A ShExML Perspective on Mapping Challenges

Already Solved Ones, Language Modifications and Future Required Actions

Herminio García González - garciaherminio@uniovi.es - @herminio_gg
Introduction

What are the mapping challenges?

• Declarative mapping rules

• Single representation for heterogeneous data sources
  • Modifiability, shareability, efficiency

• However, some mappings are still not reachable

• Defined and agreed by the community

• Need to be solved
Mapping challenges

Summary

- 9 mapping challenges
  - Inputs defined by the community to test different scenarios
  - ShExML mapping language
  - 5 challenges completely solved
  - 2 partially covered
  - 2 unaddressed
Brief ShExML Introduction
ShExML language overview

A flash introduction
ShExML language overview

A flash introduction

Prefixes

PREFIX : <http://example.com/>
ShExML language overview

A flash introduction

Prefixes

Sources (URL + variable name)

PREFIX : <http://example.com/>
SOURCE films_xml_file <https://rawgit.com/herminioqg/ShExML/master/src/test/resources/films.xml>
SOURCE films_json_file <https://rawgit.com/herminioqg/ShExML/master/src/test/resources/films.json>
A flash introduction

**Prefixes**

**Sources (URL + variable name)**

**Iterators + Fields (extract content from input files using queries)**

```shex
PREFIX : <http://example.com/>
SOURCE films_xml_file <https://rawgit.com/herminioog/ShExML/master/src/test/resources/films.xml>
SOURCE films_json_file <https://rawgit.com/herminioog/ShExML/master/src/test/resources/films.json>

ITERATOR film_xml <xpath: //film> {
    FIELD id <id>
    FIELD name <name>
    FIELD year <year>
    FIELD country <country>
    FIELD directors <directors/director>
}

ITERATOR film_json <jsonpath: $.films[*]> {
    FIELD id <id>
    FIELD name <name>
    FIELD year <year>
    FIELD country <country>
    FIELD directors <director>
}
```
ShExML language overview

A flash introduction

Prefixes

Sources (URL + variable name)

Iterators + Fields (extract content from input files using queries)

Expressions to be applied to iterators or to actual values (result will be, accordingly, an iterator or a value)
ShExXML language overview

A flash introduction

Prefixes

Sources (URL + variable name)

Iterators + Fields (extract content from input files using queries)

Expressions to be applied to iterators or to actual values (result will be, accordingly, an iterator or a value)

Shapes (give form to the output graph)
ShExML iteration model

How it works?

• Iterator (tells the engine to traverse all results)
• Fields (actual value/s to be outputted)
• Nested iterators (to traverse down in the hierarchy)
• films_xml.actors.name
• //film[i]/cast/actor[j]/name
• No need to join

ITERATOR film_xml <xpath: //film> {
  FIELD id <$id>
  FIELD name <$name>
  FIELD year <$year>
  FIELD country <$country>
  FIELD directors <crew/directors/director>
  FIELD screenwriters <crew/screenwriter>
  FIELD music <crew/music>
  FIELD photography <crew/photography>
  ITERATOR actors <cast/actor> {
    FIELD name <$name>
    FIELD role <$role>
    FIELD film <../../@id>
  }
  ITERATOR actresses <cast/actress> {
    FIELD name <$name>
    FIELD role <$role>
    FIELD film <../../@id>
  }
}
Addressed Challenges
Datatype map

The problem

• Generate datatype from input data
• Dynamic vs static
• Different possible format inputs
  • http://www.w3.org/2001/XMLSchema#integer
  • xsd:integer
  • integer
  • int
Datatype map

ShExML old syntax

• ShExML allowed to generate static datatypes
• It does not solve the previous case
• We should generalise the existing syntax for dynamic generation

```
ex:Person exPerson:[person.firstname] { 
ex: num [person.num] xsd:integer ; 
}

exPerson:John ex: num 3 .
```
Datatype map

ShExML new syntax

```
ex:Person exPerson:[person.firstname] {
  ex:num [person.num] xsd:integer ;
}
```

• Expand the content generation expression but for datatypes
• It allows different types representations
• Prefixed or not
• Works like subject/object generation expressions but for datatypes

```
ex:Person exPerson:[person.firstname] {
  ex:num [person.num] xsd:[person.dt] ;
}
```

```
exPerson:John ex:num 3 .
```
The problem

• Generate language tag from input data
• Dynamic vs static
• Different possible format inputs
  • en
  • English
Language map

ShExML old syntax

- ShExML allowed to generate static language tags
- It does not solve the previous case
- We should generalise the existing syntax for dynamic generation

```xml
ex:Person exPerson:[person.firstname] { 
  ex:lastName [person.lastname] @en ;
}
```

```xml
exPerson:Jane ex:lastName "Smith"@en .
exPerson:John ex:lastName "Doe"@en .
```
Language map

ShExML new syntax

```
ex:Person ex:Person:[person.firstname] { ex:lastName [person.lastname] @en ;
}
ex:Person ex:Person:[person.firstname] { ex:lastName [person.lastname]@[person.lang] ;
```

- Expand the content generation expression but for datatypes
- It allows different types representations
  - We can use MATCHERS to generate BCP47 conformant tags
- Works like subject/object generation expressions but for datatypes

```
exPerson: Jane  ex: lastName  "Smith"@fr .
exPerson: John  ex: lastName  "Doe"@en .
```
The problem

- Multi-language or multi-datatype values for the same subject
- Additionally, default languages or datatypes

```json
{  
  "firstname": "Doe",
  "firstname": [
    {  
      "label": "John",
      "lang": "en"
    },
    {  
      "label": "John",
      "lang": "fr"
    }
  ]
}
```

```html
exPerson:John  ex:name  "John"@fr , "John"@en .
```
Generate multiple values

ShExML solution

- Multi-value generation bounds language and datatype generation to the current index
- Using old syntax we can generate additional default triples

```shex
ex:Person ex:Person:[person.lastname] {
  ex:name [person.firstname.label]@[person.firstname.lang];
}
```

```shex
ex:Person ex:Person:[person.firstname] {
  ex:name [person.firstname]@en;
  ex:name [person.firstname]@[person.lang];
}
```
Join on literal

The problem

- Joins, by default, generate resources
- There is no way to output literals instead

```json
{
    "people": [
        {
            "id": 1,
            "firstname": "John",
            "affiliation": "Unil"
        },
        {
            "id": 2,
            "firstname": "Jane",
            "affiliation": "Uni2"
        }
    ],
    "author": [
        {
            "id": 1,
            "firstname": "John",
            "familyName": "Doe"
        },
        {
            "id": 2,
            "firstname": "Jane",
            "familyName": "Dane"
        }
    ]
}
```
Join on literal

ShExML solution

• ShExML allows (from its inception) to output resources and literals on joins
• Separation of concerns
• How to extract and transform data
• How to output data

```shex
PREFIX : <http://example.com/>
PREFIX experson: <http://example.com/person/>
PREFIX dbpedia: <http://dbpedia.org/resource/>
PREFIX schema: <http://schema.org/>

SOURCE jsonfile <https://raw.githubusercontent.com/kg-constructor/mapping-challenges/2aad9680cd731fd647b0334a7f400e4278c3/challenges/join-on-literal/input-1/input.json>

ITERATOR author <jsonpath: $.author[*]> {
  FIELD id <id>
  FIELD firstname <firstname>
  FIELD affiliation <affiliation>
}

ITERATOR people <jsonpath: $.people[*]> {
  FIELD firstname <firstname>
  FIELD familyname <familyName>
}

EXPRESSION authors <jsonfile.author UNION jsonfile.people>

EXPRESSION familyName <jsonfile.people.familyname UNION jsonfile.author.firstname JOIN jsonfile.people.firstname>

:Author experson:[authors.id] {
  :affiliation [authors.affiliation] ;
  :lastName [familyName] ;
}
```
Multi-value references

The problem

- How to deal with the expected output in a hierarchical file
- Cartesian product or respect the current relation
- Join condition poses problems in JSON files as it is not possible to go upwards
Multi-value references

ShExML solution

• ShExML allows to nest iterators
• No need for join condition
• More usable
• Thus, verbatim translation

```
ITERATOR lab <jsonpah: $> {
  FIELD labName <labName>

  ITERATOR articles <articles[*]> {
    FIELD title <title>

    ITERATOR authors <authors[*]> {
      FIELD name <name>

      ITERATOR affiliation <affiliation[*]> {
        FIELD label <label>
      }
    }
  }
}
```
Access fields outside iterators

The problem

- How to access upper fields
- JSON path doesn’t allow going upwards
- From cars how to reach owners?
Access fields outside iterators

ShExML solution

• Pushed fields allow to save value information for later use
• Popped fields allow to recover values from pushed fields
• Inspired in xR2RML’s xrr:pushDown
• Pushed and popped like in a stack
• But popped is not gone forever

```xml
ITERATOR records <jsonpath: $.records[*]> {  
  PUSHED_FIELD id <id>
  FIELD enteredBy <enteredBy>
  ITERATOR cars <cars[*]> {  
    FIELD make <make>
    POPPED_FIELD carOwner <id>
  }
}
```

Tells the engine to save the value under "id" name

Tells the engine to recover the value under "id" name and expose the value under "carOwner" name
Access fields outside iterators

Explicit vs implicit

• Delva T. et al. [1] proposed an algorithm that saves iteration information

• No need to push and pop fields, transparent to the user

• But, it saves a lot of information -> Possible bottlenecks

• Further challenge, quantify the best option in terms of:

  • Usability
  • Performance

RDF Collections

The problem

- Generate collections from multi-value references
- Different RDF Collections and Containers
  - List
  - Bag, Seq, Alt

```json
{
  "labName": "AmazingLab1",
  "article": {
    "title": "article1",
    "authors": [
      {
        "name": "Alice"
      },
      {
        "name": "Bob"
      }
    ]
  }
}
```
RDF Collections

ShExML solution

• Indicate the collection in the generation expression

• AS + (RDFList, RDFBag, RDFSeq, RDFAlt)

```shexml
exArticle:article1 a ex:Article ;
ex:hasAuthors ( exAuthor:Alice exAuthor:Bob ) .
```

```shexml
ex:Article exArticle:[labValues.articles.title] {
  a ex:Article ;
  ex:hasAuthors exAuthor:[labValues.articles.authors.name AS RDFList] ;
}
```
Addressed challenges

Coverage summary

Before

- Access fields outside iterators
- Datatype map
- Excel style
- Generate multiple values
- Join on literal
- Language map
- Multivalue references
- Process multivalue references
- RDF Collections

0 % 25 % 50 % 75 % 100 %
Addressed challenges

Coverage summary

- Access fields outside iterators
- Datatype map
- Excel style
- Generate multiple values
- Join on literal
- Language map
- Multivalue references
- Process multivalue references
- RDF Collections
Unaddressed Challenges

And How They Could Be Addressed
Excel style

Possible solution A

• How to process Excel sheets values and styles?
• Preprocess and add more columns with style
• Input as CSV

- No need to change the language and the engine
- Need to preprocess (not ideal for users)
Excel style

Possible solution B

- How to process Excel sheets values and styles?
- Support Excel sheets directly
- How to access styles?
- Need for a specific query language

```plaintext
ITERATOR excel_iterator <perRow> {
    FIELD value <A1>
    FIELD style <A1:TextColor>
}
```

Easy and straightforward for users

Too much design and implementation for a working solution
Process multivalue references

Possible solution A

- How to add data transformation functions in ShExML?
- Add support for FnO functions library
- No need to develop a dedicated function infrastructure

- All function infrastructure outside ShExML language and engine
- More dependencies to users which need to learn a third-party environment
- We lose control of this part
Process multivalue references

Possible solution B

• How to add data transformation functions in ShExML?

• Allow to define inline functions
  • Extension mechanism as semantic actions in ShEx

• All in the same environment

FUNCTION splitFunction <n => n.split(',>')

ex:Tag ex:[lab.articles.tag WITH splitFunction] {
  ex:label [lab.articles.tag WITH splitFunction] ;
}

- No third-party dependencies
- Higher flexibility
- No need to learn another tool

- Higher complexity due to the necessity to know about functional programming
Discussion and Conclusions
Discussion & Conclusions

• Added solutions try to maintain ShExML usability
  • Using similar syntax constructions
• Some challenges were already solved by ShExML
  • Separation of concerns can help to solve some of them
• Further challenges
  • Need to be carefully designed and included in the language
• First step on how the challenges can be solved
  • Solutions from other languages and joint discussion -> Unified solutions
A ShExML Perspective on Mapping Challenges

Already Solved Ones, Language Modifications and Future Required Actions

Herminio García González - garciaherminio@uniovi.es - @herminio_gg