Combining graph and tree: writing SHAX, obtaining SHACL, XSD and more

Hans-Juergen Rennau, parsQube GmbH <hrennau@yahoo.de>

Abstract

The Shapes Constraint Language (SHACL) is a data modeling language for describing and validating RDF data. This paper introduces SHAX, which is an XML syntax for SHACL. SHAX documents are easy to write and understand. They cannot only be translated into executable SHACL, but also into XSD describing XML data equivalent to the RDF data constrained by the SHACL model. Similarly, SHAX can be translated into JSON Schema describing a JSON representation of the data. SHAX may thus be viewed as an abstract data modeling language, which does not prescribe a concrete representation language (RDF, XML, JSON, …), but can be translated into concrete models validating concrete model instances.

Table of Contents

Introduction

RDF ([7], [8]) defines a unified view of information which enables the integration of wildly heterogenous data sources into a uniform graph of RDF nodes. A similar quality has XML. If understood as a data model and technology, rather than a syntax, it can translate tree-structured documents irrespective of the domain of information, the vocabulary used and even the media type (XML, HTML, JSON, CSV, …) into a uniform forest of XDM nodes. Thus both, RDF and XML, offer us the translation of heterogeneous information landscapes into homogeneous spaces of nodes, amenable to powerful tools for addressing, navigating and transforming the contents of these spaces. In spite of the similarities, though, both technologies have different and complementary key strengths, and an integration of RDF and XML technologies is a very promising goal.

A key aspect of integration is how we think about and model RDF data and XML data. Integration might be built upon a careful alignment of these concepts and models, but we must acknowledge a serious mismatch between the major data modeling languages used to describe RDF (RDFS [9] and OWL [5], [6]) and XML data (XSD [16]), respectively. RDF languages are designed for inferencing, and XML modeling concentrates on data validation. The differences do not only pose technical problems, they also reflect quite different mindsets prevailing in the RDF and XML communities.

A new situation has been created by the recent standardization of the Shapes Constraint Language (SHACL, [10]), which is an RDF data modeling and validation language. This paper attempts to identify new possibilities and takes a step towards using them. It introduces SHAX (SHAcl+Xml), which is essentially an XML syntax for SHACL, but might also be regarded as a new and very simple data modeling language which can be translated into both, SHACL and XSD, and thus target RDF and XML data alike. SHAX models are simple to write and to understand, and they may narrow the gap between RDF and XML both conceptually and technically. In particular, they can provide a platform on which to build model-driven translations between RDF graphs and XML trees.

RDF and XML: one overarching abstraction

Integration of RDF and XML can be inspired by a clear perception of their essential relationship. At first sight, RDF seems to take a totally unique and “lonely” view of information, reducing it to the atomic unit of a triple: subject, predicate, object. However, when grouping these primary units by subject resource, a secondary unit emerges, which is an entity and its properties. Similarly, a complex XML element can be viewed as an entity enclosing information content. Both, properties and content amount to an itemized description of the entity. Through the equating of properties and content items, one approaches an overarching abstraction: an *information object*. A brief, informal account follows:

- Object abstraction
  - An object is a set of properties
  - A property has a name and one or more values
  - A property value is either a data value or an object value
- XML view
  - Object = complex-typed element
  - Property = attribute or child element
  - Property name = attribute or child element name
  - Property data value = attribute or a simple-typed child element
  - Property object value = complex-typed child element
- RDF view
  - Object = IRI or blank node
  - Property = predicate
  - Property name = predicate IRI
  - Property data value = a literal
  - Property object value = IRI or blank node

Compared to XML, RDF is marked by key strengths. It captures information in an *impartial* way: RDF predicates do not presuppose that the subject is “larger”, more general or more interesting than the object. XML represents a partial view, as the containment relationship is arbitrarily directed (does a book contain authors or an author books?), usually reflecting an application-based perspective.

A second aspect is the *amphibian* nature of RDF: while it allows to perceive a solid structure composed of objects (see above), RDF data is at the same time a mere collection of triples, without any primary structure whatsoever. It is like a liquid of information, and this implies a huge benefit: RDF graphs can be merged. Finally, RDF is a natural representation of *complex graphs* (where anything can be related to anything), and such graphs are a natural representation of most non-trivial systems.

XML, however, has unrivalled power of its own. Tree structure is a natural response of the human mind to complexity, as it enables the perception of a coherent whole. It allows a switching between alternative levels of detail. Due to the “spatial” organisation of tree content, navigation can be expressed with superb elegance (XPath). Leveraging this navigation engine, the transformation of trees into all kinds of artifacts – including graphs – is accomplished with amazing ease (XQuery, XSLT).

RDF and XML functioning in cooperation may be regarded as a technological dream team. RDF excels as the organisation of “potentially useful information”; XML excels as the organisation of “ready-for-use” information. How to combine these fundamental strengths? Ultimately, both technologies are about shaping and processing datasets. If datasets can be viewed as instances of data models, the integration of RDF and XML might leverage the knowledge pent up in these models. What is clearly called for is model-driven approaches.

RDF and XML: the challenge of integration

The integration of RDF and XML – or, more generally, graph and tree technology - has many interesting aspects. For example, if RDF data describe XML resources in terms of various metadata, XML navigation of a document forest (e.g. the internet) might be supported by SPARQL queries mapping conditions on resource metadata to resource URIs.

Distributed software components communicate via messages, which are usually tree-structured datasets. If the information required by a software component is RDF-based, the typical task at hand is thus the transformation of RDF graph data into an XML or JSON tree. Inversely, if an update of RDF data is necessary and the data is supplied by messages, the transformation of trees into graph data is required.

Remembering the appropriateness of RDF to serve as an unbiased representation of “potentially useful information”, the translation of RDF data into a ready-for-use form inspires great interest. For the purpose of this paper, we concentrate on the transformation of trees into graphs and of graphs into trees. Such transformation can always be accomplished by custom code. But given the close relationship between the RDF and XML models (argued in the previous section), transformation should be facilitated by standards or products. Table 1 compiles some approaches to the transformation between RDF and XML data.

Table 1. Standards and products supporting the transformation between RDF and XML data. T2G = „tree to graph“, G2T = „graph to tree“.

Name	Kind	Direction	Remark
RDFa	W3C recommendation	T2G	Defines markup used for embedding RDF triples in HTML and XML content
GRDDL	W3C recommendation	T2G	Defines the identification of a transformation resource (e.g. XSLT) to be used for extracting RDF triples from HTML or XML content
JSON-LD	W3C recommendation	T2G, G2T	Defines the flexible representation of RDF data by JSON trees, thus transformations in both directions
NIEM	Industry standard	T2G	Defines the equivalence of NIEM XML data with a set of RDF triples
SWP	Commercial product	G2T	A framework for creating HTML and XML content containing and controlled by RDF data
GraphQL	OpenSource product	G2T	Not concerned with RDF and XML, but with abstract representations of graphs and trees: defines the flexible transformation of graphs into trees

It is interesting to note that - apart from GraphQL ([1]) - none of these approaches builds on a data model describing the graph data in terms of type definitions that might be aligned with the building blocks of trees. Equally interesting is the fact that GraphQL is not concerned with RDF and XML data, but with a data source described by an abstract type system and a data target described by an abstract tree model. These observations have an explanation: before SHACL, there was no standardized data modeling language fit for supporting the transformation from/into graph data.

The advent of SHACL creates new possibilites. For the first time, integration might leverage an RDF data modeling language which can be aligned with an XML data modeling language - perhaps enabling the alignment of RDF data model components with XML data model components.

A short introduction to SHACL

The W3C recommendation ([10]) introduces SHACL as a "language for describing and validating RDF graphs." It is defined as an RDF vocabulary which can be used to describe data structures, data types and business rules. The key abstractions are constraints and shapes. A *constraint* is a condition which particular RDF nodes must meet. A *shape* is a set of constraints, which jointly must be adhered to. A shape can be thought of as a type, which may be a simple data type or an object type for resources with properties.

Let us consider a very simple example and try to model RDF data describing flight bookings. We decide that a flight booking is represented by a resource belonging to the RDF class `e:FlightBooking`. We want to introduce several constraints: a flight booking has …

- exactly one `e:BookingDate` property, which is a date
- exactly one `e:BookingID` property, which is a string with length >= 12 and <= 40
- an optional `e:BookingChannel`, which is an integer
- one or more `e:OperatingAirlines`, resources (IRIs or blank nodes) which have …
  - exactly one `e:AirlineCode` property, which is a string of two uppercase letters
  - exactly one `e:AirlineName` property, which is a string

This model is a fit for the following RDF graph, represented in Turtle syntax:

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix e:   <http://example.org/ns/model#> .
@prefix o:   <http://example.org/ns/objects#> .

o:FB101  a                   e:FlightBooking ;
         e:BookingChannel    2 ;
         e:BookingDate       "2017-12-31"^^xsd:date ;
         e:BookingID         "123456789XYZ" ;
         e:OperatingAirline  o:AL902 , o:AL901 .

o:AL902  e:AirlineCode       "KL" ;
         e:AirlineName       "KLM - Royal Dutch Airlines" .

o:AL901  e:AirlineCode       "LH" ;
         e:AirlineName       "Lufthansa" .
```

The graph can be validated with a SHACL model, which is itself an RDF graph. Here comes a SHACL model encoding the description given above, represented in Turtle syntax.

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

# object types
# ============
e:FlightBookingType
   a sh:NodeShape ;
   sh:targetClass e:FlightBooking ;
   sh:property [
      sh:path e:BookingID ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:node e:BookingIDType ;
   ] ;
   sh:property [
      sh:path e:BookingDate ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:datatype xsd:date ;
   ] ;
   sh:property [
      sh:path e:BookingChannel ;
      sh:maxCount 1 ;
      sh:datatype xsd:integer ;
   ] ;
   sh:property [
      sh:path e:OperatingAirline ;
      sh:minCount 1 ;
      sh:node e:AirlineType ;
   ] .

e:AirlineType
   a sh:NodeShape ;
   sh:property [
      sh:path e:AirlineCode ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:node e:AirlineCodeType ;
   ] ;
   sh:property [
      sh:path e:AirlineName ;
      sh:maxCount 1 ;
      sh:datatype xsd:string ;
   ] .

# data types
# ==========
e:BookingIDType
   a sh:NodeShape ;
   sh:datatype xsd:string ;
   sh:minLength 12 ;
   sh:maxLength 40 .

e:AirlineCodeType
   a sh:NodeShape ;
   sh:datatype xsd:string ;
   sh:pattern "^[A-Z]{2}$" .
```

This structure can be read as follows: `e:FlightBookingType` is a shape modeling the members of a particular RDF class (`e:FlightBooking`), and each `sh:property` […] statement describes a mandatory or optional property of those resources, with a property name identified by `sh:path`:

```
e:FlightBookingType
   a sh:NodeShape ;
   sh:targetClass e:FlightBooking ;
   sh:property [sh:path: e:BookingID ; …];
   sh:property [sh:path: e:BookingDate ; …];
   sh:property [sh:path: e:BookingChannel ; …];
   sh:property [sh:path: e:OperatingAirline ; …];>
```

Further settings within the `sh:property[…]` statements constrain the property in terms of cardinality and value type, e.g.:

```
sh:property [
     sh:path e:BookingDate ;
     sh:minCount 1 ;
     sh:maxCount 1 ;
     sh:datatype xsd:date ;
  ] ;>
```

constrains the `e:BookingDate` property to have exactly one value which is a date.

The example demonstrated capabilities which are similar to those of the XSD language. XSD describes data structure in terms of *element contents* (attributes and child elements), whereas SHACL describes data structure in terms of *resource properties* (data and object properties). XSD constrains content items in terms of name, cardinality and a type which is either a simple data type or a model of (nested) element contents. SHACL constrains properties in terms of name, cardinality and a type which is either a simple data type or a model of (nested) resource properties.

Accepting the analogy between element contents and resource properties, one realizes that most capabilities of XSD can be expressed in SHACL. But SHACL has still more to offer. First, it supports constraints targeting *pairs* of properties – e.g. stating that the value of one property must be less than the value of another property. SHACL also allows the construction of complex constraints which are a *logical combination* of other constraints (`sh:and`, `sh:or`, `sh:not`, `sh:xone`). A property can for example be described as either a string or one of several alternative object types.

SHACL models can describe "virtual" properties, which do not correspond to a simple property IRI, but are defined as a property path expression. The following are a few examples:

```
sh:path ( ex:address ex:street )
sh:path [ sh:inversePath ex:parent ]
sh:path [ sh:alternativePath ( ex:father ex:mother ) ]
sh:path ( ex:reads [ sh:zeroOrMorePath ex:cites ] )
```

Perhaps most importantly, SHACL supports the use of *SPARQL queries* for expressing constraints of virtually unlimited complexity and specificity. Such expressions could be used to capture XSD features not explicitly supported (like choice groups, identity constraints and XSD 1.1 assertions), as well as arbitrary business rules which are typically validated by Schematron models.

To summarize, SHACL combines the capabilities of XSD and Schematron. Besides, it allows the construction of complex constraints and supports virtual properties.

SHAX – motivation

The SHACL language enables a modeling of RDF data which makes them understandable for anyone familiar with data modeling, not requiring a deep understanding of RDF. Such models facilitate the development of applications processing or creating RDF data. Apart from this mental level, SHACL-based validation enables elaborate guarantees of data quality, which may reduce code complexity.

How to use these potential benefits in practice? Several problems must be overcome:

- SHACL is defined as an *RDF vocabulary*, but only few IT professionals are familiar with RDF
- *Tool support* for processing RDF data is rather limited – transformation of SHACL data into other artifacts tends to be difficult
- A *grammar-oriented data model* - describing structured content in terms of a simple and generic content model – is not always straightforward to express because:
  - Choice groups require convoluted constructs (combining `sh:xone`, `sh:not` and `sh:and`)
  - The description of a single property may be spread over several `sh:property` statements

These difficulties might be overcome by an *XML representation* of SHACL models. It offers a simple tree structure which is trivial to write and read. An XML representation can be translated into object oriented structures (e.g. via JAXB [2]) or directly processed by powerful processing and transformation languages (XQuery [15] and XSLT [17]). The convolutions sometimes required to express basic grammatical constructs (choice groups and properties-as-building-blocks) can be hidden behind the façade of a straightforward expression of intent.

The promise of an XML representation goes beyond this simplification of writing, reading and processing of SHACL models. It may pave the way for *abstract data models*, which define data structures and data types, but refrain from prescribing the representation language, in particular RDF versus non-RDF expression like XML or JSON. This possibility could be realized by transforming the XML representation of a SHACL model alternatively into a SHACL model (applicable to RDF data) or an instance of a non-RDF data modeling language, like XSD ([16]) or JSON Schema ([3]), which describes a non-RDF representation of RDF content.

If the XML representation is designed to increase the abstraction level of a SHACL model, a single XML source might be used to generate alternative SHACL styles. An example is the use of local versus global shapes, comparable to XSD styles preferring local versus global types. Last but not least, an XML representation might prove a convenient platform for augmenting model components with *metadata* – for example metadata specifying a data source for constructing the item, or a data destination for dispatching the item.

The remainder of this paper introduces **SHAX** (= SHAcl+Xml), which can be regarded as an XML representation of (some core parts of) the SHACL language, or, alternatively, as a new data modeling language which may be translated into SHACL, XSD and JSON Schema.

SHAX – introductory examples

A SHAX model describes *object types* in terms of their *properties*. The following SHAX model describes a single object type (`e:FlightBookingType`) which represents a flight booking:

```
<shax:model defaultCard="1"
    xmlns:shax="http://shax.org" xmlns:e="http://example.org/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <!-- object types
         ============ -->
    <shax:objectType name="e:FlightBookingType" class="e:FlightBooking">
        <e:BookingID type="e:BookingIDType"/>
        <e:BookingDate type="xsd:date"/>
        <e:BookingChannel card="?" type="xsd:integer"/>
    </shax:objectType>

    <!-- data types
         ========== -->
    <shax:dataType name="e:BookingIDType"
                   base="xsd:string" minLen="12" maxLen="40"/>
</shax:model>
```

Flight bookings have three properties:

- A `BookingID`, which is a string with a length between 12 and 40 characters
- A `BookingDate`, which is an instance of `xsd:date`
- A `BookingChannel`, which is an instance of `xsd:integer`

The `BookingChannel` is optional (@card="?"), and the other properties are mandatory (@defaultCard="1"). The model is matched by the following RDF data:

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix e:   <http://example.org/ns/model#> .
```

```
@prefix o:   <http://example.org/ns/objects#> .

o:FB101 a                e:FlightBooking ;
        e:BookingChannel  2 ;
        e:BookingDate     "2017-12-31"^^xsd:date ;
        e:BookingID       "123456789XYZ" .>
```

After downloading the SHAX processor from github ([14]), we can translate our SHAX model into a SHACL model. The command line call:

```
    shax "shacl?shax=flightBooking01.shax"
```

produces the following model:

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

# object types
# ============
e:FlightBookingType
   a sh:NodeShape ;
   sh:targetClass e:FlightBooking ;
   sh:property [
      sh:path e:BookingID ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:node e:BookingIDType ;
   ] ;
   sh:property [
      sh:path e:BookingDate ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:datatype xsd:date ;
   ] ;
   sh:property [
      sh:path e:BookingChannel ;
      sh:maxCount 1 ;
      sh:datatype xsd:integer ;
   ] .

# data types
# ==========
e:BookingIDType
   a sh:NodeShape ;
   sh:datatype xsd:string ;
   sh:minLength 12 ;
   sh:maxLength 40 .>
```

In order to see this model at work, we launch the “SHACL playground” (`http://shacl.org`) in a browser and copy model and instance data into the areas labeled “Shapes Graph” and “Data Graph”. Using the “Update” buttons beneath both areas, we observe that the data are valid against the model. We also observe how various manipulations of instance data or the model cause validation errors (see area “Validation Report”, where any error reports appear).

Having seen that our SHAX model can be compiled into a functional SHACL model, we proceed to extend our model, adding a property `OperatingAirline`. This is an *object property*, as the property values are objects which have their own properties: a mandatory code (two uppercase letters) and an optional name (string). Here comes the new version:

```
<shax:model defaultCard="1"
    xmlns:shax="http://shax.org/ns/model"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:e="http://example.org/ns/model">

    <!-- object types
         ============ -->
    <shax:objectType name="e:FlightBookingType" class="e:FlightBooking">
        <e:BookingID type="e:BookingIDType"/>
        <e:BookingDate type="xsd:date"/>
        <e:BookingChannel card="?" type="xsd:integer"/>
        <e:OperatingAirline card="+" type="e:AirlineType"/>
    </shax:objectType>

    <shax:objectType name="e:AirlineType">
        <e:AirlineCode type="e:AirlineCodeType"/>
        <e:AirlineName card="?" type="xsd:string"/>
    </shax:objectType>

    <!-- data types
         ========== -->
    <shax:dataType name="e:BookingIDType"
                   base="xsd:string" minLen="12" maxLen="40"/>
    <shax:dataType name="e:AirlineCodeType"
                   base="xsd:string" pattern="^[A-Z]{2}$"/>

</shax:model>
```

Note the cardinality constraint (@card="+"), which means "one or more" values. In the final step we add a `Customer` property:

```
<shax:model defaultCard="1"
    xmlns:shax="http://shax.org/ns/model"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:e="http://example.org/ns/model">

    <!-- properties
         ========== -->
    <shax:property name="e:FlightBooking" type="e:FlightBookingType"/>

    <!-- object types
         ============ -->
    <shax:objectType name="e:FlightBookingType" class="e:FlightBooking">
        <e:BookingID type="e:BookingIDType"/>
        <e:BookingDate type="xsd:date"/>
        <e:BookingChannel card="?" type="xsd:integer"/>
        <e:OperatingAirline card="+" type="e:AirlineType"/>
        <e:Customer type="e:CustomerType"/>
    </shax:objectType>

    <shax:objectType name="e:AirlineType" class="e:Airline">
        <e:AirlineCode type="e:AirlineCodeType"/>
        <e:AirlineName card="?" type="xsd:string"/>
    </shax:objectType>

    <shax:objectType name="e:CustomerType" class="e:Customer">
        <e:LastName type="xs:string"/>
        <e:FirstName type="xs:string"/>
```

```
            <shax:choice>
                <e:PassportNumber type="xsd:string"/>
                <shax:pgroup>
                    <e:LoyaltyProgramCode type="e:LoyaltyProgramCodeType"/>
                    <e:LoyaltyProgramMemberID type="xsd:integer"/>
                </shax:pgroup>
            </shax:choice>
        </shax:objectType>

        <!-- data types
             ========== -->
        <shax:dataType name="e:BookingIDType"
                       base="xsd:string" minLen="12" maxLen="40"/>
        <shax:dataType name="e:AirlineCodeType"
                       base="xsd:string" pattern="^[A-Z]{2}$"/>
        <shax:dataType name="e:LoyaltyProgramCodeType"
                       base="xsd:integer" min="1" max="999"/>
    </shax:model>
```

Here comes a conforming RDF graph, this time represented in JSON-LD:

```
{
   "@context": {
      "xsd": "http://www.w3.org/2001/XMLSchema#",
      "e": "http://example.org/ns/model#",
      "o": "http://example.org/ns/objects#"
   },
   "@graph": [
      {
         "@id": "o:FB101",
         "@type": "e:FlightBooking",
         "e:BookingID": "123456789XYZ",
         "e:BookingDate": {"@value": "2017-12-31", "@type": "xsd:date"},
         "e:BookingChannel": 2,
         "e:OperatingAirline": [
             {
                "@id": "o:AL901",
                "e:AirlineCode": "LH",
                "e:AirlineName": "Lufthansa"
             },
             {
                "@id": "o:AL902",
                "e:AirlineCode": "KL",
                "e:AirlineName": "KLM - Royal Dutch Airlines"
             }
         ],
         "e:Customer": {
            "e:LastName": "Perez",
            "e:FirstName": "Deborah",
            "e:LoyaltyProgramCode": 800,
            "e:LoyaltyProgramMemberID": 81557
         }
      }
   ]
}
```

The SHAX model is translated into a SHACL model which is more than three times as long (see Appendix A, *SHACL model obtained from SHAX*). Note that the SHACL model does implement the choice group structure (see the `sh:xone` element), but it does not make this structure explicit.

To summarize, this section demonstrated various features of SHAX models, including object and data types, cardinality constraints and the grammatical structure of a choice group. The next section gives a more systematic overview of the SHAX language.

SHAX – building blocks

A SHAX model is represented by a `shax:model` element which contains various kinds of top-level model components:

- `shax:objectType` – defines an object type
- `shax:dataType` – defines a data type
- `shax:property` – defines a property which can be referenced by object types

Models can import other models, using `shax:import` elements.

Element `shax:model`

A SHAX model is represented by a `shax:model` element. It contains elements importing other models (`shax:import`), as well as elements representing top-level model components. The `shax:model` element has an optional @defaultCard attribute which sets a default value for cardinality constraints (@card attributes, see below). In the absence of an @defaultCard attribute, the default cardinality is “exactly one”. Note that a SHAX model does not have a target namespace.

Element `shax:objectType`

An object type is a named model of resources. Above all, it is a description of the properties which adhering resources may have. Besides, the model can constrain the resources to belong to one or more (RDF) classes and to have a certain node kind (IRI or blank node). For example:

```
<shax:objectType name="e:BookingType" class="e:Booking" nodeKind="IRI">…
```

The possible properties of an object are described by *property declarations*, which are represented by elements named after the property in question. The @card attribute expresses a cardinality constraint (?, *, +, i, i-j), and the @type attribute constrains the property values to have a particular object or data type. A data type may be a built-in type (if from the XSD namespace) or a model-defined data type (see below, `shax:dataType`). For example:

```
<shax:objectType name="e:BookingType" class="e:Booking" nodeKind="IRI">
    <e:BookingID type="e:BookingIDType"/>
    <e:BookingDate type="xsd:date"/>
    <e:BookingChannel card="?" type="xsd:integer"/>
</shax:objectType>
```

Note that the order of property declarations is significant information. An object type can *extend* another one, which means that the property declarations of the extended type can be thought of as preceding the properties of the extending type. For example:

```
<shax:objectType name="e:FlightBookingType" extends"e:BookingType">
    <e:OperatingAirline card="+" type="e:AirlineType"/>
</shax:objectType>
```

A property declaration may also be a reference to a globally defined property. Reference is implicit, recognized by the absence of a @type attribute. The object or data type of the property is specified by the referenced property declaration:

```
<shax:objectType name="e:FlightBookingType" extends"e:BookingType">
    <e:OperatingAirline card="+"/>
</shax:objectType>
```

```
   …
<shax:property name="e:OperatingAirline" type="e:AirlineType"/>
```

Note that property references can be used for defining extension points, described below. An object model may include *choices* of properties, represented by a `shax:choice` element with one child element per alternative. An alternative is either a single property or a group of properties (wrapped in a `shax:pgroup` element). Choices can be nested. For example:

```
<shax:choice>
     <e:PassportNumber type="xsd:string"/>
     <shax:pgroup>
         <e:LoyaltyProgramCode type="e:LoyaltyProgramCodeType"/>
         <shax:choice>
             <e:LoyaltyProgramMemberID type="xsd:integer"/>
             <e:PreliminaryMemberID type="xsd:string"/>
         <shax:choice>
     </shax:pgroup>
</shax:choice>
```

Element `shax:property`

A `shax:property` element defines a property in a *global* way, independent of the object type which uses it. The declaration specifies a property name (@name) and a value type (@type), which is the IRI of an object or data type. An optional @substitutes attribute specifies the names of one or more (global) properties which the given property may *substitute*. This means that references to the specified properties are also matched by properties adhering to the substituting property declaration. Note the constraint that the substituting property declaration must have a type which extends or restricts the types of all substitutable property declarations.

Substitution can be used to define *extension points*, a property representing a logical choice between the property itself and all other property declarations that are able to substitute it. For example:

```
<shax:objectType name="e:FlightBookingType" extends"e:BookingType">
    <e:Airline card="+"/>
</shax:objectType>
   …
<shax:property name="e:Airline" type="e:AirlineType"/>
<shax:property name="e:OperatingAirline" type="e:AirlineType"
               substitutes="e:Airline"/>
<shax:property name="e:MarketingAirline" type="e:MarketingAirlineType"
               substitutes="e:Airline"/>
   …
<shax:objectType name="e:MarketingAirlineType" extends="e:AirlineType">…
```

Element `shax:dataType`

A data type is a set of constraints applicable to data values. The concept of a data type follows the approach taken by XSD:

- A data type is either atomic, or a list type, or a union type
- An atomic type associates a base type with constraints, called facets
- A list type describes an ordered sequence of atomic items (in RDF represented as `rdf:List`)
- A union type is defined as a choice between two or more data types

A data type is represented by a `shax:dataType` element. An atomic type has a @base attribute specifying the base type, and most facets are specified by attributes (@pattern, @len, @minLen, @maxLen, @min, @minEx, @max, @maxEx). The enumeration facet, which specifies a list of all possible values, is represented by `shax:value` child elements. Examples:

```
<shax:dataType name="e:AirlineCodeType"
               base="xsd:string" pattern="^[A-Z]{2}$"/>

<shax:dataType name="e:GenderType" base="xsd:string">
    <shax:value>male</shax:value>
    <shax:value>female</shax:value>
</shax:dataType/>
```

A list type definition specifies the item type with an @itemType attribute; optional @minSize, @maxSize and @size attributes constrain the number of list items. A union type definition specifies the alternative types with a @memberTypes attribute. Examples:

```
<shax:dataType name="e:AirlineCodesType"
               itemType="xsd:AirlineCodeType"
               minSize="1">

<shax:dataType name="e:ExtendedCurrencyType"
               memberTypes="xsd:CurrencyCodeType xsd:int">
```

Translation into concrete data modeling languages

SHAX can be used for constructing *abstract data models* – models defining structures of information content, but not prescribing the data format of concrete representations. In order to allow *validation* of data against a SHAX model, data can be represented in different formats:

- RDF (any serialization)
- XML
- JSON

For each of these formats, a SHAX model can be translated into an appropriate data validation language (SHACL, XSD, JSON Schema). Note that these translations may involve a certain loss of information. The sequence of properties, for example, is not preserved when translating into JSON Schema, and when translating into SHACL, although it is preserved as information (using `sh:order`), it loses its impact on validation. Another example is property pair constraints (e.g. @shax:lessThan) which are only preserved when translating into SHACL.

SHAX into SHACL

The translation into a SHACL model can be performed in two different styles. These styles are similar to the XSD styles using only global types ("flat" style) or only local types ("deep" style):

- Flat – every SHAX type reference (@type) is translated into a `sh:node` constraint
- Deep – every SHAX type reference is replaced by the constraints defined by the referenced type

The flat style ensures a well-modularized SHACL model. A disadvantage concerns the error messages in case of validation errors, which are less specific. Messages from flat/deep SHACL:

```
sh:resultMessage "Value does not have shape e:BookingIDType" ;
sh:resultMessage "Value has less than 12 characters" ;
```

SHAX into XSD

The XSD(s) obtained from a SHAX model describe XML documents which are a canonical representation of a model instance. The definition of this representation (not formally specified here) has been inspired by the definition of XML/RDF equivalence described in the NIEM Naming and Design Rules ([4], section 5.6). Key points are the mapping of property IRIs to element names and the representation of resource IRIs by dedicated attributes (@shax:IRI). The following listing shows an XML representation of the RDF data shown above (Model instance, RDF).

```
<e:FlightBooking xmlns:shax="http://shax.org/ns/model"
                 xmlns:e="http://example.org/ns/model"
                 xmlns:o="http://example.org/ns/objects"
                 shax:IRI="o:FB101">
    <e:BookingID>123456789XYZ</e:BookingID>
    <e:BookingDate>2017-12-31</e:BookingDate>
    <e:OperatingAirline shax:IRI="o:AL901">
        <e:AirlineCode>LH</e:AirlineCode>
        <e:AirlineName>Lufthansa</e:AirlineName>
    </e:OperatingAirline>
    <e:OperatingAirline shax:IRI="o:AL902">
        <e:AirlineCode>KL</e:AirlineCode>
        <e:AirlineName>KLM - Royal Dutch Airlines</e:AirlineName>
    </e:OperatingAirline>
    <e:Customer>
        <e:LastName>Perez</e:LastName>
        <e:FirstName>Deborah</e:FirstName>
        <e:LoyaltyProgramCode>800</e:LoyaltyProgramCode>
        <e:LoyaltyProgramMemberID>81557</e:LoyaltyProgramMemberID>
    </e:Customer>
</e:FlightBooking>
```

The translation of SHAX into XSD is straightforward:

- `shax:dataType` is translated into `xs:simpleType`
- `shax:objectType` is translated into `xs:complexType`
- `shax:property` is translated into `xs:element`, which is top-level
- Property elements are translated into `xs:element` defined within complex type definitions; the element declaration has an @type or @ref attribute, dependent on whether or not the property element has a @type attribute

See appendix (Appendix B, *XSD model obtained from SHAX*) for a listing of the XSD obtained for the SHAX model described above (SHAX model example).

SHAX into JSON Schema

The JSON Schema obtained describes JSON documents which are a straightforward representation of a model instance. A somewhat primitive style is assumed which ignores namespaces and uses only local names. Investigation of an advanced representation model using JSON-LD is a work in progress. See appendix (Appendix C, *JSON Schema model obtained from SHAX*) for the JSON Schema obtained from the SHAX model described above (SHAX model example).

SHAX - implementation

A SHAX processor accomplishes the translation of SHAX models into SHACL, XSD and JSON Schema models. An implementation of a SHAX processor is available on github (`https://github.com/hrennau/shax`). The processor is provided as a command line tool. Example calls:

```
shax shacl?shax=airportSystem.shax
shax xsd?shax=airportSystem.shax
shax jschema?shax=airportSystem.shax
```

Prospect: SHAX for RDF-XML integration

Any transformation of RDF data into XML documents can be accomplished by custom code. However, in situations where source and/or target data are described by data models, it would be unsatisfactory if custom code were the only option. A model-driven alternative might define a way how available descriptions of source or target structures can control the transformation and thus significantly reduce the effort. A detailed investigation of such possibilities is beyond the scope of this paper, but an example is sketched. The general goal is to enable transformation implemented in a declarative way, that is, modeling the transformation, rather than coding it.

A transformation of RDF data into an XML representation might be defined in terms of an EFRUG model, which expresses transformation as the successive application of operations to RDF source data. The operations have a generic definition and respond to model facts in a configurable way. Starting with a root resource to be transformed …

- **EF** – expand and filter the resource properties recursively, ignoring irrelevant properties and expanding object property values by a nested representation of their own property values
- **R** – rename the properties, starting with a canonical translation into a qualified name and replacing those qualified names which are not satisfactory
- **U** – unwrap nodes, removing inner nodes and connecting their parent nodes to their child nodes, where simplification is desirable
- **G** – group nodes by introducing intermediate nodes which wrap subsets of related siblings, where additional structure is desirable

Note that the key step is a filtered iteration over properties, combined with recursive expansion of object properties. It is easy to implement such an iteration, given a graph model expressed in SHAX, and it should not be difficult to render the filtering configurable, perhaps using whitelists and blacklists of regular expressions.

Discussion

The SHAX language as described in this paper is only a preliminary version serving as a proof of concept. It focuses on a subset of the SHACL language which enables the definition of a traditional object type system, as this subset is key for aligning RDF models and tree models like XSD. Future work should strive for a steady increase of the supported capabilities of SHACL. It is to be hoped that the simplicity and clarity of the SHAX language will be preserved in the process.

We regard the SHACL language as an important advance, which makes the introduction of an abstract data modeling language possible – a language capable of describing graph and tree encoded information. Key aspects of this language are the definition of equivalence between RDF and non-RDF representations, and the transformation of the abstract model into concrete, validating models for each major representation language.

The greatest problem ahead may not be a technical one, but a cultural one: to arouse in both communities, RDF and XML, an awareness of how complementary their technologies are, of how incomplete they are without each other.

A. SHACL model obtained from SHAX

The following listing shows the result of translating a SHAX model into SHACL.

Tip: You can experiment with this model here: `http://shacl.org`

- SHAX source model - see: SHAX model example
- Model instance, RDF - see: Model instance, RDF

```
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix xs:  <http://www.w3.org/2001/XMLSchema#> .
@prefix e: <http://example.org/ns/model#> .

# object types
# ============
e:FlightBookingType
   a sh:NodeShape ;
   sh:targetClass e:FlightBooking ;
   sh:property [
      sh:path e:BookingID ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:node e:BookingIDType ;
   ] ;
   sh:property [
      sh:path e:BookingDate ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:datatype xsd:date ;
   ] ;
   sh:property [
      sh:path e:BookingChannel ;
      sh:maxCount 1 ;
      sh:datatype xsd:integer ;
   ] ;
   sh:property [
      sh:path e:OperatingAirline ;
      sh:minCount 1 ;
      sh:node e:AirlineType ;
   ] ;
   sh:property [
      sh:path e:Customer ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:node e:CustomerType ;
   ] .

e:AirlineType
   a sh:NodeShape ;
   sh:targetClass e:Airline ;
   sh:property [
      sh:path e:AirlineCode ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:node e:AirlineCodeType ;
   ] ;
   sh:property [
      sh:path e:AirlineName ;
      sh:maxCount 1 ;
      sh:datatype xsd:string ;
   ] .
```

```
e:CustomerType
   a sh:NodeShape ;
   sh:targetClass e:Customer ;
   sh:property [
      sh:path e:LastName ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:datatype xs:string ;
   ] ;
   sh:property [
      sh:path e:FirstName ;
      sh:minCount 1 ;
      sh:maxCount 1 ;
      sh:datatype xs:string ;
   ] ;
   sh:xone (
      [
         a sh:NodeShape ;
         sh:not
            [
               a sh:NodeShape ;
               sh:or (
                  [
                     sh:path e:LoyaltyProgramCode ;
                     sh:minCount 1 ;
                  ]
                  [
                     sh:path e:LoyaltyProgramMemberID ;
                     sh:minCount 1 ;
                  ]
               ) ;
            ] ;
         sh:property [
            sh:path e:PassportNumber ;
            sh:minCount 1 ;
            sh:maxCount 1 ;
            sh:datatype xsd:string ;
         ] ;
      ]
      [
         a sh:NodeShape ;
         sh:not
            [
               sh:path e:PassportNumber ;
               sh:minCount 1 ;
            ] ;
         sh:property [
            sh:path e:LoyaltyProgramCode ;
            sh:minCount 1 ;
            sh:maxCount 1 ;
            sh:node e:LoyaltyProgramCodeType ;
         ] ;
         sh:property [
            sh:path e:LoyaltyProgramMemberID ;
            sh:minCount 1 ;
            sh:maxCount 1 ;
            sh:datatype xsd:integer ;
         ] ;
      ]
   ) .

# data types
# ==========
e:BookingIDType
   a sh:NodeShape ;
   sh:datatype xsd:string ;
   sh:minLength 12 ;
   sh:maxLength 40 .

e:AirlineCodeType
   a sh:NodeShape ;
   sh:datatype xsd:string ;
   sh:pattern "^[A-Z]{2}$" .

e:LoyaltyProgramCodeType
   a sh:NodeShape ;
   sh:datatype xsd:integer ;
   sh:minInclusive 1 ;
   sh:maxInclusive 999 .
```

B. XSD model obtained from SHAX

The following listing shows the result of translating a SHAX model into XSD.

- SHAX source model - see: SHAX model example
- Model instance, RDF - see: Model instance, RDF
- Model instance, XML - see: Model instance, XML

```
<xs:schema xmlns:shax="http://shax.org/ns/model"
           xmlns:e="http://example.org/ns/model"
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://example.org/ns/model"
           elementFormDefault="qualified">
  <xs:import namespace="http://shax.org/ns/model" schemaLocation="shax.xsd"/>
  <xs:element name="FlightBooking" type="e:FlightBookingType"/>
  <xs:complexType name="FlightBookingType">
    <xs:complexContent>
      <xs:extension base="shax:objectBaseType">
        <xs:sequence>
          <xs:element name="BookingID" type="e:BookingIDType"/>
          <xs:element name="BookingDate" type="xs:date"/>
          <xs:element name="BookingChannel" minOccurs="0" type="xs:integer"/>
          <xs:element name="OperatingAirline" maxOccurs="unbounded"
                      type="e:AirlineType"/>
          <xs:element name="Customer" type="e:CustomerType"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="AirlineType">
    <xs:complexContent>
      <xs:extension base="shax:objectBaseType">
        <xs:sequence>
          <xs:element name="AirlineCode" type="e:AirlineCodeType"/>
          <xs:element name="AirlineName" minOccurs="0" type="xs:string"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="CustomerType">
    <xs:complexContent>
      <xs:extension base="shax:objectBaseType">
        <xs:sequence>
          <xs:element name="LastName" type="xs:string"/>
          <xs:element name="FirstName" type="xs:string"/>
          <xs:choice>
            <xs:element name="PassportNumber" type="xs:string"/>
            <xs:sequence>
              <xs:element name="LoyaltyProgramCode"
                          type="e:LoyaltyProgramCodeType"/>
              <xs:element name="LoyaltyProgramMemberID" type="xs:integer"/>
            </xs:sequence>
          </xs:choice>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:simpleType name="BookingIDType">
    <xs:restriction base="xs:string">
      <xs:minLength value="12"/>
      <xs:maxLength value="40"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="AirlineCodeType">
    <xs:restriction base="xs:string">
      <xs:pattern value="[A-Z]{2}"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="LoyaltyProgramCodeType">
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="1"/>
      <xs:maxInclusive value="999"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

C. JSON Schema model obtained from SHAX

The following listing shows the result of translating a SHAX model into JSON schema.

Tip: You can experiment with this schema here: `https://www.jsonschemavalidator.net/`

- SHAX source model - see: SHAX model example
- Model instance, RDF - see: Model instance, RDF

```
{
  "$schema":"http://json-schema.org/draft-04/schema#",
  "type":"object",
  "properties":{
    "FlightBooking":{
      "type":"object",
      "properties":{
```

```
"IRI":{
  "$ref":"#/definitions/xs:anyURI"
},
"BookingID":{
  "$ref":"#/definitions/a:BookingIDType"
},
"BookingDate":{
  "$ref":"#/definitions/xs:date"
},
"BookingChannel":{
  "$ref":"#/definitions/xs:integer"
},
"OperatingAirline":{
  "type":"array",
  "minItems":1,
  "maxItems":999,
  "items":{
    "type":"object",
    "properties":{
      "IRI":{
        "$ref":"#/definitions/xs:anyURI"
      },
      "AirlineCode":{
        "$ref":"#/definitions/a:AirlineCodeType"
      },
      "AirlineName":{
        "$ref":"#/definitions/xs:string"
      }
    },
    "additionalProperties":false
  }
},
"Customer":{
  "type":"object",
  "properties":{
    "IRI":{
      "$ref":"#/definitions/xs:anyURI"
    },
    "LastName":{
      "$ref":"#/definitions/xs:string"
    },
    "FirstName":{
      "$ref":"#/definitions/xs:string"
    },
    "PassportNumber":{
      "$ref":"#/definitions/xs:string"
    },
    "LoyaltyProgramCode":{
      "$ref":"#/definitions/a:LoyaltyProgramCodeType"
    },
    "LoyaltyProgramMemberID":{
      "$ref":"#/definitions/xs:integer"
    }
  },
  "additionalProperties":false,
  "required":[
    "LastName",
    "FirstName"
```

```
          ],
          "oneOf":[
            {
              "required":[
                "PassportNumber"
              ]
            },
            {
              "required":[
                "LoyaltyProgramCode",
                "LoyaltyProgramMemberID"
              ]
            }
          ]
        }
      },
      "additionalProperties":false,
      "required":[
        "BookingID",
        "BookingDate",
        "OperatingAirline",
        "Customer"
      ]
    }
  },
  "definitions":{
    "a:AirlineCodeType":{
      "type":"string",
      "pattern":"^[A-Z]{2}$"
    },
    "a:BookingIDType":{
      "type":"string",
      "minLength":12,
      "maxLength":40
    },
    "a:LoyaltyProgramCodeType":{
      "type":"integer",
      "minimum":1,
      "maximum":999
    },
    "xs:anyURI":{
      "type":"string"
    },
    "xs:date":{
      "type":"string"
    },
    "xs:integer":{
      "type":"integer"
    },
    "xs:string":{
      "type":"string"
    }
  }
}
```

Bibliography

[1] *GraphQL*. 2017. Facebook Inc.. http://graphql.org/.

[2] *Java Architecture for XML Binding (JAXB)*. 2017. Java Community Process. https://jcp.org/en/jsr/detail?id=222.

[3] *JSON Schema Validation: A Vocabulary for Structural Validation of JSON*. 2017. Internet Engineering Task Force. http://json-schema.org/latest/json-schema-validation.html.

[4] *National Information Exchange Model Naming and Design Rules (Version 4.0)*. 2017. NIEM Program Management Office (PMO). https://reference.niem.gov/niem/specification/naming-and-design-rules/4.0rc1/niem-ndr-4.0rc1.html.

[5] *OWL 2 Web Ontology Language Primer (Second Edition)*. 2012. World Wide Web Consortium (W3C). https://www.w3.org/TR/owl2-primer/.

[6] *OWL 2 Web Ontology Language Quick Reference Guide (Second Edition)*. 2012. World Wide Web Consortium (W3C). https://www.w3.org/TR/owl-quick-reference/.

[7] *RDF 1.1 Concepts and Abstract Syntax*. 2014. World Wide Web Consortium (W3C). https://www.w3.org/TR/rdf11-concepts/.

[8] *RDF 1.1 Primer*. 2014. World Wide Web Consortium (W3C). https://www.w3.org/TR/rdf11-primer/.

[9] *RDF Schema 1.1*. 2014. World Wide Web Consortium (W3C). https://www.w3.org/TR/rdf-schema/.

[10] *Shapes Constraint Language (SHACL)*. 2017. World Wide Web Consortium (W3C). https://www.w3.org/TR/shacl/.

[11] *SHACL Advanced Features*. 2017. World Wide Web Consortium (W3C). https://www.w3.org/TR/shacl-af/.

[12] *SHACL JavaScript Extensions*. 2017. World Wide Web Consortium (W3C). https://www.w3.org/TR/shacl-js/.

[13] *SHACL and OWL Compared*. Holger Knublauch. 2017. World Wide Web Consortium (W3C). https://www.w3.org/TR/shacl-js/.

[14] *A SHAX processor, transforming SHAX models into SHACL, XSD and JSON Schema.*. Hans-Juergen Rennau. 2018. https://github.com/hrennau/shax.

[15] *XQuery 3.1: An XML Query Language*. 2017. World Wide Web Consortium (W3C). https://www.w3.org/TR/xquery-31/.

[16] *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. 2012. World Wide Web Consortium (W3C). https://www.w3.org/TR/xmlschema11-1/.

[17] *XSL Transformations (XSLT) Version 3.0*. 2017. World Wide Web Consortium (W3C). https://www.w3.org/TR/xslt/.