\mathcal{A}$.

Definition 4. *A series of data acquisition pipelines $\mathcal{A}_0$, $\mathcal{A}_1$, ..., $\mathcal{A}_l$, where each $\mathcal{A}_i$, for $i > 1$, is a transformation that depends on one or more $\mathcal{A}_j$ for $j < i$ is a* set table specification. *$\mathcal{A}_0$ is the* primary *data acquisition pipeline.*

A set table specification gives rise to a unique set table: $\mathcal{S}_{\mathcal{A}_0}$ extended by columns contributed by $\mathcal{A}_1$, ..., $\mathcal{A}_l$. Each transformation is realized as a series of requests to an information source, after binding the parameters to *all* possible combinations of values obtained from the referred to columns of the set table constructed from the preceding data acquisition pipelines. Thus, a set table specification is evaluated serially. The primary data acquisition pipeline $\mathcal{A}_0$ gives rise to set table $\mathcal{S}_{\mathcal{A}_0}$. Then, for each set row $\mathcal{D}$ of $\mathcal{S}_{\mathcal{A}_0}$, evaluating $\mathcal{A}_1$ gives rise to a set table $\mathcal{S}_{\mathcal{A}_1}(\mathcal{D})$. By flattening all rows of $\mathcal{S}_{\mathcal{A}_1}(\mathcal{D})$ into a single row we obtain a new set row that is appended to $\mathcal{D}$. Doing this for all set rows $\mathcal{D}$ results in $\mathcal{S}_{\mathcal{A}_0\mathcal{A}_1}$. Proceeding this way, eventually $\mathcal{S}_{\mathcal{A}_0}$ is extended to set table $\mathcal{S}_{\mathcal{A}_0\mathcal{A}_1\ldots\mathcal{A}_l}$. More formally, let $n_1, \ldots, n_k$ be the names, and $[\mathcal{D}_1, \ldots, \mathcal{D}_m]$ the rows of $\hat{\mathcal{S}} \doteq \mathcal{S}_{\mathcal{A}_0\ldots\mathcal{A}_i}$. Evaluating $\mathcal{A}_{i+1}$ on each row of $\hat{\mathcal{S}}$ produces set tables $\mathcal{S}_{\mathcal{A}_{i+1}}(\mathcal{D}_1)$, ..., $\mathcal{S}_{\mathcal{A}_{i+1}}(\mathcal{D}_m)$. Since all these set tables are produced by the same data acquisition pipeline $\mathcal{A}_{i+1}$, they share the same arity, say k', and let $\hat{s}_1, \ldots, \hat{s}_{k'}$, be the selectors of $\mathcal{A}_{i+1}$. Thus $\mathcal{S}_{\mathcal{A}_0\ldots\mathcal{A}_{i+1}} = \langle N, \mathcal{T}\rangle$, where $N = \langle n_1, \ldots, n_k, \mathcal{A}_{i+1}.\hat{s}_1, \ldots, \mathcal{A}_{i+1}.\hat{s}_{k'}\rangle$, $\mathcal{T} = [\mathcal{D}'_1, \ldots, \mathcal{D}'_m]$, $\mathcal{D}'_j = [\mathcal{D}_j[n_1], \ldots, \mathcal{D}_j[n_k], \hat{\mathcal{D}}_{j1}, \ldots, \hat{\mathcal{D}}_{jk'}]$ for $1 \leq j \leq m$, and $\hat{\mathcal{D}'_{jl}} = \bigcup \mathcal{S}_{\mathcal{A}_{i+1}}(\mathcal{D}_j)[\hat{s}_l]$ for $1 \leq l \leq k'$.

Definition 5. *A* triples rule $\mathcal{R}$ *over a set table $\mathcal{S} = \langle N, \mathcal{T}\rangle$ is either (a) a triple of filters $\langle \mathcal{F}_s, \mathcal{F}_p, \mathcal{F}_o\rangle$, over $\mathcal{S}$, called the* subject, predicate *and* object filter, *respectively, or (b) a triple $\langle \mathcal{F}_s, \mathcal{F}_p, \hat{\mathcal{R}}\rangle$, over $\mathcal{S}$, where $\mathcal{F}_s$, $\mathcal{F}_p$ are the* subject *and* predicate filter, *respectively, and $\hat{\mathcal{R}}$ another triples rule.*

The implementation *of $\mathcal{R}$ is the set of RDF triples $\{(s,p,o) \mid s \in \mathcal{F}_s(\mathcal{D}), p \in \mathcal{F}_p(\mathcal{D}), o \in \mathcal{F}_o(\mathcal{D}), \mathcal{D} \in \mathcal{T}\}$ in case (a), and $\{(s,p,o) \mid s \in \mathcal{F}_s(\mathcal{D}), p \in \mathcal{F}_p(\mathcal{D}), o \in \mathcal{S}_{\hat{\mathcal{R}}}, \mathcal{D} \in \mathcal{T}\}$ in case (b), where $\mathcal{S}_{\hat{\mathcal{R}}}$ are the subjects of the implementation of $\hat{\mathcal{R}}$.*

A set of triples rules over some set tables defines a *Data-to-RDF mapping*. The relevant RDF dataset is the implementation of all its triples rules.

5 Retrieving and Interpreting Data

To define general data acquisition pipelines in practice, we consider that an information source, in response to a request, provides some source data. An information source may be either a server (RDBMS, web server/RESTful web service, SPARQL endpoint), the local file system, or a local data model (e.g. an

in-memory RDF model). The source data may be an instance of a particular data model (e.g. a SQL result set) or a document (e.g. a JSON or XML document). To obtain source data from an information source we need to specify a request (e.g. an HTTP GET request, an SQL query), as summarized in Table 1. Source data obtained as data models (e.g. SQL result sets) have a unique interpretation, but a document may be interpreted as one of several models. The effective data source is the source data interpreted: SQL and SPARQL results sets denote themselves, a JSON document a JSON Tree, an XML/HTML document a DOM or XDM, and a CSV document a table. In some cases, to obtain an effective data source, we need an intermediate interpretation of the source data as a new information source. This is the case e.g. with an RDF document, which should be seen first as an RDF dataset, from which SPARQL result sets can then be obtained.

Table 1. Information sources, requests and source data

Information source	Request	Source data
RDBMS	SQL SELECT Query	SQL Result Set
Web Server/ RESTful Service	HTTP GET/ POST Request	JSON/XML/CSV/HTML/ RDF/TXT Document
SPARQL Endpoint/ RDF model	SPARQL SELECT Query and Graph IRIs via API	SPARQL Result Set
File System	File Request	JSON/XML/CSV/HTML/ RDF/TXT Document

Table 2. Effective data sources, iterators and selectors

Effective data source	Type	Iterator	Selector
SQL Result Set	Tabular	Row Iterator	Column Name
SPARQL Result Set	Tabular	Row Iterator	Variable Name
JSON Tree	Hierarchical	JSONPath	JSONPath
XDM	Hierarchical	XPath/XQuery	XPath/XQuery
DOM	Hierarchical	CSS Selector	CSS Selector
CSV Document	Tabular	Row Iterator	Column Name/Number
Text Document	Flat	Regular Expression	Regular Expression

The model-specific iterators and selectors needed to convert effective data sources to set tables are shown in Table 2. For example, an SQL (SPARQL

result set) obtained through an SQL (SPARQL) SELECT query q that specifies attributes (variables) $n_1, \ldots, n_k$ in the SELECT statement for the returned columns (variables), returns a list of rows $[\langle v_{11}, \ldots, v_{1k}\rangle, \ldots, \langle v_{n1}, \ldots, v_{nk}\rangle]$. Using a row iterator and the column (variable) names $n_1, \ldots, n_k$ as selectors, we obtain the set table $\langle\langle n_1, \ldots, n_k\rangle, [\langle\{v_{11}\}, \ldots, \{v_{1k}\}\rangle, \ldots, \langle\{v_{n1}\}, \ldots, \{v_{nk}\}\rangle]\rangle$. Similarly, a JSONPath (XPath/XQuery, CSS) expression q splits a JSON tree [2] (XDM [15], DOM) interpretation of a JSON (XML/HTML) document $\mathcal{T}$ into a logical array of smaller JSON trees (XDMs, DOMs) $\mathcal{T}_1, \ldots, \mathcal{T}_n$. By executing as selectors JSONPath (XPath/XQuery, CSS) expressions $q_1, \ldots q_k$ over each $\mathcal{T}_1, \ldots, \mathcal{T}_n$ we get the set table $\langle\langle q_1, \ldots, q_k\rangle, [\langle C_{11}, \ldots, C_{1k}\rangle, \ldots, \langle C_{n1}, \ldots, C_{nk}\rangle]\rangle$, where C_{ij} is either the set of values (string values of text or attribute nodes) contained in the array (node set) that results from applying q_j on $\mathcal{T}_i$.

6 D2RML Specification

D2RML draws significantly from R2RML and RML, and follows the same strategy for defining mappings: triples maps, consisting of a subject map and several predicate-object maps. From RML it adopts and extends the interaction with information sources through requests, iterators and selectors. It also extends the expressive capabilities of R2RML and RML by allowing transformations, conditional and case statements, and custom RDF term generation functions. For its semantics, it relies on the data model of Sect. 4. Each triples map corresponds to a set table (Definition 5) and a set of triple rules (Definition 4) with the same subject filter over the common underlying set table. The information source, request and iterator for the original data acquisition pipeline are provided in the triples map definition. Any additional transformations are declared in the order of their application and extend incrementally the underlying set table. The selectors are implicitly declared in the included subject, predicate, object and graph maps.

6.1 Triples Maps

Triples maps are defined as in RML, but tabular data providing information sources are clearly distinguished from non-tabular ones. The inclusion of *Transformation* and *DefinedColumn* lists allow extending the primary set table.

TriplesMap ← `a rr:TriplesMap`
(`rr:logicalTable` ⟨*LogicalTable*⟩ | `dr:logicalSource` ⟨*LogicalSource*⟩)?
(`dr:transformations` (⟨*Transformation*⟩+)?
(`dr:definedColumns` (⟨*DefinedColumn*⟩+)?
(`rr:graphMap` ⟨*GraphMap*⟩)*
`rr:subjectMap` ⟨*SubjectMap*⟩ | `rr:subject` IRI
(`rr:predicateObjectMap` ⟨*PredObjMap*⟩)*

PredObjMap ← `a rr:PredicateObjectMap`
(`rr:predicateMap` ⟨*PredicateMap*⟩ | `rr:predicate` IRI)+
(`rr:objectMap` (⟨*ObjectMap*⟩|⟨*RefObjectMap*⟩) | `rr:object` (IRI|LIT))+
(`rr:graphMap` ⟨*GraphMap*⟩ | `rr:graph` IRI)*

6.2 Logical Tables and Logical Sources

A *LogicalTable* or *LogicalSource* specifies a data acquisition pipeline (excluding the separators). In the case of query supporting information sources (RDBMSs' and SPARQL services), for compatibility with R2RML, they contain also the query-relevant details of the request. The `is:parameters` predicate helps declare parameters in queries of parametric data acquisition pipelines. For other information sources, the request and any parameters are part of the *InformSource* specification. For non-tabular data information sources, *LogicalSource* should contain the definition of the iterator (`dr:iterator` and `dr:referenceFormulation`) used to split the effective data source.

D2RML introduces two special sources: The first `dr:CurrentModel`, represents the RDF model of the current RDF dataset generated by the D2RML processor, and is interpreted as an *SPARQLTable*. Since the model is constantly updated, we need to define an execution order for the triples maps. Thus, a D2RML document using `dr:CurrentModel` must specify a `is:TriplesMapOrder`, whose `is:mapOrder` defines the execution order of the triple maps as a list. The second source, `dr:SetTable`, allows the generation of a new table from the values of some selected columns (`dr:transferredColumns`) of the dependent on set table. The `dr:SetTable` instance is generated by taking the values of each column of the current row, in order of appearance, and putting them into a new table, by aligning values having the same order. Thus it restructures the data: it converts, one or more sets of values, into a table where each column in each row contains aligned single values. If used as a *LogicalSource*, `dr:SetTable` allows, as in xR2RML, interpreting row values as structured documents (eg. JSON, XML).

LogicalTable$_1$	←	`a rr:LogicalTable`	`a dr:CurrentModel`
		`dr:source` ⟨*InformSource*⟩	
		SQLTable | *SPARQLTable* | *CSVTable*	*SPARQLTable*
		(`is:parameters` (⟨*DataVariable*⟩+))?	
LogicalTable$_2$	←	`a dr:SetTable ;` (`dr:transferedColumns` (⟨*ValueRef*⟩+))?	
LogicalSource	←	(`a dr:LogicalSource ; dr:source` ⟨*InformSource*⟩) | *LogicalTable*$_2$	
		`dr:iterator` LIT `; dr:referenceFormulation` IRI	

An *SQLTable* is defined as in R2RML. A *SPARQLTable* must specify a query (`dr:sparqlQuery`) and any default or named graphs (`dr:defaultGraph`, `dr:namedGraph`). Finally a, `CSVTable` must specify a delimiter (`dr:delimiter`), whether there is a header line (`dr:headerline`), and possibly a quote, comment, escape, or record separator characters.

6.3 Information Sources

An information source may be a RDBMS, an HTTP server, a SPARQL endpoint, or the file system.

InformSource ← *RDMSSource* | *SPARQLSource* | *HTTPSource* | *FileSource*

An *RDMBSSource* must specify the type of the RDMBS (`is:rdbms`), the location (`is:location`) and any needed information for establishing the connection (e.g. username, password). An *HTTPSource* should either specify a single URI (`is:uri`) or prescribe a full HTTP request (`is:request`). The latter is specified in using the W3C's 'HTTP Vocabulary in RDF 1.0' [9] and 'Representing Content in RDF 1.0' vocabularies [10]. An *HTTPSource* may contain a list of parameters (`is:parameter`). To account for result pagination, it may contain a request iterator in the parameter list. A request iterator may be a *KeyRequestIterator* or a *CountRequestIterator*. Both of them should provide the parameter name (`is:name`) which should appear in the URI of the HTTPRequest, and an initial value (`is:initialValue`). A key request iterator must specify how to extract each time the new parameter value from the current server results in order to formulate the subsequent request, while a count request iterator must specify an increment (`is:increment`) and possibly a maximum value (`is:maxValue`). The set of iteration policies is extensible. A *SPARQLSource* must specify simply the URI of the service (`is:uri`). A *FileSource* must specify the location of one or more files (`is:path`) and their encoding (`is:encoding`). Note that, following the earlier discussion, e.g. a *FileSource* that fetches an RDF file used in conjunction with a *SPARQLTable*, is interpreted as a RDF model information source.

6.4 Transformations and Defined Columns

A *Transformation* extends the underlying set table. Since it is a parametric data acquisition pipeline, its definition includes a *LogicalTable* or *LogicalSource* and one or more *ParameterBinding*s to assign parameter values. The latter consists of a reference to a value (*ValueRef*) or a constant value, and the parameter name (`dr:parameter`) in the corresponding information source the value will be bound to. A *DefinedColumn* allows for in-line set table transformations: to add new columns by applying a series of transformations on particular set table column values without consulting external sources. A *DefinedColumn* should declare the new column name `dr:name`, the function (`dr:function`) to generate the values (eg. `op:regex`, `op:replace`), and a list of arguments (`dr:parameterBinding`). It is assumed that the parameter names are provided by the function definition.

Because a *Transformation* may need to work not directly with the value sets of the underlying set table at the level of the selectors (where any alignment between values is lost), but first with higher level iterators that preserve the alignment and then with the value set producing selectors, a transformation may declare one such iterator (`dr:bindingIterator`) for each transformation that provides parameter bindings. When, executed, the values for parameter bindings will be generated aligned according to the iterator.

Transformation ← `a dr:Transformation`
`rr:logicalTable` ⟨*LogicalTable*⟩ | `dr:logicalSource` ⟨*LogicalSource*⟩
(`dr:parameterBinding` ⟨*ParameterBinding*⟩)+
(`dr:bindingIterator` ⟨*BindingIterator*⟩)*

DefinedColumn ← `a dr:DefinedColumn ;` `dr:name` LIT `;` `dr:function` IRI
(`dr:parameterBinding` ⟨*ParameterBinding*⟩)+

ParameterBinding ← `a dr:ParameterBinding`
`dr:parameter` LIT `;` `rr:constant` LIT | *ValueRef*

BindingIterator ← `rr:column` LIT | `dr:reference` LIT
(`dr:transformationReference` ⟨*Transformation*⟩)?

6.5 Term Maps and Conditions

The definition of a *TermMap* (*SubjectMap*, *PredicateMap*, *ObjectMap*, *GraphMap* and *LanguageMap*) follows the R2RML specification with the support for filters.

SubjectMap ← `a rr:SubjectMap ;` *IRIRef* | *BlankNodeRef*
(*SubjBody CSubjBody**) | *CSubjBody*+

PredicateMap ← `a rr:PredicateMap ;` (*PredBody CPredBody**) | *CPredBody*+

ObjectMap ← `a rr:ObjectMap ;` (*ObjBody CObjBody**) | *CObjBody*+

GraphMap ← `a rr:GraphMap ;` (*GraphBody CGraphBody**) | *CGraphBody*+

LanguageMap ← `a rr:LangMap ;` (*LangBody CLangBody**) | *CLangBody*+

SubjBody ← (`rr:class` IRI)* `;` (`rr:graphMap` ⟨*GraphMap*⟩ | `rr:graph` IRI)*
(`dr:condition` ⟨*Condition*⟩)?

[Pred|Graph]Body ← *IRIRef* `;` (`dr:condition` ⟨*Condition*⟩)?

ObjBody ← *IRIRef* | *BlankNodeRef* | *LiteralRef* `;` (`dr:condition` ⟨*Condition*⟩)?

LangBody ← *LiteralRef* `;` (`dr:condition` ⟨*Condition*⟩)?

C[Subj|Pred|Obj|Graph|Lang]Body ← `dr:cases` (⟨*[Subj|Pred|Obj|Graph|Lang]Body*⟩+)

Condition ← (*ValueRef*)? `;` (`dr:booleanOperator` IRI)?
(OPERATOR (*ValueRef* | LIT) | `dr:operand` ⟨*Condition*⟩)+

RefObjectMap ← `a rr:RefObjectMap ;` `rr:parentTriplesMap` ⟨*TriplesMap*⟩
((`rr:joinCondition` ⟨*JoinCondition*⟩)+ |
(`dr:parameterBinding` ⟨*ParameterBinding*⟩)+)?

JoinCondition ← `a rr:Join ;` `rr:child` LIT `;` `rr:parent` LIT

To implement filters, a *TermMap* may contain a condition (`dr:condition`) and/or a case statement (`dr:cases`). If it contains a condition, it will be evaluated and the corresponding RDF term will be taken into account only if the condition holds. A condition statement must specify the actual value on which it will operate, and may include several tests which will be jointly evaluated using the operator specified by `dr:booleanOperator` (`op:and` or `op:or`). A test is specified either through an OPERATOR (`op:eq`, `op:le`, etc.) and a literal or *ValueRef* which define the value(s) with which the actual value will be compared using OPERATOR, or as a nested condition. Due to the underlying set table model, in general a behaviour of the operators for set arguments should be defined. In the current implementation it is assumed that a condition holds if it holds for a single value pair, but this is a point of further refinement of the language. The operation type (eg. number/string comparison) depends on the operand XSD types. A case statement includes a list of alternative realizations of a *TermMap*, each along with a condition. If the condition evaluates to true, the corresponding *TermMap* is realized, otherwise control flows to the next case.

Finally, a referring object map (*RefObjectMap*) may be defined either as in R2RML or using a *ParameterBinding*, in the case the *LogicalTable* or *LogicalSource* of the referring object's triples map is a parametric data acquisition pipeline: the *ParameterBinding* provides the parameters values to be used in the parametric data acquisition pipeline of the referring object map.

6.6 RDF Terms

RDF terms are specified as in RML, but to account for values coming from transformations, RDF terms are generated through value references, specified by two components: a compulsory `rr:column`, `rr:template` or `dr:reference`, and an optional `dr:transformationReference` to specify the underlying transformation for the `rr:column`, `rr:template` or `dr:reference`. If missing, the primary logical array is assumed. To overcome the limited data manipulation options offered by `rr:template`, a value reference may include local defined columns (`dr:definedColumns`) that are need to generate a particular RDF Term.

IRIRef ← `rr:constant` IRI | *ValueRef* ; (`rr:termType rr:IRI`)?

LiteralRef ← `rr:constant` LIT | *ValueRef* ; (`rr:termType rr:Literal`)?
((`dr:languageMap` ⟨*LanguageMap*⟩ | `rr:language` LIT) |
`rr:datatype` IRI)?

BlankNodeRef ← *ValueRef* ; (`rr:termType rr:BlankNode`)?

ValueRef ← `rr:column` LIT | `rr:template` LIT | `dr:reference` LIT
(`dr:transformationReference` ⟨*Transformation*⟩)?
(`dr:definedColumns` (⟨*DefinedColumn*⟩+)?

7 Evaluation

In Sect. 3 we identified cases where the desired mappings could not be achieved using existing RDF mapping languages. In fact, all cases were real and arose in the context of a project aiming to provide semantic analysis services over a repository of heterogeneous cultural data. This provided a real testbed for the usefulness of D2RML; here we discuss how it helped solving such mappings needs. (To save space we omit the `dr:referenceFormulation` declarations and write `dr:transformationReference` as `dr:tRef`).

Example 1 involved data restructuring: split each joint keyword into a distinct keywords and separate the each keyword's terms. To do this, we first extend the primary set table with a defined column containing the set of split keywords for each row (`op:split` splits its `input` on the provided `separator`). Then in a referring object map, we build a new table (`dr:SetTable`) from each keyword set, and perform there the splitting on the dashes. In this way we achieve a restructuring that preserves the connection of each term with the source keyword:

```
<#EventMap>
   rr:logicalTable [ dr:source <#FOSource> ; rr:tableName "Events" ] ;
   dr:definedColumns ( [
      dr:name "KW" ; dr:function op:split ;
      dr:parameterBinding [ dr:parameter "input" ; rr:column "keywords" ] ;
```

```
      dr:parameterBinding [ dr:parameter "separator" ; rr:constant "," ] ] ) ;
   rr:subjectMap [ rr:template  {@tl}{ID}" ; rr:class cge-t:Event ] ;
   rr:predicateObjectMap [
      rr:predicate cge-t:keyword ;
      rr:objectMap [
         rr:parentTriplesMap [
            rr:logicalTable [ a dr:SetTable ; dr:transferedColumns ( [ rr:column "KW" ] ) ] ;
            rr:subjectMap [ rr:class cge-t:Term ; rr:termType  rr:BlankNode ] ;
            rr:predicateObjectMap [
               rr:predicate kvoc-t:text ;
               rr:objectMap [
                  dr:definedColumns ( [
                     dr:name "TERM" ; dr:function op:split ;
                     dr:parameterBinding [ dr:parameter "input" ; rr:column "KW" ] ;
                     dr:parameterBinding [ dr:parameter "separator" ; rr:constant "-" ] ] ) ;
                  rr:column "TERM"; rr:termType rr:Literal ] ] ] ] ] .
```

In Example 2 we needed maps generating more than one subjects for each row to link periods with period collections. The solution is to define first a map to declare each collection as a `skos:ConceptScheme`, and then a second multi-subject map to link periods with period collections:

```
<#CollectionMap1>
   dr:logicalSource [ dr:source <#PeriodoSource> ; dr:iterator "$.periodCollections.*" ] ;
   rr:subjectMap [ rr:template "{@ark}{$.id}" ;
                   rr:class periodo:PeriodCollection ; rr:class skos:ConceptScheme ] .
<#CollectionMap2>
   dr:logicalSource [ dr:source <#PeriodoSource> ; dr:iterator "$.periodCollections.*" ] ;
   rr:subjectMap [ rr:template "{@ark}{$.definitions.*.id}" ] ;
   rr:predicateObjectMap [ rr:predicate skos:inScheme ;
                           rr:objectMap [ rr:template "{@ark}{$.id}" ; rr:termType rr:IRI ] ] .
```

In Example 3, to avoid joining CSV files, we needed to generate triples linking admin codes to their geonames ids, and then consult these triples to resolve admin code references when generating triples for a country's locations. To achieve this, we first define `ADMIN1Map`. Because `ADMIN1Map` should be executed first, we include a triples map order statement. According to that ordering, next is executed `CountryMap`. This map uses the transformation `ADMIN1Trans`, which extends the primary set table by a new column with the geonames id of the locations admin code. `ADMIN1Trans` operates on the `dr:CurrentModel` logical table, so as to has access to the triples generated by `ADMIN1Map`. Last comes the `AlternateNamesMap` which processes the file with the alternate names and contains conditional statements to decide which predicate it should use for each entry:

```
<#Order>
   dr:mapOrder ( <#ADMIN1Map> <#CountryMap> <#AlternateNamesMap> ) .
<#ADMIN1Map>
   rr:logicalTable [ dr:source <#ADMIN1Source> ; dr:delimiter ";" ] ;
   rr:subjectMap [ rr:template "{@geo}{##3}/" ] ;
   rr:predicateObjectMap [ rr:predicate gn:code ;
                           rr:objectMap  [ rr:column "##1" ; rr:termType  rr:Literal ] ] .
<#ADMIN1Trans>
   rr:logicalTable [
      a dr:CurrentModel ;
      dr:sparqlQuery "SELECT ?admin1uri WHERE {?admin1uri gn:code \"{@@name@@}\" }" ] ;
   dr:parameterBinding [ dr:parameter "name" ; rr:template "{##6}.{##7}" ] .
<#CountryMap>
   rr:logicalTable [ dr:source <#CountrySource> ; dr:delimiter ";" ] ;
   dr:transformations ( <#ADMIN1Trans> ) ;
```

```
   rr:subjectMap [ rr:template "{@geo}{##1}/" ; rr:class gn:Feature ] ;
   rr:predicateObjectMap [
      rr:predicate gn:parentADM1 ;
      rr:objectMap [ rr:column "admin1uri" ; dr:tRef <#ADMIN1Trans> ; rr:termType  rr:IRI ] ] .
<#AlternateNamesMap>
   rr:logicalTable [ dr:source <#AlternateNamesSource> ; dr:delimiter ";" ] ;
   rr:subjectMap [ rr:template "{@geo}{##2}/" ; rr:termType rr:IRI ] ;
   rr:predicateObjectMap [
      rr:predicateMap [
         dr:cases ( [ rr:constant gn:officialName ;
                      dr:condition [ rr:column "##5" ; op:eq "1" ] ]
                    [ rr:constant gn:alternateName ] ) ] ;
      rr:objectMap  [
        rr:column "##4" ; rr:termType  rr:Literal ; dr:languageMap [ rr:column "##3" ] ;
        dr:condition [ rr:column "##3" ; op:neq "link" ] ] ] ;
   rr:predicateObjectMap [
      rr:predicate gn:wikipediaArticle ;
      rr:objectMap [ rr:column "##4" ; rr:termType  rr:IRI ;
                     dr:condition [ rr:column "##3" ; op:eq "link" ] ] ] ] .
```

In Example 4 we needed to define three transformations on the primary set table, the last one of which depended on the first two. The first two (`DateMap`, `LocationMap`) are simple triples maps that use a date/location identification information sources. We assume that `DateMap` returns a JSON array with elements of the form { `"start":` *start-date-uri*, ‘‘`end’’:` *end-date-uri* }, as it may identify several ranges in the input. Similarly, `LocationMap` returns an array of { `"place":` *place-uri* }. Thus, executing these on the primary set table, we get the table extended with two new logical columns. Taking values from the new columns, we can then execute the `PeriodoMap` transformation. However, we need to match start with end dates; for this we need to specify a *BindingIterator* to declare that we need to iterate on the top level array returned by `DateMap`:

```
<#DateMap>
   dr:logicalSource [ dr:source <#DatifyService> ; dr:iterator "$" ] ;
   dr:parameterBinding [ dr:parameter "text" ; rr:column "chronology" ] .
<#LocationMap>
   dr:logicalSource [ dr:source <#LocalifyService> ; dr:iterator "$" ] ;
   dr:parameterBinding [ dr:parameter "text" ; rr:column "location" ] .
<#PeriodoMap>
   dr:logicalSource [ dr:source <#PeriodoService> ; dr:iterator "$" ] ;
   dr:bindingIterator [ dr:transformationReference <#DateMap> ; dr:reference "$" ] ;
   dr:parameterBinding [ dr:parameter "start" ;
                         dr:reference "$.start" ; dr:transformationReference <#DateMap> ] ;
   dr:parameterBinding [ dr:parameter "end" ;
                         dr:reference "$.end" ; dr:transformationReference <#DateMap> ] ;
   dr:parameterBinding [ dr:parameter "place" ;
                         dr:reference "$.place" ; dr:transformationReference <#LocationMap> ] .
<#InscriptionsMap>
   rr:logicalTable [ dr:source <#InscriptionsSource> ; dr:delimiter "\t" ] ;
   dr:transformations ( <#DateMap> <#LocationMap> <#PeriodoMap> ) ;
   rr:subjectMap [ rr:template "{@bci}{##1}" ; rr:class bci-t:Inscription ] ;
   rr:predicateObjectMap [
      rr:predicate kvoc-t:location ;
      rr:objectMap [ rr:reference "$.uri" ; dr:tRef <#LocationMap> ; rr:termType rr:IRI ] ] ;
   rr:predicateObjectMap [
      rr:predicate kvoc-t:date ;
      rr:objectMap [
         rr:parentTriplesMap [
            rr:subjectMap [ rr:class time:DateTimeInterval ; rr:termType  rr:BlankNode ] ;
            rr:predicateObjectMap [
               rr:predicate time:hasBeginning ;
               rr:objectMap [ dr:reference "$.start" ; dr:tRef <#DateMap> ] ] ;
```

```
        rr:predicateObjectMap [
           rr:predicate time:hasEnd ;
           rr:objectMap [ dr:reference "$.end" ; dr:tRef <#DateMap> ] ] ] ] ] ;
rr:predicateObjectMap [
   rr:predicate  kvoc-t:period ;
   rr:objectMap  [ rr:column "$.uri" ; dr:transformationReference <#PeriodoMap> ] ] .
```

Our D2RML processor is available at http://apps.islab.ntua.gr/d2rml/.

8 Conclusions

Motivated by practical cases of more complex RDF mapping needs not covered by existing languages, we presented D2RML, an extension of R2RML and RML, which, based on an abstract underlying data model, allows the orchestrated retrieval of data from diverse information sources, their transformation using relevant web services, their filtering and manipulation using simple operations, and finally their limited restructuring and mapping to RDF triples.

To offer such capabilities, D2RML adds a programming language flavor to the mapping process, but we claim that this is necessary if such languages are ever going to be widely accepted and used in practice. If in a real mapping problem scenario, the source data do not exactly reflect the structure of the target model, and the modeler needs extended data manipulation capabilities, they usually resort to a programming language, thus invalidating the very usefulness of a mapping language. Our aim was to design a mapping language that would limit the cases where this occurs and where writing custom code turns out to be unavoidable. Further extensions to the language will most probably be needed to accommodate other needs, but the underlying abstract data model provides a solid ground on which to incorporate such extensions.

Acknowledgements. We acknowledge support of this work by 'APOLLONIS' (MIS 5002738), a project implemented under the Action 'Reinforcement of the Research and Innovation Infrastructure', funded by the Operational Programme 'Competitiveness, Entrepreneurship and Innovation' (NSRF 2014-2020) and co-financed by Greece and the European Union (European Regional Development Fund).

References

1. Bischof, S., Decker, S., Krennwallner, T., Lopes, N., Polleres, A.: Mapping between RDF and XML with XSPARQL. J. Data Semant. **1**(3), 147–185 (2012)
2. Bourhis, P., Reutter, J.L., Suárez, F., Vrgoc, D.: JSON: data model, query languages and schema specification. In: PODS, pp. 123–135. ACM (2017)
3. Chortaras, A., Stamou, G.: D2RML: integrating heterogeneous data and web services into custom RDF graphs. In: LDOW. CEUR Workshop Proceedings (2018)
4. Connolly, D.: Gleaning resource descriptions from dialects of languages (GRDDL) (2007). https://www.w3.org/TR/grddl/
5. Das, S., Sundara, S., Cyganiak, R.: R2RML: RDB to RDF mapping language (2012). https://www.w3.org/TR/r2rml/

6. Dimou, A., Nies, T.D., Verborgh, R., Mannens, E., de Walle, R.V.: Automated metadata generation for linked data generation and publishing workflows. In: LDOW. CEUR Workshop Proceedings, vol. 1593 (2016)
7. Dimou, A., Sande, M.V., Colpaert, P., Verborgh, R., Mannens, E., de Walle, R.V.: RML: a generic language for integrated RDF mappings of heterogeneous data. In: LDOW. CEUR Workshop Proceedings, vol. 1184 (2014)
8. Hert, M., Reif, G., Gall, H.C.: A comparison of RDB-to-RDF mapping languages. In: I-SEMANTICS, ACM International Conference Proceeding Series, pp. 25–32. ACM (2011)
9. Koch, J., Velasco, C.A., Ackermann, P.: HTTP vocabulary in RDF 1.0 (2017). https://www.w3.org/TR/HTTP-in-RDF10/
10. Koch, J., Velasco, C.A., Ackermann, P.: Representing content in RDF 1.0 (2017). https://www.w3.org/TR/Content-in-RDF10/
11. Langegger, A., Wöß, W.: XLWrap – querying and integrating arbitrary spreadsheets with SPARQL. In: Bernstein, A., et al. (eds.) ISWC 2009. LNCS, vol. 5823, pp. 359–374. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04930-9_23
12. De Meester, B., Dimou, A., Verborgh, R., Mannens, E.: An ontology to semantically declare and describe functions. In: Sack, H., Rizzo, G., Steinmetz, N., Mladenić, D., Auer, S., Lange, C. (eds.) ESWC 2016. LNCS, vol. 9989, pp. 46–49. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47602-5_10
13. Michel, F., Djimenou, L., Zucker, C.F., Montagnat, J.: xR2RML: non-relational databases to RDF mapping language (2014). https://hal.inria.fr/hal-01066663v1/document
14. O'Connor, M.J., Halaschek-Wiener, C., Musen, M.A.: M^2: a language for mapping spreadsheets to OWL. In: OWLED. CEUR Workshop Proceedings, vol. 614 (2010)
15. Walsh, N., Snelson, J., Coleman, A.: XQuery and XPath Data Model 3.1 (2017). https://www.w3.org/TR/xpath-datamodel-31/