YARRRML tutorial

1 Introduction

YARRRML is a language for defining mappings between a range of data formats and RDF. The object of this study is to investigate the user experience with YARRRML. For this, we use three data formats: CSV, JSON and XML.

In fact, YARRRML is a more user-friendly alternative to a previous mapping language, RML. When using YARRRML, it is first translated into RML, and the RML is then used as a mapping language to generate the RDF.

In Section 2 we provide an overview of those YARRRML features used in our study; these are only a subset of the language as a whole. When using YARRRML with JSON data, we need to use JSONPath expressions. Similarly, when working with XML, we need to use XPath expressions. In Section 3 we explain those JSONPath and XPath features used in the study. Again, these are only a subset of the total. Sections 4, 5 and 6 describe how YARRRML works with CSV, JSON and XML.

This tutorial contains everything you need to know to undertake the study. Please read through the tutorial before undertaking the study. However, there is no need to memorize the tutorial. You will be free to refer to it at any time during the study.

2 Overview of YARRRML

This section outlines the structure of a YARRRML document. A more detailed description of YARRRML features will be provided in the examples in Sections 4, 5 and 6.

A YARRRML document begins with a set of RDF prefixes. The remainder of the document then consists of one or more mappings. Each mapping is given a name and defines transformations from data to RDF entities. The mapping begins by specifying the data source, i.e. the data file being used. In the case of JSON and XML, the source statement also defines an iterator which specifies over which elements of the data the mapping will be executed. In the case of CSV no iterator is required; the mapping will iterate over the rows of the file.

The mapping then specifies a transformation from data element to IRI to form the subject of one or more triples. This is then normally followed by one or more pairs of predicates and objects. There is an exception to this which will be noted in Section 4 below. In our study the subject and object will be derived from the data. In some cases the required subject or object will be URLs present in the data and in some cases the object will be an integer or character string. In other cases, the subject or object will be an IRI which will be formed from a character string in the data. In our study the predicate will always be given in the YARRRML, i.e. not derived from the data.

A YARRRML document may contain more than one mapping. These mappings may all use the same data sources. Alternatively, they may use different data sources; in this way it is possible to create triples with subjects and objects derived from different data sources. It is also possible to define conditions for the creation of a triple. In particular, we use the 'equal' condition when we want to compare data from different files.

With JSON and XML data we use JSONPath and XPath in two situations. Firstly, when specifying the iterator in the source statement. As noted above, the iterator describes with what part of the data we are working; an example might be the elements of a JSON array. Secondly, we need to use a path expression to indicate particular data items to be used to form the subjects and objects of a triple. With CSV data, as already noted, we iterate over the rows; to indicate particular data items, we use the first row as column headings.

The details of the YARRRML syntax are explained in Sections 4, 5 and 6. However, as far as possible, the questions are not concerned with syntactic details. We are concerned in this study with how participants manipulate the concepts underlying YARRRML mappings.

3 JSONPath and XPath

In Table 1 we explain the JSONPath and XPath features used in our tutorial example, or that you may wish to use in your answers. Please note that these represent a small subset of JSONPath and XPath.

Table 1: JSONPath and XPath features used in the study

JSONPath	**XPath**	**Description**
$	/	The root object / element in the data.
@	.	The current object / element.
.	/	Child operator, i.e. when between two objects indicates that the rightmost is the child of leftmost.
..	//	Recursive descent: searches recursively for the JSON name or XML tag, e.g. *..identifier* or *//identifier* .
*	*	Wildcard
[]	not applicable	A JSON array
not applicable	@	Selects attributes

4 YARRRML with CSV

To illustrate the use of YARRRML with CSV data, we take a small example. We have two CSV files: one containing company data, the other containing industry data. The companies file, ‘companies.csv’, contains information about three companies: name of company; annual revenue in millions of pounds sterling; and a code to indicate the sector in which the company operates. The industries file contains information about the three industries: the industry name and the industry code. In each file, the first row contains a header specifying the nature of the data in each column. The two files are shown in Figures 1 and 2.

```
company,revenue,industryCode
AceCo,2420,fi
BuildCo,10010,co
CableCo,990,te
```

Figure 1: companies.csv

```
industry,code
telecoms,te
finance,fi
construction,co
```

Figure 2: industries.csv

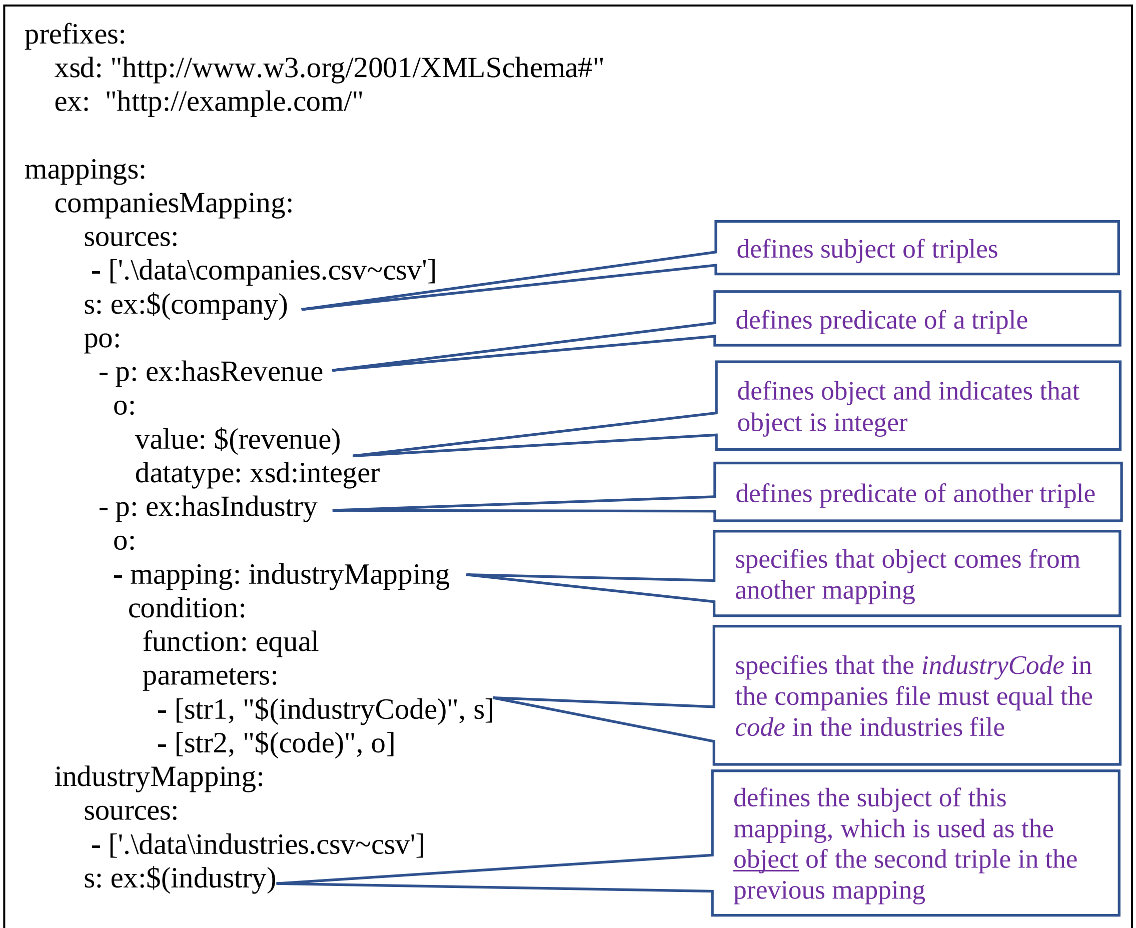
We wish to create an ontology as shown in Figure 3. The predicate ex:hasRevenue is used to link companies to their revenues, represented as integers; this is entirely defined by the companies file. However, we wish also to link companies and their industries, and for this we need compare ‘industryCode’ in the companies file with ‘code’ in the industries file. For conciseness, in Figure 3 we have used the prefix ‘ex:’ and written the revenue figures simply as integers. In the RDF created by YARRRML the prefix will be expanded and revenue figures written as, e.g.:

"2420"^^http://www.w3.org/2001/XMLSchema#integer

```
ex:AceCo     ex :hasIndustry  ex:finance ;
             ex :hasRevenue   2420 .
ex :BuildCo  ex :hasIndustry  ex :construction ;
             ex :hasRevenue   10010 .
ex:CableCo   ex:hasIndustry  ex:telecoms ;
             ex:hasRevenue   990 .
```

Figure 3: the required ontology

Figure 4 shows the YARRRML document to create Figure 3 from Figures 1 and 2. After listing the prefixes used, two mappings are defined. The first, companiesMapping, uses the companies.csv file, identified in the sources statement. The ‘~csv’ after the filename indicates the data should be interpreted as CSV data. The mapping defines two triples for each row of the companies data. ‘s:’ is used at the beginning of a line to indicate that the line defines the subject of a triple. In this case the subject is formed by using the company column from companies.csv, and preceding this with the prefix ex:. After this the line with ‘po:’ simply states that the next lines will define the predicate and object of the triple. The immediately following line states that the predicate will be ex:hasRevenue. The following three lines define the object as having a value derived from the revenue column in companies.csv, and that it will be of type xsd:integer. It is necessary to specify that the object is of type integer, because CSV data is of type string, so a conversion is required. Note that, because of the need to specify the data type, the definition of the object uses an expanded form of the syntax with three lines, rather than the more compact one line syntax used to define the subject.



```
prefixes:
  xsd: "http://www.w3.org/2001/XMLSchema#"
  ex:  "http://example.com/"

mappings:
  companiesMapping:
    sources:
     - ['.\data\companies.csv~csv']
    s: ex:$(company)
    po:
     - p: ex:hasRevenue
      o:
        value: $(revenue)
        datatype: xsd:integer
     - p: ex:hasIndustry
      o:
      - mapping: industryMapping
       condition:
        function: equal
        parameters:
         - [str1, "$(industryCode)", s]
         - [str2, "$(code)", o]
  industryMapping:
    sources:
     - ['.\data\industries.csv~csv']
    s: ex:$(industry)
```

defines subject of triples

defines predicate of a triple

defines object and indicates that object is integer

defines predicate of another triple

specifies that object comes from another mapping

specifies that the *industryCode* in the companies file must equal the *code* in the industries file

defines the subject of this mapping, which is used as the object of the second triple in the previous mapping

Figure 4: YARRRML for use with csv files in Figures 1 and 2

So far we have defined one triple, with data entirely from one file. We now wish to define a triple with the same subject, from companies.csv, but object from industries.csv. Because we are using the same subject, no new subject definition is necessary. The predicate is defined as ‘ex:hasIndustry’. To understand how the object is defined, we need to jump ahead to the final four lines of Figure 4, which describe a second mapping, industryMapping. This uses industries.csv as the data file, and has subject the industry name, preceded by the ex: prefix. This is the exception which we mentioned in Section 2. Here, the mapping does not define a predicate or an object, and the subject of this mapping is not used as the subject of a triple, but rather the object of a triple defined in another mapping. If we return to the companiesMapping, the line ‘mapping: industryMapping’ indicates the object of this new triple is to be taken from the industryMapping. However, we do not wish to associate every company with every industry. Therefore, we define a condition. The condition states the industryCode in the companies file must equal the code in the industries file. Note that the ‘s’ in the first line after ‘parameters:’ signifies that the accompanying value - $(industryCode) - comes from the file which provides the subject of the triple, i.e. companies.csv. ‘o’ in the next line signifies that the accompanying value - $(code) - comes from the file which provides the object of the triple, i.e. industries.csv.

5 YARRRML with JSON

To describe how YARRRML works with JSON, we use the same company and industry information from the previous section, represented as the JSON files shown in Figures 5 and 6. The intention is to produce the same ontology as in the previous example, i.e. as shown in Figure 3. The YARRRML is shown in Figure 7.

```
{
        "companies": [
                {
                        "company": "AceCo",
                        "revenue": 2420,
                        "industryCode": "fi"
                },
                {
                        "company": "BuildCo",
                        "revenue": 10010,
                        "industryCode": "co"
                },
                {
                        "company": "CableCo",
                        "revenue": 990,
                        "industryCode": "te"
                },
        ]
}
```

Figure 5: companies.json

```
{
        "industries": [
                {
                        "industry": "telecoms",
                        "code": "te"
                },
                {
                        "industry": "finance",
                        "code": "fi"
                },
                {
                        "industry": "construction",
                        "code": "co"
                }
        ]
}
```

Figure 6: industries.json

```
prefixes:
  xsd: "http://www.w3.org/2001/XMLSchema#"
  ex:  "http://example.com/"

mappings:
  companiesMapping:
    sources:
     - ['.\data\companies.json~jsonpath','$.companies[*]']
    s:
      value: ex:$(company)
    po:
     - p: ex:hasRevenue
       o:
         value: $(revenue)
         datatype: xsd:integer
     - p: ex:hasIndustry
       o:
       - mapping: industryMapping
         condition:
           function: equal
           parameters:
             - [str1, "$(industryCode)", s]
             - [str2, "$(code)", o]
  industryMapping:
    sources:
     - ['.\data\industries.json~jsonpath','$.industries[*]']
    s: ex:$(industry)
```

This line modified to indicate use of JSONPath and define the iterator

This line also modified to indicate use of JSONPath and define the iterator

Figure 7: YARRRML for use with JSON files in Figures 5 and 6

Figure 7 differs from Figure 4, the CSV example, in just the two lines as indicated in the figure. These are the lines which define the data sources. These lines first specify the file, which in each case is a JSON file. This is followed by '~jsonpath', to indicate that JSONPath expressions are used to traverse the JSON. Finally, there is an iterator. This indicates that, for the companies file the YARRRML should iterate over the elements of the companies array, and for the industries file, the YARRRML should iterate over the elements of the industries array. In this example, the expressions which define the values to be used for the subjects and objects of triples use single JSON names, and are thus the same as in the CSV case. In more complex examples, more complex JSONPath expressions would be required here.

6 YARRRML with XML

In subsection 6.1, we illustrate the use of YARRRML with XML files which contain tags, but no attributes. In subsection 6.2, we illustrate the use of YARRRML with XML files which contain both tags and attributes.

6.1 YARRRML with tags only

We use the same company and industry information as in the previous two examples, represented by the XML files in Figures 8 and 9, and the intention is to create the same ontology, as was shown in Figure 3.

```
<companies>
	<item>
		<company>AceCo</company>
		<revenue>2420</revenue>
		<industryCode>fi</industryCode>
	</item>
	<item>
		<company>BuildCo</company>
		<revenue>10010</revenue>
		<industryCode>co</industryCode>
	</item>
	<item>
		<company>CableCo</company>
		<revenue>990</revenue>
		<industryCode>te</industryCode>
	</item>
</companies>
```

Figure 8: companies.xml

```
<industries>
	<item>
		<industry>telecoms</industry>
		<code>te</code>
	</item>
	<item>
		<industry>finance</industry>
		<code>fi</code>
	</item>
	<item>
		<industry>construction</industry>
		<code>co</code>
	</item>
</industries>
```

Figure 9: industries.xml

Figure 10 shows the YARRRML. This differs from the previous two examples, in Figures 4 and 7, in the two lines indicating the data sources. These specify that the data will be referenced using XPath; and define the iterators used. In both cases, we iterate over the XML tags <item>…</item>. As with the CSV and JSON examples, the definition of the subject and object values is very simple, requiring only the single XML tags. More complex examples would require more complex XPath statements.

```
prefixes:
  xsd: "http://www.w3.org/2001/XMLSchema#"
  ex:  "http://example.com/"

mappings:
  companiesMapping:
    sources:
     - ['.\data\companies.xml~xpath','/companies/item']
    s:
      value: ex:$(company)
    po:
     - p: ex:hasRevenue
      o:
        value: $(revenue)
        datatype: xsd:integer
     - p: ex:hasIndustry
      o:
      - mapping: industryMapping
       condition:
        function: equal
        parameters:
         - [str1, "$(industryCode)", s]
         - [str2, "$(code)", o]
  industryMapping:
    sources:
     - ['.\data\industries.xml~xpath','/industries/item']
    s: ex:$(industry)
```

This line modified to indicate use of Xpath and define the iterator

This line also modified to indicate use of Xpath and define the iterator

Figure 10: YARRRML for use with XML files in Figures 8 and 9

6.2 YARRRML with tags and attributes

Once again, we use the same company and industry information as in the previous examples. This time we use a combination of XML tags and attributes, as shown for the revised companies.xml and industries.xml files in Figures 11 and 12. The intention is to create the same final ontology, as shown in Figure 3.

```
<companies>
        <company name = “AceCo” revenue = “2420” industryCode = “fi” />
        <company name = “BuildCo” revenue = “10010” industryCode = “co” />
        <company name = “CableCo” revenue = “990” industryCode = “te” />
</companies>
```

Figure 11: companies.xml

```
<industries>
        <industry name = “telecoms” code = “te” />
        <industry name = “finance” code = “fi” />
        <industry name = “construction” code = “co” />
</industries>
```

Figure 12: industries.xml

Figure 13 shows the YARRRML. This is similar to the previous YARRRML examples. As in the last example, we are using XPath; however the iterators have been altered. For the companies file, we iterate over the ‘company’ tags, which are nested within the ‘companies’ tags. For the industries file, we iterate over the ‘industry’ tags, which are nested within the ‘industries’ tags.

In addition, five other lines have been changed from the other examples. In each case where we are selecting a specific piece of data, since we are interested in the values of XML attributes, we use the XPath ‘@’ character to select these attribues. As a result, the ‘@’ character is used five times, including the two occurrences where we are comparing ‘industryCode’ from the companies file and ‘code’ from the industries file.

```
prefixes:
  xsd: "http://www.w3.org/2001/XMLSchema#"
  ex:  "http://example.com/"

mappings:
  companiesMapping:
    sources:
     - ['companiesAttributes.xml~xpath','/companies/company']
    s:
      value: ex:$(@name)
    po:
     - p: ex:hasRevenue
       o:
         value: $(@revenue)
         datatype: xsd:integer
     - p: ex:hasIndustry
       o:
       - mapping: industryMapping
         condition:
           function: equal
           parameters:
             - [str1, "$(@industryCode)", s]
             - [str2, "$(@code)", o]
  industryMapping:
    sources:
     - ['industriesAttributes.xml~xpath','/industries/industry']
    s: ex:$(@name)
```

Figure 13: YARRRML for use with XML files in Figures 11 and 12

END OF TUTORIAL