Human-Friendly Knowledge Graph Construction: Which one do you chose?

Ana Iglesias-Molina[1], David Chaves-Fraga[1,2], Ioannis Dasoulas[2], and Anastasia Dimou[2]

[1] Ontology Engineering Group, Universidad Politécnica de Madrid, Spain
{ana.iglesiasm,david.chaves}@upm.es
[2] KULeuven – Flanders Make@KULeuven – Leuven.AI, Belgium
{ioannis.dasoulas,anastasia.dimou}@kuleuven.be

Abstract. Knowledge Graphs (KGs) are a powerful mechanism to structure and organize data on the Web. KGs are usually constructed by declaring a set of mapping rules, specified according to the grammar of a mapping language (e.g., RML), that relates the input data sources to a domain vocabulary. However, the verbosity and (manual) definition of these rules affect their global adoption. Several user-friendly serializations for different mapping languages were proposed to facilitate users with the definition of such rules, e.g., YARRRML, SMS2, XRM, or ShExML. Still, most of them do not cover all features of the mapping languages for KG construction (e.g., constructing RDF-star), or they lack tooling support. In this paper, (i) we present a set of updates over the YARRRML serialisation to empower it with the latest necessities for constructing KGs; (ii) we implement these new features in a new open-source translator, Yatter, currently used in different real-use cases and international projects; and (iii) we qualitatively compare our proposal against similar state-of-the-art serialisations, and their associated translators over a set of conformance test cases. Our proposal advances the declarative construction of KGs and supports users in choosing an appropriate serialisation and translator for their use cases.

Keywords: Knowledge Graphs · Mapping Languages · YARRRML.

1 Introduction

Knowledge graphs have proven to be a powerful technology for integrating and accessing myriads of data available on the Web. Using mapping languages guarantees sustainable construction of knowledge graphs based on a set of declarative mapping rules [18], specified according to a mapping language's grammar (e.g., R2RML [8] and RML [10]), which relates data sources to a domain vocabulary.

Several mapping languages were proposed to construct knowledge graphs [18]. W3C recommends the Relational to RDF Mapping Language (R2RML) [8]) to construct RDF from Relational Databases. R2RML is a custom mapping language based on the RDF syntax. Multiple works extend R2RML [18] (e.g., RML [10]) enabling its use for heterogeneous data sources. Despite efforts to conceptualize and describe these mapping languages, their manual creation process, verbosity, and complexity lead to the appearance of user-friendly serializations.

Human-friendly serialisations emerged to ease the definition of mapping rules. YARRRML [13] leverages YAML to offer a user-friendly representation to define mapping rules, while ShExML [11] extends the syntax of the ShEx constraint language [15]. XRM [21] provides an abstract syntax that simulates programming languages and SMS2 [17], proposed by Stardog,[3] is loosely based on the SPARQL query language and extends the features of R2RML to create virtual KGs. Each serialisation is accompanied by a system that translates their rules into mapping languages, such as RML or R2RML (henceforth abbreviated as [R2]RML). However, these serialisations and translators were not compared with each other in terms of serialisations' features and system's characteristics, even though it would help to decide which serialisation fits each use case.

In this paper, we propose YARRRML-star, by extending YARRRML to also support RML-star [9] to construct RDF-star graphs, and improve it to adhere with the latest RML updates (e.g., datatypes, joins, etc.). We developed a translator system that implements the new features, and validate our proposal with test cases and compared it to other user-friendly serializations.

The contributions of this paper are: (i) the YARRRML-star[4], an extension of the YARRRML serialisation to fully cover RML and support for RML-star; (ii) Yatter[5], a new YARRRML system that implements the translation of the new features; (iii) a qualitative comparison of human-friendly mapping languages with respect to their expressiveness; (iv) conformance test cases[6] for YARRRML adapted from R2RML/RML test-cases and translated to other serializations; and (v) a comparison of language conformance and other features (e.g., open vs. close source code) of the associated translators.

The remaining paper is structured as follows: Section 2 describes related work, and Section 3 a set of relevant concepts to understand the rest of the paper. Section 4 describes the extension over the YARRRML serialization. Section 5 presents the implementation of these advances in a new translator, Yatter. Section 6 validates our proposals and their position compared to previous works and Section 7 outlines our conclusions and future work.

2 Related Work

Different serialisations were proposed so far to offer a user-friendly experience for generating RDF graphs. Each serialisation is accompanied with a system to translate the mapping rules to RML or directly construct the RDF graph.

YARRRML [13] is a compact serialisation for RML and R2RML mapping rules based on YAML[7] and is currently used in several projects over different do-

[3] `https://www.stardog.com/`
[4] `https://oeg-dataintegration.github.io/yarrrml-spec/`
[5] `https://github.com/oeg-upm/yatter/`
[6] `https://github.com/oeg-upm/yarrrml-validation`
[7] `https://yaml.org/`

mains [4, 16, 2]. Companies also incorporate YARRRML into their processes, e.g., the Google Enterprise Knowledge Graph where YARRRML is used to construct and reconcile an external knowledge graph[8]. However, YARRRML is currently outdated with respect to latest developments of RML (e.g., RML-star [9]).

ShExML [11] is a mapping language for heterogeneous data sources based on Shape Expressions (ShEx) [15]. Its syntax combines declarations to handle data sources with a set of shapes that define how they should be mapped. The language's operators and support of Scala functions offer multiple possibilities for data transformation. ShExML mapping rules can be translated into RML or directly used to construct RDF graphs with the ShExML Java library[9].

The Expressive RDF Mapper [21] (XRM) by Zazuko[10] offers an abstract syntax for mapping rules aiming to resemble programming languages. XRM's system translates the mappings to [R2]RML, CARML[11] or CSVW[12], and provides code-assistance and syntax validation.

Another well-known human-friendly serialization is the Stardog Mapping Syntax 2 [17] (SMS2) supporting both structured and semi-structured data sources. SMS2 is loosely based on SPARQL `CONSTRUCT` queries. It follows a FROM - TO syntax, where the FROM part resembling the data source it refers to, and the TO part resembling the RDF syntax that defines how the output will be generated. This serialisation can be used to directly create KGs.

3 Background

As we propose YARRRML-star extending YARRRML to also support RML-star, in this section we describe the basics of the YARRRML serialisation and how it translates to RML. To this end, we present an example in YARRRML (List 1) and its corresponding translation to RML (List 2). The YARRRML mapping rules (List 1) are grouped in sets unified under a mapping identifier, given below the `mappings` key (lines 1-2,18). Each rule set describes how to access the input data sources and how the triples will be constructed from these data sources. The input data sources description is specified below the `sources` key (lines 4-5,19-20). Within, the name, path and format of the file are specified.

The `graphs` key assigns a named graph to the triples (line 3). The `subjects` key defines the IRI or Blank Node of the subjects to be generated (lines 6, 21), and the `predicateobjects` key is used for the predicate-object pairs (lines 7-16,22-24). The `predicates` (line 9) and `objects` (line 10) keys define how predicate IRIs and object terms (IRIs, Blank Nodes or Literals) are generated.

[8] `https://cloud.google.com/enterprise-knowledge-graph/docs/entity-reconciliation-console`
[9] `https://github.com/herminiogg/ShExML`
[10] `https://zazuko.com`
[11] `https://github.com/carml/carml`
[12] `https://www.w3.org/ns/csvw`

```
 1 mappings:
 2  personTM:
 3   graphs: :pole-vaulters
 4   sources:
 5    - [jump.csv~csv]
 6   subjects: :$(ID)
 7   predicateobjects:
 8    - [:name, $(PERSON), en~lang]
 9    - predicates: :jumps
10      objects:
11       - mapping: jumpTM
12         condition:
13          function: equal
14          parameters:
15           - [str1, $(ID)]
16           - [str2, $(ID)]
17
18  jumpTM:
19   sources:
20    - [jump.csv~csv]
21   subjects: :$(ID)-$(MARK)
22   predicateobjects:
23    - [:date, $(DATE)]
24    - [:mark, $(MARK), xsd:float]
```

Listing 1: YARRRML mapping rules.

```
<#personTM>
 a rr:TriplesMap ;
 rml:logicalSource [
  rml:source "jump.csv" ;
  rml:referenceFormulation ql:CSV
  ] ;
 rml:subjectMap [
  rr:template ":{ID}" ] ;
  rr:graphMap [
   rr:constant :pole-vaulters ];
 rr:predicateObjectMap [
  rr:predicate :name ;
  rml:objectMap [
   rml:reference "PERSON";
   rr:language "en" ] ] ;
 rr:predicateObjectMap [
  rr:predicate :jumps;
  rr:objectMap [
   rr:parentTriplesMap <#jumpTM>;
   rr:joinCondition [
    rr:child "ID";
    rr:parent "ID" ] ] .
<#jumpTM>
 a rr:TriplesMap ;
 rml:logicalSource [
  rml:source "jump.csv" ;
  rml:referenceFormulation ql:CSV
  ] ;
 rml:subjectMap [
  rr:template ":{ID}-{MARK}" ] ;
 rr:predicateObjectMap [
  rr:predicate :date ;
  rml:objectMap [
   rml:reference "DATE" ] ] ;
 rr:predicateObjectMap [
  rr:predicate :mark ;
  rml:objectMap [
   rml:reference "MARK";
   rr:datatype xsd:float
] ] ] .
```

Listing 2: RML mapping rules translated from List 1.

It is usually more common to use the abbreviated syntax, that needs none of the keys abovementioned (line 8,23-24). Following this alternative, the first element of the array corresponds to the predicate, the second to the object, and optionally, the language tag (line 8) or datatype (line 24). Lastly, join conditions may be used when the desired object is the subject of another mapping set (lines 9-16). This condition requires the name of the target mapping set (line 11), and a similarity function, (usually `equal`). This function evaluates when the data values of the source data specified (lines 15-16) are the same to create the triple. The input parameters for this funciton that refer to the data values are `str1` for the current mapping set, and `str2` for the referencing mapping set.

Subjects, predicates, objects and graphs are terms that can be generated as constant values (i.e. the same term is always generated in all triples) or dynamic values (i.e. the term changes with the data value, that is enclosed inside "`$()`"). The YARRRML serialisation[13] also includes description of target data output (`targets`) [19], and application of functions[14] (`function`).

The YARRRML features are translated to RML (List 2) as follows: mapping rule sets are denoted by `rr:TriplesMap` (lines 1-2); input data sources with `rml:LogicalSource` (lines 3-6); subjects with `rr:SubjectMap` (lines 7-10); named graphs with `rr:graphMap` (lines 9-10); predicate-object pairs with `rr:PredicateObjectMap` (lines 11-22,31-40); language tags with `rr:language` (line 15); datatypes with `rr:datatype` (line 39); joins with `rr:joinCondition` (lines 16-22). Term maps can be divided in three categories: `rr:constant` for constant values, `rml:reference` for data fields and `rr:template` for terms that have a constant value and one or more data fields enclosed by "{}".

4 Extending YARRRML

We extend the YARRRML serialisation to support the RDF-star construction and two updates: the dynamic datatypes and language tags, and a shortcut for join conditions. These are recent features in RML that so far were not considered in YARRRML. To illustrate the extensions, we use a CSV file as input (List 3).

```
ID , PERSON      , COUNTRY , MARK , DATE                 , DATE-TYPE
1  , Lisa Ryzih  , de      , 4.40 , 2022-03-21           , date
2  , Xu Huiqin   , zh      , 4.55 , 2022-03-19T17:23:37  , dateTime
```

Listing 3: Contents of the `jump-source` logical source.

4.1 YARRRML-star

RDF-star introduces the notion of RDF-star triples, i.e. triples that are subjects or objects of another triple. These triples are enclosed using "<<" and ">>", and can be (1) quoted, if they only appear in a graph embedded by another triple (List 4 lines 2,4); or (2) asserted, if the quoted triple is also generated outside the triple where it is quoted (lines 1-4). We extend YARRRML to specify how we can construct RDF-star graphs, aligned with the RML-star specification [14].

```
1  :1 :jumps 4.40 .
2  << :1 :jumps 4.80 >> :date "2022-03-21" .
3  :2 :jumps 4.55 .
4  << :2 :jumps 4.85 >> :date "2022-03-19T17:23:37" .
```

Listing 4: RDF-star triples.

RDF-star triples can be created in YARRRML-star (List 5) by referencing existing Triples Maps with the tags (1) `quoted` for quoted asserted triples (line 10) and (2) `quotedNonAsserted` for quoted non-asserted triples. The triple that the rule set `jumpTM` creates is used as subject in the rule set `dateTM`, creating RDF-star triples (List 4). The equivalent mapping rules in RML are shown in List 6.

[13] `https://rml.io/yarrrml/spec/`

[14] `https://rml.io/yarrrml/spec/#functions`

```
mappings:
 jumpTM:
  sources: jump-source
  subjects: :$(ID)
  predicateobjects:
   - [:jumps, $(MARK)]
 dateTM:
  sources: jump-source
  subjects:
   quoted: jumpTM
  predicateobjects:
   - [:date, $(DATE)]
```

Listing 5: YARRRML-star mapping rules.

```
<#jumpTM>
  a rr:TriplesMap ;
  rml:logicalSource :jump-source;
  rml:subjectMap [
    rr:template ":{ID}" ] ;
  rr:predicateObjectMap [
    rr:predicate :jumps ;
    rml:objectMap [
      rml:reference "MARK" ] ] .
<#dateTM>
  a rr:TriplesMap ;
  rml:logicalSource :jump-source ;
  rml:subjectMap [
    rml:quotedTriplesMap <#jumpTM> ];
  rr:predicateObjectMap [
    rr:predicate :date ;
    rml:objectMap [
      rml:reference "DATE" ] ] .
```

Listing 6: RML-star mapping rules translated from List 5.

4.2 Additional updates

We enable YARRRML-star with other new features that have been incorporated into RML in the last years. We extend YARRRML to assign datatypes and language tags dynamically to objects. They are generated with the data values, in the following examples the datatype is generated dynamically with the data field `DATA-TYPE` (List 7), and the language tag with `COUNTRY` (List 8). They translates into RML as Lists 9 and 10 show.

```
- [:date, $(DATE), xsd:$(DATE-TYPE)]
```

Listing 7: Dynamic datatype.

```
- [:name, $(PERSON), $(COUNTRY)~lang]
```

Listing 8: Dynamic language tag.

```
rr:predicateObjectMap [
 rr:predicate :jumpsOnDate ;
 rml:objectMap [
  rml:reference "DATE";
  rml:datatypeMap [
   rr:template "xsd:{DATE-TYPE}"]]];
```

Listing 9: Dynamic datatype in RML translated from List 7.

```
rr:predicateObjectMap [
 rr:predicate :name ;
 rml:objectMap [
  rml:reference "PERSON";
  rml:languageMap [
   rml:reference "COUNTRY"]]];
```

Listing 10: Dynamic language tag in RML translated from List 8.

We also incorporate a shortcut for specifying join conditions (List 11). This shortcut follows the functions' syntax[14]. It is specified as the function `join` that takes as parameters the mapping identifier (with the `mapping=` parameter key) and the similarity function to perform the join. This function can, in turn, take as parameters data values. Quoted and non-asserted triples can also be generated within join conditions by using the parameter keys `quoted=` and `quotedNonAsserted=` respectively. In the example, the join condition is performed using the `equal` function to create as objects the subjects of the mapping set `jumpTM` if the values of the fields `ID` from source and `ID` from target mapping set are the same.

All the described updates have been proposed for the YARRRML specification and is currently under review by the KG Construction Community Group[15].

```
- predicates: :jumps
  objects:
   - function: join(mapping=jumpTM, equal(str1=$(ID), str2=$(ID)))
```

Listing 11: Abbreviated syntax for join conditions.

5 Yatter

Yatter [5] is a new open-source bi-directional YARRRML translator that supports the aforementioned new features. Yatter receives as input a mapping document in the YARRRML serialisation and the desirable output format (R2RML or RML), or the other way around.

Algorithm 1 presents the procedure implemented by the system to translate an input YARRRML mapping document into [R2]RML. First, the namespaces defined in YARRRML are added together with a set of predefined ones (e.g., `foaf`[16]`, rml`[17]`, rdf`[18]) that are used in by [R2]RML. Second, functions, targets, and databases are identified in the entire YARRRML document and translated into RML. Each of them generates a global identifier mapped into a hash table that can be used by any `rr:TermMap`. For external source declaration, their identifiers are also mapped into a hash table for the next steps. Regarding the RDF-star support, the mapping rules are parsed to identify if it contains `quoted` or `quotedNonAsserted` keys. This determines if the translation requires producing RML-star mapping rules. If true, a hash table is also created for the mapping instances of `rml:NonAssertedTriplesMap`.

In YARRRML, lists of sources and subjects maps can be defined within the same triples map, but [R2]RML triples maps may contain only one source and one subject. Hence, for each mapping document, a list of sources and subjects is first collected and then translated depending on the desirable output format ([R2]RML[-star]). Nevertheless, multiple predicate maps and object maps are allowed within the same triples map, and they are directly translated to RML. Finally, the cartesian product of sources and subject maps together with the predicate object maps is combined to generate the desirable triples map. Before returning the mapping rules, Yatter validates that the generated output is a valid RDF graph.

[15] `https://github.com/kg-construct/yarrrml-spec/pull/4`
[16] `http://xmlns.com/foaf/0.1/`
[17] `http://semweb.mmlab.be/ns/rml#`
[18] `http://www.w3.org/1999/02/22-rdf-syntax-ns#`

Algorithm 1: YARRRML-star translation algorithm

```
Result: [R2]RML mapping document
$input\_m \longleftarrow yarrrml\_rules$;
$format \longleftarrow output\_format$;
$output\_m \longleftarrow \emptyset$;
$output\_m.add(translate\_prefixes(input\_m))$;
if $format == RML$ then
    $output\_m.add(translate\_functions(input\_m))$;
    $output\_m.add(translate\_targets(input\_m))$;
    $is\_star, non\_asserted\_maps \leftarrow analyze\_rml\_star(input\_m)$;
end
$output\_m.add(translate\_databases\_access(input\_m))$;
$ext\_sources \leftarrow get\_external\_sources(input\_m)$;
for $tm \in M.get\_triples\_map()$ do
    $source\_list \leftarrow translate\_source(format, get\_source\_list(ext\_sources, tm))$;
    $subject\_list \leftarrow translate\_subject(is\_star, format, get\_subject\_list(tm))$;
    $predicates\_objects \leftarrow translate\_predicates\_objects(is\_star, format, tm)$;
    for $s \in source\_list$ do
        for $subj \in subject\_list$ do
            $m \leftarrow combine(s, subj, predicates\_objects, non\_asserted\_maps))$;
            $output\_m.add(m)$;
        end
    end
end
return $validate(output\_m)$;
```

Although YARRRML leaves the RDF-based syntax of the mapping rules to be processed only by the knowledge graph construction systems, we also provide a human-readable output considering previous experiences [4, 7, 6]. This helps knowledge engineers in complex data integration contexts to easier understand if the mapping document in YARRRML represents the desirable rules of [R2]RML and, hence, if the constructed knowledge graph will be correct or not. Thus, the output mapping follows a Turtle-based syntax, using predicate object lists within blank node properties[19], as recommended by the [R2]RML specifications. We also ensure the same mapping rules' order as they are defined in YARRRML. Functions, targets, and databases appear first, while for each `rr:TriplesMap`, the sequence is: source, subject map, and the set of predicate object maps.

The source code of Yatter is openly available under Apache 2.0 license[5]. Following open science best practices, each release automatically generates a dedicated DOI to ensure reproducibility in any experimental evaluation[20]. The development is under continuous integration using GitHub Actions and the YARRRML test-cases (Section 6) have more than 80% code coverage. Yatter is available through PyPi as a module[21] to be easily integrated in any Python development.

[19] `https://www.w3.org/TR/turtle/#unlabeled-bnodes`
[20] `https://doi.org/10.5281/zenodo.7024500`
[21] `https://pypi.org/project/yatter/`

6 Validation

We validate the extensions to YARRRML and the developed implementation by proposing and testing a set of test cases, and comparing to other proposed user-friendly serialisations and corresponding systems.

6.1 YARRRML Test Cases

Test cases are a common method to evaluate the conformance of a system [1, 12]. To the best of our knowledge, previous R2RML [20] and RML [12] test cases were not translated to any human-friendly serialisation (e.g., YARRRML). Relying on [R2]RML test cases, we propose a set of representative test cases (including also the new features presented in this work) to assess the conformance of any YARRRML translator system. The proposed test cases require to be two-fold defined: to cover the complete vocabulary of the serialisation, and also have the flexibility to declare the rules (e.g., shortcuts or location of the keys).

We follow a systematic methodology for creating the YARRRML test cases. We analyzed the [R2]RML test cases and observed that several assess correct data generation. Since YARRRML serves as user-friendly serialisation for another mapping language, the focus of its test cases is not on assessing data correctness, but on covering the language expressiveness. Hence, we select 15 R2RML test cases that cover the R2RML features and manually translate them into YARRRML. Since RML is a superset of R2RML, it introduces modifications with respect to R2RML to include the definition of heterogenous datasets (e.g., `rr:LogicalTable` is superseded by `rml:LogicalSource`). We propose 8 new test cases to cover these features.

For features not covered by the RML test cases, we follow a similar procedure. We inspected the RML-star test cases [3], and translated to YARRRML the ones that provide a complete coverage of this extension. From the 16 test cases proposed to assess the conformance of RML-star, we adapt 6. Finally, as there are still no test cases proposed for RML-Target, RML-FNML, RML dynamic language tags and datatypes, we proposed another 21 test cases to cover them.

In total, we defined 50 YARRRML test cases and their corresponding translation to RML or R2RML. They are openly available[22] to be used by any YARRRML-compliant system. Yatter passes all test cases successfully.

6.2 Serialisations Comparison

We compare a set of user-friendly serialisations and languages, namely SMS2 [17], XRM [21], ShExML [11] and YARRRML [13] incorporating the updates described in Section 4, regarding their expressiveness. To that end, we study 15 features that tackle usual characteristics and functionalities in mapping languages. We describe each and discuss how each serialisation addresses it (Table 1).

LF1. Subject Term Type. This feature indicates what kind of RDF[-star] term the language can generate as subject. In RDF, subjects can be IRIs or

[22] `https://github.com/oeg-upm/yarrrml-validation`

Table 1: Features of user-friendly serialisations. BN stands for blank node, L for literal, ST for RDF-star triple, C for constant and D for dynamic. Underlined features indicate the updates of YARRRML-star, while "*" indicates that a feature is possible with the implementation but not explicit in the serialisation.

	ShExML	**SMS2**	**XRM**	**YARRRML-star**
LF1	BN, IRI	BN, IRI, ST	IRI	BN, IRI, ST
LF2	C, D (1..1)	C, D, (1..1)	C, D, (1..1)	C, D (0..1)
LF3	IRI	IRI	IRI	IRI
LF4	C (1..1)	C, D (1..1)	C (1..1)	C, D (1..N)
LF5	BN, IRI, L	BN, IRI, L, ST	IRI, L	BN, IRI, L, ST
LF6	C, D (1..1)	C, D (1..N)	C, D (1..1)	C, D (1..N)
LF7	C, D (0..1)	C (0..1)	C (0..1)	C, D (0..1)
LF8	C, D (0..1)	C, D (0..1)	C (0..1)	C, D (0..1)
LF9	C (0..1)	C (0..1)	C, D (0..N)	C, D (0..N)
LF10	(1..N)	(1..N)	(1..N)	(1..N)
LF11	Input	Input	Input	Input, output
LF12	Yes	No*	No*	Yes
LF13	Yes	No	No	No
LF14	Yes	Yes	No	Yes
LF15	Yes	No	No	Yes

blank nodes, while in RDF-star they can also be RDF-star triples. All serialisations enable the creation of subjects at least as IRIs, SMS2 and YARRRML additionally implement RDF-star triples and, along with ShExML, blank nodes.

LF2. Subject Generation. This feature indicates if subjects can be generated as constant or dynamic values; and how many subject declarations are allowed at a time. In dynamically generated values, the subject value changes with a field in the data source. In our example, the subject uses the field "`ID`" to generate different subject for each row of input data (List 1 line 6) . All serialisations can generate constant and dynamic subjects. For each set of rules, exactly one subject declaration is expected, i.e. one subject for predicate-object pairs. YARRRML can also accept no subject declaration, producing a blank node.

LF3. Predicate Term Type. This feature indicates if the serialisation is able to generate an IRI for a predicate and all serialisations do so.

LF4. Predicate Generation. This feature indicates if predicates can be generated as constant or dynamic values; and how many predicate declarations are allowed at a time. In dynamically generated values, the subject value changes with a field in the data source. SMS2 and YARRRML enable dynamic predicates, and YARRRML is also able to handle more than one predicate, which avoids repeating the same object for different predicates.

LF5. Object Term Type. This feature indicates what kind of RDF[-star] term the serialisation is able to generate as object. The serialisations can generate the same kinds of terms as in subjects (LF1), with the addition of literals.

LF6. Object Generation. This feature indicates if objects can be generated as constant or dynamic values; and how many predicate declarations are allowed at a time. As for subjects, all serialisations can generate constant and dynamic objects. In addition, SMS2 and YARRRML allow more than one at a time, which avoids repeating the same predicate for different objects.

LF7. Datatype. This feature indicates if datatypes can be specified constant or dynamically. All serialisations enable the optional declaration of constant datatypes, but ShExML and YARRRML also enable dynamic datatypes.

LF8. Language Tag. This feature indicates if language tags can be constant or dynamic. Just as for datatypes, all serialisations enable the optional declaration of constant language tags, XRM is the only not allowing dynamic.

LF9. Named Graph. This feature indicates if named graphs can be assigned to the generated statements and how (constant or dynamically). All serialisations enable their optional declaration as constant IRI. XRM and YARRRML also enable more than one graph assignation, and allow dynamic values.

LF10. Data references. This features indicates how many data references a term can contain when generated dynamically (i.e. when its value changes with the input data). It applies to subjects, predicates, objects, datatypes, language tags and named graphs when the serialisation allows dynamic generation. All serialisations allow more than one data reference for dynamic generation.

LF11. Data Description. This feature indicates if the input or output data (e.g., format, iteration, name, path, etc.) can be described. All serialisations can describe input data source, and YARRRML also provides the output data source.

LF12. Data Linking. This feature indicates if explicit data linking (e.g. join, fuzzy linking, etc) can be performed with mapping rules. ShExML and YARRRML provide specific features for this end; in XRM and SMS2, however, it is not explicit, but it is possible by using SQL queries.

LF13. Nested Hierarchies. This feature indicates if different levels of a hierarchy source can be accessed in the same data iteration. ShExML is the only language that implements this feature. It is not implemented in YARRRML since is not supported in RML either as a language feature in the time of writing.

LF14. Functions. This indicates if data transformations are applicable to input data (e.g. lowercase). XRM is the only serialisation not supporting it.

LF15. Conditions. This feature indicates if a statement is generated or not depending on a condition. Only ShExML and YARRRML implement this.

Discussion. All serialisations offer a rich variety of mapping features, but ShExML and YARRRML have a richer selection. SMS2 leverages the SPARQL syntax, lowering the learning curve for SPARQL users. At the same time, data processing is limited to basic SPARQL functions and the user is unable to integrate custom ones. XRM is designed to mimic natural language and adds minimal overhead with its own syntax keywords, which also makes it easy-to-learn, but provides a more limited variety of features.

Table 2: Features of the systems that support the different languages.

	ShExML translator	**Stardog**	**XRM translator**	**YARRRML parser**	**Yatter**
SF1	Open Source	Closed source	Closed source	Open Source	Open Source
SF2	Java	Java	Java	Javascript	Python
SF3	RDB, CSV, JSON, XML, RDF	RDB, NoSQL, CSV, JSON, GraphQL	RDB, CSV, XML	RDB, NoSQL, CSV, JSON, XML	RDB, NoSQL, CSV, JSON, XML
SF4	ShExML	R2RML, SMS, SMS2	XRM	YARRRML, R2RML, RML	YARRRML, R2RML, RML
SF5	RML, RDF	RDF	R2RML, RML, CSVW, CARML	R2RML, RML, YARRRML	R2RML, RML, YARRRML
SF6	N/A	Yes	N/A	No	Yes
SF7	Yes	Yes	N/A	No	Yes
SF8	Yes	N/A	N/A	No	Yes

6.3 Systems Comparison

We also compare the systems that support the aforementioned serialisations: ShExML translator[9], Stardog[3] (with focus on how Stardog maps data sources to RDF graphs, using R2RML or SMS2), XRM translator [21], and YARRRML-parser[23] and our system, Yatter [5]. We study 8 system features to draw conclusions about them including:

SF1. Availability. Stardog and XRM are commercial systems and their implementation is not available. ShExML Java library[9], YARRRML-parser[23] and Yatter[24] are all available as GitHub repositories.

SF2. Programming Language. ShExML, XRM and Stardog are built in Java, YARRRML-parser in Javascript and Yatter in Python.

SF3. Input Data Sources. This feature indicates the data source formats that the system can translate, given the corresponding mapping rules. All systems support relational databases and CSV files as input data sources. Only Stardog and Yatter support NoSQL data sources.

SF4. Input Serialisation. This feature indicates the input mapping serialisation. All systems support their corresponding mapping serialisation. Additionally, Stardog can transform R2RML mapping rules to RDF graphs, whereas both YARRRML systems can translate R2RML or RML files into YARRRML.

SF5. Output Serialisation. This indicates the output mapping serialisation. XRM and YARRRML systems translate their mapping rules to [R2]RML, while XRM also supports CARML and CSVW. Stardog directly constructs the RDF graph. ShExML generates both RML mapping rules and RDF graphs.

[23] `https://github.com/RMLio/yarrrml-parser`

[24] `https://github.com/oeg-upm/yatter/`

SF6. RDF-star Support. Only Stardog and Yatter support this feature. Stardog added RDF-star statement support in one of their latest releases using the "Edge Properties" configuration. Yatter improves upon YARRRML-parser by also enabling the construction of RDF-star graphs.

SF7. Dynamic Language Tag Support. ShExML, Yatter and Stardog provide support for this feature.

SF8. Dynamic Datatype Tag Support. Yatter and ShExML are the only systems that enable the reference of datatypes dynamically.

Additionally, based on the YARRRML test cases, we develop the corresponding test cases for the other analyzed serialisations. The results of the conformance test of the analysed systems are presented online[22].

Discussion. All systems provide a solid user experience and are -mostly- highly conformant with their corresponding serialisation. ShExML is especially useful for integrating different data sources and formats, but lacks RDF-star support, writing functions results cumbersome and the translation to RML is incomplete. Stardog works smoothly with its proprietary Stardog databases, but managing several different sources becomes complex as a different mapping rule set is required per source. XRM is installed within a coding editor, and helps actively the writing process with suggestions and warnings. YARRRML-parser supports most of the functionalities that are also implemented in Yatter but still it does not support the latest RML features. YARRRML-parser translates functions to non-standard set of RML rules, while our implementation supports the specification proposed by the W3C CG on Knowledge Graph Construction[25].

6.4 Use Cases

We present two use cases with YARRRML-star and Yatter:

Constructing KGs for Research-Performing Organizations. In a previous work [4] we used YARRRML-star and Yatter to support the creation of KGs for research supporting organizations with R2RML. Thanks to this setup, we created a fluent and iterative pipeline for testing and debugging the created mapping rules in a complex environment, where almost 2000 tables were mapped into RDF. Additionally, the easy-to-read RML outcome helped the knowledge engineer to easily identify and fix errors during the construction of the KGs.

The EU Public Procurement Data Space (PPDS). The EU PPDS aims to construct a decentralized KG[26], declaratively mapping the procurement data from each EU member state into the e-Procurement Ontology (ePO)[27]. YARRRML-star is the selected serialisation to ensure the maintainability of the KG construction, and it is currently used together with Yatter to develop the initial pilots. In the latest pilots, open Spanish public procurement data extracted from their national platform was mapped, where the mapping rules[28] contain dynamic language tags and complex XPath expressions to extract the desirable data.

[25] `https://w3id.org/kg-construct/rml-fnml`

[26] `https://europa.eu/!qx9WxQ`

[27] `https://docs.ted.europa.eu/EPO/latest/`

[28] `https://github.com/oeg-upm/yatter/tree/main/test/projects/PPDSTC`

7 Conclusions and Future Work

In this paper, we present YARRRML-star, an extension of YARRRML serialisation to fully cover the RML specification supported by a new translator, Yatter. Additionally, we compare YARRRML-star with other human-friendly mapping serialisations in terms of language features and system support over a set of conformance test cases. We demonstrated the impact of our approach over two real use cases, situating YARRRML-star and Yatter as a promising setup for constructing knowledge graphs in complex environments.

In future work, we plan to extend YARRRML-star to support collections and containers[29] in both the serialization and in Yatter. We are also planning to include more test cases to verify correctness in the inverse translation from RML or R2RML mapping rules to YARRRML-star.

Acknowledgement

The work presented in this paper is partially funded by Knowledge Spaces project (Grant PID2020-118274RB-I00 funded by MCIN/AEI/ 10.13039/501100011033) and partially supported by Flanders Make, the strategic research centre for the manufacturing industry. David Chaves-Fraga is supported by the Madrid Government (Comunidad de Madrid-Spain) under the Multiannual Agreement with Universidad Politécnica de Madrid in the line Support for R&D projects for Beatriz Galindo researchers, in the context of the V PRICIT (Regional Programme of Research and Technological Innovation).

References

1. Arenas-Guerrero, J., Iglesias-Molina, A., Chaves-Fraga, D., Garijo, D., Corcho, O., Dimou, A.: Morph-KGC^{star}: Declarative generation of RDF-star graphs from heterogeneous data. Semantic Web (Under Review) (2023)
2. Chatterjee, A., Nardi, C., Oberije, C., Lambin, P.: Knowledge graphs for covid-19: An exploratory review of the current landscape. Journal of personalized medicine **11**(4), 300 (2021)
3. Chaves, D., Iglesias, A., Garijo, D., Guerrero, J.A.: kg-construct/rml-star-test-cases: v1.1 (May 2022). https://doi.org/10.5281/zenodo.6518802
4. Chaves-Fraga, D., Corcho, O., Yedro, F., Moreno, R., Olías, J., De La Azuela, A.: Systematic Construction of Knowledge Graphs for Research-Performing Organizations. Information **13**(12), 562 (2022)
5. Chaves-Fraga, D., Gonzalez, M., Doña, D.: oeg-upm/yatter (Feb 2023). https://doi.org/10.5281/zenodo.7643310, `https://doi.org/10.5281/zenodo.7643310`
6. Chaves-Fraga, D., Priyatna, F., Cimmino, A., Toledo, J., Ruckhaus, E., Corcho, O.: Gtfs-madrid-bench: A benchmark for virtual knowledge graph access in the transport domain. Journal of Web Semantics **65**, 100596 (2020)

[29] `https://w3id.org/kg-construct/rml-collections-containers`

7. Corcho, O., Chaves-Fraga, D., Toledo, J., Arenas-Guerrero, J., Badenes-Olmedo, C., Wang, M., Peng, H., Burrett, N., Mora, J., Zhang, P.: A high-level ontology network for ict infrastructures. In: International Semantic Web Conference. pp. 446–462. Springer (2021)
8. Das, S., Sundara, S., Cyganiak, R.: R2RML: RDB to RDF Mapping Language. W3C Recommendation, World Wide Web Consortium (W3C) (2012), `http://www.w3.org/TR/r2rml/`
9. Delva, T., Arenas-Guerrero, J., Iglesias-Molina, A., Corcho, O., Chaves-Fraga, D., Dimou, A.: RML-star: A Declarative Mapping Language for RDF-star Generation. In: International Semantic Web Conference, ISWC, P&D. vol. 2980. CEUR Workshop Proceedings (2021), `http://ceur-ws.org/Vol-2980/paper374.pdf`
10. Dimou, A., Vander Sande, M., Colpaert, P., Verborgh, R., Mannens, E., Van de Walle, R.: RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data. In: Proceedings of the 7th Workshop on Linked Data on the Web. vol. 1184. CEUR Workshop Proceedings (2014), `http://ceur-ws.org/Vol-1184/ldow2014\_paper\_01.pdf`
11. García-González, H., Boneva, I., Staworko, S., Labra-Gayo, J.E., Cueva-Lovelle, J.M.: ShExML: improving the usability of heterogeneous data mapping languages for first-time users. PeerJ Computer Science **6**, e318 (nov 2020). https://doi.org/10.7717/peerj-cs.318
12. Heyvaert, P., Chaves-Fraga, D., Priyatna, F., Corcho, O., Mannens, E., Verborgh, R., Dimou, A.: Conformance test cases for the rdf mapping language (rml). In: Iberoamerican Knowledge Graphs and Semantic Web Conference. pp. 162–173. Springer (2019)
13. Heyvaert, P., De Meester, B., Dimou, A., Verborgh, R.: Declarative rules for linked data generation at your fingertips! In: European Semantic Web Conference. pp. 213–217. Springer (2018)
14. Iglesias-Molina, A., Arenas-Guerrero, J., Delva, T., Dimou, A., Chaves-Fraga, D.: RML-star. W3C Draft Community Group Report (May 2022), `https://kg-construct.github.io/rml-star-spec/`
15. Prud'hommeaux, E., Labra Gayo, J., Solbrig, H.: Shape expressions: An RDF validation and transformation language. In: Proceedings of the 10th International Conference on Semantic Systems (2014)
16. Rojas, J.A., Aguado, M., Vasilopoulou, P., Velitchkov, I., Van Assche, D., Colpaert, P., Verborgh, R.: Leveraging semantic technologies for digital interoperability in the european railway domain. In: The Semantic Web–ISWC 2021: 20th International Semantic Web Conference, ISWC 2021, Virtual Event, October 24–28, 2021, Proceedings 20. pp. 648–664. Springer (2021)
17. Stardog: Sms2 (stardog mapping syntax 2) (2022), `https://docs.stardog.com/virtual-graphs/mapping-data-sources`
18. Van Assche, D., Delva, T., Haesendonck, G., Heyvaert, P., De Meester, B., Dimou, A.: Declarative rdf graph generation from heterogeneous (semi-) structured data: A systematic literature review. Journal of Web Semantics p. 100753 (2022)
19. Van Assche, D., Delva, T., Heyvaert, P., De Meester, B., Dimou, A.: Towards a more human-friendly knowledge graph generation & publication. In: ISWC2021, the International Semantic Web Conference. vol. 2980. CEUR (2021)
20. Villazón-Terrazas, B., Hausenblas, M.: R2RML and Direct Mapping Test Cases. W3C Note, W3C (2012), `http://www.w3.org/TR/rdb2rdf-test-cases/`
21. Zazuco: Expressive rdf mapper (xrm) (2022), `https://zazuko.com/products/expressive-rdf-mapper/`