Shape Expressions: An RDF validation and transformation language

Eric Prud'hommeaux
World Wide Web Consortium
(W3C) MIT, Cambridge, MA, USA
eric@w3.org

Jose Emilio Labra Gayo
University of Oviedo
Dept. of Computer Science
C/Calvo Sotelo, S/N
labra@uniovi.es

Harold Solbrig
Mayo Clinic
College of Medicine,
Rochester, MN, USA

ABSTRACT

RDF is a graph based data model which is widely used for semantic web and linked data applications. In this paper we describe a Shape Expression definition language which enables RDF validation through the declaration of constraints on the RDF model. Shape Expressions can be used to validate RDF data, communicate expected graph patterns for interfaces and generate user interface forms. In this paper we describe the syntax and the formal semantics of Shape Expressions using inference rules. Shape Expressions can be seen as domain specific language to define Shapes of RDF graphs based on regular expressions.

Attached to Shape Expressions are semantic actions which provide an extension point for validation or for arbitrary code execution such as those in parser generators. Using semantic actions, it is possible to augment the validation expressiveness of Shape Expressions and to transform RDF graphs in a easy way.

We have implemented several validation tools that check if an RDF graph matches against a Shape Expressions schema and infer the corresponding Shapes. We have also implemented two extensions, called GenX and GenJ that leverage the predictability of the graph traversal and create ordered, closed content, XML/Json documents, providing a simple, declarative mapping from RDF data to XML and Json documents.

Categories and Subject Descriptors

I.2.4 [**Knowledge Representation Formalisms and Methods**]: Representation languages; H.3.5 [**Online Information Services**]: Web-based services

General Terms

Theory

Keywords

RDF, Graphs, Validation, Transformation

1. INTRODUCTION

RDF [17] can be considered the *lingua franca* of semantic web and linked data technologies. RDF has been successfully employed as a data integration language and there are a number of applications using RDF as a database technology or as an interoperability layer.

Despite RDF's emerging popularity, XML and relational databases remain far more widely deployed, in part because those technologies offer good validation tools to define and exchange data conforming to specified schemas.

Most data representation languages used in conventional settings offer some sort of input validation like parsing grammars for domain-specific languages, XML Schema, RelaxNG or Schematron for XML, and DDL for SQL. While the distributed nature of RDF affects the notions of *validity*, tool chains need to be established to publish interface definitions and ensure data integrity.

In the case of RDF, although there are standards for inference like RDF Schema and OWL, these technologies employ Open World and Non-Unique Name Assumptions that creates difficulties for validation purposes [22]. The Shape Expressions language (ShEx) is intended to perform the same function for RDF graphs as Schema languages to XML. It can be used to validate documents, communicate expected graph patterns for interfaces, and generate user interface forms and code.

The syntax and semantics of Shape Expressions are designed to be familiar to users of regular expressions (specially RelaxNG). The conspicuous difference is that RDF data is a set (of triples) while regular expression data is a sequence (of characters). Regular expressions correlate an ordered pattern of atomic characters and logical operators against an ordered sequence of characters. Shape Expressions correlate an ordered pattern of pairs of predicate and object classes and logical operators against an *unordered* set of arcs in a graph.

Throughout the paper we will employ a simple example data model for an issue reporting system with two types of entities: *Issue* and *User*. The data model is represented in Figure 1.

The logical operators in Shape Expressions, grouping, conjunction, disjunction and cardinality constraints, are defined to match as closely as possible their counterparts in regular expressions languages like RelaxNG. Our example can be realized in RelaxNG using the Compact Syntax schema as described in figure 2. Notice that we allow users to have either a name or a combination of given names and one family name.

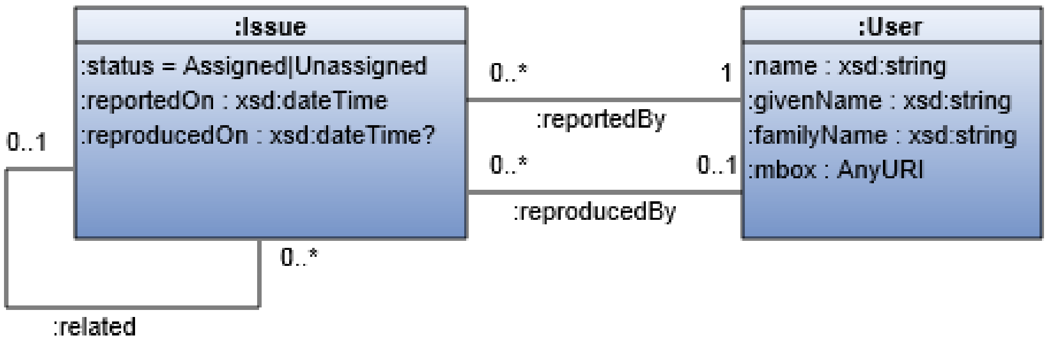

Figure 1: Example data model for issue tracking

```
IssueContent = element :Issue {
  element :status
   {":assigned" |":unassigned" },
  element :reportedBy { UserContent },
  element :reportedOn { xsd:dateTime },
 ( element :reproducedBy { UserContent },
   element :reproducedOn { xsd:date }
 )?
}
UserContent =
 ( element :name {xsd:string}
 | element :givenName {xsd:string}+,
   element :familyName {xsd:string}
 ),
 element :mbox { xsd:anyURI }?
```

Figure 2: Example data model in RelaxNG

In Figure 3 we represent the issue reporting data model using the Shape Expressions language[1]

```
<IssueShape> {
    :status ( :unassigned
              :assigned ),
    :reportedBy @<UserShape>,
    :reportedOn xsd:date,
    ( :reproducedBy @<UserShape>
    , :reproducedOn xsd:date
    )?,
    :related @<IssueShape>*
}
<UserShape> {
    ( foaf:name xsd:string
    | foaf:givenName xsd:string+ ,
      foaf:familyName xsd:string
    ),
    foaf:mbox shex:IRI ?
}
```

Figure 3: Example of Shape Expressions to validate issues

[1]In the examples we omit prefix declarations. Most prefixes are the common ones that can be retrieved from `http://prefix.cc` or a default `http://example.org/`

Figure 4 presents an instance in Turtle syntax that validates against the previous Shape Expressions schema.

```
:Issue1
 :status        :unassigned ;
 :reportedBy   :Bob ;
 :reportedOn   "2013-01-23"^^xsd:date ;
 :reproducedBy :Thompson.J ;
 :reproducedOn "2013-01-23"^^xsd:date .

:Bob
 foaf:givenName "Bob" ;
 foaf:familyName "Smith" ;
 foaf:mbox <mail:bob@example.org> .

:Thompson.J
 foaf:givenName "Joe", "Joseph" ;
 foaf:familyName "Thompson" ;
 foaf:mbox <mail:joe@example.org> .
```

Figure 4: Turtle example

The semantics of Shape Expression validation acts as a type inference system which infers a type (shape) for a given node in an RDF graph. In this example, the validator infers that `:Issue1` has shape `IssueShape` and that `:Bob` and `:Thompson.J` have shape `UserShape`[2].

`:Issue1`	$\rightsquigarrow$	`:IssueShape`
`:Bob`	$\rightsquigarrow$	`:UserShape`
`:Thompson.J`	$\rightsquigarrow$	`:UserShape`

The Shape Expression language extends the validation functionality with the inclusion of semantic actions attached to productions. These semantic actions can provide extensible validation or code callbacks familiar to users of parser generators like yacc or ANTLR.

The paper is structured as follows. The next section introduces the Shape Expressions language. Section 3 defines the abstract syntax. In section 4 we describe the semantics and in section 5 we describe how we can employ Shape Expressions to transform RDF documents to different languages like XML or Json.

[2]The example can be executed using the online RDF Shape Expressions validator http://rdfshape.weso.es.

2. SHAPE EXPRESSIONS

In this section, we present an overview of the Shape Expressions language employing the issue example described in section 1. Further examples of Shape Expressions are available at: `https://www.w3.org/2001/sw/wiki/ShEx_Examples`. At the moment of this writing an W3C Working Group charter about RDF Data Shapes has been created[3] so it is possible that the language will change in the future.

2.1 Labeled Shape Expression

A shape expression is a labelled pattern for a set of RDF Triples sharing a common subject. Syntactically, it is a pairing of a label, which can be an IRI or a blank node, and a rule enclosed in brackets (`{ }`). Typically, this rule is a conjunction of constraints separated by commas (`,`).

As an example, the `IssueShape` described in Figure 3 must have exactly one `ex:status` predicate, and the target of that predicate must be either `ex:unnasigned` or `ex:assigned`. It must have exactly one `ex:reportedBy` predicate whose target will be a valid `UserShape`, and exactly one `ex:reportedOn` predicate whose target must be of type `xsd:dateTime`.

If the shape has a `ex:reproducedBy` predicate, it must have a `ex:reproducedOn` predicate of the appropriate type. Finally, it may have any number of `ex:related` predicates, all of which must conform to the `IssueShape` type.

2.2 Alternatives

The `UserShape` declared in Figure 3 has either exactly one `foaf:name` of type `xsd:string` or a combination of at least one `foaf:givenName` and one `foaf:familyName`.

A valid `UserShape` instance could optionally have one `foaf:mbox` that must be an IRI.

Using alternatives and conjunctions, it is trivial to validate RDF collections. For example, we can declare a list of integers as:

```
<listOfInt> {
   rdf:first xsd:integer
 , ( rdf:rest  ( rdf:nil )
   | rdf:rest @<listOfInt>
   )
}
```

2.3 Cardinality

The `<IssueShape>` example includes a group with a cardinality of 0 or 1.

```
 ( ex:reproducedBy @<UserShape>
 , ex:reproducedOn xsd:dateTime
 )?
```

This requires that the data have neither or both of those properties. The example also contains a repetition cardinality:

```
 ex:related @<IssueShape> *
```

which declares that issues can be related to 0 or `n` other issues. Notice that it is also possible to have recursive references.

Cardinality constraints may be expressed as one of `?, +, *` or as one or two integers in `{}`s.

[3] `http://www.w3.org/2014/data-shapes/charter`

2.4 Matching names

The Shape Expression language contains the following possibilities to match names of properties:

Name	**Example**	**Description**
NameTerm	`foaf:name`	Matches the given IRI (`foaf:name`)
NameStem	`foaf:~`	Matches a IRI that starts with a given IRI. In this case, matches an IRI that starts by the IRI associated with the `foaf` prefix.
NameAny	`- foaf:name`	Matches with any predicate except those excluded by the '-' operator. In this case, any predicate except `foaf:name`

The symbol dot (`.`) matches with any name and can be represented as *NameAny* with an empty set of excluded predicates.

2.5 Matching values

The Shape Expression language contains the following possibilities to match values or objects of triples

Name	**Example**	**Description**
ValueType	`xsd:date`	The value has the type expressed by the IRI
ValueSet	`(:assigned :unassigned)`	The object is one of the list of nodes in the ValueSet
ValueAny	`- xsd:int`	The value has any objects except those excluded by the `-` operator. As in the case of names, the dot `.` lsymbol can be used to represent an any value.
ValueStem	`foaf:~`	The value starts with the IRI
ValueRef	`@<UserShape>`	The value is an IRI or blank node which has the shape expressed by the value reference.

2.6 Start Rule

Some grammar languages provide a starting point for validating documents or generating forms. In Shape Expressions, the starting point is specified by the `start` keyword.

It is not necessary to identify a particular node in the graph for validation operations. Nor is it necessary to provide a `start` point for all operations. For instance, generating a sequence of forms obviously needs to start somewhere, but some documents can be validated by optimistically testing each shape expression against each node in the graph. This exhaustive search is more expensive and raises the possibility that a document validates or executes actions in a way that the author of the document did not intend.

2.7 Semantic Actions

The `<IssueShape>` example above includes both `ex:reportedOn` and `ex:reproducedOn` dateTimes. It would be reasonable in the interest of data quality to ensure that the `ex:reproducedOn` dateTime, if present, were temporally after the `ex:reportedOn` dateTime.

While ShEx itself has no built-in functionality for comparing dateTimes, specific extensions may offer that functionality by means of semantic actions. Semantic actions have the syntax %lang{ code %} which mean the Shape Expression validator calls the processor of language *lang* with the code *code* and the values (s,p,o) of the current triple that is being validated.

The example below includes semantic actions to test date order in either Javascript or SPARQL. In Javascript, the semantic actions (marked as `%js`) check that if there is a `:reproducedOn` value, then it must be bigger than the value of `:reportedOn` which was saved in the variable `report`. The SPARQL semantic action does the same using a `FILTER`.

```
:reportedOn xsd:dateTime
 %js{ report = _.o;
      return true; %},
( :reproducedBy @<EmployeeShape>,
  :reproducedOn xsd:dateTime
  %js{ return _.o.lex > report.lex; %}
  %sparql{ ?s ex:reportedOn ?rpt .
            FILTER (?o > ?rpt) %}
)?
```

Notice that semantic actions provide an extension mechanism which will depend on a validator supporting the corresponding language. In section 5 we will show how we can employ semantic actions to transform RDF to XML and Json using two very simple languages called *GenX* and *GenJ*.

3. ABSTRACT SYNTAX

This section presents the operational semantics of Shape Expressions. The semantics has been inspired by RelaxNG semantics [23].

As in the case of RelaxNG, we define a simplified abstract syntax that covers the kernel of the Shape Expressions Language.

For brevity, we concentrate on the main parts of the Shape Expression language and omit some proposed extensions like shape inclusion and inheritance, incoming arcs and relation predicates, language tagged literals, etc.

$Schema$	::=	$Schema(rules : Set(Shape))$
$Shape$	::=	$Shape(label : Label, rule : Rule)$
$Rule$	::=	$Arc(n : NameClass, v : ValueClass)$
	|	$And(rule_1 : Rule, rule_2 : Rule)$
	|	$Or(rule_1 : Rule, rule_2 : Rule)$
	|	$OneOrMore(rule : Rule)$
	|	$ActionRule(a : Action)$
	|	$Empty$
$Label$	::=	$IRI \mid BlankNode$
$NameClass$	::=	$NameTerm(t : IRI)$
	|	$NameAny(excl : Set(IRI))$
	|	$NameStem(s : IRI)$
$ValueClass$	::=	$ValueType(type : IRI)$
	|	$ValueSet(s : Set(RDFNode))$
	|	$ValueAny(excl : Set(IRI))$
	|	$ValueStem(stem : IRI)$
	|	$ValueRef(l : Label)$
$Action$	::=	$Action(label : Label, code : String)$

The full syntax of Shape Expressions can be transformed to the simplified abstract syntax. For example, there is no need to define optional and star as they can be defined in terms of $OneOrMore$ (+) and $Empty$:

$Optional(rule)$	=	$Or(rule, Empty)$
$Star(rule)$	=	$Or(OneOrMore(rule), Empty)$

A common situation when matching an RDF node against a shape is that the node matches the shape but contains some remaining triples. By default the Shape Expression language employs open shapes which means that it will match even if there are some remaining triples. It is also possible to use closed shapes in which case, it only matches if there are no remaining triples.

In section 4, we define the semantics using closed shapes because any open shape can be defined in terms of a closed shape adding the conjunction "`. .*`" which means that the rule can also match if there are remaining triples.

4. SEMANTICS

In this section we define the semantics of the Shape Expression language. The semantics is defined formally using axioms and inference rules. Axioms are propositions that are provable unconditionally. An inference rule consists of one or more antecedents and exactly one consequent. An antecedent is either positive or negative. If all the positive antecedents of an inference rule are provable and none of the negative antecedents are provable, then the consequent of the inference rule is provable. We begin with some basic definitions on RDF and Shape typings.

4.1 RDF definitions

An RDF model can be defined as a set of RDF triples. Each RDF triple is a three-tuple $\langle s, p, o\rangle \in (\mathcal{I} \cup \mathcal{B}) \times \mathcal{I} \times (\mathcal{I} \cup \mathcal{B} \cup \mathcal{L})$, where $\mathcal{I}$ is a set of IRIs, $\mathcal{B}$ a set of blank nodes, and $\mathcal{L}$ a set of literals. The components s, p, o are called, the subject, the predicate, and the object of the triple, respectively. We declare the following operations on RDF graphs:

$\circledcirc$	=	Empty graph (empty set of triples)
$[t]$	=	Singleton set with triple t
$t \rtimes g$	=	Result of adding triple t to graph g
$g.triples(s)$	=	subset of g with triples $\langle s, \_, \_\rangle$
$g_1 \oplus g_2$	=	Union of graphs g_1 and g_2.

Notice that in the semantics, $g_1 \oplus g_2$ will be used to divide a given graph g into two sub-graphs g_1 and g_2 whose union is g.

4.2 Shape typings

The validation semantics can be seen as a type inference system where each shape defines a candidate type. A type will be defined as a mapping from subjects $(\mathcal{I} \cup \mathcal{B})$ to Shape Labels $\mathcal{S}$. We define the following definitions on shape typings:

$\odot$	=	Empty typing
$n \rightarrow s : t$	=	Add shape type s to node n in typing t
$t_1 \uplus t_2$	=	Combine typings t_1 and t_2

4.3 Matching shapes

The expression $\Gamma \vdash n \simeq_s s$ represents the shape typings generated when matching a node n with a shape s in the context Γ.

The context contains the current typing which can be accessed through $\Gamma.typing$. The expression $\Gamma\{n \rightarrow t\}$ means the addition of type t to n in context Γ. The semantic definition of $\simeq_s$ is depicted in Figure 5.

4.4 Matching Rules

The expression $\Gamma \vdash g \simeq_r r$ represents the typings generated when matching a set of triples g with a rule r. Figure 6 represents the inference rules that generate those typings.

4.5 Matching Names

Matching names is a boolean expression of the form $\Gamma \vdash p \simeq_n n$ which is $true$ if predicate p matches nameValue n in context Γ. The semantics is represented in Figure 7 where it defines three rules which depend of the type of term: $NameTerm$, $NameAny$ or $NameStem$. We use the definitions:

$matchStem(stem, iri)$	=	$true$ if iri has stem $stem$
$matchStems(stems, iri)$	=	$true$ if iri has stem one of $stems$

Notice that $matchAny$ is defined in terms of $matchStems$ because it matches any value except those that appear in the $stems$ list.

4.6 Matching values

Matching values is an expression of the form $\Gamma \vdash p \simeq_v n$. Although in most of the rules, it returns a boolean, in the case of $ValueRef$, given that it has to check the typing of another node in a referenced shape it can return a shape typing. In this way, this

$$MatchShape\frac{g.triples(n) = ts \qquad \Gamma\{n \to shape.label\} \vdash ts \simeq_r shape.rule \rightsquigarrow t}{\Gamma \vdash n \simeq_s shape \rightsquigarrow t}$$

Figure 5: Inference rule to match shapes

$$Or_1\frac{\Gamma \vdash g \simeq_r r_1 \rightsquigarrow t}{\Gamma \vdash g \simeq_r \text{Or}(r_1,r_2) \rightsquigarrow t} \qquad Or_2\frac{\Gamma \vdash g \simeq_r r_2 \rightsquigarrow t}{\Gamma \vdash g \simeq_r \text{Or}(r_1,r_2) \rightsquigarrow t}$$

$$And\frac{\Gamma \vdash g_1 \simeq_r r_1 \rightsquigarrow t_1 \qquad \Gamma \vdash g_2 \simeq_r r_2 \rightsquigarrow t_2}{\Gamma \vdash g_1 \oplus g_2 \simeq_r \text{And}(r_1, r_2) \rightsquigarrow t_1 \uplus t_2}$$

$$Empty\frac{}{\Gamma \vdash \circledcirc \simeq_r \text{Empty} \rightsquigarrow \odot}$$

$$OneOrMore_1\frac{\Gamma \vdash g \simeq_r \text{r} \rightsquigarrow t}{\Gamma \vdash g \simeq_r \text{OneOrMore } r \rightsquigarrow t} \qquad OneOrMore_2\frac{\Gamma \vdash g_1 \simeq_r \text{r} \rightsquigarrow t_1 \qquad \Gamma \vdash g_2 \simeq_r \text{OneOrMore } r \rightsquigarrow t_2}{\Gamma \vdash g_1 \oplus g_2 \simeq_r \text{OneOrMore } r \rightsquigarrow t_1 \uplus t_2}$$

$$Arc\frac{\Gamma \vdash triple.pred \simeq_n \text{n} \qquad \Gamma \vdash triple.obj \simeq_v \text{v} \rightsquigarrow t}{\Gamma \vdash [triple] \simeq_r \text{Arc}(n,v) \rightsquigarrow t}$$

Figure 6: Inference rules for Shape expression rules

expression returns a shape typing as the general case. The inference rules are depicted in Figure 8.

Special care has to be taken for recursive definitions. To avoid infinite loops, we add the current type to the context in the definition of *matchShape* and in the definition of *ValueRef* we added two cases: one where the object has already the label shape so there is no need to check again its shape, and the other where the referenced object does not have a shape a it is necessary to check it.

5. TRANSFORMING RDF USING SHAPE EXPRESSIONS

The Shape Expressions semantics lays out a repeatable call sequence for dispatching semantic extensions. The examples in the previous sections describe how these can be used to extend the expressiveness of validation.

Parser generators are generally used to map input text into some native representation of an abstract syntax tree. Because these trees are highly specialized, we instead demonstrate the concept with a mapping to an XML tree. This process is frequently referred to as *semantic lowering* and requires mapping an RDF graph (unordered set of triples) to an ordered tree of XML elements and attributes. (An analogous process meeting the use cases of the Linked Data Platform would map an RDF graph to a sequence of SQL updates.)

GenX was developed to map RDF clinical care records to the XML representation being standardized by the HL7 FHIR working group [4]. The expression of FHIR in RDF is more attractive to the XML-focused participants when the data can then be *round-tripped* back to XML. The expressiveness required to map to FHIR/XML includes: transform RDF IRIs to XML `xsd:anyURI`, for XML fragments: control over emitted element and attribute names, obtaining values from the objects of triples, simple string transformations of RDF IRIs and literals and arbitrary placement of the created elements and attributes within the hierarchy.

The syntax of *GenX* is described in table 1.

Feature	Description
`$IRI`	The namespace for an element
`<name>`	The local name for an element
`@<name>`	The name of an attribute
`=<expr>`	An XPath function to transform the value. For example: `@status =substr(19)`
=	Don't emit the value as attribute or element. For example `reproduced =`
`[n]`	Place the value up n elements in the hierarchy. For example: `[-1]@date`

Table 1: GenX Language Summary

As an example, Figure 9 adds GenX semantic actions to the issues example. The output generated by the validation process applied to 4 is presented in Figure 10.

[4]`http://www.hl7.org/implement/standards/fhir/`

$$NameTerm\,\frac{\Gamma \vdash pred = t}{\Gamma \vdash pred \simeq_n \mathrm{NameTerm}(t)}$$

$$NameAny\,\frac{\Gamma \vdash not(matchStems(excl, pred))}{\Gamma \vdash pred \simeq_n \mathrm{NameAny}(excl)} \qquad NameStem\,\frac{\Gamma \vdash matchStem(s, pred)}{\Gamma \vdash pred \simeq_n \mathrm{NameStem}(s)}$$

Figure 7: Matching names

$$ValueType\,\frac{\Gamma \vdash obj.type = t}{\Gamma \vdash obj \simeq_v \mathrm{ValueType}(t) \rightsquigarrow \odot} \qquad ValueSet\,\frac{\Gamma \vdash obj \in s}{\Gamma \vdash obj \simeq_v \mathrm{ValueSet}(s) \rightsquigarrow \odot}$$

$$ValueAny\,\frac{\Gamma \vdash not(matchStems(excl, obj))}{\Gamma \vdash obj \simeq_v \mathrm{ValueAny}(excl) \rightsquigarrow \odot} \qquad ValueStem\,\frac{\Gamma \vdash matchStem(s, obj)}{\Gamma \vdash obj \simeq_v \mathrm{ValueStem}(s) \rightsquigarrow \odot}$$

$$ValueRef\,\frac{\Gamma \vdash obj \rightarrow label}{\Gamma \vdash obj \simeq_v \mathrm{ValueRef}(label) \rightsquigarrow \Gamma.typing} \qquad ValueRef\,\frac{\Gamma \vdash label : shape \qquad \Gamma \vdash obj \simeq_s shape \rightsquigarrow t}{\Gamma \vdash obj \simeq_v \mathrm{ValueRef}(label) \rightsquigarrow t}$$

Figure 8: Matching values

```
%GenX{ issue $http://ex.example/xml %}
<IssueShape> {
  ex:status (ex:unassigned
            ex:assigned)
    %GenX{@status =substr(19)%},
  ex:reportedBy @<UserShape>
    %GenX{ reported = %},
  ex:reportedOn xsd:dateTime
    %GenX{ [-1]@date %},
  (ex:reproducedBy @<UserShape>,
   ex:reproducedOn xsd:dateTime
    %GenX{ @date %}
  )?
    %GenX{ reproduced = %},
  ex:related @<IssueShape>*
} %GenX{ @id %}
<UserShape> {
  (foaf:name xsd:string
     %GenX{ full-name %}
   | foaf:givenName xsd:string+
     %GenX{ given-name %},
     foaf:familyName xsd:string
     %GenX{ family-name %}
  ),
  foaf:mbox shex:IRI ?
     %GenX{ email %}
}
```

Figure 9: Issue example extended with GenX semantic actions

```
<issue xmlns="http://ex.example/xml"
       id="Issue1" status="unassigned">
 <reported date="2013-01-23">
   <given-name>Bob</given-name>
   <family-name>Smith</family-name>
   <email>mail:bob@example.org</email>
 </reported>
 <reproduced date="2013-01-23">
  <given-name>Joe</given-name>
  <given-name>Joseph</given-name>
  <family-name>Thompson</family-name>
  <email>mail:joe@example.org</email>
 </reproduced>
</issue>
```

Figure 10: XML output generated by validator with GenX semantic actions

The value of the Shape Expressions semantic actions can be demonstrated by both the utility of individual language extensions like GenX and the simplicity with which they can be added to a validator.

It is trival to add a language called *GenJ* to emit a customized dialect of JSON-LD [21]. The presence of a label for the semantic action permits actions in multiple languages to be expressed in the same schema.

6. IMPLEMENTATIONS AND APPLICATIONS

Currently, there are 4 implementations of Shape Expressions in progress:

- *FancyShExDemo*[5] was the first prototype implementation in Javascript. It handles semantic actions and implements GenX

[5]http://www.w3.org/2013/ShEx/FancyShExDemo

and GenJ. It supports a form-based system with dynamic validation during the edition process and SPARQL queries generation.

- *JSShexTest*[6], developed by Jesse van Dam is another Javascript implementation. It supports both the SHEXc and SHEX/RDF syntax of Shape Expressions and contains a validation semantics for testing purposes based on truth tables.

- Shexcala[7]: an implementation developed in Scala originally based on the type inference semantics presented in this paper. It supports validation against an RDF file and against an SPARQL endpoint. We have also created RDFShape[8], an online RDF validator tool based on Shexcala.

  The initial Shexcala implementation used a backtracking monad similar to the semantics of logic programming languages [12]. Later, we implemented a more efficient algorithm based on regular expression derivatives [5] which was also used in RelaxNG [6].

- Haws[9]: a Haskell implementation based on the type inference semantics presented in this paper and implemented using a backtracking monad. This implementation can be seen as an executable monadic semantics of Shape Expressions [13].

Shape Expressions have been employed for data portal documentation. For example, the RDF data model of the WebIndex[10] and LandPortal[11] projects have been documented using Shape Expressions. The documentation defines templates for the different shapes of nodes and for the triples that can be retrieved when dereferencing those nodes. For example, the template of an observation in the WebIndex data portal is defined as:

```
<Observation> {
  rdf:type ( qb:Observation )
, rdfs:label rdf:langString ?
, cex:md5-checksum xsd:string ?
, cex:computation @<Computation>
, dcterms:issued xsd:date ?
, dcterms:publisher (:WebFoundation)
, qb:dataSet @<DataSet>
, wfonto:ref-area @<Area>
, cex:indicator @<Indicator>
, wfonto:ref-year @<Year>
, cex:value xsd:float
}
```

These templates define the dataset structure in an intuitive way and can be used to generate specialized visualizations for the different shapes when browsing the data portal.

One important aspect is that shapes are different from types. Both portals are using a similar model based on RDF Data Cube where the main entities are observations of the same type qb:Observation. However, the shapes of observation resources in each portal are different. For example, in the LandPortal we used the time ontology while in the WebIndex we just used years codified as integers. Separating shapes from types offers a good separation of concerns: types work at a semantic level while shapes are more intended as interfaces between linked data portals.

Shape Expressions are also being employed in the development of more specialized validators. For example, the Vaskos project[12] is developing a SKOS validator using a combination between Shape Expressions and SPARQL queries.

7. RELATED WORK

There has been an increasing interest on RDF validation languages. Most of the approaches were presented at the W3c Workshop on RDF Validation [18] and can be summarized in the following categories:

- Inference based approaches, which try to adapt RDF Schema or OWL to express validation semantics. The use of Open World and Non-unique name assumption limits the validation possibilities. In fact, what triggers constraint violations in closed world systems leads to new inferences in standard OWL systems. [7, 22, 14] propose the use of OWL expressions with a Closed World Assumption to express integrity constraints.

- SPARQL-based approaches use the SPARQL Query Language to express the validation constraints. SPARQL has much more expressiveness than Shape Expressions and can even be used to validate numerical and statistical computations [11]. However, we consider that the Shape Expressions language will be more usable by people familiar with validation languages like RelaxNG. Nevertheless, Shape Expressions can be translated to SPARQL queries. In fact, we have implemented a translator from Shape Expressions to SPARQL queries. This translator combined with semantic actions expressed in SPARQL can offer the same expressiveness as other SPARQL approaches with a more succinct and intuitive syntax.

  SPARQL Inferencing Notation (SPIN)[9] constraints associate RDF types or nodes with validation rules. These rules are expressed as SPARQL ASK queries where `true` indicates an error or CONSTRUCT queries which produce instances of `spin:ConstraintViolation`. SPIN constraints use the expressiveness of SPARQL plus the semantics of the `?this` variable standing for the current subject. The authors of SPIN have also created a data quality constraints library [13] with numerous SPIN templates for common patterns.

  There have been other proposals using SPARQL combined with other technologies, Simister and Brickley[20] propose a combination between SPARQL queries and property paths which is used in Google and Kontokostas et al [10] proposed *Databugger* a Test-driven framework which employs SPARQL query templates that are instantiated into concrete quality test queries. We consider that Shape Expressions can also be employed in the same scenarios as SPARQL while the specialized validation nature of Shape Expressions can lead to more efficient implementations.

- Grammar based approaches define a domain specific language to declare the validation rules. OSLC Resource Shapes [19] have been proposed as a high level and declarative description of the expected contents of an RDF graph expressing

[6]https://github.com/jessevdam/shextest
[7]http://labra.github.io/ShExcala/
[8]http://rdfshape.weso.es
[9]http://labra.github.io/haws/
[10]http://data.webfoundation.org/webindex/2013
[11]http://landportal.weso.es
[12]http://vaskos.chemaar.cloudbees.net/
[13]http://semwebquality.org/

constraints on RDF terms. Shape Expressions have been inspired by OSLC although they offer more expressive power.

Dublin Core Application Profiles [8] also define a set of validation constraints using Description Templates with less expressiveness than Shape Expressions.

The main inspiration for Shape Expressions has been RelaxNG [23], a Schema language for XML that offers a good trade-off between expressiveness and validation efficiency. The semantics of RelaxNG has also been expressed using inference rules in the specification document [16] and is based on tree grammars [15]. In the case of Shape Expressions the underlying semantics can be defined in terms of regular bag expresions [4].

Transforming RDF to different formats has also been challenging and several approaches have appeared. XSLT+SPARQL [2] proposed to add functions to XSLT that provided the ability to query SPARQL endpoints and to use standard XSLT to process the SPARQL XML results format.

The lowering use case can in principle be met within the W3C standards of SPARQL and XSLT. This involves a SPARQL query to generate SPARQL XML Results followed by XSLT of those results into the desired XML structure. One problem is that SPARQL results are tabular (like SQL) which means that the tree structure must be extracted by ordering the results and comparing successive rows in order to detect when a new branch of the tree is required. This transformation process is not easy and can be error-prone while the Shape Expressions validator automatically handles it.

XSPARQL[1, 3] merges SPARQL queries into XQuery. The expressivity of XSPARQL is considerably higher than that of GenX though, of course, lower than the more general semantic action mechanism in Shape Expressions. In service to the lowering use case, the GenX expressivity is sufficient for practical use cases like the FHIR RDF-to-XML one, as well as presenting a terser and potentially more intuitive declaration of the transformation.

8. CONCLUSIONS AND FUTURE WORK

Shape Expressions can be seen as a Domain Specific Language to define the shape of RDF graphs. They offer a more expressive way to define sets of graph shapes than OSLC's Resource Shapes or Dublin Core's Application Profiles. There are trade-offs between expressiveness and implementability, but compared to schema languages in other data models, Shape Expressions represent a conservative point in that spectrum, emulating mostly the expressiveness of RelaxNG.

As a language, it offers an opportunity to publish and enforce defined interfaces, as well as to serve as a *yacc-like* compiler complier for semantic web data. From a tooling perspective, it can be used stand-alone to validate RDF graphs and endpoints. Combined It can also be used to generate SPARQL queries which perform that same task on widely deployed infrastructure.

The complexity of the validation algorithms for Shape Expressions offers some theoretical challenges related to regular bag expressions that have been tackled in [4]. We have recently implemented a regular expression derivatives algorithm which seems more efficient although it is necessary to establish a common framework to asess the performance metrics of the different approaches.

It is the authors' hope that Shape Expressions be critically examined with respect to intuitiveness, utility and aesthetics as the RDF world shifts its focus more on the critical problems of validation, transformation and interface specification.

9. REFERENCES

[1] W. Akhtar, J. Kopecky, T. Krennwallner, and A. Polleres. XSPARQL: Traveling between the XML and RDF worlds and avoiding the XSLT pilgrimage. In M. Hauswirth, M. Koubarakis, and S. Bechhofer, editors, *Proceedings of the 5th European Semantic Web Conference*, LNCS, Berlin, Heidelberg, June 2008. Springer Verlag.

[2] D. Berrueta, J. E. Labra, and I. Herman. XSLT+SPARQL: Scripting the semantic web with SPARQL embedded into XSLT stylesheets. In *Proceedings of 4th Workshop on Scripting for the Semantic Web*. 5th European Semantic Web Conference (ESWC2008), 2008.

[3] S. Bischof, S. Decker, T. Krennwallner, N. Lopes, and A. Polleres. Mapping between RDF and XML with XSPARQL. *Journal on Data Semantics*, 1:147–185, 2012.

[4] I. Boneva, J. E. Labra, S. Hym, E. G. Prud'hommeau, H. Solbrig, and S. Staworko. Validating RDF with Shape Expressions. *ArXiv e-prints*, (1404.1270), Apr. 2014.

[5] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, Oct. 1964.

[6] J. Clark. An algorithm for RELAX NG validation. `http://www.thaiopensource.com/relaxng/derivative.html`, 2002.

[7] K. Clark and E. Sirin. On RDF validation, stardog ICV, and assorted remarks. In *RDF Validation Workshop. Practical Assurances for Quality RDF Data*, Cambridge, Ma, Boston, September 2013. W3c, `http://www.w3.org/2012/12/rdf-val`.

[8] K. Coyle and T. Baker. Dublin core application profiles. separating validation from semantics. In *RDF Validation Workshop. Practical Assurances for Quality RDF Data*, Cambridge, Ma, Boston, September 2013. W3c, `http://www.w3.org/2012/12/rdf-val`.

[9] H. Knublauch. SPIN - Modeling Vocabulary. `http://www.w3.org/Submission/spin-modeling/`, 2011.

[10] D. Kontokostas, P. Westphal, S. Auer, S. Hellmann, J. Lehmann, R. Cornelissen, and A. Zaveri. Test-driven evaluation of linked data quality. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 747–758, Republic and Canton of Geneva, Switzerland, 2014. International World Wide Web Conferences Steering Committee.

[11] J. E. Labra and J. M. Alvarez Rodríguez. Validating statistical index data represented in RDF using SPARQL queries. In *RDF Validation Workshop. Practical Assurances for Quality RDF Data*, Cambridge, Ma, Boston, September 2013. W3c, `http://www.w3.org/2012/12/rdf-val`.

[12] J. E. Labra, J. M. Cueva, M. C. Luengo, and A. Cernuda. Specification of logic programming languages from reusable semantic building blocks. *Electronic Journal on Theoretical Computer Science*, 62:220–233, 2002.

[13] J. E. Labra Gayo. Reusable semantic specifications of programming languages. In *6th Brazilian Symposium on Programming Languages*, 2002.

[14] B. Motik, I. Horrocks, and U. Sattler. Adding Integrity Constraints to OWL. In C. Golbreich, A. Kalyanpur, and B. Parsia, editors, *OWL: Experiences and Directions 2007 (OWLED 2007)*, Innsbruck, Austria, June 6–7 2007.

[15] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of xml schema languages using formal language theory.

ACM Trans. Internet Technol., 5(4):660–704, Nov. 2005.

[16] OASIS Committee Specification. RELAX NG Specification:. http://relaxng.org/spec-20011203.html, 2001.

[17] RDF Working Group W3C. RDF - semantic web standards. http://www.w3.org/RDF/, 2004.

[18] RDF Working Group W3c. W3c validation workshop. practical assurances for quality rdf data, September 2013.

[19] A. G. Ryman, A. L. Hors, and S. Speicher. OSLC resource shape: A language for defining constraints on linked data. In C. Bizer, T. Heath, T. Berners-Lee, M. Hausenblas, and S. Auer, editors, *Linked data on the Web*, volume 996 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[20] S. Simister and D. Brickley. Simple application-specific constraints for rdf models. In *RDF Validation Workshop. Practical Assurances for Quality RDF Data*, Cambridge, Ma, Boston, September 2013. W3c, `http://www.w3.org/2012/12/rdf-val`.

[21] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, and N. Landström. JSON-LD 1.0: AJSON-based Serialization for Linked Data. `http://www.w3.org/TR/json-ld/`, 2014.

[22] J. Tao, E. Sirin, J. Bao, and D. L. McGuinness. Integrity constraints in OWL. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI-10)*. AAAI, 2010.

[23] E. van der Vlist. *Relax NG: A Simpler Schema Language for XML*. O'Reilly, Beijing, 2004.