D2RML: Integrating Heterogeneous Data and Web Services into Custom RDF Graphs

Alexandros Chortaras
National Technical University of Athens
Athens, Greece
achort@cs.ntua.gr

Giorgos Stamou
National Technical University of Athens
Athens, Greece
gstam@cs.ntua.gr

ABSTRACT

In this paper, we present the D2RML Data-to-RDF Mapping Language, as an extension of the R2RML mapping language, which significantly enhances its abilities to collect data from diverse data sources and transform them into custom RDF graphs. The definition of D2RML is based on a simple formal abstract data model, which is needed to clearly define its semantics, given the diverse types of data representation standards used in practice. D2RML allows web service-based data transformations, simple data manipulation and filtering, and conditional maps, so as to improve the selectivity of RDF mapping rules and facilitate the generation of higher quality RDF data stores, through a lightweight, easy to write and modify specification.

CCS CONCEPTS

• **Information systems → Information integration**; **Web data description languages**; *Query languages*; *Web services*;

KEYWORDS

RDF mapping language, Data integration, Web service integration

ACM Reference Format:
Alexandros Chortaras and Giorgos Stamou. 2018. D2RML: Integrating Heterogeneous Data and Web Services into Custom RDF Graphs. In *Proceedings of Linked Data on the Web 2018 (LDOW2018).* ACM, New York, NY, USA, 10 pages.

1 INTRODUCTION

In the past years, a considerable amount of work has been done on developing methodologies for mapping relational databases to RDF graphs. Several approaches, mapping languages and systems have been proposed, including two W3C recommendations [1, 8]. This work has mainly been motivated by the need to integrate the huge amount of information contained in existing relational databases with the emerging Semantic Web, and make them part of the Linked Data cloud.

LDOW2018, April 2018, Lyon, France

Following the Linked Data growth, several research institutions and companies such as DBpedia[1], WordNet[2], OpenStreetmap[3], offer now access to their huge datastores through SPARQL endpoints or RESTful web services. Even more recently, the expansion of cloud computing and the exciting developments in the field of machine learning and the subsequent revival of interest in artificial intelligence applications has resulted in the emergence of cloud platforms and marketplaces that offer intelligent data analysis web services, often representing their output using Linked Open Data vocabularies and resources, such as DBedia Spotlight[4], Google's Cloud Natural Language[5] and Microsoft's Computer Vision API[6]. These services typically deliver data using some structured data exchange format (usually JSON or XML documents).

Thus, if until recently the question was how to integrate existing data with the Semantic Web, now part of the question is also how to use all these available data and diverse services in a coordinated and integrated manner to selectively pick and aggregate data into custom data stores to power new intelligent applications. In this respect, aggregating data into custom RDF data stores is of particular interest not only because they allow direct integration with the Linked Data cloud, but also because intelligence can be added on top of the data by including e.g. axiomatic knowledge in the form of OWL2 [20] axioms. As a matter of fact, recent work on efficient algorithms and methods for reasoning with tractable fragments of ontologies (e.g. [3], [21]) has allowed the development of practical systems that provide inferencing over semantic data.

In this environment, we propose D2RML, a generic Data-to-RDF Mapping Language, whose aim is to facilitate the generation of custom RDF data stores by selectively collecting and integrating data from diverse data sources and web services into as much as possible high quality RDF data stores. Our purpose is to provide a formal basis for defining transformation-oriented general Data-to-RDF mappings, as well as, while staying within the mapping language approach, to transfer as much as possible of the burden for generating such data stores in practice from writing code or using heavyweight data workflow solutions, to writing easy understandable and modifiable specifications.

The rest of the paper is organized as follows: In Section 2 we briefly discuss related work with emphasis on R2RML and RML, which are the starting points for our work. In Section 3 we define the simple theoretical data model that underlies D2RML. In Section 4 we describe how several widely used information sources can be cast onto our model, and in Section 5 we present the formal specification of D2RML. Section 6 presents an extensive realistic

[1] http://dbpedia.org/sparql/
[2] http://wordnet-rdf.princeton.edu/
[3] http://api.openstreetmap.org/
[4] http://www.dbpedia-spotlight.org/api/
[5] https://cloud.google.com/natural-language/
[6] https://azure.microsoft.com/en-us/services/cognitive-services/computer-vision/

use case that showcases the expressivity and practical usefulness of the proposed language, and Section 7 concludes the paper.

2 RELATED WORK

Several languages and systems have been proposed to map relational databases to RDF (RDB-to-RDF mapping languages). A comparative analysis is presented in [14], which determines fifteen desirable features (e.g. support for transformation functions, named graphs, integrity constraints) that such languages should have, and discusses how they are or are not supported by the several languages. Existing RDB-to-RDF mapping languages vary considerably in the flexibility they allow in defining mappings, from the rigid Direct Mapping [1] approach that automatically translates the data of a relational database into an RDF graph representation following the database schema, to the R2RML language [8] that allows the user to define custom views and mapping rules (expressed as RDF graphs), and satisfies most of the fifteen desirable features.

The development of mapping languages and practical systems for translating data sources other than relational databases to RDF graphs has also been attempted. Closer to the relational model are CSV/TSV documents and spreadsheets, which retain the tabular format. Tools for converting from these data sources include XLWrap [18], TaRQL[7], Vertere[8], and M^2 [22]. In all such tools, for each table row one or more RDF resources are generated, and for each column one or more RDF triples about the respective resources are generated. Other formats, such as XML, diverge considerably from tabular data owing to their hierarchical structure, and the systems that have been proposed to translate XML to RDF graphs rely on XSLT transformations (e.g. XML2RDF[9]), XPath (e.g. Tripliser[10]), XQuery (e.g. XSPARQL [2]) or on embedding within the XML documents links to transformation algorithms, typically XSLT transformations (GRDDL [6]). All such tools rely on syntactical transformations of parts of the XML structure to RDF triples. Another framework to assist the transformation of XML and JSON data sources is xCurator [13] which focus on delivering high-quality linked data. Apart from the above, there exist also tools, in the form of web services (e.g. The Datatank[11]) or parts of other infrastructures (e.g. Virtuoso Sponger[12]) that provide custom solutions to work with data from different formats and possibly construct RDF graphs out of them. These tools, however, are general data processing and transformation tools and not designed to directly support semantic mappings of general data to RDF triples.

To resolve the polymorphy of tools and focus on the semantic aspects of the Data-to-RDF mapping process, several works extend the W3C recommended R2RML language to support other data formats. These include KR2RML [23], xR2RML [19] and RML [9]. These proposals are a considerable advance with respect to custom system solutions, because they are based on an existing, clean, mapping-oriented standard, and allow backward compatibility, and in most cases extensibility. It should be noted, however, that simply extending the R2RML standard to support other data source types, does not necessarily carry on all its features into the other data types. E.g. select conditions and transformation functions are supporting implicitly by R2RML by relying on the expressivity of the SQL query language, but this is not fully portable in a straightforward extension to the case of XML or JSON documents.

2.1 R2RML and RML

R2RML works with logical tables (`rr:LogicalTable`), which may be either base tables or views (`rr:BaseTableOrView`) defined by specifying an appropriate table name (`rr:tableName`), or result sets (`rr:R2RMLView`) obtained by executing a query (`rr:sqlQuery`). Each logical table is mapped to RDF triples using one or more triples maps (`rr:TriplesMap`). A triples map is a complex rule that maps each row in the underlying logical table to several RDF triples. The rule has two parts: a subject map (`rr:SubjectMap`) that generates the subject of all RDF triples that will be generated from each row of the logical table, and several predicate-object maps (`rr:PredicateObjectMap`) that in turn consist of predicate maps (`rr:PredicateMap`) and object maps (`rr:ObjectMap`) or referencing object maps (`rr:RefObjectMap`). A predicate map determines predicates for the to-be generated RDF triples for the given subject, and the object maps their objects. A subject map may include several IRIs (`rr:class`) that will be used as objects to generate triples with the predicate `rdf:type` for the particular subjects. A subject map or predicate-object map may have also one or more graph maps (`rr:GraphMap`) associated with it, which specify the target graph of the resulting RDF triples. Referring object maps allow joining two different 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, and may contain (`rr:joinCondition`) a join condition (`rr:Join`) specified by a reference to a column name of the current and parent triples map (`rr:child` and `rr:parent`, respectively). The IRIs and literals that will be used as RDF triple subjects, predicates, objects, or RDF graph names may be either declared constants (`rr:constant`), or obtained from the underlying table, view or result set by specifying the desired column name (`rr:column`) that will act as value source, or generated through a string template (`rr:template`) to concatenate column values and custom strings. String templates offer only very rudimentary options to manipulate actual database values and generate custom IRIs and literals.

RML extends R2RML by allowing other sources (e.g. JSON or XML files) apart from logical tables (`rml:LogicalSource`), that may be used in an interlinked manner, by defining data iterators (`rml:iterator`) to split the data obtained from such sources into base elements on which each mapping rule will be applied, and by allowing particular references (`rml:reference`), in the form of subelement selectors within the base element, to define the value sources to be used for the generation of IRIs and literals. Both the iterators and the references depend on the underlying data source, and may be XPath queries, JSONPath queries, CSV column names or SPARQL return variable names. Their type is declared using the `rml:referenceFormulation` predicate.

With respect to the specification of the actual access to the data sources, R2RML leaves the issue to the implementation. The assumption is that each R2RML document applies to data from a unique database. In contrast, RML, which allows multiple sources

[7] https://github.com/tarql/tarql/ [8] https://github.com/knudmoeller/Vertere-RDF/
[9] http://www.gac-grid.de/project-products/Software/XML2RDF.html
[10] http://daverog.github.io/tripliser/ [11] http://thedatatank.com/
[12] http://vos.openlinksw.com/owiki/wiki/VOS/VirtSponger

and cross-references between the retrieved data, must include the data source descriptions within the RML document. To describe them, it suggests the use of some recommended or widely-used vocabularies such as DCAT[13], D2RQ[14], CSVW[15], Hydra[16], SPARQL-SD[17] to access files, relational databases, CSV/TSV files, web APIs and SPARQL endpoints, respectively. However, these vocabularies have been developed mainly for APIs and data sources to inform clients about their exact properties and services they offer, and not as a form of formulating requests to them. E.g. to retrieve data from a web API that paginates the results using next page access keys, knowledge on how to formulate each time the subsequent HTTP request is needed; this is not covered for example by Hydra. Similarly, a SPARQL-SD specification provides information about the supported SPARQL version, the default entailment regime, the default named graph, etc., which are not useful to a client, at the time of formulating a request.

3 DATA MODEL

In this section, we extend the table-based model underlying the R2RML language to support complex, non-tabular data, that can be obtained from various information sources (such as JSON or XML document returning sources). To do this we consider that instead of logical tables, RDF triples are generated from set tables. In the following we represent an RDF triple as a tuple $\langle s, p, o\rangle$, where s is the *subject*, p the *property* or *predicate* and o the *object*.

Definition 3.1. 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 over some domains. 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 as the i-th elements of $\mathcal{D}_1, \ldots, \mathcal{D}_m$, for $1 \le i \le k$, 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 $(1 \le i \le k)$ in a set row $\mathcal{D}$ by $\mathcal{D}[n_k]$. We also denote 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 $\mathcal{S}$ by $\mathcal{S}[n_k]$, which we call a *column* of $\mathcal{S}$. Let also $\mathrm{dom}(n)$ denote the domain of column n. It should be underlined, that for a particular set row $\mathcal{D}$ and the different possible names n_i, the several sets $\mathcal{D}[n_i]$ may have different numbers of values, 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 3.2. A *filter* $\mathcal{F}$ over a set table $\mathcal{S}$ of arity k is a tuple $\langle n, f\rangle$, where n is a column name and $f : \mathrm{dom}(n) \to \mathrm{dom}(n)$ 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 $\mathcal{F}(\mathcal{D})$. Clearly, f may be the identity function.

Definition 3.3. A *triples rule* $\mathcal{R}$ over a set table $\mathcal{S} = \langle N, \mathcal{T}\rangle$ is 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. The *implementation* of $\mathcal{R}$ is the set of RDF triples

$$\{(s, p, o) \mid s \in \mathcal{F}_s(\mathcal{D}),\ p \in \mathcal{F}_p(\mathcal{D}),\ o \in \mathcal{F}_o(\mathcal{D}),\ \mathcal{D} \in \mathcal{T}\}.$$

A set of triples rules over one or more set tables defines a *Data-to-RDF mapping*. Using the above simple model we can define Data-to-RDF mappings for any information sources that can give rise to one or more set tables. The triple store represented by a Data-to-RDF mapping is then the implementation of all its triples rules.

We consider an *information source* to be any online software system that can deliver structured data upon request. The information source may be a data repository (e.g. a relational database, an RDF store, an XML file stored in some directory) or an implementation of a service or an algorithm (e.g. a RESTful web service) that may process some input data and deliver some structured output. The *request*, in the form of a query (e.g. an SQL or SPARQL SELECT query) or message (e.g. an HTTP GET or POST request) in a format supported by the information source, includes all input data and parameters required by the information source to generate and deliver the output. The *reply*, or *effective data source*, is the output produced by the information source, upon processing the request. The reply may be delivered to the client in a native format (e.g. as an SQL result set), or in a generic document format (e.g. as a JSON or XML document).

To accommodate the several possible information sources in our model, we consider, as in RML, that the effective data source groups some set of autonomous elements (e.g. rows of an SQL result set, elements of a JSON array). The division of the reply in these autonomous elements is achieved through an *iterator*. Hence, an effective data source together with an iterator specifies a *logical array*, through whose items the iterator eventually iterates. Each item of a logical array may itself be a complex data structure (a new effective data source), so in order to extract from it lists of values to construct set rows and use them as subjects, predicates and objects of RDF triples, we need some *selectors*. Thus, the role of the selectors is to transform a logical array into a set table.

Definition 3.4. The triple $\mathcal{A} = \langle \mathcal{I}, t, \mathcal{L}\rangle$, where $\mathcal{I}$ is a information source and request specification, t an iterator specification, and $\mathcal{L}$ a set of selectors, is a *data acquisition pipeline*.

It follows that each data acquisition pipeline $\mathcal{A}$ gives rise to a unique set table $\mathcal{S}_\mathcal{A}$. A data acquisition pipeline may be parametric, in the sense that the information source or request specification may contain parameters. Given a non-parametric data acquisition pipeline $\mathcal{A}$, 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 $\mathcal{S}_\mathcal{A}$. We call such a parametric data acquisition pipeline a *transformation* of $\mathcal{A}$.

Definition 3.5. A series of data acquisition pipelines $\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_l$, 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, which is $\mathcal{S}_{\mathcal{A}_0}$ extended by columns contributed by transformations $\mathcal{A}_1, \ldots, \mathcal{A}_l$. A trivial set table specification consists only of the primary data acquisition pipeline $\mathcal{A}_0$. Each transformation in a set table

[13] https://www.w3.org/TR/vocab-dcat/
[14] http://d2rq.org/d2rq-language
[15] https://www.w3.org/TR/tabular-metadata/
[16] https://www.hydra-cg.com/spec/latest/core/
[17] https://www.w3.org/TR/sparql11-service-description/

specification is realized as a series of requests to the respective 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. In particular, to evaluate a set table specification, we must evaluate serially the data acquisition pipelines, extending at each step the previously obtained set table: 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 (by merging the respective column values of each row) we obtain a new set row that is appended to $\mathcal{D}$. Doing this for all set rows $\mathcal{D}$ results in $\mathcal{S}_{\mathcal{A}_0\mathcal{A}_1}$. By applying this process iteratively, eventually $\mathcal{S}_{\mathcal{A}_0}$ is extended with additional columns 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 $\mathcal{S}_{\mathcal{A}_{i+1}}(\mathcal{D}_1), \ldots, \mathcal{S}_{\mathcal{A}_{i+1}}(\mathcal{D}_m)$. 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 column names, say $\hat{n}_1, \ldots, \hat{n}_{k'}$. Thus $\mathcal{S}_{\mathcal{A}_0\ldots\mathcal{A}_{i+1}} = \langle N, \mathcal{T}\rangle$, where $N = \langle n_1, \ldots, n_k, \hat{n}_1, \ldots, \hat{n}_{k'}\rangle$, $\mathcal{T} = [\mathcal{D}'_1, \ldots, \mathcal{D}'_m]$, $\mathcal{D}'_j = [\mathcal{D}_j[n_1], \ldots, \mathcal{D}_j[n_k], \hat{\mathcal{D}}_{j1}, \ldots, \hat{\mathcal{D}}_{jk'}]$ for $1 \le j \le m$, and $\hat{\mathcal{D}}'_{jl} = \bigcup \mathcal{S}_{\mathcal{A}_{i+1}}(\mathcal{D}_j)[\hat{n}_l]$ for $1 \le l \le k'$.

The row flattening step is intentional: $\mathcal{S}_{\mathcal{A}_0}$ provides the original data that we want to extend through transformations, ie. by appending new columns containing new properties of that data. Since, as mentioned above, all values contained in a particular row and column of $\mathcal{S}_{\mathcal{A}}$ are equivalent with respect to the values in the sets of the other columns of the current row, the flattening behaviour maintains this relationship between values, without introducing non-desired hierarchical dependencies. Finally, the primary data acquisition pipeline may be itself parametric. In this case, the evaluation is done exactly as described above, but the set rows generated by $\mathcal{S}_{\mathcal{A}_0}$ are not appended to the set table on which it depends, but initiate a new set table.

4 INFORMATION SOURCES AND REPLIES

We now study how several information and effective data sources used in real applications can be accommodated by our model. We discuss relational databases, RESTful web services, JSON, XML, CSV/TSV documents, and SPARQL endpoints.

4.1 Relational Databases

In relational databases data is organized into one or more tables (or relations) of columns (or attributes) and rows (or tuples). Each table column has a name. Data are retrieved by issuing an SQL SELECT query and the results are packed as a result set, which is essentially a row-by-row iteratable table along with its metadata. Because relational database management systems (RDBMS) use native formats to implement the data stores and the result formats, communications with RDBMSs' are done using special protocols (such as ODBC, JDBC) to implement clients for particular RDMBSs'. Practical access requires several parameters to be specified (e.g. server location, database name, user name, password, access driver), which are usually grouped in the so-called connection string and are programming language implementation dependent. There is no standard for representing connection strings in RDF form. D2RQ Mapping Language [7] allows a JDBC-dependent RDF definition of connection strings and is used by RML to specify RDBMS connectivity.

An implementation provided with a RDBMS connection specification can connect to the particular RDBMS, pose an SQL SELECT query q that specifies attributes $n_1, \ldots, n_k$ in the SELECT statement for the returned columns, and obtain as result a list of rows $[\langle v_{11}, \ldots, v_{1k}\rangle, \ldots, \langle v_{n1}, \ldots, v_{nk}\rangle]$. Using, a trivial row iterator and column names $n_1, \ldots, n_k$ as selectors, the results of q can be converted to the following set table: $\langle\langle n_1, \ldots, n_k\rangle, [\langle\{v_{11}\}, \ldots, \{v_{1k}\}\rangle, \ldots, \langle\{v_{n1}\}, \ldots, \{v_{nk}\}\rangle]\rangle$

Table 1: Information sources, requests and replies

Information Source	*Request*	*Effective Data Source*
RDBMS	SQL SELECT Query	SQL Result Set
SPARQL Endpoint	SPARQL SELECT Query and RDF graph IRIs via HTTP Message	SPARQL Result Set via HTTP Message
RESTful Web Service	HTTP GET/POST Request	JSON/XML/CSV/TSV Document
JSON/XML/CSV/TSV Document	HTTP GET Request	JSON/XML/CSV/TSV Document

Table 2: Effective Data Sources, iterators and selectors

Effective Data Source	*Iterator*	*Selector*
SQL Result Set	Row Iterator	Column name
SPARQL Result Set	Row Iterator	Variable name
JSON Document	JSONPath query	Flat JSONPath query
XML Document	XPath query	Flat XPath query
CSV/TSV Document	Row Iterator	Column name

4.2 RESTful Web Services

RESTful web services are services based on the REST principles [11], and are usually implemented using the HTTP protocol. Typically, a data retrieving RESTful service accepts an HTTP request and delivers the result in a self-descriptive text message (e.g. an HTML, XML, JSON, plain text). Here we are interested in structured reply services, i.e. services whose reply is in one of the XML, JSON or CSV/TSV formats. To access a RESTful web service, the elements of the appropriate HTTP request have to be specified. These include the method (GET or POST), the URI (including the query string in the case of a GET message), any headers, and the body (for passing parameters in the case of a POST message). All these can be specified in RDF using the W3C's Working Group Notes 'HTTP Vocabulary in RDF 1.0' [16] and 'Representing Content in RDF 1.0' [17]. Thus, we can assume that an HTTP client that can consume an HTTP Vocabulary and Representing Content in RDF 1.0 description to create an HTTP request, can use a RESTful web service and obtain as result a structured document. Although not strictly qualifying as RESTful web services, we include in this category also URIs that simply deliver structured documents (e.g.

URIs to static JSON/XML files), since the communication is performed in exactly in the same way through HTTP messages.

A practical consideration usually related with some RESTful web services, is that the APIs that implement the services, to avoid extremely long replies, perform pagination of the results and do not return the full set of results as one document, but as a series of smaller documents: in most cases, each returned document contains some keys that can be used by the client in the subsequent request to instruct the server to return the next set of results. The pagination schema may get non-trivial, as in the case of MediaWiki[18].

4.3 SPARQL Endpoints

SPARQL endpoints are URIs at which a SPARQL Protocol service listens [10]. SPARQL Protocol is built on top of HTTP and as such it can be treated as a RESTful web service. However, since special SPARQL Protocol clients, in the form of APIs, exist (e.g. Apache Jena[19]) that hide from the user the cumbersome details of building and decoding the necessary HTTP request and reply messages it is useful to provide support also for this type of interaction. The situation is similar to the RDBMS case: The request is a SPARQL SELECT (possibly along with some default and named RDF graph IRIs) instead of an SQL SELECT query, and the effective data source is a result set, whose column names are the return variable names specified in the SPARQL query. Thus, the translation of the reply to a set table is done in exactly the same way. The only essential thing that changes is the specification of the access to the SPARQL endpoint for which a single URI is enough.

4.4 JSON Documents

A JSON document [15] may be modeled as a JSON tree [4]. A JSON tree is an edge-labeled tree, whose root represents the entire document. A node may have either string- or integer-labeled children, but not both. A node with string-labeled outgoing edges represents a set of JSON key-value pairs: the edge label is the key and the edge destination the corresponding value. A node with integer-labeled outgoing edges represents an array: the edge label is the array index and the edge destination the corresponding value. Value nodes, are either leaf nodes having a string or integer label, or JSON trees.

In the absence of an official standard, to select values from a JSON document that meet specific conditions, in practice the JSONPath [12] specification is used, which is inspired by XPath. JSONPath queries select nodes of a JSON tree that meet a certain path condition, and group them into a JSON array, which is the result of the query. Since a JSON array is a JSON document, the result of a JSONPath query is always a JSON document. We will say that a JSONPath query is *flat* if the result JSON tree has depth 1, ie. is an array of simple values.

Hence, an iterator for a JSON tree $\mathcal{T}$ is any relevant JSONPath query q, which splits $\mathcal{T}$ into a logical array of smaller JSON trees $\mathcal{T}_1, \ldots, \mathcal{T}_n$, and the selectors are flat JSONPath queries $q_1, \ldots q_k$ that are executed over each $\mathcal{T}_1, \ldots, \mathcal{T}_n$ to deliver a set table from the underlying logical array. Thus $\mathcal{T}$, after applying iterator q and selectors $q_1, \ldots q_k$, yields 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 the set of values contained in the array that results from applying q_j on $\mathcal{T}_i$.

[18] https://www.mediawiki.org/wiki/API:Query [19] https://jena.apache.org/

4.5 XML Documents

An XML document may also be modeled using a tree [5], however its structure differs from a JSON tree. The core part of an XML document is represented in the tree by element, attribute and text nodes. Each element node corresponds to an element of the XML document and has a name (the element name) and children that are all the enclosed elements. It may also have as child a text node, that holds in its string value the characters in the CDATA section of the element. Each element node may have associated with it also a set of attribute nodes that represent the attributes of the element, which, however, are not considered to be children of the element node. Each attribute node has a name (the attribute name) and a string value that holds the respective attribute value. Relying on this model, the XPath language allows to select particular nodes from the tree that meet certain conditions. Unlike in the case of JSON, the result is not itself an XML document, but a set of the nodes that match the query criteria. We will say that an XPath query is *flat* if the result contains only text or attribute nodes.

Hence, we can consider as iterator for an XML document tree $\mathcal{T}$ any relevant non-flat XPath query q that splits $\mathcal{T}$ into a logical array of nodes $N_1, \ldots, N_n$. Since the query is non-flat, these nodes are element nodes, and can be treated as smaller XML document trees $\mathcal{T}_1, \ldots, \mathcal{T}_n$. The selectors are then flat XPath queries $q_1, \ldots q_k$ that are executed over each one of these smaller XML documents. Thus, $\mathcal{T}$ after applying iterator q and selectors $q_1, \ldots q_k$ yields 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} are the sting values of the text or attribute nodes in the node set obtained by applying q_j on $\mathcal{T}_i$.

4.6 CSV/TSV Documents

CSV/TSV documents are textual representations of tabular data. Each line represents a data row, expect possibly from the first row that contains the names of the columns. Hence, the situation is similar to the RDBMS case, with no need of a query to be specified. The name tuple consists of the names of the columns in the file (or of their numbering) and the row sets of the actual rows of the table. The only thing the needs to be specified are the formatting details (eg. delimiter, escape separator, quote character).

5 D2RML SPECIFICATION

D2RML draws significantly from R2RML and RML, and follows the same simple syntactical strategy for defining mappings: Triples maps, which consist of a subject map and several predicate object maps. From RML it adopts and appropriately extends the way to define the interaction with information sources through requests, iterators and selectors. Moreover, it significantly extends the expressive capabilities of R2RML and RML by allowing transformations, conditional statements, and custom IRI generation functions.

For its semantics, D2RML relies on the data model described in Section 3. Each triples map is essentially a set table specification of Def. 3.3 and a specification of a set of triple rules of Def. 3.5 with the same subject filter over the common underlying set table. The information source, request and iterator of the original data acquisition pipeline is directly provided in the triples map definition. Any transformations to be added to the set table specification

Table 3: Namespaces used in D2RML documents

Prefix	IRI
rr	http://www.w3.org/ns/r2rml#
dr	http://islab.ntua.gr/ns/d2rml#
op	http://islab.ntua.gr/ns/d2rml-op#
is	http://islab.ntua.gr/ns/d2rml-is#
http	http://www.w3.org/2011/http#
cnt	http://www.w3.org/2011/content#

are declared in the order of their application. The selectors are implicitly declared in the subject, predicate, object and graph maps. Several triples map are allowed to coexist in the a D2RML document, in which case several distinct set tables are generated.

We define D2RML using a BNF-like notation. Terminal symbols are written in `monospace`, and non-terminals in *italics*. Non-terminals within angle brackets represent RDF nodes. Parenthesis specify the scope of alternatives (separated by |) and of the standard quantifiers ?, *, and +. Terminal symbols not explicitly defined in the specification are written in SMALLCAPS. The namespaces are defined in Table 3. D2RML is compatible with R2RML, but not fully compatible with RML, so it does not directly extend its namespace.

5.1 Triples Maps

A triples map is defined as in R2RML and RML, but tabular data providing information sources are clearly distinguished from non-tabular by using `rr:logicalTable` for tabular data providing information sources, and `dr:logicalSource` for the rest.

TriplesMap ← a `rr:TriplesMap`
`rr:logicalTable` ⟨*LogicalTable*⟩ | `dr:logicalSource` ⟨*LogicalSource*⟩
(`dr:transformations` (⟨*Transformation*⟩+))?
`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 | LITERAL))+
(`rr:graphMap` ⟨*GraphMap*⟩ | `rr:graph` IRI)*

5.2 Logical Tables and Logical Sources

The *LogicalTable* and *LogicalSource* nodes provide details about the primary information source used to generate the set table. In the case of query supporting information sources (such as RDBMSs' and SPARQL endpoints), for backward compatibility with R2RML, they contain also the query-relevant details of the request that should be sent to the information source. The `is:parameters` predicate may be used to declare parameter names in queries that participate in parametric data acquisition pipelines. For other information sources (such as RESTful web services), the request, and any parameters, are included in the *InformationSource* specification itself. For non-tabular data providing information sources, *LogicalSource* contains also the definition of the iterator (`dr:iterator` and `dr:referenceFormulation`) that will be used to split the effective data source into a logical array. Since the effective data source format is fixed, the object of `dr:referenceFormulation` determines also the form of all selectors that will be applied on the particular effective data source.

LogicalTable ← a `rr:LogicalTable`
`dr:source` ⟨*InformationSource*⟩
SQLTable | *SPARQLTable* | *CSVTable*
(`is:parameters` (⟨*DataVariable*⟩+))?

LogicalSource ← a `dr:LogicalSource`
`dr:source` ⟨*InformationSource*⟩
`dr:iterator` LITERAL
`dr:referenceFormulation` IRI

SQLTable ← a `rr:BaseTableOrView`
`rr:tableName` LITERAL
| a `rr:R2RMLView`
`rr:sqlQuery` LITERAL
(`rr:sqlVersion` IRI)?

SPARQLTable ← a `dr:SPARQLTable`
`dr:sparqlQuery` LITERAL
(`dr:sparqlVersion` IRI)?
(`dr:defaultGraph` IRI)*
(`dr:namedGraph` IRI)*

CSVTable ← a `dr:TextTable`
`dr:delimiter` LITERAL
`dr:headerline` BOOLEAN
(`dr:quoteCharacter` LITERAL)?
(`dr:commentCharacter` LITERAL)?
(`dr:escapeCharacter` LITERAL)?
(`dr:recordSeparator` LITERAL)?

5.3 Information Sources

The version of D2RML presented here provides definitions for implementing data acquisition pipelines involving RDBMSs', RESTful web services and SPARQL endpoints. Extensions for additional sources are expected in subsequent versions.

InformationSource ← *RDMSSource* | *SPARQLService* | *HTTPSource*

RDMSSource ← a `is:RDBMSSource`
`is:rdbms` IRI
`is:location` LITERAL
(`is:username` LITERAL)?
(`is:password` LITERAL)?
(`is:database` LITERAL)?

SPARQLService ← a `is:SPARQLService`
`is:uri` URI

HTTPSource ← a `is:HTTPSource`
`is:request` ⟨*HTTPRequest*⟩ | `is:uri` URI
(`is:parameters` (⟨*Parameter*⟩+))?

Parameter ← *DataVariable* | *SimpleKeyRequestIterator*

DataVariable ← a `is:DataVariable`
`is:name` LITERAL

SimpleKeyRequestIterator ← a `is:SimpleKeyRequestIterator`
`is:name` LITERAL
`dr:reference` LITERAL
`dr:referenceFormulation` LITERAL
`is:initialValue` LITERAL

In an *RDBMSSource*, `is:rdbms` determines the specific RBMBS (eg. MySQL, PostgreSQL). An *HTTPSource* is specified in terms of a *HTTPRequest* which should be a `http:Request` and specify the details of the HTTP message to be sent. An *HTTPSource* may contain

parameters in case the web service is part of a parametric data acquisition pipeline, or it paginates the results. Data parameters are identified by a name (`is:name`). For paginated results, the above specification allows, as an example, iterated requests through a request iterator that should be part eg. of the web service URI and whose values, apart from the initial value (`is:initialValue`) are extracted each time from the previous reply using a selector. Extensions are possible to support additional pagination policies.

5.4 Transformations

A triples map definition may include a list of transformations that should be applied in the declared order to the set table derived from the primary information source. Since a transformation is itself a parametric data acquisition pipeline, its definition includes the specification of an *InformationSource* through a `rr:logicalTable` or `dr:logicalSource` and one or more *ParameterBinding*s. A *ParameterBinding* 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.

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

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

5.5 Term Maps and Conditions

The definitions of term maps (i.e. of subject maps, graph maps, predicate maps and object map) follow the R2RML specification with the addition of filters.

SubjectMap ← a `rr:SubjectMap`
IRIRef | *BlankNodeRef*
(*SubjectBody CaseSubjectBody**) | *CaseSubjectBody+*

PredicateMap ← a `rr:PredicateMap`
(*PredicateBody CasePredBody**) | *CasePredBody+*

ObjectMap ← a `rr:ObjectMap`
(*ObjectBody CaseObjectBody**) | *CaseObjectBody+*

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

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

PredicateBody ← *IRIRef*
(`dr:condition` ⟨*Condition*⟩)?

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

GraphBody ← *IRIRef*
(`dr:condition` ⟨*Condition*⟩)?

CaseSubjectBody ← `dr:cases` (⟨*SubectBody*⟩+)

CasePredBody ← `dr:cases` (⟨*PredicateBody*⟩+)

CaseObjectBody ← `dr:cases` (⟨*ObjectBody*⟩+)

CaseGraphBody ← `dr:cases` (⟨*GraphBody*⟩+)

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

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

JoinCondition ← a `rr:Join`
`rr:child` LITERAL
`rr:parent` LITERAL

To support filters, a *SubjectMap*, *GraphMap*, *PredicateMap* or *ObjectMap* may contain a condition (`dr:condition`) and/or a case statement (`dr:cases`). If a term map contains a condition statement, this will be evaluated and the corresponding subject, graph, predicate or object value will be included in the respective value set only if the condition evaluates to true. Each condition statement should first specify the *actual* value on which it will operate (as a *ValueRef*), and may include several tests which will be jointly evaluated using the boolean operator specified by `dr:booleanOperator` (`op:and` or `op:or`). Each test is specified either through an OPERATOR and a LITERAL which define a constant value with which the actual value will be compared using OPERATOR, or as a nested condition. An OPERATOR is a common operator such as `op:eq`, `op:le`, `op:leq`, `op:ge`, `op:geq`, `op:matches`, etc. The type of the operation (eg. number or string comparison) depends on the XSD type of the LITERAL provided as operand. If a nested condition does not specify a value reference, it inherits it from the enclosing condition.

The case statement offers alternatives for realizing a term map: It contains a list of alternative term maps, each along with a condition. If the condition evaluates to true the term map is realized, otherwise control flows to the next case.

Finally, a referring object map (*RefObjectMap*) may be defined by a *ParameterBinding*, instead of by a R2RML `JoinCondition`. This is how set table specifications with parametric primary data acquisition pipelines are defined: the parametric set table specification corresponds to the parent triples map of *RefObjectMap*, and the *ParameterBinding* provides the parameters values.

5.6 IRIs, Literals and Blank Nodes

In R2RML, RDF terms are generated using the `rr:constant`, the `rr:column` and `rr:template` predicates; to these, RML adds the `rml:reference` option. D2RML follows the same strategy, but to account for values coming from transformations, RDF terms are generated through value references (*ValueRef*s), specified by two distinct components: a compulsory `rr:column`, `rr:template` or `dr:reference`, and an optional `dr:transformationReference` to specify the transformation that provides the logical array for the respective `rr:column`, `rr:template` or `dr:reference`. If missing, the primary logical array is assumed.

Although `rr:template` allows some minimal flexibility in defining custom IRIs or literals, the overall mechanism is quite restrictive, since no simple transformations (e.g. replace particular characters etc.) can be applied on the values obtained from the underlying set tables. D2RML addresses this issue by allowing simple functions to be applied on the raw values obtained from effective data sources. Thus, a *ValueRef* may include definitions of one or more

defined columns (dr:definedColumns) that are constructed by applying a series of functional transformations on particular set table column values and may be used in a rr:column or rr:template. A defined column should declare the new column name dr:name it will be referred by, the function (dr:function) that will generate the custom values (eg. op:regex, op:replace), and a list of arguments, in the form of one or more dr:parameterBindings. The parameter names should be provided by the function definition.

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

LiteralRef ← rr:constant LITERAL | *ValueRef*
(rr:termType rr:Literal)?
(rr:language LITERAL | rr:datatype IRI)?

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

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

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

6 USE CASE

In this section, we present a realistic use case for D2RML, involving true data and readily available web services and data repositories. The aim is to extract an extensive set of textual or URI features for a set of cultural items, in order to subsequently use them to perform several tasks such as clustering and similarity ranking. We assume that we want to extract features in several ways (e.g. directly from the metadata, from applying named entity extraction, image analysis, etc.), and that we want to keep information about the source of each feature so that we can use them selectively to test how they affect the clustering or similarity algorithm performance.

As primary information source of cultural items we use Europeana Collections[20], in particular the collection provided by TopFoto[21], which consists of 60,882 black and white images of the 1930s, along with their metadata. This collection can be obtained through the Europeana API. The D2RML specification for getting the effective data source for this collection is the following:

```
<#EuropeanaAPI>
   a is:HTTPSource ;
   is:request [
      http:absoluteURI "http://www.europeana.eu/api/v2/search.json?
            wskey=A*******W&rows=20&cursor={@@cursor@@}&profile=rich&
            query=europeana_collectionName%3A%222024904_Ag_EU_
            EuropeanaPhotography_TopFoto_1013%22" ;
      http:methodName "GET" ;
   ] ;
   is:parameters ( [ a is:SimpleKeyRequestIterator ;
                     is:name "cursor" ;
                     is:initialValue "*" ;
                     dr:reference "$.nextCursor" ;
                     dr:referenceFormulation is:JSONPath ; ] ) .
```

The specification includes a is:SimpleKeyRequestIterator as parameter, because the API returns the results in pages, and each page contains a key to accessing the next page (nextCursor). An extract from the response obtained from executing the above is the following JSON document, which contains a list of items modeled using the Europeana Data Model (EDM):

```
{
  "nextCursor": "AoE/GC8yMDI0OTA0L3Bob3Rv****=",
  "items": [
    {
      "id": "/2024904/photography_ProvidedCHO_TopFoto_co_uk_EU061905",
      "dcDescription": [
          "Former chief inspector Berrett decorated by the king.\n
            Former chief detective inspector James Berrett of
            Scotland Yard was decorated by the King at the royal
            invesititure at Buckingham Palace. "
      ],
      "edmIsShownBy": [
        "http://www.topfoto.co.uk/imageflows/imagepreview/f=EU061905"
      ],
      "edmConcept": [
        "http://bib.arts.kuleuven.be/photoVocabulary/12003",
        "http://data.europeana.eu/concept/base/1711"
      ],
      "type": "IMAGE"
    },  ...
  ]
}
```

Most fields are self-explanatory. edmConcept contains a list of Open Linked Data resources that have been associated to each item by the provider to characterize the respective item content. To generate RDF triples for this information, as well as for the type of each item, we define the following D2RML document:

```
<#EuropeanaMapping>
   dr:logicalSource [ dr:source <#EuropeanaAPI> ;
                      dr:iterator "$.items" ;
                      dr:referenceFormulation is:JSONPath ; ] ;
   rr:subjectMap [
      dr:definedColumns ( [
         dr:name "SID" ;
         dr:function op:extractMatch ;
         dr:parameterBinding [ dr:parameter "input" ;
                               dr:reference "$.id" ; ] ;
         dr:parameterBinding [ dr:parameter "regex" ;
                               rr:constant "^.*_(.*)$" ; ] ;
      ] ) ;
      rr:template  "http://islab.ntua.gr/resources/tp/{SID}" ;
      dr:cases ( [
        rr:class <http://islab.ntua.gr/ml/Image> ;
        dr:condition [ dr:reference "$.type" ;
                       op:eq "IMAGE"^^xsd:string ; ] ;
      ] [
        rr:class <http://islab.ntua.gr/ml/Other> ;
      ] ) ;
   ] ;
   rr:predicateObjectMap  [
      rr:predicate  <http://islab.ntua.gr/ml/edmConcept> ;
      rr:objectMap  [ dr:reference "$.edmConcept" ;
                      rr:termType  rr:IRI ; ] ;
   ] .
```

Note the use of a defined column to construct custom RDF subject IRIs. The particular defined column applies the regular expression ^.*_(.*)$ on the id field of each item and uses the value of the first capturing group, named SID. The above specification generates the following RDF triples for the first item:

```
<http://islab.ntua.gr/resources/tp/EU061905>
         <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
                  <http://islab.ntua.gr/ml/Image> .
<http://islab.ntua.gr/resources/tp/EU061905>
         <http://islab.ntua.gr/ml/edmConcept>
                 <http://bib.arts.kuleuven.be/photoVocabulary/12003> .
<http://islab.ntua.gr/resources/tp/EU061905>
         <http://islab.ntua.gr/ml/edmConcept>
```

[20] https://www.europeana.eu/portal/en [21] http://www.topfoto.co.uk/

```
                <http://data.europeana.eu/concept/base/1711> .
```

Since we want to extract several features, we can invoke services to the analyze metadata. An option is to use DBpedia Spotlight to extract named entities from the textual descriptions. To do this, we need a transformation that takes the description of each item (dcDescription) and invokes DBpedia Spotlight on it. We first define the relevant information source:

```
<#DBpediaSpotlightAPI>
   a is:HTTPSource ;
   is:request [
      http:absoluteURI "http://model.dbpedia-spotlight.org/en/
            annotate?text={@@text@@}&confidence=0.5&support=0&
            spotter=Default&disambiguator=Default&policy=whitelist&
            types=&sparql=" ;
      http:methodName "GET" ;
      http:headers ( [ http:fieldName "Accept" ;
                       http:fieldValue "application/xml" ; ] ) ;
   ] ;
   is:parameters ( [ a is:DataVariable ;
                     is:name "text" ;    ] ) .
```

The respective effective data source has the following XML format

```
<Annotation text="Former chief inspector Berrett decorated by the king.
     \nFormer chief detective inspector James Berrett of Scotland Yard
     was decorated by the King at the royal invesititure at Buckingham
      Palace." confidence="0.5" support="0"
      types="" sparql="" policy="whitelist">
   <Resources>
      <Resource URI="http://dbpedia.org/resource/Inspector"
         support="972" types="" surfaceForm="detective inspector"
         offset="69" similarityScore="1.0"
         percentageOfSecondRank="0.0"/>
      ...
   </Resources>
</Annotation>
```

which includes all detected named entities (Resource) as DBpedia resources (URI). We next define the transformation

```
<#SpotlightTransformation>
   dr:logicalSource [ dr:source <#DBpediaSpotlightAPI> ;
                      dr:iterator "/Annotation/Resources/Resource" ;
                      dr:referenceFormulation is:XPath ; ] ;
   dr:parameterBinding [ dr:parameter "text" ;
                         dr:reference "$.dcDescription" ; ] .
```

and add the transformation and a new predicate object map to the <#EuropeanaMapping> triples map:

```
<#EuropeanaMapping>
   ...
   dr:transformations ( <#SpotlightTransformation> ) ;
   rr:predicateObjectMap  [
      rr:predicate  <http://islab.ntua.gr/ml/DBpediaResource> ;
      rr:objectMap  [
         dr:reference "/Resource/@URI" ;
         dr:transformationReference <#SpotlightTransformation> ;
         rr:termType  rr:IRI ;
      ] ;
   ] .
```

When executed, it generates the following additional triples:

```
<http://islab.ntua.gr/resources/tp/EU061905>
        <http://islab.ntua.gr/ml/DBpediaResource>
                <http://dbpedia.org/resource/Inspector> .
<http://islab.ntua.gr/resources/tp/EU061905>
        <http://islab.ntua.gr/ml/DBpediaResource>
                <http://dbpedia.org/resource/James_Berrett> .
```

We further extend the set of features by using DBpedia ontology to get the types of the retrieved DBpedia resources. For this we need a second transformation, dependent on the first one, that consults a DBpedia endpoint. The information source definition is

```
<#DBpediaSPARQLService>
   a is:SPARQLService ;
   is:uri "http://dbpedia.org/sparql" .
```

and the transformation

```
<#DBpediaTransformation>
   dr:logicalSource [
      dr:source <#DBpediaSPARQLService> ;
      dr:query "SELECT ?dbpediatype WHERE
                  { <{@@resource@@}> a ?dbpediatype }" ;
      is:parameters ( [ a is:DataVariable;
                        is:name "resource" ; ] ) ;
   ] ;
   dr:parameterBinding [
      dr:parameter "resource" ;
      dr:reference "/Resource/@URI" ;
      dr:transformationReference <#SpotlightTransformation> ;
   ] .
```

Finally, we modify <#EuropeanaMapping> to add the new transformation and a add new predicate object map:

```
<#EuropeanaMapping>
   ...
   dr:transformations ( <#SpotlightTransformation>
                        <#DBpediaTransformation> ) ;
   rr:predicateObjectMap [
      rr:predicate <http://islab.ntua.gr/ml/DBpediaConcept> ;
      rr:objectMap [
         rr:column "dbpediatype" ;
         dr:transformationReference <#DBpediaTransformation> ;
         rr:termType rr:IRI ;
         dr:condition [
            op:matches "http://dbpedia\\.org/ontology/.*" ;
         ] ;
      ] ;
   ] .
```

Note that the mapping includes a conditional statement. It has been included because the query returns not only DBpedia ontology concepts as types, but also FOAF, YAGO, Schema, Wikidata, and other resources, which we do not want to include in our results. Eventually, this map generates the following RDF triples:

```
<http://islab.ntua.gr/resources/tp/EU061905>
        <http://islab.ntua.gr/ml/DBpediaConcept>
                <http://dbpedia.org/ontology/Athlete> .
<http://islab.ntua.gr/resources/tp/EU061905>
        <http://islab.ntua.gr/ml/DBpediaConcept>
                <http://dbpedia.org/ontology/Person> .
<http://islab.ntua.gr/resources/tp/EU061905>
        <http://islab.ntua.gr/ml/DBpediaConcept>
                <http://dbpedia.org/ontology/Agent> .
```

Finally, we can use computer vision technologies to analyze the image of each item (the URI is provided by the edmIsShownBy field in the document returned by the Europeana API) to detect objects that appear in it. To this end we use Microsoft's Computer Vision API, that is offered as a RESTful web service. Thus, we add a new information source including the required request parameters

```
<#ComputerVisionAPI>
   a is:HTTPSource
   is:request [
     http:absoluteURI "https://westcentralus.api.cognitive.microsoft.
           com/vision/v1.0/analyze?visualFeatures=Categories&
           language=en" ;
      http:methodName "POST" ;
      http:headers ( [ http:fieldName "Content-Type" ;
                       http:fieldValue "application/json" ; ]
            [ http:fieldName "Ocp-Apim-Subscription-Key" ;
              http:fieldValue "3*************************b" ; ] ) ;
      http:body [ a cnt:ContentAsText ;
                  cnt:chars "{\"url\" : \"{@@imageURL@@}\" }" ; ] ;
   ] ;
   is:parameters ( [ a is:DataVariable ;
                     is:name "imageURL" ; ] ) .
```

which produces the following JSON-formatted effective data source:

```
{
   "categories": [ {
      "name": "people_group",
      "score": 0.578125
   } ],
   "requestId": "3b28df72-abf5-488c-86f4-b2c6a7eb9703"
}
```

Based on this, we define the transformation

```
<#ImageTransformation>
   dr:logicalSource [ dr:source <#ComputerVisionAPI> ;
                      dr:iterator "$.categories" ;
                      dr:referenceFormulation is:JSONPath ; ] ;
   dr:parameterBinding [ dr:parameter "imageURL" ;
                         dr:reference "$.edmIsShownBy" ; ] .
```

to generate a logical array from `categories` that contains the names of the detected objects, and modify `<#EuropeanaMapping>` by adding the new transformation and a new predicate object map:

```
<#EuropeanaMapping>
   dr:transformations ( <#SpotlightTransformation>
                  <#DBpediaTransformation> <#ImageTransformation> ) ;
   rr:predicateObjectMap  [
      rr:predicate <http://islab.ntua.gr/ml/ComputerVisionTerm> ;
      rr:objectMap [
         dr:reference "$.name" ;
         dr:transformationReference <#ImageTransformation> ;
         rr:termType rr:Literal ;
         dr:condition [
            dr:reference "$.score" ;
            dr:transformationReference <#ImageTransformation> ;
            op:geq "0.4"^^xsd:decimal ;
         ] ;
      ] ;
   ] ;
```

The above object map applies a filter in order to keep only objects that have been detected with relatively high confidence (`score`). Eventually, the above map adds the following RDF triple:

```
<http://islab.ntua.gr/resources/tp/EU061905>
        <http://islab.ntua.gr/ml/ComputerVisionTerm>
                 "people_group" .
```

The RDF triples generated by all the above predicate-object maps make up the desired RDF graph. In terms of performance, for executing the above D2RML document, our implementation of D2RML processor[22] took about 7 minutes per 100 Europeana items.

7 CONCLUSIONS

We presented D2RML, a Data-to-RDF mapping language, which based on an abstract data model, allows the orchestrated retrieval of data from several information sources, their transformation and extension using relevant web services, their filtering and manipulation using simple operations, and finally their mapping to RDF graphs. It combines the mapping approach of R2RML and RML with workflow approaches, by allowing the definition of easy to write and understand, homogenous views of the underlying data and services in a lightweight document. We developed D2RML on top of a formal abstract data model, so as to formally define its semantics and allow future extensions. We also presented a realistic use case, which demonstrates the capabilities of the proposed language in real settings, by delivering a unified and coordinated access to Linked Data data stores and other services in a clean specification without the need of code writing or heavy-weight solutions.

[22] Available as a web service at http://apps.islab.ntua.gr/d2rml/

ACKNOWLEDGEMENTS

We acknowledge support of this work by the project 'APOLLONIS' (MIS 5002738) which is 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] Marcelo Arenas, Alexandre Bertails, Eric Prud'hommeaux, and Juan Sequeda. 2012. A Direct Mapping of Relational Data to RDF. (2012). https://www.w3.org/TR/rdb-direct-mapping/
[2] Stefan Bischof, Stefan Decker, Thomas Krennwallner, Nuno Lopes, and Axel Polleres. 2012. Mapping between RDF and XML with XSPARQL. *J. Data Semantics* 1, 3 (2012), 147–185.
[3] Barry Bishop, Atanas Kiryakov, Damyan Ognyanoff, Ivan Peikov, Zdravko Tashev, and Ruslan Velkov. 2011. OWLIM: A family of scalable semantic repositories. *Semantic Web* 2, 1 (2011), 33–42.
[4] Pierre Bourhis, Juan L. Reutter, Fernando Suárez, and Domagoj Vrgoc. 2017. JSON: Data model, Query languages and Schema specification. In *PODS*. ACM, 123–135.
[5] James Clark and Steve DeRose. 2016. XML Path Language (XPath) Version 1.0. (2016). https://www.w3.org/TR/xpath/
[6] Dan Connolly. 2007. Gleaning Resource Descriptions from Dialects of Languages (GRDDL). (2007). https://www.w3.org/TR/grddl/
[7] Richard Cyganiak, Chris Bizer, Jörg Garbers, Oliver Maresch, and Christian Becker. 2012. The D2RQ Mapping Language. (2012). http://d2rq.org/d2rq-language
[8] Souripriya Das, Seema Sundara, and Richard Cyganiak. 2012. R2RML: RDB to RDF Mapping Language. (2012). https://www.w3.org/TR/r2rml/
[9] Anastasia Dimou, Miel Vander Sande, Pieter Colpaert, Ruben Verborgh, Erik Mannens, and Rik Van de Walle. 2014. RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In *LDOW (CEUR Workshop Proceedings)*, Vol. 1184. CEUR-WS.org.
[10] Lee Feigenbaum, Gregory Todd Williams, Kendall Grant Clark, and Elias Torres. 2013. SPARQL 1.1 Protocol. (2013). https://www.w3.org/TR/sparql11-protocol/
[11] Roy T. Fielding and Richard N. Taylor. 2000. Principled design of the modern Web architecture. In *ICSE*. ACM, 407–416.
[12] Stefan Gössner and Stephen Frank. 2007. JSONPath. (2007). http://goessner.net/articles/JsonPath/
[13] Oktie Hassanzadeh, Soheil Hassas Yeganeh, and Renée J. Miller. 2011. Linking Semistructured Data on the Web. In *WebDB*.
[14] Matthias Hert, Gerald Reif, and Harald C. Gall. 2011. A comparison of RDB-to-RDF mapping languages. In *I-SEMANTICS (ACM International Conference Proceeding Series)*. ACM, 25–32.
[15] Internet Engineering Task Force (IETF). 2014. The JavaScript Object Notation (JSON) Data Interchange Format. (2014). https://tools.ietf.org/html/rfc7159
[16] Johannes Koch, Carlos A Velasco, and Philip Ackermann. 2017. HTTP Vocabulary in RDF 1.0. (2017). https://www.w3.org/TR/HTTP-in-RDF10/
[17] Johannes Koch, Carlos A Velasco, and Philip Ackermann. 2017. Representing Content in RDF 1.0. (2017). https://www.w3.org/TR/Content-in-RDF10/
[18] Andreas Langegger and Wolfram Wöß. 2009. XLWrap - Querying and Integrating Arbitrary Spreadsheets with SPARQL. In *International Semantic Web Conference (Lecture Notes in Computer Science)*, Vol. 5823. Springer, 359–374.
[19] Franck Michel, Loïc Djimenou, Catherine Faron Zucker, and Johan Montagnat. 2014. xR2RML: Non-Relational Databases to RDF Mapping Language. (2014). https://hal.inria.fr/hal-01066663v1/document
[20] Boris Motik, Peter F. Patel-Schneider, and Bijan Parsia. 2012. OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition). (2012). https://www.w3.org/TR/owl2-syntax/
[21] Yavor Nenov, Robert Piro, Boris Motik, Ian Horrocks, Zhe Wu, and Jay Banerjee. 2015. RDFox: A Highly-Scalable RDF Store. In *International Semantic Web Conference (2) (Lecture Notes in Computer Science)*, Vol. 9367. Springer, 3–20.
[22] Martin J. O'Connor, Christian Halaschek-Wiener, and Mark A. Musen. 2010. M^2: A Language for Mapping Spreadsheets to OWL. In *OWLED (CEUR Workshop Proceedings)*, Vol. 614. CEUR-WS.org.
[23] Jason Slepicka, Chengye Yin, Pedro A. Szekely, and Craig A. Knoblock. 2015. KR2RML: An Alternative Interpretation of R2RML for Heterogenous Sources. In *COLD (CEUR Workshop Proceedings)*, Vol. 1426. CEUR-WS.org.