# Combining graph & tree …

*Writing SHAX, obtaining SHACL, XSD and more*

Hans-Jürgen Rennau, parsQube GmbH
Presented at xmlprague 2018, February 9, 2018

XML technology is very powerful, but also very limited. The more you are aware of the power, the keener your interest in reducing the limitations. A key problem is rooted in the very paradigm of XML, which is tree-structured information. This leads to the challenge of combining XML tree technology with RDF graph technology.

# What to expect

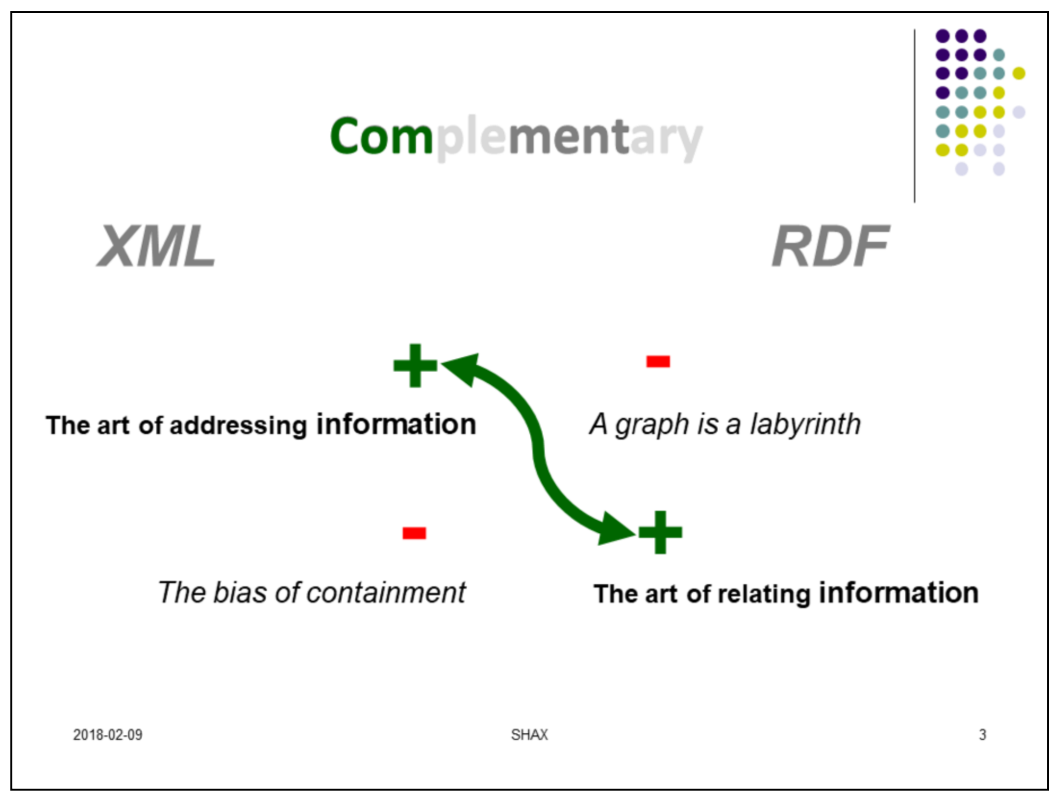
- Integrate graph and tree – why? how?

- SHACL - new RDF schema language (W3C)
- SHAX - XML syntax for SHACL

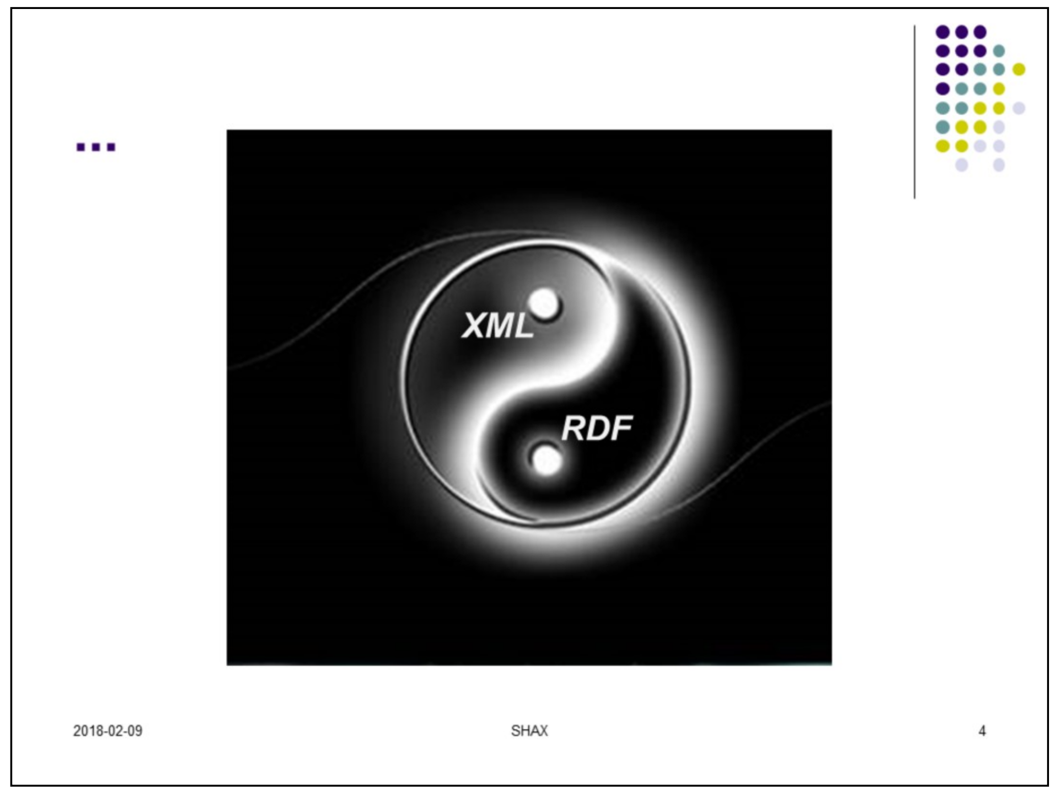
- SHAX = abstract modeling language
  = describing RDF, XML, JSON, ...

2018-02-09 SHAX 2

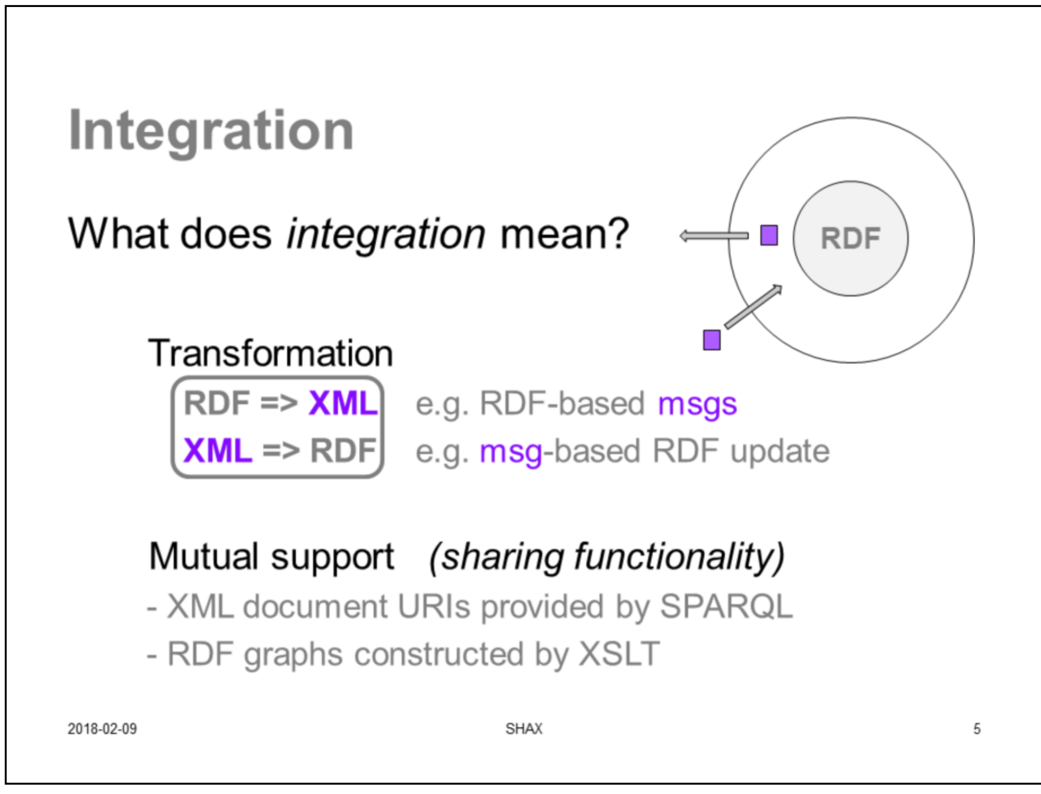
A brief look at what to expect. I shall start with an argument why the integration of graph and tree is so compelling a challenge. We shall glance at SHACL, the new schema language for RDF, and then I shall introduce you to SHAX, an XML syntax for SHACL. In the end, I shall argue that SHAX is not only an XML syntax for SHACL, but a data modeling language in its own right, an abstract language which cannot only validate RDF, but also XML and JSON data.



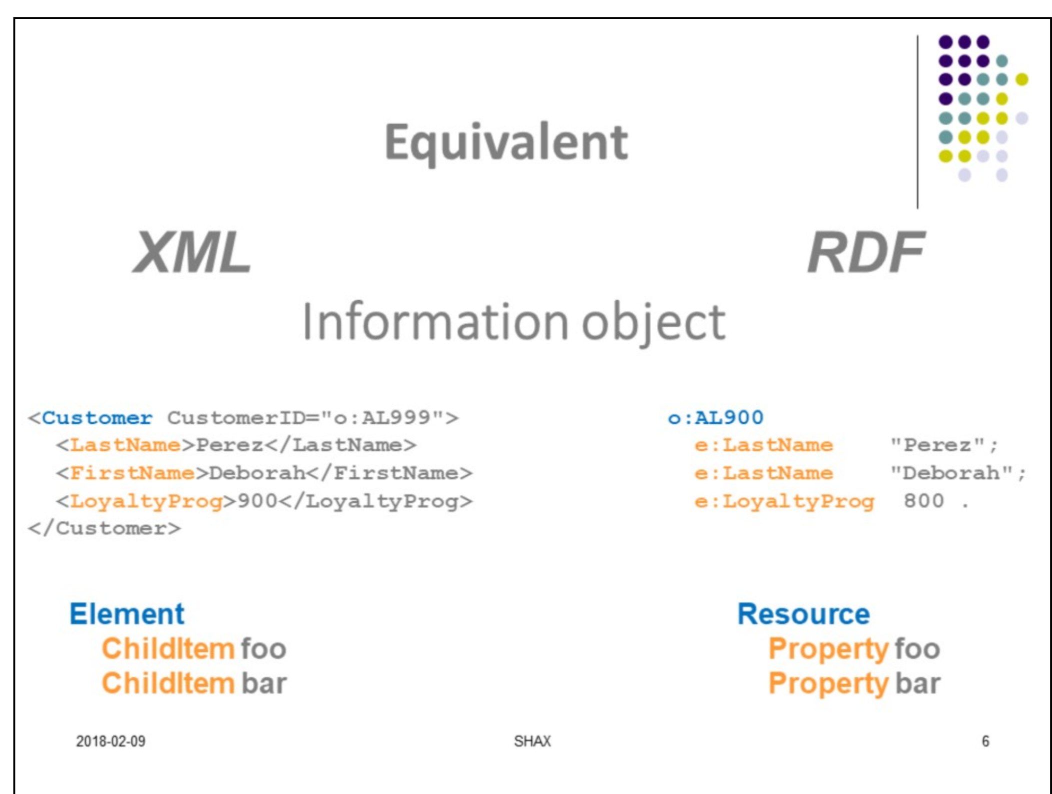

The power of XML technology is based on an ingenious concept of addressing information, XPath. It presupposes tree structure. On the one hand, this is no problem, as trees are ubiquitous. On the other hand, we should remember that trees are based on the containment relationship – for example, a book element contains author elements., and containment is pretty arbitrary. Is it not the other way around, an author should contain books? It depends on the focus of your interest, on where you stand. Like in real life, where the tree is behind the house, but the house is behind the tree. This means that XML structure is tied to specific perspectives and may be not appropriate for enterprise modeling and for enterprise data repositories. This cuts XML off from realizing its full potential. Enters RDF, which excels in relating information without assuming containment. The RDF model is fit for capturing the complex reality of an enterprise, and RDF triple stores may be used as a single point of truth, servicing a wide range of information needs. However, what you get is a graph, and graphs are hard to understand and process. Conclusion: XML and RDF are complementary.




The more you think about it, the more amazing the complementary character of XML and RDF becomes. It is mind-boggling. There is so much promise in combining the two intelligently.




What does their integration mean? Above all, two things: free transformation between the two formats, and mutual support – one technology using functionality offered by the other. For example, an XQuery processor might launch SPARQL queries in order to discover relevant document URIs. But the heart of integration is, I think, transformation. It enables us to use RDF triple stores as a single point of truth which communicates with the world via tree-structured messages.



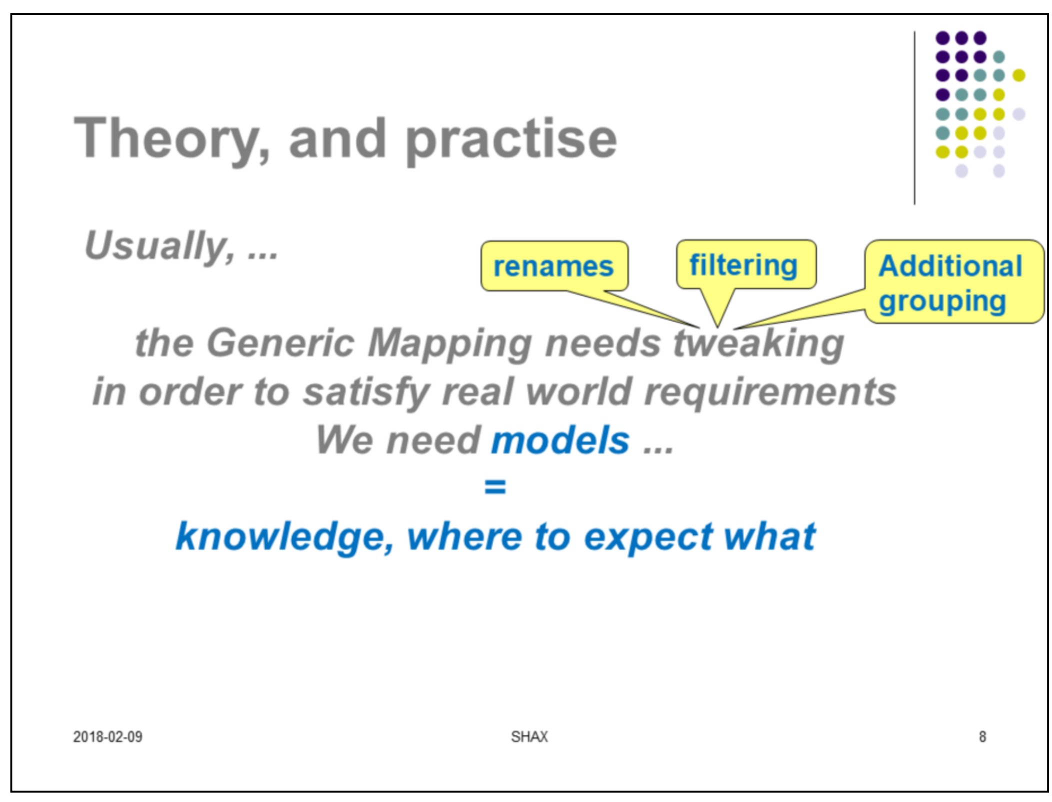

How to approach the challenge of transformation? Let us start by exploring the intrinsic relationship between the two. In spite of their outward differences, XML and RDF are built on common ground - a common abstraction – which I suggest to call an information object. It is a container of named values, which may be atomic (like strings or numbers) or themselves containers of named values. RDF calls the containers and their values: resources and their properties. XML calls them elements and their child elements and attributes. I call it an information object and its properties.

# Generic mapping (= canonical transformation)

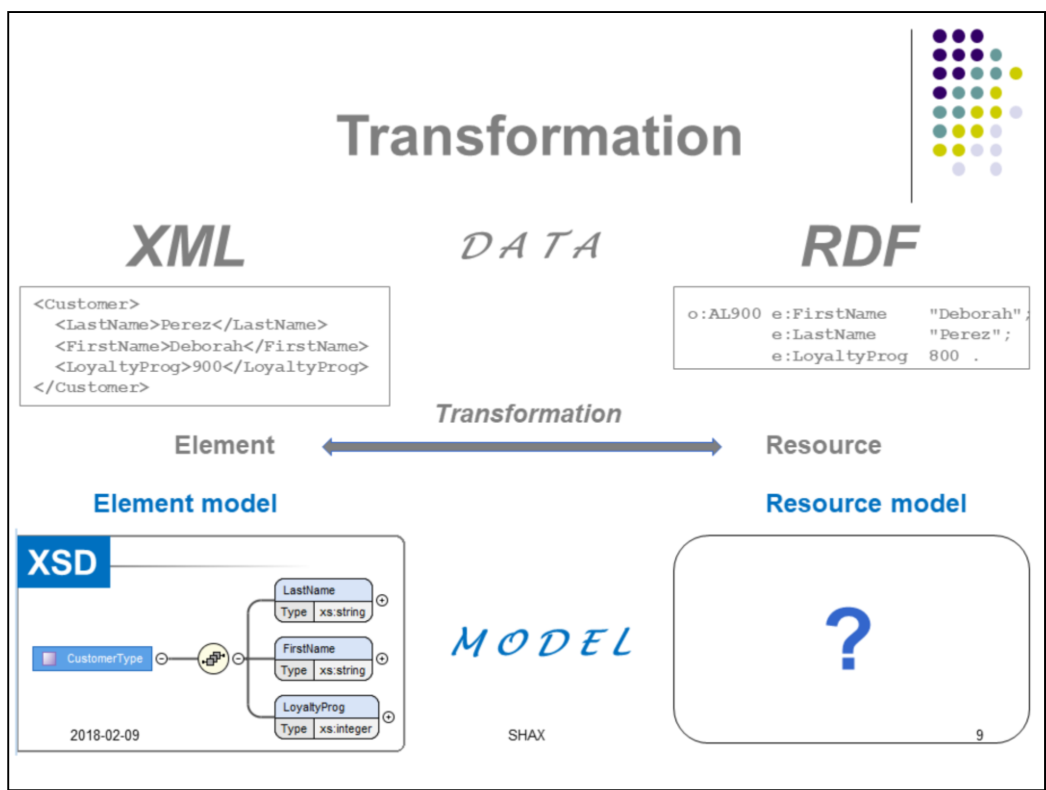
| RDF | XML |
|---|---|
| Resource node | *Complex element* |
| Blank node | *Complex element without @rdf:about* |
| Literal | *Simple element content or attribute value* |
| Resource IRI | *Value of attribute @rdf:about;*<br>*Alternative: value found at a configurable XPath* |
| Property IRI | *Element name, attribute name* |

2018-02-09 SHAX 7

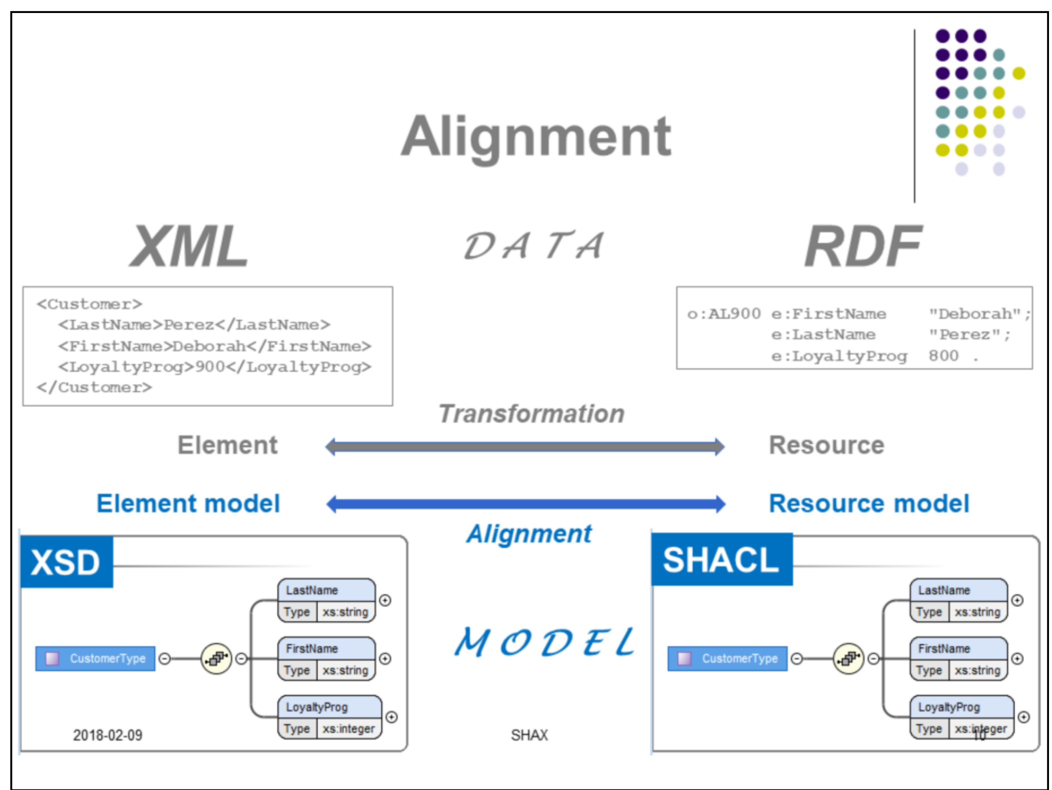
This perception leads immediately to a generic mapping between RDF and XML data – a canonical transformation!



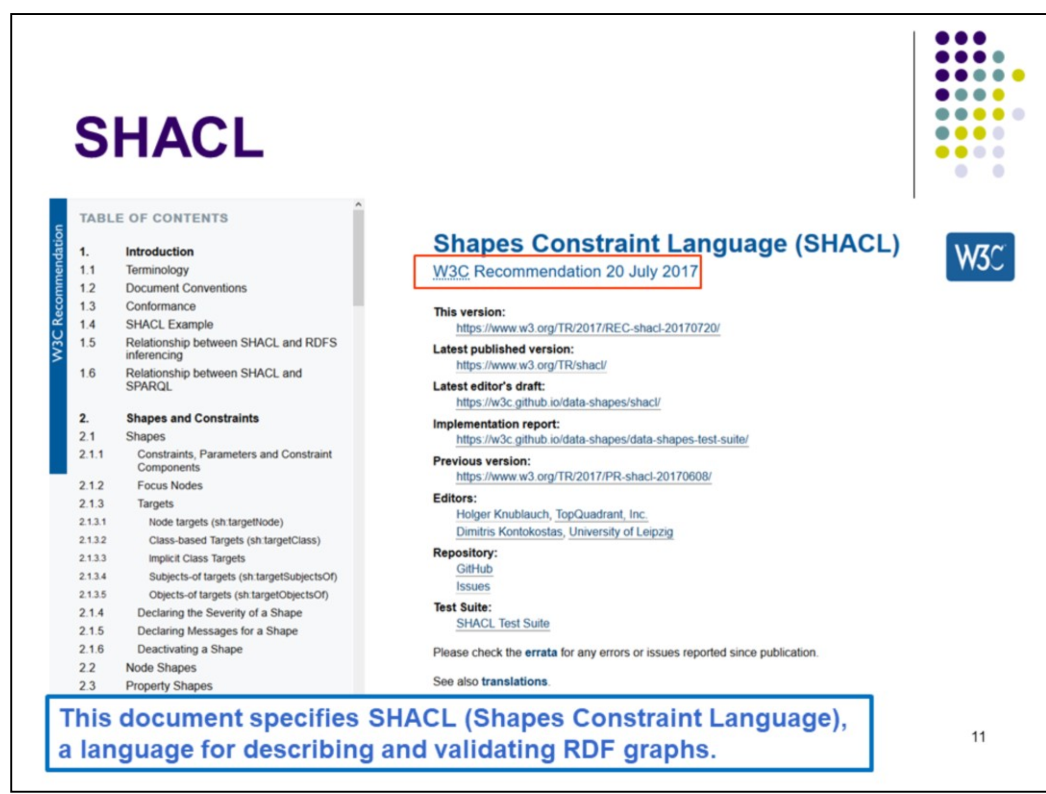

In theory, we are almost done: we can translate the generic mapping into generic code, and then we have a transformation machine for XML and RDF. But usually the result of our generic mapping will fail to meet the real world requirement to be focussed, intuitive and elegant. Think of a message – we want it to look purpose-built, and not like a translation of something else. As in natural language, a translation betraying its being a translation is ugly. We need to tweak the model, and to do so we must know exactly where to expect what. Which is another way of saying: we need models – on both sides of the wall.



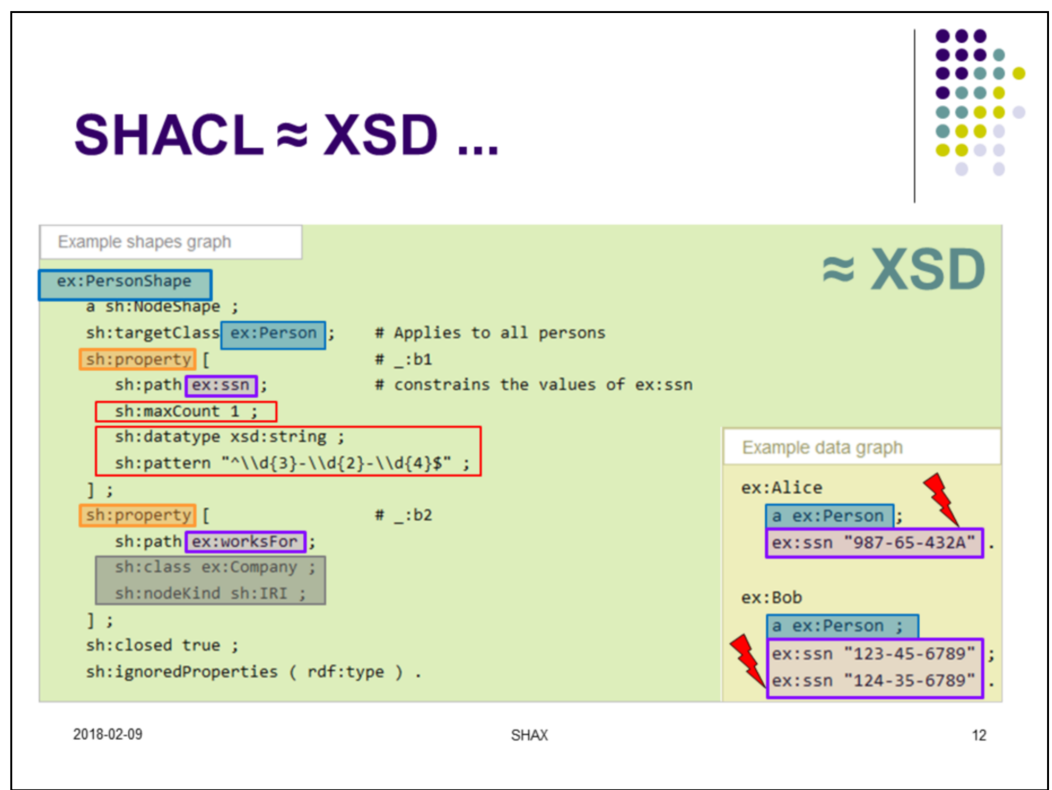

On the XML side, we have XSD, which gives us what we need – a prcise picture of the grammatical structure. With the RDF side, we have a problem. OWL, the main modeling language of RDF, is designed for inference, not for validation and prediction. OWL models cannot tell us exactly where to expect what. They are not appropriate for guiding transformation.



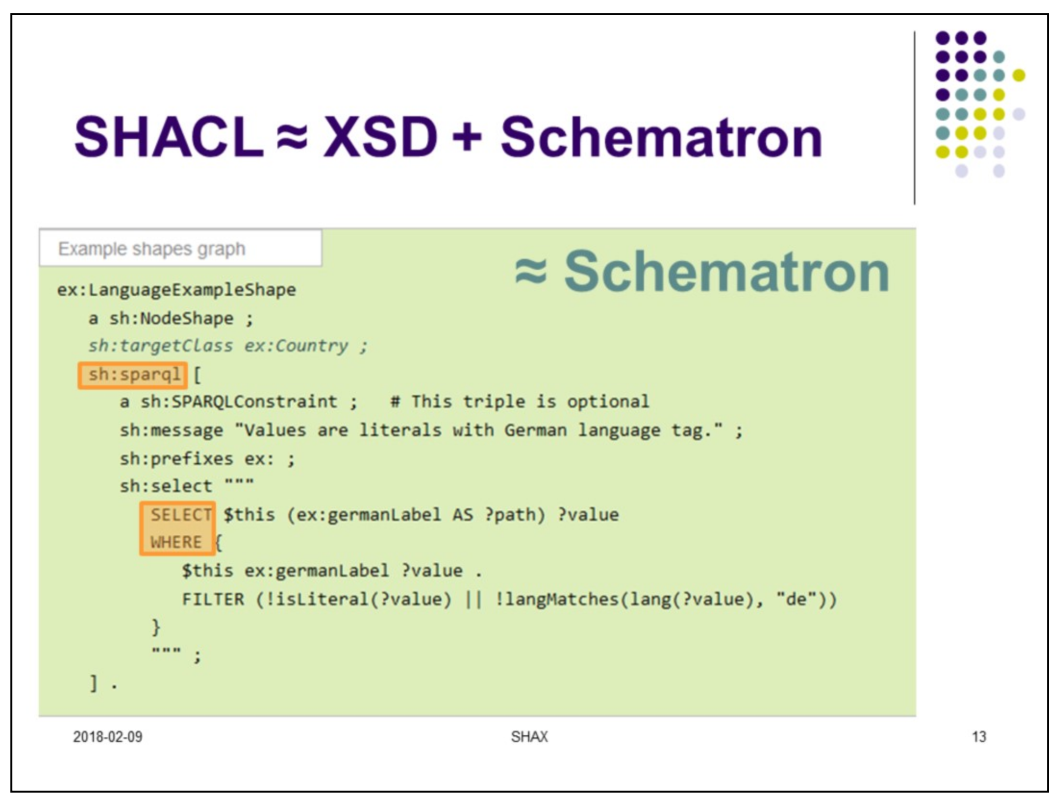

Fortunately, last year a new schema language for RDF has appeared – SHACL. Like XSD, SHACl can describe the data grammar precisely, and thus we have two models which can be aligned. This should enable high quality transformation of data.




SHACL is a W3C recommendation. The first sentence of the spec characterizes the language as a language for describing and validating RDF graphs.




The first thing to notice is that SHACL is itself an RDF vocabulary – just like XSD is an XML vocabulary. This little model (taken from the introduction of the spec) describes resources belonging to the RDF class „Person“. According to the model, such resources have two properties – a social security number and the companies the person works for. The model names the properties and constrains their cardinality and data types. On the right-hand side you see RDF data which belong to the Person class. They have the expected properties, and yet they violate the shape – in one case, a regex pattern, in the other a cardinality constraint. The model describes and validates the RDF data exactly like an XSD describes and validates XML data. At a first and superficial glance, SHACL is XSD for graphs.




But SHACL has also features similar to Schematron. So a short formular for SHACL is XSD + Schematron for RDF.

## SHACL – a breakthrough for integration?

- Upside
  - RDF data can be validated (at last!)
  - RDF data can be modelled for dev purposes = *exact picture where to expect what*
- Issues
  - RDF syntax ...
    - Few professionals are familiar
    - Very limited tool support
  - Grammatical structure not always explicit (No explicit support for choice groups )



Now – is SHACL a breakthrough for the integration of XML and RDF? I think it is a significant and necessary step. At last, RDF data can be modelled in a way which gives an exact picture where to expect what. But there are also issues...

# Why SHAX (XML syntax for SHACL)

- Simplicity
  - Simple to write and read
  - Simple to transform and process
  - Simple to generate (e.g. from XSD)
- Abstraction
  - More abstract representation of grammatical intent
  - Single source for multiple model styles
- Unification

  SHAX => SHACL, XSD, JSON Schema

2018-02-09 SHAX 15

The solution to these problems might be XML – an XML syntax for SHACL, which I call SHAX. It promises benefits which I divide into three categories. Let us check to which degree SHAX realizes these potential benefits.

```
<shax:model xmlns:shax="http://shax.org/ns/model" xmlns:e="http://..."

  <shax:objectType name="e:PersonShape" targetClass="e:Person">
    <e:ssn base="xsd:string" pattern="^\\d{3}-\\d{2}-\\d{4}$" card="?"/>
    <e:worksFor class="e:Company" card="*"/>
  </shax:objectType>

</shax:model>
```

SHAX

Example shapes graph

```
ex:PersonShape
   a sh:NodeShape ;
   sh:targetClass ex:Person ;    # Applies to all persons
   sh:property [                 # _:b1
      sh:path ex:ssn ;           # constrains the values of ex:ssn
      sh:maxCount 1 ;
      sh:datatype xsd:string ;
      sh:pattern "^\\d{3}-\\d{2}-\\d{4}$" ;
   ] ;
   sh:property [                 # _:b2
      sh:path ex:worksFor ;
      sh:class ex:Company ;
      sh:nodeKind sh:IRI ;
   ] ;
   sh:closed true ;
   sh:ignoredProperties ( rdf:type ) .
```

SHACL

To get started, this slide shows a SHAX representation of the trivial SHACL model shown before. A resource is modelled by an objectType element. Its child elements represent the properties of the resource, after which they are named. Their attributes specify cardinality constraints and type details. In order to get a better impression of SHAX, let us evolve our model a little...

```
<shax:model xmlns:shax="http://shax.org/ns/model" xmlns:e="http://..."

  <shax:objectType name="e:PersonShape" targetClass="e:Person">
    <e:ssn base="xsd:string" pattern="^\\d{3}-\\d{2}-\\d{4}$" card="?"/>
    <e:worksFor type="e:CompanyShape" card="*"/>
  </shax:objectType>


  <shax:objectType name="e:CompanyShape" targetClass="e:Company">
    <e:companyName type="xsd:string"/>
    <e:companyCode type="xsd:string"/>
  </shax:objectType>

</shax:model>
```

SHAX

17

First we notice that the second property of a person does not yet specify type information – what is the structure of a company? We introduce a second object type describing company resources, and we let the worksFor property element reference that type definition via an @type attribute.

```
<shax:model xmlns:shax="http://shax.org/ns/model" xmlns:e="http://..."

  <shax:objectType name="e:PersonShape" targetClass="e:Person">
    <e:ssn base="xsd:string" pattern="^\\d{3}-\\d{2}-\\d{4}$" card="?"/>
    <e:worksFor type="e:CompanyShape" card="?"/>
  </shax:objectType>


  <shax:objectType name="e:CompanyShape" targetClass="e:Company">
    <e:companyName          type="xsd:string"/>
    <shax:choice>
       <e:companyCode       type="xsd:string"/>
       <shax:pgroup>
          <e:repoCode       type="xsd:integer"/>
          <e:repoEntryNumber type="xsd:integer"/>
       </shax:pgroup>
    </shax:choice>
  </shax:objectType>

</shax:model>
```

SHAX

18

Then we add a little grammar – our company model contains a choice of properties. This is straightforward, using a shax:choice element, whose child elements represent the choice branches. If a branch contains several properties, they are wrapped in a shax:pgroup element.

```
<shax:model xmlns:shax="http://shax.org/ns/model" xmlns:e="http://..."

  <shax:objectType name="e:PersonShape" targetClass="e:Person">
    <e:ssn      type="e:SsnShape"      card="?"/>
    <e:worksFor type="e:CompanyShape" card="*"/>
  </shax:objectType>


  <shax:objectType name="e:CompanyShape" targetClass="e:Company">
    <e:companyName            type="xsd:string"/>
    <shax:choice>
       <e:companyCode         type="xsd:string"/>
       <shax:pgroup>
          <e:repoCode         type="xsd:integer"/>
          <e:repoEntryNumber  type="xsd:integer"/>
       </shax:pgroup>
    </shax:choice>
  </shax:objectType>


  <shax:dataType name="e:SsnShape"
                 base="xsd:string"
                 pattern="^\\d{3}-\\d{2}-\\d{4}$"/>
</shax:model>
```

SHAX

19

Now we get rid of the local declaration of a simple data type – we introduce a global data type definition which is referenced by the property element again via @type attribute.

```
<shax:model xmlns:shax="http://shax.org/ns/model" xmlns:e="http://..."

  <shax:objectType name="e:PersonShape" targetClass="e:Person">
    <e:ssn       type="e:SsnShape"      card="?"/>
    <e:worksFor type="e:CompanyShape" card="*" ordered="true"/>
  </shax:objectType>


  <shax:objectType name="e:CompanyShape" targetClass="e:Company">
    <e:companyName             type="xsd:string"/>
    <shax:choice>
       <e:companyCode          type="xsd:string"/>
       <shax:pgroup>
          <e:repoCode          type="xsd:integer"/>
          <e:repoEntryNumber type="xsd:integer"/>
       </shax:pgroup>
    </shax:choice>
  </shax:objectType>


  <shax:dataType name="e:SsnShape"
                 base="xsd:string"
                 pattern="^\\d{3}-\\d{2}-\\d{4}$"/>
</shax:model>
```

SHAX

20

Finally, we introduce order into our model – we want the values of the worksFor property to be an ordered list. To achieve this, we just add an @ordered attribute to the property element.

```
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix shax: <http://shax.org/ns/model#> .
@prefix _e: <http://shax.org/ns/model/element-equivalent#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix e: <http://example.org/ns/model#> .

e:PersonShape
   a sh:NodeShape;
   sh:targetClass e:Person;
   sh:property [
      sh:path e:ssn;
      sh:maxCount 1;
      sh:node e:SsnShape;
   ];
   sh:property [
      sh:path e:worksFor;
      sh:node shax:ListType;
      sh:property [
         sh:path ([sh:zeroOrMorePath rdf:rest] rdf:first);
         sh:node e:CompanyShape;
      ];
   ].
e:CompanyShape
   a sh:NodeShape;
   sh:targetClass e:Company;
   sh:property [
      sh:path e:companyName;
      sh:minCount 1;
      sh:maxCount 1;
      sh:datatype xsd:string;
   ];
   sh:xone (
      [
         a sh:NodeShape ;
         sh:not [
            a sh:NodeShape;
            sh:or (
               [
                  sh:path e:repoCode ;
                  sh:minCount 1 ;
               ]
               [
                  sh:path e:repoEntryNumber ;
                  sh:minCount 1 ;
               ]
            );
         ] ;
      sh:property [
            sh:path e:companyCode;
            sh:minCount 1;
            sh:maxCount 1;
            sh:datatype xsd:string;
         ] ;
      ]
      [
         a sh:NodeShape ;
         sh:not [
            sh:path e:companyCode;
            sh:minCount 1;
         ] ;
         sh:property [
            sh:path e:repoCode;
            sh:minCount 1;
            sh:maxCount 1;
            sh:datatype xsd:integer;
         ] ;
         sh:property [
            sh:path e:repoEntryNumber;
            sh:minCount 1;
            sh:maxCount 1;
            sh:datatype xs:integer;
         ] ;
      ]
   ).
e:SsnShape
   a sh:NodeShape;
   sh:datatype xsd:string;
   sh:pattern "^\\d{3}-\\d{2}-\\d{4}$".

shax:ListType
   a sh:NodeShape ;
   sh:xone (
      [   a sh:NodeShape ;
          sh:in (rdf:nil) ;
      ]
      [   a sh:NodeShape ;
          sh:property [
              sh:path rdf:first ;
              sh:minCount 1;
              sh:maxCount 1;
          ] ;
          sh:property [
              sh:path rdf:rest ;
              sh:minCount 1;
              sh:maxCount 1;
          ]
      ]
   ) .
```

SHACL

21

For comparison, here is the SHACL code into which our SHAX model is compiled. It think it is more difficult to read and to write, and probably more difficult to process or generate.

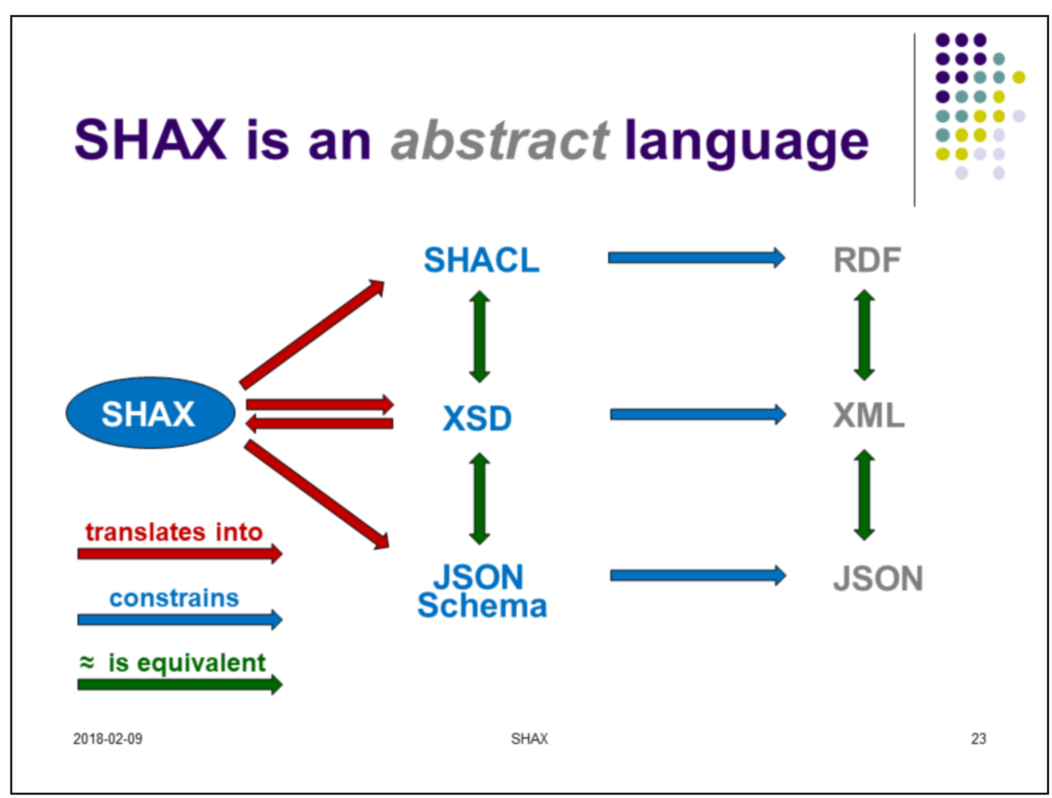
# SHAX – summary

*XSD equivalent*

- Building blocks
  - objectType ~ xs:complexType
  - dataType ~ xs:simpleType
  - property ~ xs:element (top-level)
  - import ~ xs:import, xs:include
- Advanced features
  - Union types ~ xs:union
  - Type extension ~ xs:extension
  - Substitution groups ~ @substitutionGroup

2018-02-09 SHAX 22

After the example, a brief summary of SHAX – its building blocks and advanced features. The example showed objectTypes and dataTypes. We also saw properties, but they were locally defined within the objectType. Properties can also be globally defined and referenced within the object types. Global properties are equivalent to top-level element declarations in XSD.




SHAX can be translated into SHACL – but also into XSD and JSON Schema. Thus SHAX can be viewed as an abstract modeling language: used to build abstract data models which constrain structures and data types, and yet do not presuppose a particular data representation language (XML, RDF, JSON). The next slides give you an impression of SHAX and its translations.

```
<shax:model xmlns:shax="http://shax.org/ns/model" xmlns:e="http://..."

  <shax:objectType name="e:PersonShape" targetClass="e:Person">
    <e:ssn       type="e:SsnShape"      card="?"/>
    <e:worksFor type="e:CompanyShape" card="*" ordered="true"/>
  </shax:objectType>


  <shax:objectType name="e:CompanyShape" targetClass="e:Company">
    <e:companyName            type="xsd:string"/>
    <shax:choice>
       <e:companyCode         type="xsd:string"/>
       <shax:pgroup>
          <e:repoCode         type="xsd:integer"/>
          <e:repoEntryNumber type="xsd:integer"/>
       </shax:pgroup>
    </shax:choice>
  </shax:objectType>


  <shax:dataType name="e:SsnShape"
                 base="xsd:string"
                 pattern="^\\d{3}-\\d{2}-\\d{4}$"/>
</shax:model>
```

SHAX

First, once more, our little SHAX model.

```
<xs:schema xmlns:a="http://example.org/ns/model"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://example.org/ns/model"
           elementFormDefault="qualified">
  <xs:complexType name="PersonShape">
    <xs:sequence>
      <xs:element name="ssn"      type="a:SsnShape"     minOccurs="0"/>
      <xs:element name="worksFor" type="a:CompanyShape" minOccurs="0"
                                                        maxOccurs="unbounded"
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="CompanyShape">
    <xs:sequence>
      <xs:element name="companyName"         type="xs:string"/>
      <xs:choice>
        <xs:element name="companyCode"       type="xs:string"/>
        <xs:sequence>
          <xs:element name="repoCode"        type="xs:integer"/>
          <xs:element name="repoEntryNumber" type="xs:integer"/>
        </xs:sequence>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="SsnShape">
    <xs:restriction base="xs:string">
      <xs:pattern value="\\d{3}-\\d{2}-\\d{4}"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

XSD

It can be translated into this XSD ....

# JSON Schema

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "type":"object",
  "properties":{
    "Person":{
      "type":"object",
      "properties":{
        "ssn":{
          "$ref":"#/definitions/a:SsnShape"
        },
        "worksFor":{
          "type":"array",
          "maxItems":999,
          "items":{
            "type":"object",
            "properties":{
              "companyName":{
                "$ref":"#/definitions/xs:string"
              },
              "companyCode":{
                "$ref":"#/definitions/xs:string"
              },
              "repoCode":{
                "$ref":"#/definitions/xs:integer"
              },
              "repoEntryNumber":{
                "$ref":"#/definitions/xs:integer"
              }
            },
            "additionalProperties":false,
            "required":[
              "companyName"
            ],
            "oneOf":[
              {
                "required":[
                  "companyCode"
                ]
              },
              {
                "required":[
                  "repoCode",
                  "repoEntryNumber"
                ]
              }
            ]
          }
        }
      },
      "additionalProperties":false
    }
  },
  "definitions":{
    "a:SsnShape":{
      "type":"string",
      "pattern":"^\\\\d{3}-\\\\d{2}-\\\\d{4}$"
    },
    "xs:integer":{
      "type":"integer"
    },
    "xs:string":{
      "type":"string"
    }
  }
}
```

.... or into this JSON schema.

```
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix shax: <http://shax.org/ns/model#> .
@prefix _e: <http://shax.org/ns/model/element-equivalent#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix e: <http://example.org/ns/model#> .

e:PersonShape
   a sh:NodeShape;
   sh:targetClass e:Person;
   sh:property [
      sh:path e:ssn;
      sh:maxCount 1;
      sh:node e:SsnShape;
   ];
   sh:property [
      sh:path e:worksFor;
      sh:node shax:ListType;
      sh:property [
         sh:path ([sh:zeroOrMorePath rdf:rest] rdf:first);
         sh:node e:CompanyShape;
      ];
   ].
e:CompanyShape
   a sh:NodeShape;
   sh:targetClass e:Company;
   sh:property [
      sh:path e:companyName;
      sh:minCount 1;
      sh:maxCount 1;
      sh:datatype xsd:string;
   ];
   sh:xone (
      [
         a sh:NodeShape ;
         sh:not [
            a sh:NodeShape;
            sh:or (
               [
                  sh:path e:repoCode ;
                  sh:minCount 1 ;
               ]
               [
                  sh:path e:repoEntryNumber ;
                  sh:minCount 1 ;
               ]
            );
         ] ;
      sh:property [
            sh:path e:companyCode;
            sh:minCount 1;
            sh:maxCount 1;
            sh:datatype xsd:string;
         ] ;
      ]
      [
         a sh:NodeShape ;
         sh:not [
            sh:path e:companyCode;
            sh:minCount 1;
         ] ;
         sh:property [
            sh:path e:repoCode;
            sh:minCount 1;
            sh:maxCount 1;
            sh:datatype xsd:integer;
         ] ;
         sh:property [
            sh:path e:repoEntryNumber;
            sh:minCount 1;
            sh:maxCount 1;
            sh:datatype xs:integer;
         ] ;
      ]
   ).
e:SsnShape
   a sh:NodeShape;
   sh:datatype xsd:string;
   sh:pattern "^\\d{3}-\\d{2}-\\d{4}$".

shax:ListType
   a sh:NodeShape ;
   sh:xone (
      [   a sh:NodeShape ;
          sh:in (rdf:nil) ;
      ]
      [   a sh:NodeShape ;
          sh:property [
              sh:path rdf:first ;
              sh:minCount 1;
              sh:maxCount 1;
          ] ;
          sh:property [
              sh:path rdf:rest ;
              sh:minCount 1;
              sh:maxCount 1;
          ]
      ]
   ) .
```

SHACL

Or into this SHACL.

# SHAX can be *generated* ...

**from XSD**

2018-02-09 SHAX 28

SHAX is easy to read and write. But even more easy it is to generate them – from XSD. Thus tons of modeling work can be launched into the RDF space.

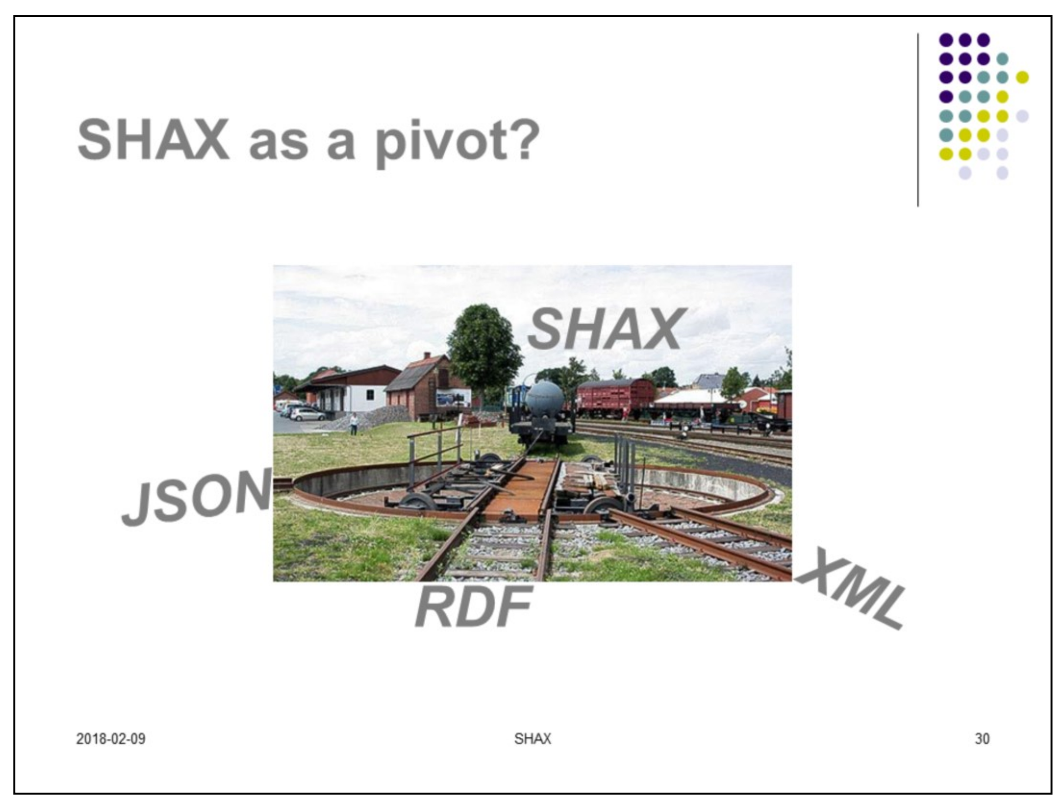
## SHAX processor

- SHAX => SHACL
- SHAX => XSD
- SHAX => JSON Schema
- XSD => SHAX

https://github.com/hrennau/shax

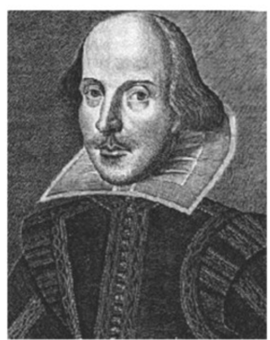
```
call shax "shacl?shax=person.shax"  > person.shacl.ttl
call shax "xsd  ?shax=person.shax"  > person.xsd
call shax "jsch ?shax=person.shax"  > person.jschema
call shax "shax ?xsd=person.xsd"    > person.shax
```

The translation work is accomplished by a SHAX processor. A prototype is available at github. Disclaimer: the translation into JSON Schema is still work in progress and will be released by the end of the month.




So I wonder if SHAX might be used as a pivot, a turning point where to bring your model work in order to let it travel into a different country. Much work ahead, to make everything as robust and comprehensive as needed, but I think the concept has been proved. Anybody showing interest – reporting bugs or requesting features – would help enormously.

# *Idea still in need of other minds ...*



***Shaxpeare***

2018-02-09 SHAX 31

In particular conceptionally, I feel that SHAX is an idea which is still in need of other minds. Perhaps someone will join me who shares my interest in elaborating the concept of an abstract modeling language – in connecting the seemingly unconnected. Could this be YOU?

# Thank you !

SHACL~~~XSD~~~JSON Schema
XSD~~~JSON Schema~~~SHACL
JSON Schema~~~SHACL~~~XSD
SHACL~~~XSD~~~JSON Schema
XSD~~~JSON Schema~~~SHACL
JSON Schema~~~SHACL~~~XSD

SHAX

SHAX

32