ShEx & SHACL compared

RDF Validation tutorial

Jose Emilio Labra Gayo
WESO Research group
University of Oviedo, Spain

Eric Prud'hommeaux
World Wide Web Consortium
MIT, Cambridge, MA, USA

Iovka Boneva
LINKS, INRIA & CNRS
University of Lille, France

Dimitris Kontokostas
GeoPhy
http://kontokostas.com/

Several common features…

Similar goal: describe and validate RDF graphs

Both employ the word "shape"

Node constraints similar in both languages

Constraints on incoming/outgoing arcs

Both allow to define cardinalities

Both have RDF syntax

Both have an extension mechanism

But some differences…

Underlying philosophy

Syntactic differences

Notion of a shape

Syntactic differences

Default cardinalities

Shapes and Classes

Recursion

Repeated properties

Property pair constraints

Uniqueness

Extension mechanism

Underlying philosophy

ShEx is more schema based

Shapes schemas look like grammars

Focus on validation results:

- Result shape maps
- Info about conforming and non-conforming nodes

SHACL is more constraint based

Shapes ≈ collections of constraints

Main focus: validation errors

No info about conforming nodes

- How to difficult to distinguish between conforming nodes and nodes that have been ignored?
- RDFShape offers info about conforming node also

Semantic specification

ShEx semantics: mathematical concepts

Well-founded semantics*

Support for recursión and negation

Inspired by type systems and RelaxNG

*Semantics and Validation of Shapes Schemas for RDF
Iovka Boneva Jose Emilio Labra Gayo Eric Prud'hommeaux
ISWC'17

SHACL semantics = textual description + SPARQL

SHACL terms described in natural language

SPARQL fragments used as helpers

Recursion is implementation dependent

SHACL-SPARQL based on pre-binding

Syntactic differences

ShEx design focused on human-readability

Followed programming language design methodology

1. Abstract syntax
2. Different concrete syntaxes
   - Compact
   - JSON-LD
   - RDF
   - ...

SHACL design focused on RDF vocabulary

Design centered on RDF terms

Lots of rules to define valid shapes graphs

https://w3c.github.io/data-shapes/shacl/#syntax-rules

No compact syntax

Compact Syntax

ShEx compact syntax designed along the language

Test-suite with long list of tests

Round-trippable with JSON-LD syntax

SHACL has no compact syntax

A WG Note proposed a compact syntax

- It covered a subset of SHACL core
- No longer supported and no implementations

Boolean operators and repeated properties

ShEx contains Boolean operators and grammar based operators

2-level language:

Shape expressions: AND, OR, NOT

Triple expressions: grouping (;), alternative (|)

```
:Product {
 :code IRI ;
 :code xsd:integer
}
```

It means a product with 2 codes (one IRI, and one integer)

☺

SHACL contains Boolean operators (and, or, not, xone)

Only top-level expressions

. means conjunction

```
:Product {
 :code IRI .
 :code xsd:integer
}
```

It means the code of a product must be an IRI and an integer

```
:p1 :code <http://code.org/P123> ;
     :code 123 .
```

☹

☺ = conforms to Shape

☹ = doesn't conform

RDF vocabulary

ShEx vocabulary ≈ abstract syntax

ShEx RDF vocabulary obtained from the abstract syntax

- ShEx RDF serializations typically more verbose
- They can be round-tripped to Compact syntax

```
:User a sx:Shape;
  sx:expression [ a sx:EachOf ;
    sx:expressions (
      [ a sx:TripleConstraint ;
        sx:predicate schema:name ;
        sx:valueExpr [ a sx:NodeConstraint ;
         sx:datatype xsd:string ]
      ]
      [ a sx:TripleConstraint ;
        sx:predicate schema:birthDate ;
        sx:valueExpr [ a sx:NodeConstraint ;
          sx:datatype xsd:date ] ;
          sx:min 0
      ] )
  ].
```

SHACL is designed as an RDF vocabulary

Some rdf:type declarations can be omitted

- SHACL RDF serialization typically more readable

```
:User a sh:NodeShape ;
  sh:property [ sh:path schema:name ;
   sh:minCount 1; sh:maxCount 1;
   sh:datatype xsd:string
  ];
  sh:property [ sh:path schema:birthDate ;
   sh:maxCount 1;
   sh:datatype xsd:date
  ] .
```

Notion of Shape

In ShEx, shapes only define structure of nodes

Shape maps select which nodes are validated with which shapes

Goal: flexibility and reusability

Shape

```
:User IRI {
 schema:name xsd:string
}
```

Shape map

```
:alice@:User,
{FOCUS rdf:type :Person}@:User
```

In SHACL, shapes define structure and can have target declarations

Shapes can be associated with nodes or sets of nodes through target declarations

Shapes may be less reusable in other contexts

Shape

```
:User a sh:NodeShape, rdfs:Class ;
  sh:targetClass :Person ;
  sh:targetNode  :alice ;
  sh:nodeKind    sh:IRI ;
  sh:property [
   sh:path     schema:name ;
   sh:datatype xsd:string
  ] .
```

target declarations (`rdfs:Class`, `sh:targetClass`, `sh:targetNode`)

structure (`sh:nodeKind`, `sh:property`)

Default cardinalities

ShEx: default = (1,1)

```
:User {
 schema:givenName xsd:string
 schema:lastName  xsd:string
}
```

SHACL: default = (0,unbounded)

```
:User a sh:NodeShape ;
 sh:property [ sh:path schema:givenName ;
  sh:datatype xsd:string ;
 ];
 sh:property [ sh:path schema:lastName ;
  sh:datatype xsd:string ;
 ] .
```

```
:alice schema:givenName "Alice" ;
       schema:lastName  "Cooper" .

:bob   schema:givenName "Bob", "Robert" ;
       schema:lastName  "Smith", "Dylan" .

:carol schema:lastName  "King" .

:dave  schema:givenName 23;
       schema:lastName  :Unknown .
```

= conforms to Shape

= doesn't conform

Property paths

ShEx shapes describe neighborhood of focus nodes: direct/inverse properties

Examples with paths can be simulated by nested shapes

- Sometimes requiring auxiliary recursive shapes
- More control about internal cardinalities

```
:GrandSon {
 :parent { :parent . + } + ;
 (:father . | :mother .) + ;
^:knows  :Person
}
```

SHACL shapes can also describe whole property paths following SPARQL paths

```
:GrandSon a sh:NodeShape ;
  sh:property [
   sh:path (schema:parent schema:parent) ;
   sh:minCount 1
  ];
  sh:property [
   sh:path [
    sh:alternativePath (:father :mother) ]
  ];
  sh:minCount 1
 ] ;
 sh:property [
   sh:path [sh:inversePath :knows ] ]
   sh:node :Person ;
   sh:minCount 1
 ]
.
```

Inference

ShEx doesn't mess with inference

- Validation can be invoked before or after inference
- rdf:type is considered an arc as any other
  - No special meaning
  - The same for rdfs:Class, rdfs:subClassOf, rdfs:domain, rdfs:range, ...

Some constructs have special meaning

- The following constructs have special meaning in SHACL
  - rdf:type
  - rdfs:Class
  - rdfs:subClassOf
  - owl:imports
- Other constructs like rdfs:domain, rdfs:range,... have no special meaning
- sh:entailment can be used to indicate that some inference is required

Inference and triggering mechanism

ShEx has no interaction with inference

It can be used to validate a reasoner

```
:User {
 schema:name xsd:string
}
```

In SHACL, RDF Schema inference can affect which nodes are validated

Some implicit RDFS inference but not all

```
:User a sh:NodeShape, rdfs:Class ;
 sh:property [ sh:path schema:name ;
   sh:datatype xsd:string;
].
```

```
:Teacher rdfs:subClassOf :User .
:teaches rdfs:domain :Teacher .

:alice a :Teacher ;
       schema:name 23 .

:bob   :teaches :Algebra ;
       schema:name 34 .

:carol :teaches :Logic;
       schema:name "King" .
```

	With or without RDFS inference	No RDFS inference	RDFS inference
:alice	☹	☹	☹
:bob	☹	Ignored	☹
:carol	☺	Ignored	☺

☺ = conforms to Shape

☹ = doesn't conform

Repeated properties

ShEx (;) operator handles repeated properties

SHACL needs qualifiedValueShapes for repeated properties

Example. A person must have 2 parents, one male and another female

```
:Person {
 :parent {:gender [:Male ] } ;
 :parent {:gender [:Female ] }
}
```

Direct approximation (wrong):

```
:Person a sh:NodeShape;
  sh:property [ sh:path :parent;
    sh:node [ sh:property [ sh:path :gender ;
     sh:hasValue :Male ; ]] ;
  ];
  sh:property [ sh:path :parent;
    sh:node [ sh:property [ sh:path :gender ;
     sh:hasValue :Female ]];
  ]
.
```

This says that a person must have a parent which is at the same time male and female

Repeated properties

ShEx (;) operator handles repeated properties

SHACL needs qualifiedValueShapes for repeated properties

Example. A person must have 2 parents, one male and another female

```
:Person {
 :parent {:gender [:Male ] } ;
 :parent {:gender [:Female ] }
}
```

Solution with qualifiedValueShapes:

```
:Person a sh:NodeShape, rdfs:Class ;
 sh:property [ sh:path :parent;
  sh:qualifiedValueShape [ sh:property [ sh:path :gender ;
  sh:hasValue :Male ] ] ;
  sh:qualifiedMinCount 1; sh:qualifiedMaxCount 1
 ];
 sh:property [ sh:path :parent;
  sh:qualifiedValueShape [ sh:property [ sh:path :gender ;
   sh:hasValue :Female ] ] ;
   sh:qualifiedMinCount 1; sh:qualifiedMaxCount 1
 ] ;
 sh:property [ sh:path :parent;
   sh:minCount 2; sh:maxCount 2
 ]
.
```

It needs to count all possibilities

Recursion

ShEx handles recursion

Well founded semantics

```
:Person {
 schema:name  xsd:string ;
 schema:knows @:Person*
}
```

Recursive shapes are undefined in SHACL*

Implementation dependent

Direct translation generates recursive shapes

```
:Person a sh:NodeShape ;
 sh:property [ sh:path schema:name ;
  sh:datatype xsd:string
 ];
 sh:property [ sh:path schema:knows ;
  sh:node :Person
 ]
.
```

Undefined because it is recursive

*Semantics and Validation of Recursive SHACL
Julien Corman, Juan L. Reutter and Ognjen Savkovic, ISWC'18

Recursion (with target declarations)

ShEx handles recursion

Well founded semantics

```
:Person {
 schema:name  xsd:string ;
 schema:knows @:Person*
}
```

Recursive shapes are undefined in SHACL

Implementation dependent

Can be simulated with target declarations

- Example with target declatations
- It needs discriminating arcs

```
:Person a sh:NodeShape, rdfs:Class ;
 sh:property [ sh:path schema:name ;
  sh:datatype xsd:string
 ];
 sh:property [ sh:path schema:knows ;
  sh:class :Person
 ]
.
```

It requires all nodes to have rdf:type Person

Recursion (with property paths)

ShEx handles recursion

Well founded semantics

```
:Person {
 schema:name  xsd:string ;
 schema:knows @:Person*
}
```

Recursive shapes are undefined in SHACL

Implementation dependent

Can be simulated property paths

```
:Person a sh:NodeShape ;
 sh:property [
  sh:path schema:name ; sh:datatype xsd:string ];
 sh:property [
  sh:path [sh:zeroOrMorePath schema:knows];
  sh:node :PersonAux
 ].

:PersonAux a sh:NodeShape ;
 sh:property [
  sh:path schema:name ; sh:datatype xsd:string
 ].
```

Closed shapes

In ShEx, closed affects all properties

```
:Person CLOSED {
  schema:name xsd:string
| foaf:name  xsd:string
}
```

In SHACL, closed only affects properties declared at top-level

Properties declared inside other shapes are ignored

```
:Person a sh:NodeShape ;
  sh:targetNode :alice ;
  sh:closed true ;
  sh:or (
   [ sh:path schema:name ; sh:datatype xsd:string ]
   [ sh:path foaf:name ; sh:datatype xsd:string ]
  ) .
```

☺ (ShEx)

```
:alice schema:name "Alice" .
```

☹ (SHACL)

☺ = conforms to Shape
☹ = doesn't conform

Closed shapes and paths

Closed in ShEx acts on all properties

```
:Person CLOSED {
   schema:name xsd:string |
   foaf:name xsd:string
}
```

In SHACL, closed ignores properties mentioned inside paths

```
:Person a sh:NodeShape ;
  sh:closed true ;
  sh:property [
    sh:path [
     sh:alternativePath
       ( schema:name foaf:name )
    ] ;
    sh:minCount 1; sh:maxCount 1;
    sh:datatype xsd:string ] ;
.
```

☺ ☹

```
:alice schema:name "Alice".
```

☺ = conforms to Shape

☹ = doesn't conform

Property pair constraints

This feature was posponed in ShEx 2.0

ShEx 2.1 is expected to add support for value comparisons

Not supported in ShEx 2.0

```
:UserShape {
 $<givenName> schema:givenName xsd:string ;
 $<firstName> schema:firstName xsd:string ;
 $<birthDate> schema:birthDate xsd:date ;
 $<loginDate> :loginDate xsd:date ;
 $<givenName> =  $<firstName> ;
 $<givenName> != $<lastName> ;
 $<birthDate> <  $<loginDate>
}
```

SHACL supports equals, disjoint, lessThan, ...

```
:UserShape a sh:NodeShape ;
 sh:property [
   sh:path schema:givenName ;
   sh:datatype xsd:string ;
   sh:disjoint schema:lastName
 ] ;
 sh:property [
   sh:path foaf:firstName ;
   sh:equals schema:givenName ;
 ] ;
 sh:property [
   sh:path schema:birthDate ;
   sh:datatype xsd:date ;
   sh:lessThan :loginDate
 ] .
```

Uniqueness (defining unique Keys)

This feature was postponed in ShEx 2.0

Not supported in ShEx 2.0

```
:UserShape {
  schema:givenName xsd:string ;
  schema:lastName xsd:string ;
  UNIQUE(schema:givenName, schema:lastName)
}
```

Not supported in ShEx 2.0

```
:Country {
 schema:name . + ;
 UNIQUE(LANGTAG(schema:name))
}
```

No support for generic unique keys

sh:uniqueLang offers partial support for a very common use case

Uniqueness can be done with SHACL-SPARQL

```
:Country a sh:NodeShape ;
 sh:property [
  sh:path schema:name ;
  sh:uniqueLang true
] .
```

Modularity

ShEx has EXTERNAL and import keywords

- EXTERNAL declares that a shape definition should be retrieved elsewhere
- Import declaration

SHACL supports owl:imports

- SHACL processors follow owl:imports declarations

```
:UserShape a sh:NodeShape ;
 sh:property [ sh:path schema:name ;
 sh:datatype xsd:string
] .
```

http://example.org/UserShapes

```
<> owl:imports <http://example.org/UserShapes>

:TeacherShape a sh:NodeShape;
   sh:node :UserShape ;
   sh:property [ sh:path :teaches ;
    sh:minCount 1;
] ;
```

Reusability - Extending shapes (1)

ShEx shapes can be extended by composition

```
:Product {
  schema:productId xsd:string
  schema:price xsd:decimal
}

:SoldProduct @:Product AND {
 schema:purchaseDate xsd:date ;
 schema:productId /^[A-Z]/
}
```

SHACL shapes can also be extended by composition

Extending by composition

```
:Product a sh:NodeShape, rdfs:Class ;
  sh:property [ sh:path schema:productId ;
    sh:datatype xsd:string
  ];
  sh:property [ sh:path schema:price ;
    sh:datatype xsd:decimal
  ].

:SoldProduct a sh:NodeShape, rdfs:Class ;
  sh:and (
   :Product
    [ sh:path schema:purchaseDate ;
      sh:datatype xsd:date]
    [ sh:path schema:productId ;
     sh:pattern "^[A-Z]" ]
  ) .
```

Reusability - Extending shapes (2)

In ShEx, there is no special treatment for `rdfs:Class, rdfs:subClassOf, ...`

By design, ShEx has no concept of Class

- It is not possible to extend by declaring subClass relationships
- No interaction with inference engines

SHACL shapes can also be extended by leveraging subclasses

Extending by leveraging subclasses

```
:Product a sh:NodeShape, rdfs:Class ;
  ...as before...

:SoldProduct a sh:NodeShape, rdfs:Class ;
  rdfs:subClassOf :Product ;
  sh:property [ sh:path schema:productId ;
     sh:pattern "^[A-Z]"
   ] ;
  sh:property [ sh:path schema:purchaseDate ;
     sh:datatype xsd:date
  ] .
```

SHACL subclasses may differ from RDFS/OWL subclases

Annotations

ShEx allows annotations but doesn't have predefined annotations yet

Annotations can be declared by //

```
:Person {
 // rdfs:label "Name"
 // rdfs:comment "Name of person"
 schema:name xsd:string ;
}
```

SHACL allows any kind of annotations and has some non-validating built-in annotations

Built-in properties: `sh:name`, `sh:description`, `sh:defaultValue`, `sh:order`, `sh:group`

```
:Person a sh:NodeShape ;
 sh:property [
  sh:path        schema:name ;
  sh:datatype    xsd:string ;
  sh:name        "Name" ;
  sh:description "Name of person"
  rdfs:label     "Name";
 ] .
```

Apart of the built-in annotations, SHACL can also use any other annotation

Validation report

ShEx 2.0 defines a result shape map

- It contains both positive and negative node/shape associations

SHACL defines a validation report

- Describes only the structure of errors
- Some properties can be used to control which information is shown
  - sh:message
  - sh:severity

Extension mechanism

ShEx uses semantic actions

Semantic actions allow any future processor

- They can be used also to transform RDF

SHACL has SHACL-SPARQL

SHACL-SPARQL allows new constraint components defined in SPARQL

- [See example in next slide]

It is possible to define constraint components in other languages, e.g. Javascript

Stems

ShEx can describe stems

Stems are built into the language

Example:

The value of :homePage starts by <http://company.com/>

```
:Employee {
 :homePage [ <http://company.com/> ~ ]
}
```

Stems are not built-in

Can be defined using SHACL-SParql

```
:StemConstraintComponent
 a sh:ConstraintComponent ;
 sh:parameter [ sh:path :stem  ] ;
  sh:validator [ a sh:SPARQLAskValidator ;
   sh:message "Value does not have stem {$stem}";
   sh:ask """
    ASK {
     FILTER (!isBlank($value) &&
       strstarts(str($value),str($stem)))
     }"""
 ] .
```

```
:Employee a sh:NodeShape ;
 sh:targetClass :Employee ;
 sh:property [
  sh:path :homePage ;
  :stem <http://company.com/>
 ] .
```

Further info

Further reading:

- Validating RDF data, chapter 7. http://book.validatingrdf.com/bookHtml013.html

Other resources:

- SHACL WG wiki: https://www.w3.org/2014/data-shapes/wiki/SHACL-ShEx-Comparison
- Phd Thesis: Thomas Hartmann, Validation framework of RDF-based constraint languages. 2016, https://publikationen.bibliothek.kit.edu/1000056458

End

This presentation is part of the set:

https://figshare.com/articles/Validating_RDF_Data/7159802