Greenfox – a schema language for validating file systems

Hans-Juergen Rennau, parsQube GmbH
<hans-juergen.rennau@parsqube.de>

Abstract

Greenfox is a schema language for validating file systems. One key feature is an abstract validation model inspired by the SHACL language. Another key feature is a view of the file system which is based on the XDM data model and thus supports a set of powerful expression languages (XPath, foxpath, XQuery). Using their expressions as basic building blocks, the schema language unifies navigation within and between resources and access to the structured contents of files with different mediatypes.

Table of Contents

Introduction

How to validate data against expectations? Major options are visual inspection, programatic checking and validation against a schema document (e.g. XSD, RelaxNG, Schematron, JSON Schema) or a schema graph (e.g. SHACL). Schema validation is in many scenarios the superior approach, as it is automated and declarative. But there are also limitations worth considering when thinking about validation in general.

First, schema languages describe instances of a particular format or mediatype only (e.g. XML, JSON, RDF), whereas typical projects involve a mixture of mediatypes. Therefore schema validation tends to describe the state of resources which are pieces from a jigsaw puzzle, and the question arises how to integrate the results into a coherent whole.

Second, several schema languages of key importance are grammar based and therefore do not support “incremental validation” – starting with a minimum of constraints, and adding more along the way. We cannot use XSD, RelaxNG or JSON Schema in order to express some very specific key expectation, without saying many things about the document as a whole, which may be a task requiring disproportional effort. Rule based schema languages (like Schematron) do support incremental validation, but they are inappropriate for comprehensive validation as accomplished by grammar based languages.

As a consequence, schema validation enables isolated acts of resource validation, but it cannot accomplish the integration of validation results. Put differently, schema validation may contribute to, but cannot accomplish, system validation. The situation might change in an interesting way if we had a schema language for validating *file system contents* – arbitrary trees of files and folders. This simple abstraction suffices to accommodate any software project, and it can accommodate system representations of very large complexity.

This document describes an early version of **greenfox**, a schema language for validating file system contents. By implication, it can also be viewed as a schema language for the validation of *systems*. Such a claim presupposes that a meaningful reflection of system properties, state and behaviour can be represented by a collection of data (log data, measurement results, test results, configurations, …) distributed over a set of files arranged in a tree of folders. It might then sometimes be possible to translate meaningful definitions of system validity into constraints on file system contents. At other times it may not be possible, for example if the assessment of validity requires a tracking of realtime data.

The notion of system validation implies that extensibility must be a key feature of the language. The language must not only offer a scope of expressiveness which is immediately useful. It must at the same time serve as a *framework*, within which current capabilities, future extensions and third-party contributions are uniform parts of a coherent whole. The approach we took is a generalization of the key concepts underlying SHACL [x], a validation language for RDF data. These concepts serve as the building blocks of a simple metamodel of validation, which offers guidance for extension work.

Validation relies on the key operations of navigation and comparison. File system validation must accomplish them in the face of divers mediatypes and the necessity to combine navigation within as well as between resources. In response to this challenge, greenfox is based on a *unified data model* (XDM) [x] and a *unified navigation model* (foxpath/XPath) [x] [x] [x] [x] built upon it.

Validation produces results, and the more complex the system, the more important it may become to produce results in a form which combines maximum precision with optimal conditions for integration with other resources. This goal is best served by a *vocabulary* for expressing validation results and schema contents in a way which does not require any context for being understood. We choose an RDF based definition of validation schema and validation results, combined with a bidirectional mapping between RDF and more intuitive representations, XML and JSON. For practical purposes, we assume the XML representation to be the form most frequently used.

Before providing a more detailed overview of the greenfox language, a detailed example should give a first impression of how the language can be used.

Getting started with greenfox

This section illustrates the development of a greenfox schema designed for validating a file system tree against a set of expections. Such a validation can also be viewed as validation of the system “behind” the file system tree, represented by its contents.

The system – system S

Consider **system S** – an imaginary system which is a collection of web services. We are going to validate a *file system representation* which is essentially a set of test results, accompanied by resources supporting validation (XSDs, codelists and data about expected response messages). The following listing shows a file system tree which is a representation of system S, as observed at a certain point in time:

```
system-s
. resources
. . codelists
. . . codelist-foo-article.xml
. . xsd
. . . schema-foo-article.xsd
. testcases
. . test-t1
. . . config
. . . . msg-config.xml
. . . input
. . . . getFooRQ*.xml
. . . output
. . . . getFooRS*.xml
. . +test-t2   (contents: see test-t1)
. . usecases
. . . usecase-u1
. . . . usecase-u1a
. . . . . +test-t3   (contents: see test-t1)
```

The concrete file system tree must be distinguished from the expected file system tree, which is described by the following rules.

Table 1. Rules defining "validity" of the considered file system.

File or folder	File path	Expectation
folder	resources/codelists	Contains one or more codelist files
file	resources/codelists/*	A codelist file; name not constrained; must be an XML document containing `<codelist>` elements with a @name attribute and `<entry>` children
folder	resources/xsd	Contains one or more XSDs describing services messages
file	resources/xsd/*	An XSD schema file; name not constrained
folder	.//test-*	A test case folder, containing `input`, `xoutput` and `config` folders; apart from these only optional `log-*` files are allowed
folder	.//test-*/config	Test case config folder, containing file `msg-config.csv`
file	.//test-*/config/msg-config.csv	A CSV file with three columns: request file name, response file name, expected return code
folder	.//test-*/input	Test case input folder, containg request messages

File or folder	File path	Expectation
file	.//test-*/input/*	A file representing a request message; name extension `.xml` or `.json`; mediatype corresponding to name extension
folder	.//test-*/output	Test case output folder, containing response messages
file	.//test-*/output/*	A file representing a response message; name extension `.xml` or `.json`; mediatype corresponding to name extension

The number and location of testcase folders (`test-*`) are unconstrained. This means that the testcase folders may be grouped and wrapped in any way, although they must not be nested. So the use of a `testcases` folder wrapping all testcase folders - and the use of `usecase*` folders adding additional substructure - is allowed, but must not be expected. The placing of XSDs in folder `resources/xsd`, on the other hand, is obligatory, and likewise the placing of codelist documents in folder `resources/codelists`. The names of XSD and codelist files are not constrained.

Structural expectations include also a conditional constraint:

- For every request message, there must be a response message with a name obtained by replacing in the request file name `RQ` with `RS` (e.g. `getFooRQ.*` and `getFooRS.*`)

Besides the structural expectations, there are also content-related expectations:

- For every response message in XML format, there is exactly one XSD against which it can be validated
- Every response message in XML format is valid against the appropriate XSD
- Response message items (XML elements or JSON fields) with name `fooValue` must be found in the codelist with name `foo-article`
- Response message return codes must be as configured by the corresponding row in `msg-config.csv` (applies to XML and JSON responses alike)

Building a greenfox schema "system S"

Now we create a greenfox schema which enables us to validate the file system against these expectations. An initial version only checks the existence of non-empty XSD and codelists folders:

```
<greenfox greenfoxURI="http://www.greenfox.org/ns/schema-examples/system-s"
          xmlns="http://www.greenfox.org/ns/schema">

  <domain path="\tt\greenfox\resources\example-system\system-s"
          name="system-s">

      <!-- *** System root folder shape *** -->
      <folder foxpath="." id="systemRootFolderShape">

        <!-- *** XSD folder shape -->
        <folder foxpath=".\\resources\xsd" id="xsdFolderShape">
            <targetSize msg="No XSD folder found" count="1"/>
            <file foxpath="*.xsd" id="xsdFileShape">
                <targetSize msg="No XSDs found" minCount="1"/>
            </file>
        </folder>

        <!-- *** Codelist folder shape -->
        <folder foxpath=".\\resources\codelists" id="codelistFolderShape">
            <targetSize msg="No codelist folder found" count="1"/>
            <file foxpath="*[is-xml(.)]" id="codelistFileShape">
```

```
                <targetSize msg="No codelist files found" minCount="1"/>
            </file>
          </folder
        </folder>
  </domain>
</greenfox>
```

The `<domain>` element represents the root folder of a file system tree to be validated. The folder is identified by a mandatory @path attribute.

A `<folder>` element describes a set of folders selected by a target declaration. Here, the target declaration is a foxpath expression, given by a @foxpath attribute. Foxpath [2] [3] [4] is an extended version of XPath 3.0 which supports file system navigation, node tree navigation and a mixing of file system and node tree navigation within a single path expression. Note that file system navigaton steps are connected by a backslash operator, rather than a slash, which is used for node tree navigation steps. The foxpath expression is evaluated in the context of a folder selected by the target declaration of the *containing* `<folder>` element (or `<domain>`, if there is no containing `<folder>`). Evaluation "in the context of a folder" means that the initial context item is the file path of that folder, so that relative file system path expressions are resolved in this context (see [3], [4] for details). For example, the expression

```
.\\resources\xsd
```

resolves to the `xsd` folders contained by a `resources` folder found at any depth under the context folder, `system-s`. Similarly, a `<file>` element describes the set of files selected by its target declaration, which is a foxpath expression evaluated in the context of a folder selected by the parent `<folder>`'s target declaration.

A `<folder>` element represents a **folder shape**, which is a set of **constraints** applying to a **target**. The target is a (possibly empty) set of folders, selected by a **target declaration**, e.g. a foxpath expression. The constraints of a folder shape are declared by child elements of the shape element.

Likewise, a `<file>` element represents a **file shape**, defining a set of constraints applying to a target, which is a set of files selected by a target declaration. Folder shapes and file shapes are collectively called **resource shapes**.

The expected number of folders or files belonging to the target of a shape can be expressed by declaring a **constraint**. A constraint has a kind (identified by the **constraint component IRI**) and a set of arguments passed to the **constraint parameters**. For every kind of constraint, a characteristic set of mandatory and optional constraint parameters is defined in terms of name, type and cardinality. In a schema document, a constraint is either declared by a *constraint element* or by *constraint attributes* attached to an element representing a set of constraints or a shape. Here, we declare a `TargetSize` constraint, which is represented by a `<targetSize>` child element of a file or folder shape. The element has three optional attributes, @minCount, @maxCount and @count, representing three different constraints. A constraint can be thought of as a function which consumes constraint parameter values and a *resource value*, representing the resource being validated; and which returns a validation result. Here, the resource value is the number of target resources selected, and the constraint parameter `minCount` is set to the value "1". If the constraint is violated, the validation result is a `<gx:red>` element which contains the message (if any) specified by @msg on the constraint element, along with a set of information items identifying the violating resource (@filePath), the constraint (@constraintComp and @constraintID) and its parameter values (@minCount). Example result:

```
<gx:red msg="No XSDs found"
    filePath="C:/tt/greenfox/resources/example-system/system-s/resources/xsd"
    constraintComp="targetMinCount"
    constraintID="TargetSize_2-minCount"
    resourceShapeID="xsdFileShape"
    minCount="1"
```

```
    actCount="0"
    targetFoxpath="*.xsd"/>
```

In a second step we extend our schema with a folder shape whose target consists of *all testcase folders in the system*:

```
<!-- *** Testcase folder shape *** -->
<folder foxpath=".\\test-*[input][output][config]" id="testcaseFolderShape">
  <targetSize msg="No testcase folders found" minCount="1"/>
  <folderContent
       closedMsg="Testcase member(s) other than input/output/config, log-*."
       closed="true">
    <memberFolders names="input, output, config"/>
    <memberFiles names="log-*" minCount="0" maxCount="*"/>
  </folderContent>
    …
</folder>
```

The target includes all folders found at any depth under the current context folder (`system-s`), matching the name pattern `test-*` and having (at least) three members `input`, `output` and `config`. The `<targetSize>` constraint checks that the system contains at least one such folder. The `<folderContent>` constraint is checked for each folder in the target, thus for each testcase folder. The constraint disallows any additional members except for optional files matching `log-*`, of which any number is allowed (note the @minCount and @maxCount attributes).

We proceed with a file shape which targets the `msg-config.csv` file in the config folder of the test case:

```
<!-- *** msg config file shape -->
  <file foxpath="config\msg-config.csv" id="msgConfigFileShape" ...>
    <targetSize msg="Config file missing" count="1"/>
    ...
  </file>
```

For any testcase folder which does not contain a file `config/msg-config.csv`, a violation of the `targetSize` constraint will be reported.

We want to be more specific: to constrain the *file contents*. The file must be a CSV file, and the third column (which according to the header row is called `returnCode`) must contain a value which is `OK` or `NOFIND` or matches the pattern `ERROR_*`. We add attributes to the `<file>` element which specify how to **parse the CSV file into an XML representation** (@mediatype, @csv.separator, @csv.header). As with other non-XML mediatypes (e.g. JSON or HTML), an XML view enables us to leverage XPath and *express* a selection of content items, preparing the data material for meaningful and complex validation.

We insert into the file shape an `<xpath>` element which describes a selection of content items and defines a constrait which these items must satisfy (expressed by the `<in>` child element):

```
<!-- *** msg config file shape -->
<file foxpath="config\msg-config.csv" id="msgConfigFileShape"
      mediatype="csv" csv.separator="," csv.withHeader="yes">
  ...
  <!-- *** Check - configured return codes ok? -->
  <xpath expr="//returnCode"
         inMsg="Config file contains unknown return code">
    <in>
      <eq>OK</eq>
      <eq>NOFIND</eq>
      <like>ERROR_*</like>
```

```
    </in>
  </xpath>
</file>
```

The item selection is defined by an XPath expression (provided by @expr), and the constraint is specified by the <in> child element: an item must either be equal to one of the strings “OK” or “NOFIND”, or it must match the glob pattern “ERROR_*”.

It is important to understand that the XPath expression is evaluated in the context of the **document node** of the document obtained by parsing the file. Here comes an example of a conformant message definition file:

```
request,response,returnCode
getFooRQ1.xml,getFooRS1.xml,OK
getFooRQ2.xml,getFooRS2.xml,NOFIND
getFooRQ3.xml,getFooRS3.xml,ERROR_SYSTEM
```

while this example violates the `xpath-in` constraint:

```
request,response,returnCode
getFooRQ1.xml,getFooRS1.xml,OK
getFooRQ2.xml,getFooRS2.xml,NOFIND
getFooRQ3.xml,getFooRS3.xml,ERROR-SYSTEM
```

According to the conceptual framework of greenfox, the `<xpath>` element does not, as one might expect, represent a constraint, but a **value shape**. A value shape is a container combining a single **value mapper** with a set of constraints: the value mapper maps the focus resource to a value (“resource value”), which is validated against each one of the constraints. Greenfox supports two kinds of value mapper – XPath expression and foxpath expression, and accordingly there are two variants of a value shape – **XPath value shape** (represented by an `<xpath>` element) and **Foxpath value shape** (`<foxpath>`). See section “Schema building blocks” for detailed information about value shapes.

We proceed to check *request message files*: for each such file, there must be a response file in the output folder, with a name derived from the request file name (replacing the last occurrence of substring “RQ” with “RS”). This is a constraint which does not depend on file contents, but on file system contents found “around” the focus resource. A check requires navigation of the file system, rather than file contents. We solve the problem with a Foxpath value shape:

```
<!-- *** Request file shape *** -->
<file foxpath="input\(*.xml, *.json)" id="requestFileShape">
  ...
  <!-- *** Check - request with response ? -->
  <foxpath
      expr="..\..\output\*\file-name(.)"
      containsXPathMsg="Request without response"
      containsXPath="$fileName !
                     replace(., '(.*)RQ(.*)$', '$1RS$2')"/>
```

A Foxpath value shape combines a foxpath expression (@expr) with a set of constraints. The expression maps the focus resource to a value, which is validated against all constraints. Here we have an expression which maps the focus resource to a list of file names found in the `output` folder. A single constraint, represented by the @containsXPath attribute, requires the expression value to contain the value of an XPath expression, which maps the request file name to the response file name. The constraint is satisfied if and only if the response file is present in the `output` folder.

As with XPath value shapes, it is important to be aware of the evaluation context. We have already seen that in an XPath value shape the initial context item is the *document node* obtained by parsing the text of the focus resource into an XML representation. In a Foxpath value shape the initial context item is the *file path* of the focus resource, which here is the file path of a request file. The foxpath expression

starts with two steps along the parent axis (`..\..`) which lead to the enclosing `testcase` folder, from which navigation to the response files and their mapping to file names is trivial:

```
..\..\output\*\file-name(.)
```

A Foxpath value shape does not require the focus resource to be parsed into a document, as the context is a file path, rather than a document node. Therefore, a Foxpath value shape can also be used in a folder shape. We use this possibility in order to constrain the `codelists` folder to contain `<codelist>` elements with a @name attribute and at least one `<entry>` child:

```
<!-- *** Codelist folder shape -->
<folder foxpath=".\\resources\codelists" id="codelistFolderShape">
    ...
    <!-- *** Check - folder contains codelists? -->
    <foxpath expr="*.xml/codelist[entry]/@name"
             minCoutMsg="Codelist folder without codelists"
             minCount="1"/>
    ...
</folder>
```

Note the aggregative view enabled by the foxpath language: we do not bother with individual files but perform a “mixed” navigation, starting with file system navigation to all *.xml files, continuing within their collected content (… `/codelist[entry]/@name`), arriving at @name attributes on non-empty `<codelist>` elements.

Now we turn to the *response message files*. They must be “fresh”, that is, have a timestamp of last modification which is after a limit timestamp provided by a call parameter of the system validation. This is accomplised by a `lastModified` constraint, which references the parameter value. Besides, response files must not be empty (`fileSize` constraint):

```
<!-- *** Response file shape *** -->
<file foxpath="output\(*.xml, *.json)" mediatype="xml-or-json">
    ...
    <!-- *** Check - response fresh? *** -->
    <lastModified ge="${lastModified}"
                  geMsg="Stale output file"/>

    <!-- *** Check - response non-empty? *** -->
    <fileSize gt="0"
              gtMsg="Empty output file"/>
    ...
</file>
```

The placeholder `${lastModified}` is substituted by the value passed to the greenfox processor as input parameter and declared in the schema as a *context parameter*:

```
<greenfox ... >
  <!-- *** External context *** -->
  <context>
    <field name="lastModified"/>
  </context>
  ...
</greenfox>
```

We have several expecations related to the contents of response files. If the response is an XML document (rather than JSON), it must be valid valid against some XSD found in the XSD folder. XSD validation is triggered by an `xsdValid` constraint, with a foxpath expression locating the XSD(s) to be used:

```
<!-- *** Response file shape *** -->
```

```
<file foxpath="output\(*.xml, *.json)" mediatype="xml-or-json">
    ...
    <!-- *** Check - schema valid? (only if XML) -->
    <ifMediatype eq="xml">
        <xsdValid msg="Response msg not XSD valid"
                  xsdFoxpath="$domain\resources\xsd\\*.xsd"/>
    </ifMediatype>
</file>
```

It is not necessary to specify an individual XSD – the greenfox processor inspects all XSDs matching the expression and selects for each file to be validated the appropriate XSD. This is achieved by comparing name and namespace of the root element with local name and target namespace of all element declarations found in the XSDs selected by the foxpath expression. If not exactly one element declaration is found, an error is reported, otherwise XSD validation is performed. Note the variable reference `$domain`, which can be referenced in any XPath or foxpath expression and which provides the file path of the domain folder.

The next condition to be checked is that certain values from the response (selected by XPath `//*:fooValue`) are found in a particular codelist. Here we use an XPath value shape which contains an `ExprValueEqFoxpath` constraint, represented by the @eqFoxpath attribute:

```
<!-- *** Response file shape *** -->
<file foxpath="output\(*.xml, *.json)" mediatype="xml-or-json">
    ...
   <!-- *** Check - known article number? -->
   <xpath expr="//*:fooValue"
          eqFoxpathMsg="Unknown foo article number"
          eqFoxpath="$domain\\codelists\*.xml
                     /codelist[@name eq 'foo-article']/entry/@code"/>

   </xpath>
</file>
```

As always with an XPath value shape, the XPath expression (@expr) selects the content items to be checked. The `ExprValueEqFoxpath` constraint works as follows: it evaluates the foxpath expression provided by constraint parameter `eqFoxpath` and checks that every item of the value to be checked also occurs in the value of the foxpath expression. As here the foxpath expression returns all entries of the appropriate codelist, the constraint is satisfied if and only if every `fooValue` element in the response contains a string found in the codelist.

Note that this value shape works properly for both, XML and JSON responses. Due to the @mediatype annotation on the file shape, which is set to `xml-or-json`, the greenfox processor first attempts to parse the file as an XML document. If this does not succeed, it attempts to parse the file as a JSON document and transform it into an equivalent XML representation. In either case, the XPath expression is evaluated in the context of the document node of the resulting XDM node tree. In such cases one has to make sure, of course, that the XPath expression can be used in both structures, original XML and XML capturing the JSON content, which is the case in our example.

As a last constraint, we want to check the return code of a response. The expected value can be retrieved from the message config file, a CSV file in the `config` folder: it is the value found in the third column (named `returnCode`) of the row in which the second column (named `response`) contains the file name of the response file. We use a Foxpath value shape with an expression fetching the expected return value from the CSV file. This is accomplished by a mixed navigation, starting with file system navigation leading to the CSV file, then drilling down into the file and fetching the item of interest. The value against which to compare is retrieved by a trivial XPath expression (@eqXPath):

```
<!-- *** Response file shape *** -->
<file foxpath="output\(*.xml, *.json)" mediatype="xml-or-json">
    ...
```

```
    <!-- *** Check - return code expected? *** -->
    <foxpath expr="..\..\config\msg-config.csv\csv-doc(., ',', 'yes')
                  //record[response eq $fileName]/returnCode"
             eqXPathMsg="Return code not the configured value"
             eqXPath="//*:returnCode"/>
    </foxpath>
</file>
```

The complete schema is shown in the appendix A1. To summarize, we have developed a schema which constrains the presence and contents of folders, the presence and contents of files, and relationships between contents of different files, in some cases belonging to different mediatypes. The devlopment of the schema demanded familiarity with XPath, but no programming skills beyond that.

Basic principles

The "Getting started" section has familiarized you with the basic building blocks and principles of greenfox schemas. They can be summarized as follows.

- A file system is thought of as containing two kinds of resources, **folders** and **files**
- Resources are validated against **resource shapes**
- There are two kinds of resource shapes – **folder shapes** and **file shapes**
- A resource shape is a set of **constraints** which apply to a resource being validated
- Every violation of a constraint produces a **validation result** describing the violation and identifying resource and constraint
- The resources validated against a shape are called its **focus resources**
- A resource shape may have a **target declaration** which selects a set of focus resources
- A target declaration can be a resource name, a relative file path or a foxpath expression
- Constraints can apply to **resource properties** like the last modification time or the file size
- Constraints can apply to a **resource value**, which is a value to which the resource is mapped by an expression
- A **value shape** combines an expression mapping the focus resource to a resource value, and a set of constraints against which to validate the resource value
- The expression used by a value shape may be an **XPath expression** or a **foxpath expression**
- The **foxpath context item** used by a value shape is the file path of the focus resource
- The **XPath context item** used by a value shape is the root of an XDM node tree representing the content of the focus resource, or the file path of the focus resource if an XDM node tree could not be constructed
- **XDM node tree representations** of file resources can be controlled by mediatype related attributes on a file shape
- When validating resources against resource shapes, the heterogeneity of mediatypes can be hidden by a **unified representation as XDM node tree**
- When validating resources against resource shapes, the heterogeneity of navigation (within resource contents and between resources) can be hidden by a **unified navigation language** (foxpath)

RDFe example

This section introduces RDFe by building an example in several steps.

Getting started

Consider an XML document describing drugs (contents taken from drugbank [2]):

```
<drugs xmlns="http://www.drugbank.ca">
  <drug type="biotech" created="2005-06-13" updated="2018-07-02">
```

```
    <drugbank-id primary="true">DB00001</drugbank-id>
    <drugbank-id>BTD00024</drugbank-id>
    <drugbank-id>BIOD00024</drugbank-id>
    <name>Lepirudin</name>
    <!-- more content here -->
    <pathways>
      <pathway>
        <!-- more content here -->
        <enzymes>
          <uniprot-id>P00734</uniprot-id>
          <uniprot-id>P00748</uniprot-id>
          <uniprot-id>P02452</uniprot-id>
          <!-- more content follows -->
        </enzymes>
      </pathway>
    </pathways>
    <!-- more content here -->
  </drug>
  <!-- more drugs here -->
</drugs>
```

We want to map parts of these descriptions to an RDF representation. First goals:

- Assign an IRI to each drug
- Construct triples describing the drug

The details are outlined in the table below. Within XPath expressions, variable $drug references the XML element representing the resource.

Table 2. A simple model deriving RDF resource descriptions from XML data.

Resource IRI expression (XPath)		
$drug/db:drugbank-id[@primary = 'true']/concat('drug:', .)		
Property IRI	**Property type**	**Property value expression (XPath)**
rdf:type	xs:string	'ont:drug'
ont:name	xs:string	$drug/name
ont:updated	xs:date	$drug/@updated
ont:drugbank-id	xs:string	$drug/db:drugbank-id[@primary = 'true']
ont:drugbank-altid	xs:string	$drug/db:drugbank-id[not(@primary = 'true')]
ont:enzyme	IRI	$drug//db:enzymes/db:uniprot-id/concat('uniprot:', .)

This model is easily translated into an RDFe document, also called a **semantic map:**

```
<re:semanticMap iri="http://example.com/semap/drugbank/"
                targetNamespace="http://www.drugbank.ca"
                targetName="drugs"
                xmlns:re="http://www.rdfe.org/ns/model"
                xmlns:xs="http://www.w3.org/2001/XMLSchema"
                xmlns:db="http://www.drugbank.ca">

  <re:namespace iri="http://example.com/resource/drug/" prefix="drug"/>
  <re:namespace iri="http://example.com/ontology/drugbank/" prefix="ont"/>
  <re:namespace iri="http://www.w3.org/2000/01/rdf-schema#" prefix="rdfs"/>
  <re:namespace iri="http://bio2rdf.org/uniprot:" prefix="uniprot"/>

  <re:resource modelID="drug"
```

```
                assertedTargetNodes="/db:drugs/db:drug"
                targetNodeNamespace="http://www.drugbank.ca"
                targetNodeName="drug"
                iri="db:drugbank-id[@primary = 'true']/concat('drug:', .)"
                type="ont:drug">
    <re:property iri="rdfs:label"
                 value="db:name"
                 type="xs:string"/>
    <re:property iri="ont:updated"
                 value="@updated"
                 type="xs:date"/>
    <re:property iri="ont:drugbank-id"
                 value="db:drugbank-id[@primary = 'true']"
                 type="xs:string"/>
    <re:property iri="ont:drugbank-alt-id"
                 value="db:drugbank-id[not(@primary = 'true')]"
                 type="xs:string"/>
    <re:property iri="ont:enzyme"
                 value=".//db:enzymes/db:uniprot-id/concat('uniprot:', .)"
                 type="#iri"/>
  </re:resource>

</re:semanticMap>
```

The triples are generated by an **RDFe processor**, to which we pass the XML document and the semantic map. Command line invocation:

```
  shax "rdfe?dox=drugs.xml,semap=drugbank.rdfe.xml"
```

The result is a set of RDF triples in Turtle [7] syntax:

```
@prefix drug: <http://example.com/resource/drug/> .
@prefix ont: <http://example.com/ontology/drugbank/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix uniprot: <http://bio2rdf.org/uniprot:> .

drug:DB00001
        rdf:type            ont:drug ;
        rdfs:label          "Lepirudin" ;
        ont:updated         "2018-07-02"^^xs:date ;
        ont:drugbank-id     "DB00001" ;
        ont:drugbank-alt-id "BTD00024" ;
        ont:drugbank-alt-id "BIOD00024" ;
        ont:enzyme          uniprot:P00734 ;
        ont:enzyme          uniprot:P00748 ;
        ont:enzyme          uniprot:P02452 ;
             …
drug:DB00002
        rdf:type ont:drug ;
             …
drug:DB00003
        rdf:type ont:drug ;
             …
…
```

Some explanations should enable a basic understanding of how the semantic map controls the output. The basic building block of a semantic map is a **resource model**. It defines how to construct the triples describing a resource represented by an XML node:

```
<re:resource modelID="drug"
             assertedTargetNodes="/db:drugs/db:drug"
             iri="db:drugbank-id[@primary eq 'true']/concat('drug:', .)"
             type="ont:drug">
  <re:property iri="rdfs:label"
    value="db:name"
    type="xs:string"/>
  <!-- more property models here -->
</re:resource>
```

The @iri attribute on `<resource>` provides an XPath expression yielding the resource IRI. The expression is evaluated *in the context of the XML node representing the resource*. Note how the expression language XPath is used in order to describe the IRI as a concatenation of a literal prefix and a data-dependent suffix. Every node returned by the expression in @assertedTargetNodes, evaluated *in the context of the input document*, is mapped to a resource description as specified by this resource model element.

Each `<property>` child element adds to the resource model a **property model**. It describes how to construct triples with a particular property IRI. The property IRI is given by @iri, and the property values are obtained by evaluating the expression in @value, *using the node representing the resource as context node*. (In our example, the value expressions are evaluated in the context of a `<drug>` element.) As the examples show, the XPath language may be used freely, for example combining navigation with other operations like concatenation. The datatype of the property values is specified by the @type attribute on `<property>`. The special value `#iri` signals that the value is an IRI, rather than a typed literal. Another special value, `#resource`, will be explained in the following section.

Linking resources

Our drug document references articles:

```
<drugs xmlns="http://www.drugbank.ca">
  <drug type="biotech" created="2005-06-13" updated="2018-07-02">
    <drugbank-id primary="true">DB00001</drugbank-id>
   <!-- more content here -->
    <general-references>
      <articles>
        <article>
          <pubmed-id>16244762</pubmed-id>
          <citation>
            Smythe MA, Stephens JL, Koerber JM, Mattson JC: A c…</citation>
        </article>
        <!-- more articles here -->
      </articles>
    </general-references>
   <!-- more content here -->
</drugs>
```

RDF is about connecting resources, and therefore our RDF data will be more valuable if the description of a drug references *article IRIs* which give access to *article resource descriptions* - rather than including properties with literal values which represent properties of the article in question, like its title and authors.

Assume we have access to a document describing articles:

```
<articles>
  <article>
    <pubmed-id>16244762</pubmed-id>
    <url>https://doi.org/10.1177/107602960501100403</url>
    <doi>10.1177/107602960501100403</doi>
```

```
    <authors>
      <author>Smythe MA</author>
      <author>Stephens JL</author>
      <author>Koerber JM</author>
      <author>Mattson JC</author>
    </authors>
    <title>A comparison of lepirudin and argatroban outcomes</title>
    <keywords>
      <keyword>Argatroban</keyword>
      <keyword>Lepirudin</keyword>
      <keyword>Direct thrombin inhibitors</keyword>
    </keywords>
    <citation>Smythe MA, Stephens JL, Koerber JM, Mattson JC: A ...</citation>
    <abstract> Although both argatroban and lepirudin are used ...</abstract>
  </article>
  <!—more articles here -->
</articles>
```

We write a second semantic map for this document about articles:

```
<re:semanticMap  iri="http://example.com/semap/articles/"
                 targetNamespace="" targetName="articles" …>
  <re:namespace iri="http://example.com/resource/article/" prefix="art"/>
  <!-- more namespace descriptors here -->
  <re:resource modelID="article" iri="pubmed-id/concat('art:', .)"
               targetNodeNamespace=""
               targetNodeName="article"
               type="ont:article">
   <re:property iri="ont:doi" value="doi" type="xs:string"/>
   <re:property iri="ont:url" value="url" type="xs:string"/>
   <re:property iri="ont:author" value=".//author" list="true" type="xs:string"
   <re:property iri="ont:title" value="title" type="xs:string"/>
   <re:property iri="ont:keyword" value="keywords/keyword" type="xs:string"/>
   <re:property iri="ont:abstract" value="abstract" type="xs:string"/>
   <re:property iri="ont:citation" value="citation" type="xs:string"/>
  </re:resource>
</re:semanticMap/>
```

and we extend the resource model of a drug by a property *referencing the article resource*, relying on its XML representation provided by an `<article>` element:

```
<re:property iri="ont:ref-article"
  value="for $id in .//db:article/db:pubmed-id return
         doc('/ress/drugbank/articles.xml')//article[pubmed-id eq $id]"
  type="#resource"/>
```

The value expression fetches the values of `<pubmed-id>` children of `<article>` elements contained by the `<drug>` element, and it uses these values in order to navigate to the corresponding `<article>` element *in a different document*. This document need not be provided by the initial input – documents can be discovered during processing. While the items obtained from the value expression are `<article>` *elements*, the triple objects must be article *IRIs* giving access to article *resource descriptions*. Therefore two things must be accomplished: first, the output must include triples describing the referenced articles; second, the `ont:ref-article` property of a drug must have an object which is the article IRI used as the subject of triples describing this article. The article IRI, as well as the triples describing the article are obtained by applying the *article resource model* to the article element. All this is accomplished by the RDFe processor whenever it detects the property type `#resource`. Our output is extended accordingly:

```
drug:DB00001 a ont:drug ;
```

```
   rdfs:label        "Lepirudin" ;
   …
   ont:ref-article   art:16244762 ;
         …
art:16244762 a ont:article ;
   ont:abstract  "Although both argatroban and lepirudin are used for ..." ;
   ont:author    "Stephens JL" , "Koerber JM" , "Mattson JC" , "Smythe MA" ;
   ont:citation  "Smythe MA, Stephens JL, Koerber JM, Mattson JC: A com … " ;
   ont:doi       "10.1177/107602960501100403" ;
   ont:keyword   "Argatroban" , "Lepirudin" , "Direct thrombin inhibitors" ;
   ont:title     "A comparison of lepirudin and argatroban outcomes" ;
   ont:url       "https://doi.org/10.1177/107602960501100403" .
```

Adding a dynamic context

The property model which we just added to the resource model for drugs contains a “difficult” value expression – an expression which is challenging to write, to read and to maintain:

```
  for $id in .//db:article/db:pubmed-id return
  doc('/products/drugbank/articles.xml')//article[pubmed-id eq $id]"
```

We can simplify the expression by defining a **dynamic context** and referencing a context variable. A `<context>` element represents the *constructor* of a dynamic context:

```
<re:semanticMap  iri="http://example.com/semap/drugbank/" …>
  ...
  <re:context>
    <re:var name="articlesURI" value="'/products/drugbank/articles.xml'"/>
    <re:var name="articlesDoc" value="doc($articlesURI)"/>
  </re:context>
      …
</re:semanticMap>
```

The values of context variables are specified by XPath expressions. Their evaluation context is the root element of an input document, so that variable values may reflect document contents. A context constructor is evaluated once for each input document. The context variables are available in any expression within the semantic map containing the context constructor (excepting expressions in preceding siblings of the `<var>` element defining the variable). Now we can simplify our expression to

```
  for $id in .//db:article/db:pubmed-id return
  $articlesDoc//article[pubmed-id eq $id]
```

As a context constructor may also define *functions*, we may further simplify the value expression by turning the navigation to the appropriate `<article>` element into a function. The function is defined by a `<fun>` child element of `<context>`. We define a function with a single formal parameter, which is a pubmed ID:

```
<re:context>
  <re:var name="articlesURI" value="'/products/drugbank/articles.xml'"/>
  <re:var name="articlesDoc" value="doc($articlesURI)"/>
  <re:fun name="getArticleElem" params="id"
          code="$articlesDoc//article[pubmed-id eq $id]"/>
  </re:function>
</re:context>
```

Expressions in this semantic map can reference the function by the name `getArticleElem`. A new version of the value expression is this:

```
.//db:article/db:pubmed-id/$getArticleElem(.)
```

For each input document a distinct instance of the context is constructed, using the document root as context node. This means that the context may reflect the contents of the input document. The following example demonstrates the possibility: in order to avoid repeated navigation to the `<article>` elements, we introduce a dictionary which maps all Pubmed IDs used in the input document to `<article>` elements:

```
<re:var name="articleElemDict"
        value="map:merge(distinct-values(//db:article/db:pubmed-id)
                         ! map:entry(., $getArticleElem(.)))"/>
```

An updated version of the value expression takes advantage of the dictionary:

```
.//db:article/db:pubmed-id/$articleElemDict(.)
```

The dictionary contains only those Pubmed IDs which are actually used in a particular input document. For each input document, a distinct instance of the dictionary is constructed, which is bound to the context variable `$articleElemDict` whenever data from that document are evaluated.

RDFe language

RDFe is an XML language for defining the mapping of XML documents to RDF triples. A mapping is described by one or more RDFe documents. An RDFe document has a `<semanticMap>` root element. All elements are in the namespace `http://www.rdfe.org/ns/model` and all attributes are in no namespace. Document contents are constrained by an XSD (found here: [8], `xsd` folder). The following treesheet representation [5] [13] of the schema uses the pseudo type `re:XPATH` in order to indicate that a string must be a valid XPath expression, version 3.1 or higher.

```
semanticMap
. @iri . ... ... ... ... ... ty: xs:anyURI
. @targetNamespace . ... ... ty: Union({xs:anyURI}, {xs:string: len=0},
                                        {xs:string: enum=(*)})
. @targetName .. ... ... ... ty: Union({xs:NCName}, {xs:string: enum=(*)})
. targetAssertion* . ... ... ty: re:XPATH
. . @expr? . ... ... ... ... ty: re:XPATH
. import*
. . @href .. ... ... ... ... ty: xs:anyURI
. namespace*
. . @iri ... ... ... ... ... ty: xs:anyURI
. . @prefix  ... ... ... ... ty: xs:NCName
. context?
. . _choice_*
. . 1 var .. ... ... ... ... ty: re:XPATH
. . 1 . @name .. ... ... ... ty: xs:NCName
. . 1 . @value?  ... ... ... ty: re:XPATH
. . 2 fun .. ... ... ... ... ty: re:XPATH
. . 2 . @name .. ... ... ... ty: xs:NCName
. . 2 . @params? ... ... ... ty: xs:string: pattern=#(\i\c*(\s*,\s*\i\c*)?)?#
. . 2 . @as? ... ... ... ... ty: xs:Name
. . 2 . @code? . ... ... ... ty: re:XPATH
. resource*
. . @modelID ... ... ... ... ty: xs:NCName
. . @assertedTargetNodes? .. ty: re:XPATH
. . @iri? .. ... ... ... ... ty: re:XPATH
. . @type? . ... ... ... ... ty: List(xs:Name)
. . @targetNodeNamespace? .. ty: Union({xs:anyURI}, {xs:string: len=0},
                                        {xs:string: enum=(*)})
```

```
. . @targetNodeName? ... ... ty: Union({xs:NCName}, {xs:string: enum=(*)})
. . targetNodeAssertion* ... ty: re:XPATH
. . . @expr? ... ... ... ... ty: re:XPATH
. . property*
. . . @iri . ... ... ... ... ty: xs:anyURI
. . . @value ... ... ... ... ty: re:XPATH
. . . @type? ... ... ... ... ty: Union({xs:Name},
                                      {xs:string: enum=(#iri|#resource)})
. . . @list? ... ... ... ... ty: xs:boolean
. . . @objectModelID? .. ... ty: xs:Name
. . . @card? ... ... ... ... ty: xs:string: pattern=#[?*+]|\d+(-(\d+)?)?|-\d+#
. . . @reverse?  ... ... ... ty: xs:boolean
. . . @lang? ... ... ... ... ty: re:XPATH
. . . valueItemCase*
. . . . @test .. ... ... ... ty: re:XPATH
. . . . @iri? .. ... ... ... ty: xs:anyURI
. . . . @value?  ... ... ... ty: re:XPATH
. . . . @type? . ... ... ... ty: Union({xs:Name},
                                      {xs:string: enum=(#iri|#resource)})
. . . . @list? . ... ... ... ty: xs:boolean
. . . . @objectModelID?  ... ty: xs:Name
. . . . @lang? . ... ... ... ty: re:XPATH
```

RDFe model components

This section summarizes the main components of an RDFe based mapping model. Details of the XML representation can be looked up in the treesheet representation shown in the preceding section.

Semantic extension

A **semantic extension** is a set of one or more semantic maps, together defining a mapping of XML documents to a set of RDF triples. A semantic extension comprises all semantic maps explicitly provided as input for an instance of RDFe processing, as well as all maps directly or indirectly imported by these (see below).

Semantic map

A **semantic map** is a specification how to map a class of XML documents (defined in terms of target document constraints) to a set of RDF triples. It is represented by a `<semanticMap>` element and comprises the components summarized below.

Table 3. Semantic map components and their XML representation.

Model component		**XML representation**
Semantic map IRI		@iri
Target document constraint		
	Target document namespace	@targetNamespace
	Target document local name	@targetName
	Target assertions	<targetAssertion>
Semantic map imports		<import>
RDF namespace bindings		<namespace>
Context constructor		<context>
Resource models		<resource>

A *semantic map IRI* identifies a semantic map unambiguously. The map IRI should be independent of the document URI.

The *target document constraint* is a set of conditions met by any XML document to which the semantic map may be applied. The constraint enables a decision whether resource models from the semantic map can be used in order to map nodes from a given XML document to RDF resource descriptions. A target document assertion is an XPath expression, to be evaluated in the context of a document root. A typical use of target document assertions is a check of the API or schema version indicated by an attribute of the input document.

A semantic map may *import* other semantic maps. Import is transitive, so that any map reachable through a chain of imports is treated as imported. Imported maps are added to the semantic extension, and no distinction is made between imported maps and those which have been explicitly supplied as input.

RDF namespace bindings define prefixes used in the output for representing IRI values in compact form. Note that they are *not* used for resolving namespace prefixes used in XML names and XPath expressions. During evaluation, XML prefixes are always resolved according to the in-scope namespace bindings established by namespace declarations (`xmlns`).

Context constructor and *resource models* are described in subsequent sections.

Resource model

A **resource model** is a set of rules how to construct triples describing a resource which is viewed as represented by a given XML node. A resource model is represented by a `<resource>` element and comprises the components summarized below.

Table 4. Resource model components and their XML representation.

Model component		**XML representation**
Resource model ID		@modelID
Resource IRI expression		@iri
Target node assertion		@assertedTargetNodes
Target node constraint		
	Target node namespace	@targetNodeNamespace
	Target node local name	@targetNodeName
	Target node assertions	<targetNodeAssertion>
Resource type IRIs		@type
Property models		<property>

The *resource model ID* is used for purposes of cross reference. A resource model has an implicit resource model IRI obtained by appending the resource model ID to the semantic map IRI (with a hash character (“#”) inserted in between if the semantic map IRI does not end with “/” or “#”).

The *resource IRI expression* yields the IRI of the resource. The expression is evaluated using as context item the XML node used as target of the resource model.

A *target node assertion* is an expression to be evaluated in the context of each input document passed to an instance of RDFe processing. The expression yields a sequence of nodes which MUST be mapped to RDF descriptions. Note that the processing result is not limited to these resource descriptions, as further descriptions may be triggered as explained in the section called “Linking resources”.

A *target node constraint* is a set of conditions which is evaluated when selecting the resource model which is appropriate for a given XML node. It is used in particular when a property model treats XML

nodes returned by a value expression as representations of an RDF description (for details see the section called “Linking resources”).

Resource type IRIs identify the RDF types of the resource (`rdf:type` property values). The types are specified as literal IRI values.

Property models are explained in the following section.

Property model

A **property model** is represented by a `<property>` child element of a `<resource>` element. The following table summarizes the major model components.

Table 5. Property model components and their XML representation.

Model component	XML representation
Property IRI	@iri
Object value expression	@value
Object type (IRI or token)	@type
Object language tag	@lang
Object resource model (IRI or ID)	@objectModelID
RDF list flag	@list
Reverse property flag	@reverse
Conditional settings	<valueItemCase>

The *property IRI* defines the IRI of the property. It is specified as a literal value.

The *object value expression* yields XDM items [11] which are mapped to RDF terms in accordance with the settings of the property model, e.g. the object type. For each term a triple is constructed, using the term as object, a subject IRI obtained from the IRI expression of the containing resource model, and a property IRI as specified.

The *object type* controls the mapping of the XDM items obtained from the object value expression to RDF terms used as triple objects. The object type can be an XSD data type, the token `#iri` denoting a resource IRI, or the token `#resource`. The latter token signals that the triple object is the subject IRI used by the resource description obtained for the value item, which must be a node. The resource description is the result of applying to the value node an appropriate resource model, which is either explicitly specified (@objectModelID) or determined by matching the node against the target node constraints of the available resource models.

The *language tag* is used to turn the object value into a language-tagged string.

The *object resource model* is evaluated in conjunction with object type `#resource`. It identifies a resource model to be used when mapping value nodes yielded by the object value expression to resource descriptions.

The *RDF list flag* indicates whether or not the RDF terms obtained from the object value expression are arranged as an RDF list (default: no).

The *reverse flag* can indicate that the items obtained from the object value expression represent the subjects, rather than objects, of the triples to be constructed, in which case the target node of the containing resource model becomes the triple object.

Conditional settings is a container for settings (e.g. property IRI or object type IRI) applied only to those value items which meet a condition. The condition is expressed by an XPath expression which references the value item as an additional context variable (`rdfe:value`).

Context constructor

Using RDFe, the construction of RDF triples is based on the evaluation of XPath expressions. Evaluation can be supported by an **evaluation context** consisting of variables and functions accessible within the expression. The context is obtained from a *context constructor* represented by a `<context>` element. A distinct instance of the context is constructed for each XML document containing a node which is used as context node by an expression from the semantic map defining the context. The context constructor is a collection of variable and function constructors. Variable constructors associate a name with an XQuery expression providing the value. Function constructors associate a name with an XQuery function defined in terms of parameter names, return value type and an expression providing the function value. As the expressions used by the variable and function constructors are evaluated in the context of the root element of the document in question, variable values as well as function behaviour may reflect the contents of the document. Variable values may have any type defined by the XDM data model, version 3.1 [11] (sequences of items which may be atom, node, map, array or function). Context functions are called within expressions like normal functions, yet provide behaviour defined by the semantic map and possibly dependent on document contents.

Evaluation

Semantic maps are evaluated by an **RDFe processor**. This section describes the processing in an informal way. See also Appendix A, *Processing semantic maps - formal definition*.

Input / Ouput

Processing input is

- An initial set of XML documents
- A set of semantic map documents

Processing output is a set of RDF triples, usually designed to express semantic content of the XML documents.

The set of **contributing semantic maps** consists of the set explicitly supplied, as well as all semantic maps directly or indirectly imported by them.

The set of **contributing XML documents** is not limited to the *initial* input documents, as expressions used to construct triples my access other documents by derefencing URIs found in documents or semantic maps. This is an example of navigation into a document which may not have been part of the initial set of input documents:

```
<re:property iri="ont:country" type="#resource"
             value="country/@href/doc(.)//country"/>
```

RDFe thus supports a linked data view.

Hybrid triples and preliminary resource description

Understanding the processing of semantic maps is facilitated by the auxiliary concepts of a “hybrid triple” and a “preliminary resource description”. When a property model uses the type specification `#resource`, the nodes obtained from the object value expression of the property model are viewed as *XML nodes representing resources*, and the triple objects are the *IRIs of these resources*. The resource is identified by the combined identities of XML node and resource model to be used in order to map the node to a resource description. When this resource has already been described in an earlier phase of the evaluation, the IRI is available and the triple can be constructed. If the resource description has not yet been created, the IRI is still unknown and the triple cannot yet be constructed. In this situation, a **hybrid triple** is constructed, using the pair of XML node and resource model ID as object. A hybrid

triple is a preliminary representation of the triple eventually to be constructed. A resource description is called **preliminary** or **final**, dependent on whether or not it contains hybrid triples. A preliminary description is turned into a final description by creating for each hybrid triple a resource description and replacing the hybrid triple object by the subject IRI used by that description. The resource description created for the hybrid triple object may itself contain hybrid triples, but in any case it provides the IRI required to finalize the hybrid triple currently processed. If the new resource description is preliminary, it will be finalized in the same way, by creating for each hybrid triple yet another resource description which also provides the required IRI. In general, the finalization of preliminary resource descriptions is a recursive processing which ends when any new resource descriptions are final.

Asserted target nodes

The scope of processing is controlled by the **asserted resource descriptions**, the set of resource descriptions which MUST be constructed, given a set of semantic maps and an *initial* set of XML documents. Such a description is identified by an XML node representing the resource and a resource model ID identifying the model to be used for mapping the node to an RDF description. (Note that for a single XML node more than one mapping may be defined, that is, more than one resource model may accept the same XML node as a target.) The **asserted target nodes** of a resource model are the XML nodes to which the resource model must be applied in order to create all asserted resource descriptions involving this resource model.

Any **additional resource descriptions** are only constructed if they are required in order to construct an asserted resource description. An additional resource description is required if without this description another description (asserted or itself additional) would be preliminary, that is, contain hybrid triples. As the discovery of required resource descriptions may entail the discovery of further required resource descriptions, the discovery process is recursive, as explained in the section called “Hybrid triples and preliminary resource description”.

The asserted target nodes of a resource model are determined by the **target node assertion** of the resource model, an expression evaluated in the context of each *initial* XML document. Note that the target node assertion is not applied to XML documents which do not belong to the initial set of XML documents. Such additional documents contribute only additional resource descriptions, no asserted resource descriptions. Initial documents, on the other hand, may contribute asserted and/or additional descriptions.

Processing steps

The processing of semantic maps can now be described as a sequence of steps:

1. For each resource model identify its asserted target nodes.
2. For each asserted target node create a resource description (preliminary or final).
3. a. Map any hybrid triple object to a new resource description
   b. Replace the hybrid triple object by the IRI provided by the new resource description
4. If any resource descriptions created in (3) contain hybrid triples, repeat (3)
5. The result is the set of all RDF triples created in steps (2) and (3).

For a formal definition of the processing see Appendix A, *Processing semantic maps - formal definition*.

RDFe for non-XML resources

The core capability of the XPath language is the navigation of XDM node trees, and this navigation is the “engine” of RDFe. The W3C recommendations defining XPath 3.1 ([9] and [10]) do not define functions parsing HTML and CSV, and the function defined to parse JSON into node trees (`fn:json-to-xml`) uses a generic vocabulary which makes navigation awkward. Implementation-defined XPath extension functions, on the other hand, which parse JSON, HTML and CSV into navigation-friendly node trees are common (e.g. BaseX [1] functions `json:parse`, `html:parse` and

`csv:parse`). An RDFe processor may offer **implementation-defined support** for such functions and, by implication, also enable the mapping of non-XML resources to RDF triples.

Conformance

An **RDFe processor** translates an initial set of XML documents and a set of semantic maps to a set of RDF triples.

Minimal conformance

Minimal conformance requires a processing as described in this paper. It includes support for **XPath 3.1 expressions** in any place of a semantic map where an XPath expression is expected:

- targetAssertion/@expr
- targetNodeAssertion/@expr
- var/@value
- fun/@code
- resource/@iri
- resource/@assertedTargetNodes
- property/@value
- property/@lang
- valueItemCase/@test
- valueItemCase/@value
- valueItemCase/@lang

Optional feature: XQuery Expressions Feature

If an implementation provides the **XQuery Expressions Feature**, it must support XQuery 3.1 [12] expressions in any place of a semantic map where an XPath expression is expected.

Implementation-defined extension functions

An implementation may support implementation-defined XPath extension functions. These may in particular enable the parsing of non-XML resources into XDM node trees and thus support the RDFe-defined mapping of non-XML resources to RDF triples.

Implementation

An implementation of an RDFe processor is available on github [8] (`https://github.com/hrennau/shax`). The processor is provided as a command line tool (`shax.bat`, `shax.sh`). Example call:

```
shax rdfe?dox=drug*.xml,semap=drugbank.*rdfe.xml
```

The implementation is written in XQuery and requires the use of the BaseX [1] XQuery processor. It supports the XQuery Expressions Feature and all XPath extension functions defined by BaseX. This includes functions for parsing JSON, HTML and CSV into node trees (`json:parse`, `html:parse`, `csv:parse`). The implementation can therefore be used for mapping any mixture of XML, JSON, HTML and CSV resources to an RDF graph.

Discussion

The purpose of RDFe is straightforward: to support the mapping of XML data to RDF data. Why should one want to do this? In a “push scenario”, XML data are the primary reality, and RDF is a

means to augment it by an additional representation. In a “pull scenario”, an RDF model comes first, and XML is a data source used for populating the model. Either way, the common denominator is information content which may be represented in alternative ways, as a tree or as a graph. The potential usefulness of RDFe (and other tools for mapping between tree and graph, like RDFa [6], JSON-LD [4] and GraphQL [3]) depends on the possible benefits of switching between the two models. Such benefits emerge from the complementary character of these alternatives.

A tree representation offers an optimal reduction of complexity, paying the price of a certain arbitrariness. The reduction of complexity is far more obvious than the arbitrariness. Tree structure decouples amount and complexity of information. A restaurant menu, for example, is a tree, with inner nodes like starters, main courses, desserts and beverages, perhaps further inner nodes (meat, fish, vegetarian, etc.) and leaf nodes which are priced offerings. Such representation fits the intended usage so well that it looks natural. But when integrating the menu data from all restaurants in a town - how to arrange intermediate nodes like location, the type of restaurant, price category, ratings, …? It may also make sense to pull the menu items out of the menus, grouping by name of the dish.

A graph representation avoids arbitrariness by reducing information to an essence consisting of resources, properties and relationships – yet pays the price of a certain unwieldiness. Graph data are more difficult to understand and to use. If switching between tree and graph were an effortless operation, what could be gained by “seeing” in a tree the graph which it represents, and by “seeing” in a graph the trees which it can become?

Figure 1. La clairvoyance, Rene Magritte, 1936

A painting suggesting a thorough consideration of the relationship between XML and RDF.

Think of two XML documents, one representing `<painter>` as child element of `<painting>`, the other representing `<painting>` as child element of `<painter>`. From a tree-only perspective they are stating different facts; from a graph-in-tree perspective, they are representing the *same information*, which is about painters, paintings and a relationship between the two. Such intuitive insight may be *inferred* by a machine if machine-readable instructions for translating both documents into RDF are available. Interesting opportunities for data integration and quality control seem to emerge. A document-to-document transformation, for example, may be checked for semantic consistency.

If the potential of using tree and graph quasi-simultaneously has hardly been explored, so far, a major reason *may* be the high “resistence” which hinders a flow of information between the two models. RDFe addresses one half of this problem, the direction tree-to-graph. RDFe is meant to complement approaches dealing with the other half, e.g. GraphQL [3].

RDFe is committed to XPath as *the* language for expressing mappings within a forest of information. The conclusion that RDFe is restricted to dealing with XML data would be a misunderstanding, due to oversight that any tree structure (e.g. JSON and any table format) can be parsed into an XDM node tree and thus become accessible to XPath navigation. Another error would be to think that RDFe is restricted to connecting information *within* documents, as XPath offers excellent support for inter-document navigation (see also the example given in the section called “Linking resources”). Contrary to widespread views, XPath may be understood and used as a universal language for tree navigation - and RDFe might accordingly serve as a general language for mapping information forest to RDF graph.

A. Processing semantic maps - formal definition

The processing of semantic maps is based on the building block of an **RDFe expression (rdfee)**. An rdfee is a pair consisting of an **XML node** and a **resource model**:

```
rdfee ::= (xnode, rmodel)
```

The XML node is viewed as representing a resource, and the resource model defines how to translate the XML node into an RDF resource description. An rdfee is an expression which can be resolved to a set of tripels.

Resource models are contained by a **semantic map**. A set of semantic maps is called a **semantic extension (SE)**. A semantic extension is a function which maps a set of XML documents to a (possibly empty) set of RDF triples:

```
triple* = SE(document+)
```

The mapping is defined by the following rules, expressed in pseudo-code.

Section 1: Top-level rule

```
triples(docs, semaps) ::=
    for rdfee in rdfees(docs, semaps):
        rdfee-triples(rdfee, semaps)
```

Section 2: Resolving an rdfee to a set of triples

```
rdfee-triples(rdfee, semaps) ::=
    for pmodel in pmodels(rdfee.rmodel),
    for value in values(pmodel, rdfee.xnode):
    (
        resource-iri(rdfee.rmodel, rdfee.xnode),
        property-iri(pmodel, rdfee.xnode),
        triple-object(value, pmodel, semaps)
    )

values(pmodel, xnode) ::=
    xpath(pmodel/@value, xnode, containing-semap(pmodel))

resource-iri(rmodel, xnode) ::=
    xpath(rmodel/@iri, xnode, containing-semap(rmodel))

property-iri(pmodel, xnode) ::=
    xpath(pmodel/@iri, xnode, containing-semap(pmodel))

triple-object(value, pmodel, semaps) ::=
    if object-type(value, pmodel) = "#resource":
        resource-iri(rmodel-for-xnode(value, pmodel), value)
    else:
        rdf-value(value, object-type(value, pmodel))

rmodel-for-xnode(xnode, pmodel, semaps) ::=
    if pmodel/@objectModelID:
        rmodel(pmodel/@objectModelID, semaps)
    else:
        best-matching-rmodel-for-xnode(xnode, semaps)

best-matching-rmodel-for-xnode(xnode, semaps):
[Returns the rmodel which is matched by xnode and, if several rmodels
are matched, is deemed the best match; rules for “best match” may evolve;
current implementation treats the number of target node constraints as a
measure of priority – the better match is the rmodel with a greater number
of constraints; an explicit @priority à la XSLT is considered a future
option.]
```

```
object-type(value, pmodel):
[Returns the type to be used for a value obtained from the value expression;
value provided by pmodel/@type or by pmodel/valueItemCase/@type.]

rdf-value(value, type):
[Returns a literal with lexical form = string(value), datatype = type.]
```

Section 3: Resolving input documents to a set of rdfees

```
rdfees(docs, semaps) ::=
 for rdfee in asserted-rdfees(docs, semaps):
        rdfee,
        required-rdfees(rdfee, semaps)

Sub section: asserted rdfees

asserted-rdfees(docs, semaps) ::=
    for doc in docs,
    for semap in semaps:
        if doc-matches-semap(doc, semap):
            for rmodel in rmodels(semap),
            for xnode in asserted-target-nodes(rmodel, doc):
                (xnode, rmodel)

asserted-target-nodes(rmodel, doc) ::=
    xpath(rmodel/@assertedTargetNodes, doc, containing-semap(rmodel))

Sub section: required rdfees

required-rdfees(rdfee, semaps) ::=
    for pmodel in pmodels(rdfee.rmodel),
    for value in values(pmodel, rdfee.xnode):
        required-rdfee(value, pmodel, semaps)

required-rdfee(xnode, pmodel, semaps) ::=
    if object-type(xnode, pmodel) = "#resource":
        let rmodel ::= rmodel-for-xnode(value, pmodel, semaps),
        let required-rdfee ::= (xnode, rmodel):
            required-rdfee,
            required-rdfees(required-rdfee, semaps )
```

Section 4: auxilliary rules

```
doc-matches-semap(doc, semap):
[Returns true if doc matches the target document constraints of semap.]

xnode-matches-rmodel(xnode, rmodel):
[Returns true if xnode matches the target node constraints of rmodel.]

rmodel(rmodelID, semaps) ::=
[Returns the rmodel with an ID matching rmodelID.]

rmodels(semap) ::= semap//resource

pmodels(rmodel) ::= rmodel/property
```

```
containing-doc(xnode) ::= xnode/root()

containing-semap(semapNode) ::= semapNode/ancestor-or-self::semanticMap

xpath(xpath-expression, contextNode, semap) ::=
[Value of xpath-expression, evaluated as XPath expression using contextNode
as context node and a dynamic context including all in-scope variables from
the dynamic context constructed for the combination of the document
containing contextNode and semap.]
```

Bibliography

[1] *BaseX*. 2019. BaseX GmbH. http:// basex.org.

[2] *DrugBank 5.0: a major update to the DrugBank database for 2018.*. 2017. DS Wishart, YD Feunang, AC Guo, EJ Lo, A Marcu, JR Grant, T Sajed, D Johnson, C Li, Z Sayeeda, N Assempour, I Iynkkaran, Y Liu, A Maciejewski, N Gale, A Wilson, L Chin, R Cummings, D Le, A Pon, C Knox, and M Wilson. Nucleic Acids Res. 2017 Nov 8.. https://www.drugbank.ca/. 10.1093/nar/gkx1037.

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

[4] *JSON-LD 1.0. A JSON-based Serialization for Linked Data*. 2014. World Wide Web Consortium (W3C). https://www.w3.org/TR/json-ld/.

[5] *Location trees enable XSD based tool development.*. Hans-Juergen Rennau. 2017. http://xmllondon.com/2017/xmllondon-2017-proceedings.pdf.

[6] *RDFa Core 1.1 – Third Edition.*. 2015. World Wide Web Consortium (W3C). https://www.w3.org/TR/rdfa-core/.

[7] *RDF 1.1 Turtle*. 2014. World Wide Web Consortium (W3C). https://www.w3.org/TR/turtle/.

[8] *A SHAX processor, transforming SHAX models into SHACL, XSD and JSON Schema.*. Hans-Juergen Rennau. 2017. https://github.com/hrennau/shax.

[9] *XML Path Language (XPath) 3.1*. 2017. World Wide Web Consortium (W3C). https://www.w3.org/TR/xpath-31/.

[10] *XPath and XQuery Functions and Operators 3.1*. 2017. World Wide Web Consortium (W3C). https://www.w3.org/TR/xpath-functions-31/.

[11] *XQuery and XPath Data Model 3.1*. 2017. World Wide Web Consortium (W3C). https://www.w3.org/TR/xpath-datamodel-31/.

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

[13] *xsdplus - a toolkit for XSD based tool development*. Hans-Juergen Rennau. 2017. https://github.com/hrennau/xsdplus.