Mapping Hierarchical Sources into RDF using the RML Mapping Language

Anastasia Dimou*, Miel Vander Sande*, Jason Slepicka†,
Pedro Szekely†, Erik Mannens*, Craig Knoblock† and Rik Van de Walle*
*Ghent University – iMinds – Multimedia Lab
Gaston Crommenlaan 8 bus 201, B-9050 Ledeberg-Ghent, Belgium
Email: {firstname.surname}@ugent.be
†University of Southern California, Information Sciences Institute
Department of Computer Science, USA
Email: {slepicka,pszekely,knoblock}@isi.edu

***Abstract*—Incorporating structured data in the Linked Data cloud is still complicated, despite the numerous existing tools. In particular, hierarchical structured data (e.g., JSON) are underrepresented, due to their processing complexity. A uniform mapping formalisation for data in different formats, which would enable reuse and exchange between tools and applied data, is missing. This paper describes a novel approach of mapping heterogeneous and hierarchical data sources into RDF using the RML mapping language, an extension over R2RML (the W3C standard for mapping relational databases into RDF). To facilitate those mappings, we present a toolset for producing RML mapping files using the Karma data modelling tool, and for consuming them using a prototype RML processor. A use case shows how RML facilitates the mapping rules' definition and execution to map several heterogeneous sources.**

I. Introduction

Most of the data that we would like to have as Linked Data currently exists in formats other than RDF; much of it exists in relational databases. Many languages [1], tools and different approaches [2] have been proposed to convert data from relational databases to RDF. In 2012, the "R2RML: RDB to RDF Mapping Language" [1] became a W3C recommendation, standardizing a language for mapping relational databases to RDF. While databases account for a significant amount of data, a growing number of datasets are represented in other formats. For example, government and scientific data are often published in spreadsheets or text delimited (e.g., CSV) files. Vast amounts of data are accessible via Web APIs that return data in XML or JSON. ProgrammableWeb.org, an index of Web APIs reports over 10,000 different APIs in 2014, 74 of which return their results in RDF, while the number that produce JSON or XML is over 5,000 each. In contrast to RDB systems, different solutions were introduced for these different serializations, but neither uniform nor corresponding language.

In order to leverage these heterogeneous data in Linked Data applications, not only do we need solutions to express mappings from multiple data formats into RDF but we also need those solutions to map their *cross-file* links, too. Current approaches that automatically define classes and properties from the schema of the source data (e.g., map the XML schema to an ontology or define an RDF property for each object in a JSON file) solve the problem on a *per-file* basis, while in the effort to generalise the approach, a lot of the data semantics are getting lost. Furthermore, hierarchical sources are usually preprocessed, only providing partial access to their data.

In this paper, we present a solution that supports the definition of mappings and the generation of an RDF representation of data in hierarchical format. To this end, we use RDF Mapping Language (RML), a language to specify mappings for heterogeneous and hierarchical serializations into RDF, according to an RDF schema or ontology of the user's choice. RML is defined as a superset of the R2RML mapping language. We accompany RML with a toolset that makes the language operational and practical. The toolset consists of Karma, an integrated development environment that facilitates the creation of mappings for a large subset of RML, and an implementation of a prototype RML processor to execute them.

The rest of the paper is organized as follows: Section II discusses related solutions existing today. Section III describes how the RML language handles the mapping of hierarchical sources to RDF. Section III-C describes in detail how RML extends R2RML and III-D summarizes the extension. Section IV describes how Karma facilitates the creation of RML mapping definitions and section V addresses the challenges of implementing an RML processor. Finally, sections VI and VII present the solution's evaluation, conclusions and future work.

II. State of the Art

Several solutions exist to execute mappings from different file structures and serializations to their RDF representations. To be more precise, different mapping languages beyond R2RML are defined for databases [1], which, in turn, already has several implementations[2]. Similarly, mapping languages were defined to support conversion from data in CSV and spreadsheets to the RDF. The XLWrap's mapping language [3], the Mapping Master's M2 [4] and Vertere[3] are a few of them.

In the case of mappings from XML to RDF, there is more diversity on the proposed solutions. To the best of our

[1] http://www.w3.org/TR/r2rml/

[2] http://www.w3.org/2001/sw/rdb2rdf/wiki/Implementations

[3] https://github.com/knudmoeller/Vertere-RDF

978-1-4799-4003-5/14 $31.00 © 2014 IEEE
DOI 10.1109/ICSC.2014.25

knowledge, there are no mapping languages defined. Instead, the different tools rely mostly on existing XML solutions. Krextor [5] and the AstroGrid-D[4] mapping tools rely on XSLT, while other implementations, like Tripliser[5] and the XSPARQL [6], deploy mappings using XPath and XQuery. Finally, there are some other approaches that try to map the XML schema to OWL ontologies. These XML solutions lead to mappings on the syntactic level rather than on the semantic level or fail to provide a solution applicable to a broader domain. Furthermore, those solutions can not be extended to cover other file serializations beyond XML. Besides the afore-mentioned XML solutions, XSPARQL approximates a *Global-As-View* approach that performs dynamic query translation to convert different sources to RDF. XSPARQL generates RDF by integrating data from XML files and relational databases, once it has mapped them to RDF, and interlinks them with other RDF data. Tarql's[6] function follows also a querying approach to convert CSV to RDF.

Most existing tools deploy mappings from a certain source format to RDF (*source-centric approaches*). There are only a few tools that provide mappings from various source formats to RDF but even those tools deal with the different files they support separately following again a *source-centric* approach. Datalift[7], DataTank[8], Open Refine[9], Simile RDFizers[10] and Virtuoso Sponger[11] are a few of the most well-known.

III. Mapping heterogeneous resources using RML

While R2RML efficiently handles mapping definitions of data in relational databases to RDF, no uniform mapping language exists to support other formats. RML is defined as its superset, aiming to extend its applicability and broaden its scope beyond tabular structures, and define mappings of data in heterogeneous formats. In this section, we briefly introduce R2RML, discuss its limitations due to its assumption of a tabular input and describe how RML extends R2RML to handle hierarchical structures.

A. Mapping relational databases to RDF using R2RML

The input to an R2RML mapping is a relational database and the output an RDF dataset. The mapping to the RDF data model is based on one or more *Triples Maps* and occurs over a Logical Table iterating on a *per-row* basis. A Triples Map specifies rules that generates a number of RDF triples from each row and consists of three main parts: the *Logical Table*, the *Subject Map* and zero or more *Predicate-Object Maps*. For instance, the triples of Listing 4, lines 24 - 31 are generated from Table I by applying the Triples Map in Listing 1.

A Logical Table (`rr:LogicalTable`) is either an SQL table or an R2RMLView. In the former case, a Logical Table is

[4]http://www.gac-grid.de/project-products/Software/XML2RDF.html

[5]http://daverog.github.io/tripliser/

[6]https://github.com/cygri/tarql

[7]http://datalift.org

[8]http://datatank.com

[9]http://openrefine.org

[10]http://simile.mit.edu/wiki/RDFizers

[11]http://virtuoso.openlinksw.com/dataspace/doc/dav/wiki/Main/VirtSponger

NAME	BIRTH_DATE	DEATH_DATE
Robert Theodore McCall	1919-12-23	2010-02-26
Ronald Anderson	1929-12-06	

TABLE I
THE SQL TABLE "ARTWORKS" WITH INFORMATION ABOUT ARTISTS.

represented by a source that has exactly one `rr:tableName` property that points to the table's name *[line 4]*. In the latter case, a Logical Table stand for the results of executing an SQL query (the R2RMLView) against the input database and is represented by a source that has exactly one `rr:sqlQuery` property, whose value is a valid SQL query.

The Subject Map (`rr:SubjectMap`) defines the rule that generates unique identifiers (URIs) for the resources and is used as the subject of all the RDF triples that are generated from this Triples Map *[line 5]*. Therefore, the RDF triples generated from one row in the Logical Table using a certain Triples Map all share the same subject.

The Predicate-Object Map (`rr:PredicateObjectMap`) *[line 6]* defines the rule that generates pairs of *Predicate Maps* and *Object Maps* that, together with the subjects generated by the Subject Map, generates one or more RDF triples for each row. A Predicate-Object Map consists of Predicate Maps *[line 7]*, which define the rule that generates the triple's predicate and Object Maps or Referencing Object Maps *[line 8]*, which defines the rule that generates the triple's object.

```
1  @prefix rr:  <http://www.w3.org/ns/r2rml#>.
2  @prefix ex:  <http://www.example.com/>.
3  <#ArtistMapping>
4   rr:logicalTable [ rr:tableName "ARTISTS"; ];
5   rr:subjectMap [ rr:template "http://ex.com/{NAME}" ];
6   rr:predicateObjectMap [
7    rr:predicate ex:birth_date;
8    rr:objectMap [ rr:column "BIRTH_DATE" ];];
9   rr:predicateObjectMap [
10   rr:predicate ex:status;
11   rr:objectMap [ rr:constant ex:artist ] ].
```

Listing 1. An R2RML mapping document for the database table ARTISTS.

The Subject Map, the Predicate Map and the Object Map are *Term Maps*, namely rules that generate an RDF term (an URI, a blank node or a literal) from a Logical Table row. A Term Map can be a *constant-valued term map* (`rr:constant`) that always generates the same RDF term, *[line 11]*, or a *column-valued term map* (`rr:column`) that is the data value of a referenced column in a given Logical Table row *[line 8]*, or a *template-valued term map* (`rr:template`) that is a valid string template that can contain referenced columns *[line 5]*. The referenced columns are column names that exist in the triples map's Logical Table.

Furthermore, R2RML supports cross-references between Triples Maps, when the subject of a Triples Map is the same as the object generated by a Predicate-Object Map. A Referencing Object Map (`rr:RefObjectMap`) is used then to point to the Triples Map that generates on its Subject Map the corresponding resource, the so-called Referencing Object Map's *Parent Triples Map*. If the Triples Maps refer to different Logical Tables, a join between the Logical Tables is required. The *join condition* (`rr:joinCondition`) performs the join exactly as a join is executed in SQL. The join condition consists of a reference to a column name that exists in the Logical Table of the triples map that contains the Referencing Object Map (`rr:child`) and a reference to a column name that exists in the Logical Table of the Referencing Object Map's Parent Triples Map (`rr:parent`).

B. Dealing with hierarchy and heterogeneity

Three features of relational databases' tables make R2RML mappings fairly easy to process: (i) each row is a self-contained extract of data that can be processed independently; (ii) the columns in each row can be referred to unambiguously; (iii) for each reference to a column in a single row, a unique value is returned. In hierarchical data, there is no concept of a row to iterate over, an element name can appear multiple times in the hierarchy, and a reference can map to a collection of values. This requires the following adaptations:

(i) Explicit reference to the iteration pattern: This lends R2RML a simple *"per row"* iteration model that R2RML Triples Maps take advantage of. Re-using R2RML to map data from a CSV file to the RDF model is straightforward, because a CSV file shares the same features. However, hierarchical sources do not have such an implicit iteration model, nor an implicit way of referring to their values, as columns are in the case of database tables. Therefore, the R2RML mapping language needs an explicit reference to an iteration pattern, before it can be used for data in other structures and serializations.

(ii) Abstract reference to the input data: A generic mapping language needs to deal with different data formats and serializations which use different ways to refer to their data values (e.g., XML's elements and attributes or JSON's objects). When targeting a generic and extendable solution, the database-specific references from the core model of R2RML need to be excluded. To this end, any reference to the input file should remain detached from the core model and be defined in a form relevant to the input file's serialization. In this regard, R2RML considers the columns' names as a reference point derived from its own query language SQL. Thus, the references to the data in other serializations should be addressed as they are defined at their corresponding query languages, e.g., XPath for XML serializations or JSONPath for JSON serializations. As a more generic model is targeted, it is crucial not to impose a uniform way to refer to the data, but rather provide a loose-coupling between the references to input data using their query languages and the mapping language definition.

(iii) More than one triple per Predicate-Object Map: R2RML refers to the input data with column names and, for each reference, only expects a single value returned, as a certain column name occurs only once on each row. Considering this, it is assumed that, as a mapping definition has a single subject definition, only one subject will be generated on every iteration and only one triple for every subsequent Predicate-Object mapping definition. In contrast, in hierarchical input sources, a certain reference point (e.g., an element in the case of XML) may occur more than once on a certain iteration. Therefore, a generic solution should handle those multiple occurrences that occasionally generate more subjects and more triples after a single mapping definition.

Overall, in order to deal with other serializations, the scope of R2RML needs to be broadened. A solution is required that preserves the mapping definitions as in R2RML but tries to efficiently tackle the burdens posed by the hierarchical structure of data being, for example, in XML and JSON files. We propose RML that deals with the aforementioned issues.

In the next section, we provide a description of RML mapping definitions, while a concrete example of an iteration over the hierarchical sources is described in Section V. In the remainder of this section, we consider a simple scenario where we have two source files to be mapped to their RDF representation. The first file (Listing 2) is an XML file with information about artists. The file contains exactly the same data as the database table (Table I). The second file (Listing 3) is a JSON file with information about artworks in a museum. We map the two files into the RDF data model considering the mapping definitions as provided in Listing 5. The expected output is demonstrated in Listing 4.

```
<Artists> ... ...
 <Artist>
  <Name>Robert Theodore McCall</Name>
  <Birth_Date>1919-12-23</Birth_Date>
  <Death_Date>2010-02-26</Death_Date>
 </Artist>
 <Artist>
  <Name>Ronald Anderson</Name>
  <Birth_Date>1929-12-06</Birth_Date>
  <Death_Date/>
 </Artist> ... ...
</Artists>
```

Listing 2. artist.xml file with information about the artists.

```
[ ... { "Title": "Apollo 11 Crew",
    "Artist": "Ronald Anderson",
    "Ref":  "NPG_70_36",
    "Sitter": [ { "Name": "Neil Armstrong",
          "Birth Date": "1930-08-05" },
        { "Name": "Buzz Aldrin",
          "Birth Date": "1930-01-20" },
        { "Name": "Michael Collins" } ],
       "DateOfWork": "1969" },
  { "Title":  "Neil Armstrong",
    "Artist":  "Robert Theodore McCall",
    "Ref":   "S_NPG_2010_51",
    "Sitter":  [ { "Name": "Neil Armstrong" } ],
    "DateOfWork": "2009" }, ... ]
```

Listing 3. museum.json file with information about artworks in a museum.

C. Mapping hierarchical sources to RDF using RML

In this section, we explain how RML extends R2RML to handle hierarchical sources. We describe how database-specific components are decoupled and we introduce new concepts to cover each requirement approached in the previous section.

Specifying the input data: In R2RML a Triples Map consists of a Logical Table, a Subject Map and multiple Predicate-Object maps. The R2RML Logical Table is extended in RML to a resource named *Logical Source* (`rml:LogicalSource`) which is used to specify the input file whose data needs to be mapped. In R2RML, the Logical Table was specified providing the table's name. In the case of RML, the *source* (`rml:source`) property is introduced to specify the source file. The reference to the source file can be a URI or a relative reference to a local

```
@prefix ex:  <http://ex.com/>.
@prefix crm: <http://www.cidoc-crm.org/cidoc-crm/> .

ex:NPG_70_36 a crm:E22_Man-Made_Object ;
 crm:P102_has_title "Apollo 11 Crew" ;
 crm:P14_carried_out_by ex:Ronald+Anderson ;
 crm:P62_depicts ex:Neil+Armstrong,
     ex:Buzz+Aldrin, ex:Michael+Collins.

ex:S_NPG_2010_51 a crm:E22_Man-Made_Object ;
 crm:P102_has_title "Neil Armstrong" ;
 crm:P14_carried_out_by ex:Robert+Theodore+McCall ;
 crm:P62_depicts ex:Neil+Armstrong.

ex:Neil+Armstrong a crm:E21_Person ;
 rdfs:label "Neil Armstrong".

ex:Buzz+Aldrin a crm:E21_Person ;
 rdfs:label "Buzz Aldrin".

ex:Michael+Collins a crm:E21_Person ;
 rdfs:label "Michael Collins".

ex:Ronald+Anderson a crm:E21_Person ;
 rdfs:label "Ronald Anderson";
 ex:birth_date "1929-12-06"^^xsd:date.

ex:Robert+Theodore+McCall a crm:E21_Person ;
 rdfs:label "Robert Theodore McCall";
 ex:birth_date "1919-12-23"^^xsd:date;
 ex:death_date "2010-02-26"^^xsd:date.
```

Listing 4. The RDF output.

```
1  @prefix rr:  <http://www.w3.org/ns/r2rml#>.
2  @prefix rml: <http://semweb.mmlab.be/ns/rml#> .
3  @prefix ql:  <http://semweb.mmlab.be/ns/ql#> .
4  @prefix crm: <http://www.cidoc-crm.org/cidoc-crm/> .
5  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
6  @prefix ex:  <http://ex.com/>.
7  @prefix xsd: <http://www.w3.org/2001/XMLSchema#>.
8
9  <#ArtworkMapping>
10  rml:logicalSource [
11   rml:source "museum.json";
12   rml:referenceFormulation ql:JSONPath;
13   rml:iterator "$.[*]"; ];
14  rr:subjectMap [
15   rr:template "http://ex.com/{Ref}";
16   rr:class crm:E22_Man-Made_Object; ];
17  rr:predicateObjectMap [
18   rr:predicate crm:P102_has_title;
19   rr:objectMap [ rml:reference "Title"; ] ];
20  rr:predicateObjectMap [
21   rr:predicate crm:P14_carried_out_by;
22   rr:objectMap [
23    rr:parentTriplesMap <#ArtistMapping>;
24     rr:joinCondition [
25     rr:child "Artist";
26     rr:parent "/Artists/Artist/Name" ] ] ];
27  rr:predicateObjectMap [
28   rr:predicate crm:P62_depicts;
29   rr:objectMap [
30     rr:parentTriplesMap <#AstronautMapping> ] ].
31
32 <#AstronautMapping>
33  rml:logicalSource [
34   rml:source "museum.json";
35   rml:referenceFormulation ql:JSONPath;
36    rml:iterator "$.[*].Sitter"; ];
37  rr:subjectMap [
38   rr:template "http://ex.com/{Name}";
39   rr:class crm:E21_Person ];
40  rr:predicateObjectMap [
41   rr:predicate rdfs:label;
42   rr:objectMap [ rml:reference "Name" ] ] .
43
44 <#ArtistMapping>
45  rml:logicalSource [
46   rml:source "artist.xml";
47   rml:referenceFormulation ql:XPath;
48   rml:iterator "/Artists/Artist" ];
49  rr:subjectMap [
50   rr:template "http://ex.com/{Name}";
51   rr:class crm:E21_Person; ];
52  rr:predicateObjectMap [
53   rr:predicate rdfs:label;
54   rr:objectMap [ rml:reference "/Artists/Artist/Name"; ]; ];
55  rr:predicateObjectMap [
56   rr:predicate ex:birth_date;
57   rr:objectMap [
58    rml:reference "Birth Date";
59    rr:datatype xsd:date; ]];
60  rr:predicateObjectMap [
61    rr:predicate ex:status;
62    rr:objectMap [ rr:constant ex:artist ] ].
```

Listing 5. The RML mapping definition for the XML file.

file. In our example, line 11 of Listing 5 points to the JSON file in Listing 3 and line 46 points to the XML file in Listing 2.

Referring to the input data: R2RML uses the column names to refer to the input table as its query language SQL does. Similarly, RML considers that the references to the data should be addressed as they are defined at their corresponding query languages, e.g., XPath for XML serializations or JSONPath for JSON serializations. To this end, a *Reference Formulation* (`rml:referenceFormulation`) declaration is introduced allowing for each Triples Map to define the expected form of references to the data. Each subsequent reference to the input data (`rml:reference`) should be a valid expression of the specified reference formulation, as a valid column name is for R2RML. In our example, line 12 and line 47 of Listing 5 specify the Reference Formulation used for the references to the input dataset.

Iterating over the input data: As mentioned earlier, the iteration pattern cannot always be inferred in RML, but needs to be specified for every Triples Map. Therefore, the *iterator* (`rml:iterator`) property is introduced. The iterator determines the iteration model over the file, considering a pattern that repeats several times over the data, and specifies the extract of data to be mapped during each iteration. In R2RML, each iteration returns a row of a database table, while in RML each iteration returns an extract of data to be mapped. In the example in Listing 5, line 13 specifies the iteration pattern over a JSON file, while line 36 specifies another iteration pattern for a different Triples Map that generates other RDF triples. Line 48 determines the iteration pattern over an XML file.

Referring explicitly and implicitly to the input data: Any further reference to the extract of data, for example within the Subject Map or the Object Map, should also conform with the Reference Formulation specified at the Triples Map's Logical Source. The reference can be explicitly defined with the full path *[line 54]*, or relative to the Triples Map's iterator *[line 50]*, where both of them refer to the same data value of the current extraction. As R2RML does not allow cross-row references, RML does not allow references to data outside the current extract. Possible references are restricted towards data on a lower level in the hierarchical structure. Therefore, a single Triples Map cannot directly create triples which contain RDF terms generated on previous or future points in the iteration. However, this can be achieved by performing joins, as described later on.

Generating a subject but multiple objects on an iteration: While in R2RML a certain column name occurs only once, in

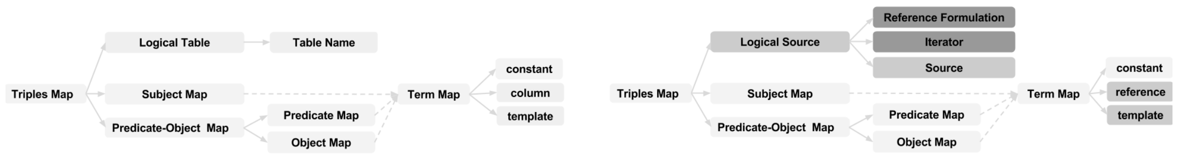

Fig. 1. On the left: R2RML diagram. On the right: RML diagram with R2RML features in light grey, extensions over R2RML in darker grey and additions over R2RML in dark grey.

the case of RML, an expression specified at a Term Map could be satisfied more times. As generating a single subject per iteration is a fundamental assumption of R2RML's definition, RML keeps the restriction that any reference used in the Subject Map definition should occur only once in the extract of data returned from a certain iteration. In our example, line 38 could not be the Subject Map in the place of line 11, as the latter is satisfied more than once per iteration. However, RML does not put any restrictions when the reference included in a Predicate or Object Map is satisfied more than once and, thus multiple predicates or objects are generated.

Integrated mapping of input data: As R2RML supports *cross-table* references, RML aims to support *cross-file* references by extending the Referencing Object Map. In RML, as in R2RML, if a Referencing Object Map refers to the same input source, a Parent Triples Map definition is only required to generate object resources considering another Triples Map definition. In RML though, such a reference is only valid when the Parent Triples Map's iteration pattern is nested deeper in the hierarchy than the child map. In our example, the `<#ArtworkMapping>` refers to the `<#AstronautMapping>` whose iteration pattern *[line 36]*, is deeper in the hierarchy compared to the one specified for the child map *[line 13]*. In this sense, the parent-child relationship is inverted in relation to the document hierarchy but consistent with R2RML naming conventions. In RML, the parent and child maps can also refer to different files. The join condition's Child indicates the reference to the data value extracted at the current iteration. The Parent Reference indicates a data extract of the Parent Triples Map. Each one of the Child and Parent References are specified in the syntax the Reference Formulation specified in the Logical Source of the Triples Map foresees. In our example, the museum's mapping is enriched with additional data from the XML file regarding the artists. A joint mapping definition is considered to keep the used sources aligned and have a single point of definition for the resources generated for the museum's artists and the resources generated for the artists based on the complementary file. As the join occurs between two references that are in different formats, the Child Reference at line 25 is defined using the JSONPath according to its Triples Map's Reference Formulation, while the Parent Reference at line 26 is defined using the XPath syntax as this is the language specified in the Parent Triples Map of the Referencing Object Map. Therefore, the Child Reference and the Parent Reference of a Join Condition may be defined using different Reference Formulations, if the Triples Map refers to sources of different formats.

D. Summary of RML extensions to R2RML

As displayed in Figure 1, RML keeps R2RML's cornerstones intact but extends those features that were *relational-database specific* aiming to broaden its scope. In summary, RML extends R2RML's Logical Table to a Logical Source `rml:LogicalSource` which can be, not only a relational database's table, but also a source file. In the same context, instead of exclusively defining table names, RML can support any reference to any source file within its Logical Source. While in R2RML it is mandatory that a table will be used as an input, in RML a broader range of input sources is supported. Therefore, RML introduces the Reference Formulation property `rml:referenceFormulation` that clarifies which file format is parsed and how the references to the extracts of this input are defined. Furthermore, since RML needs to process files that do not have an explicit iteration pattern as relational databases have, the iterator property `rml:iterator` is introduced to be used when the iteration pattern needs to be specified. Since specifing hierarchical data elements requires a more rich language than just column names, the `rr:column` property of R2RML is extended in RML as `rml:reference` and its value is expected to be a valid expression in the language specified by the Reference Formulation.

An RML mapping definition follows the same synax as R2RML. The RML vocabulary namespace is http://semweb.mmlab.be/ns/rml# and the preferred prefix is *rml*. More details regarding RML can be found at http://semweb.mmlab.be/rml.

IV. CREATING RML DOCUMENTS

To provide users with a semi-automatic way to quickly generate RML mapping definitions, an extension was built for Karma, an information integration tool. More specifically, the Karma GUI and semantic model learning capabilities were enhanced to accommodate RML's needs.

Figure 2 is a stylized example of how a user would model the example museum JSON data source. For each data element, the user assigns semantic types to describe the relationships between the data element. The figure shows the user attempting to assign the semantic type `rdfs:label` to the Artist field to describe a class (e.g., `crm:E21_Person`). The figure also shows a list of semantic types suggested by Karma to the user by reasoning about OWL Ontologies, learning from past user actions, and using Conditional Random Fields. With each new semantic type and class, Karma develops a graphical representation of the relationships. The classes become internal nodes in a graph and the semantic relationships between the classes become links that tie the classes together. Meanwhile, the semantic types connect the

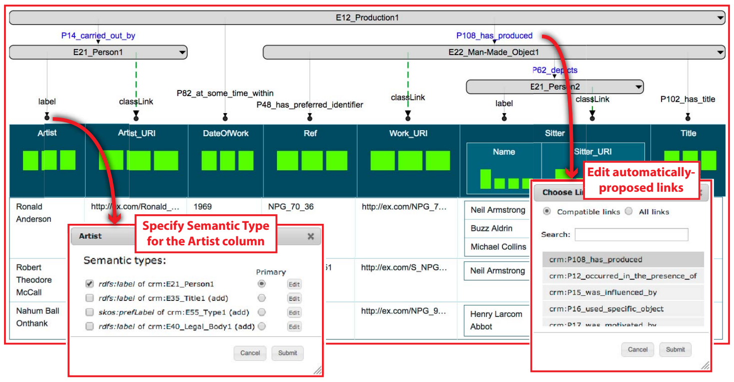

Fig. 2. Building the RML mapping for the museum dataset using Karma.

classes to the data, which form external nodes. Each class gets assigned an *id*, if there are multiple instances of the class, for instance, `crm:E21_Person` has two: the Artist (*E21_Person1*) and the Sitter (*E21_Person2*). After each addition to the graph, Karma attempts to create a minimum weight Steiner Tree that best describes the relationships between the data and then it adds links to the graph. If the user does not agree with the automatically proposed links, they are able to edit them.

Generating RML Mapping from Karma Model

Once the user is satisfied with the model, Karma generates an RML mapping document from the Steiner Tree graph representation. It begins with generating Triples Map for the root of the tree and all subsequent internal nodes of the tree. To create a Subject Map for the node *E22_Man-Made_Object1*, Karma starts with outputting the node's class, (`crm:E22_Man-Made_Object`), as the class of the Subject Map. Karma then searches the node's links for one marked as a classLink, which corresponds to the node's URI. Karma uses this to generate the template for Subject Map. If no such link exists, no template is added, because the internal node is a blank node. If one does exist, like *Work_URI* in this case, Karma normally outputs a template containing just the path to the data element in the appropriate reference formulation. For instance, the expression in JSONPath would be `$[*].Work_URI`.

In this example, however, the *Work_URI* data element does not correspond to the source data. In Karma, users can derive new data elements from the source data by performing cleaning and transformation actions in ways that may not map to RML. One such action that will map to RML, however, is leveraging the ability to execute user-provided Python code to generate URIs instead of its usual purpose of transforming and cleaning data. Karma can parse the Python code to see if it can be translated into a valid template. If so, the template of line 15 is added to the Subject Map instead of the new data element. Otherwise, the original non-compliant template is generated from the new data element and the user is notified.

To create the Logical Source for the Triples Map, Karma begins by outputting the worksheet's name *[line 11]*, as the Source name and adds JSONPath as the `rml:referenceFormulation` according to the source data behind the worksheet: JSON. Accordingly, it also generates an `rml:iterator` in the appropriate reference formulation. If a data element containing the URI exists for the node represented by the Triples Map, Karma outputs a path expression that leads to the parent of that data element, whether it is the root of the document or a nested element. Alternatively, if the Triples Map represents a blank node, Karma searches through the data elements of the leaf nodes to find the deepest nested table and generates a path expression for that data element's parent likewise.

After generating the Subject Map and Logical Source, Karma processes each of the node's links to generate Predicate-Object Maps. For each link, Karma outputs a Predicate Object Map containing the link's semantic type. If the link target is a leaf node, namely a data element in the source data, Karma generates an Object Map, like for `crm:P102_has_title`. For that Object Map, Karma outputs an `rml:reference` that captures the path to the field in the appropriate Reference Formulation relative to the iterator, in this case *Title*. If the link targets another internal node, Karma generates a Referencing Object Map with an `rr:parentTriplesMap` pointing to the Triples Map that corresponds to the link target.

V. PROCESSING RML DOCUMENTS

Unlike R2RML, implementing a processor for RML is more complex, since it deals both with tabular (relational databases and CSV) and hierarchical sources (XML and JSON). Therefore, it demands a scalable and abstract approach to support the extracts of data in different structures (tabular and hierarchical) and different formats (e.g., XML and JSON serializations of hierarchical resources) in a uniform way.

While RML syntax is used for the mapping document definition, in order to deal with the aforementioned extraction's caveats, RML relies on expressions in a target language (`rml:referenceFormulation`). Such an expression is used wherever values need to be extracted from the source, namely whenever an RDF Term Map or an iterator (`rml:iterator`) appears. To ensure consistency, the expression should be a valid expression according to the language specified in the Triples Map (`rml:referenceFormulation`). In order to deal with these embedded expressions, an RML processor is required to have a modular architecture where the *extraction* and *mapping* modules are executed independently of each other. When the RML mappings are processed, the *mapping* module deals with the mappings' execution as defined at the mapping document in RML syntax, while the *extraction* module deals with the target language expressions. An extractor corresponding to the specified target language executes the expression and returns the specified value. Therefore, the role of the extractor is limited in parsing the defined source and providing to the *mapping* module the corresponding extract of data as specified.

An RML processor can be implemented using two alternative models: *mapping-driven* or *data-driven*. In the former case, the processing is driven by the *mapping* module. It requests an extract of data from the *extraction* module, considering the iteration pattern specified at the Logical Source. In the latter case, the processing is driven by the *extraction* module. It passes an extract of data to the *mapping* module, which applies the mapping rules valid for the particular extract. The expressivity of such languages is usually very high. However, to limit complexity and increase efficiency of implemented processors, a well-defined set of constraints is included in the RML definition. This advocates *streaming* solutions, since datasets do not always fit in the processor's memory. A side-effect of a streaming approach, is the inability to support some features of expression languages. For instance, XPath has look-ahead functionality that requires access to data which is not yet known. Nevertheless, in practice, most of the expressions only require functionality within this subset. As a result, the W3C XSLT 3.0 Working Draft [7] already mentions a streaming specification. For functionality not supported by streaming, a fallback mechanism to in-memory processing can be provided.

As a proof of concept, we created a Java implementation of an RML processor[12]. In the remainder of this section, we describe the processing of Triples Maps in RML, disambiguating further the processing of Referencing-Object Maps.

A. *Processing Triples Maps*

Unlike R2RML, data can no longer be approached in a purely row-based fashion as hierarchical structures follow different or arbitrary iteration patterns. Therefore, this iteration pattern is defined at the Logical Source. The iterator (`rml:iterator`) enables the processor to traverse through the source on a per data-extract (e.g., XML element) manner. For each extract of data, the Subject Map and the Predicate-Object Maps of the Triples Map are applied. The expressions present in `rml:reference` or `rr:template` refer to values in the current extract, analogue to the column name references of a row in R2RML. Unless the Term Maps to be executed define constants, the extraction module takes care of the processing of the expressions used and returns the corresponding values to the mapping module. Although these patterns allow a processing strategy analogue to R2RML, there is an important behavioural distinction. For tabular data (e.g., CSV or RDB), the combination of row and column guarantees a single value is returned, a fundamental assumption of R2RML. However, with the path expression languages used for hierarchical sources, such assumption cannot be made. Therefore, RML expects the combination of data-extract and expression to return one or more values. Note that the RML requires expressions outside the iterator to retrieve only text nodes.

When the iteration is executed, each data-extract is passed to the mapping module. The Subject Map is executed first and computes the subject of all generated triples of this Triples Map. The expressions used to define the subject's URI template should be validated only once per extract. In compliance to R2RML and despite the distinct behaviour mentioned above, this expression is only allowed to return a single value. After the Subject Map is generated, the Predicate-Object Maps are executed. A single Predicate-Object Map can contain n_{pom} Predicate Maps and m_{pom} Object Maps. Even though the expressions used to define the subject's URI should be validated only once per extract, the same restriction is not applicable for the Predicate-Object Maps in the case of RML. For each Predicate Map, an expression returns i_{pom} values, constructing an equal amount of predicates. For each Object Map, an expression returns j_{pm} values, constructing an equal amount of objects. Therefore, a Predicate-Object Map generates a total of $(n_{pom} \times i_{pom}) \times (m_{pom} \times j_{pom})$ triples in the case of RML.

B. *Processing Referencing Object Maps*

Processing Referencing Object Maps handles different levels of complexity. In the simplest case, the Parent Triples Map, defined in the Referencing Object Map, and the current Triples Map share the same source and the subject pattern of the Parent Triples Map is the same or narrower of the one at the current Triples Map *[line 30]*. This allows the processor to evaluate both Term Maps together, generating both subjects together and combining them in the desired triple.

In case the Referencing Object Map uses a different Logical Source, a Join Condition needs to be specified, as in R2RML *[line 23]*. However, RML supports referring to a source file of different structure and serialization. The mapping module of the processor triggers the extractor of the extraction module to iterate over the source file according to the iterator's pattern. For each data-extract, the parent's reference value is compared to the child's reference value. When both values are equal, the URI of the Object Map is generated executing the Subject Map of the Referencing Object Map against the matching extract of data. Then, the generated URI is assigned as the generated object of the original Predicate Object Map.

As soon as the triples are generated, they can be either loaded to a triple store or written to a file. Overall, an RML processor needs to be accompanied not only by an RML processor but also by other validators for each separate extractor to validate the expressions of the target languages.

VI. Evaluation

We evaluate our approach comparing our solution to Datalift [8], one of the pioneer tools for mapping data in different file formats to RDF. Datalift was selected as it also offers the full stack of defining and executing mappings of

[12]https://github.com/mmlab/RMLProcessor

various raw data (CSV, RDF, XML and SHP files) to RDF and is a representative XML approach. For our evaluation, we used data of events taking place during Gent festival. The data is in XML files, one for each day of the ten days of festivities and are available at http://semweb.mmlab.be/rml/example. We evaluate our approach based on the following criteria:

Semantic quality of the RDF *output:* Karma proposes a semantic representation for the data which is crucial for the enhanced quality of the generated output. Since RML supports defining mappings of more sources in a combined way, Karma can propose mapping recommendations of enhanced quality as it builds a more comprehensive understanding of the domain. Datalift neither recommends nor maps multiple files together.

Datalift limits further the eventual generated RDF output due to its *entity-per-row* assumption when mapping tabular data, which is often encountered to other solutions, too, and the restrictive XSLT transformation, one of the most common approaches for mapping data in XML. Each line of tabular data becomes a resource and each column an RDF property as the W3C direct mapping suggests. In the case of XML files, a generic XSLT transformation is performed to produce RDF from a wide range of XML documents. On the contrary, an RML based implementation defines triples of different subjects for a certain data extract, achieving semantically richer output.

For our evaluation, we relied on Karma to provide us with a mapping recommendation and tried to reproduce it on Datalift. Our RML based solution generated an RDF output with resources of different types and links among data of different input sources, while Datalift provided an output of a single type and no links between data of different input sources.

Assistance in defining the mappings: Datalift provides a User Interface as Karma, but is limited on assisting users to coordinate the mapping procedure and fire its execution. Datalift follows a two-steps manual procedure on a *per-file* basis to edit the mappings. Initially, it converts data to raw RDF without taking into account vocabularies, namespaces or links to existing datasets. Users then select from a list of available classes and properties to assign to the raw RDF but interlinking resources is difficult, if not impossible. In contrast, besides proposing a mapping recommendation, Karma, relying on RML, accomplishes the mappings in one step. Over the course of building a model for a complicated dataset, Karma saves the users of debugging and iteratively going over the output to verify that all triples are generated and linked properly by graphically reasoning about and verifying the model. The users can also get instant feedback on their model by testing which triples are generated, in contrast with Datalift users.

Reusability of the mapping definitions: Datalift users need to redefine the mappings for every new file mapped, because the mapping definitions are tied to the implementation and the specified source; as it happens to most of the proposed solutions so far. An RML based solution reuses the same mapping definitions for similar input sources, either of the same format or not, and across different tools for editing and executing mappings (mapping documents generated using Karma are used to perform the mapping execution on RML processor). For our evaluation, we reused the same RML mappings for the ten input files, unlike Datalift where we needed to redefine the mappings for each one of the ten files.

Scalability of mappings' execution: Datalift processes the mapping of different files depending on their format, thus it can not be easily extended to support e.g., mappings of data in JSON, as the full functionality needs to be implemented from the beginning. An RML processor, in contrast, is source-independent and, thus easily extended to cover other input sources, as the core of the mappings execution is uniformly implemented and any new extractor can be easily configured.

Overall, our solution relying on RML mapping formalisation is optimal as semantically richer and better interlinked output is achieved, while mappings are reused for sources describing the same domain and are interoperable across different tools.

VII. Conclusions and Future Work

In this paper, we presented a novel approach for mapping heterogeneous hierarchical sources into RDF using the RML mapping language. We presented how R2RML is extended to support hierarchically structured data and described a semi-automatic approach to building RML mapping documents using Karma as well how to execute the mappings to build RDF with a prototype RML processor.

For future work, RML can be extended to support views on sources, built by queries. This captures the issue of data cleaning and transformation to enhance the mapping's applicability. Next, the efficiency of RML processing can be improved. A possible optimization is the use of execution plans that efficiently arrange the execution order depending on their dependencies. Finally, RML could be used to specify the triples' provenance, by taking advantage of the RDF-nature of the mapping documents.

References

[1] M. Hert, G. Reif, and H. C. Gall, "A comparison of RDB-to-RDF mapping languages," in *Proceedings of the 7th International Conference on Semantic Systems*, ser. I-Semantics '11. ACM, 2011, pp. 25–32.

[2] S. S. Sahoo, W. Halb, S. Hellmann, K. Idehen, T. Thibodeau Jr, S. Auer, J. Sequeda, and A. Ezzat, "A survey of current approaches for mapping of relational databases to rdf," *W3C RDB2RDF Incubator Group Report*, 2009.

[3] A. Langegger and W. Wöß, "XLWrap – Querying and Integrating Arbitrary Spreadsheets with SPARQL," in *Proceedings of the 8th International Semantic Web Conference*, ser. ISWC '09. Springer-Verlag, 2009, pp. 359–374.

[4] M. J. O'Connor, C. Halaschek-Wiener, and M. A. Musen, "Mapping Master: a flexible approach for mapping spreadsheets to OWL," in *Proceedings of the 9th International Semantic Web Conference on The Semantic Web - Volume Part II*, ser. ISWC'10. Springer-Verlag, 2010, pp. 194–208.

[5] C. Lange, "Krextor - an extensible framework for contributing content math to the Web of Data," in *Proceedings of the 18th Calculemus and 10th international conference on Intelligent computer mathematics*, ser. MKM'11. Springer-Verlag, 2011, pp. 304–306.

[6] S. Bischof, S. Decker, T. Krennwallner, N. Lopes, and A. Polleres, "Mapping between rdf and xml with xsparql," *Journal on Data Semantics*, vol. 1, no. 3, pp. 147–185, 2012.

[7] M. Kay and Saxonica, "Xsl transformations (xslt) version 3.0 working draft," http://www.w3.org/TR/xslt-21/#streaming, 2012.

[8] F. Scharffe, G. Atemezing, R. Troncy, F. Gandon, S. Villata, B. Bucher, F. Hamdi, L. Bihanic, G. Képéklian, F. Cotton, J. Euzenat, Z. Fan, P.-Y. Vandenbussche, and B. Vatant, "Enabling Linked Data publication with the Datalift platform," in *Proc. AAAI workshop on semantic cities*, 2012.