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	.//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 trees**
- 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)

Information model

This section describes the information model underlying the operations of greenfox.

Part 1: resource model

A **file system tree** is a tree whose nodes are file system resources – folders and files.

A **file system resource** has an identity, resource properties, derived resource properties and resource values.

The **resource identity** of a file system resource can be expressed by a combination of file system identity and a file path locating the resource within the file system.

A **resource property** has a name and a value which can be represented by an XDM value.

A **derived resource property** is a property of a resource property value, or of a derived resource property value, which can be represented by an XDM value.

A **resource value** is the XDM value of an expression evaluated in the context of a resource property or derived resource property.

Folder resources

A folder has the following **resource properties**.

Table 2. Resource properties of a folder resource, as currently evaluated by greenfox. More properties may be added, e.g. representing access rights.

Property name	Value type	Description
[name]	`xsd:string`	The folder name; optional – the file system root folder does not have a name
[parent]	Folder resource	The XDM representation of resource identity is its file path
[children]	Folder and file resources	The XDM representation of resource identity is its file path
[last-modified]	`xsd:dateTime`	May be out of sync when comparing values of resources from different machines

A folder has the following **derived resource properties**.

Table 3. Derived resource properties of a folder resource, as currently evaluated by greenfox.

Property name	Value type	Description
[filepath]	`xsd:string`	The names of all ancestor folders and the folder itself, separated by a slash
[foxpath-value]	Mapping: foxpath expression string => XDM value	A mapping of foxpath expressions to an XDM value, which is the value obtained by evaluating the expression in the context of the resource folder's [filepath] value

Resource values are obtained by applying [foxpath-value] to the text of a foxpath expression.

File resources

A file has the following **resource properties**.

Table 4. Resource properties of a file resource, as currently evaluated by greenfox. More properties may be added, e.g. representing access rights.

Property name	Value type	Description
[name]	`xsd:string`	Mandatory – a file must have a name
[parent]	Folder resource	The XDM representation of resource identity is its file path
[last-modified]	`xsd:dateTime`	May be out of sync when comparing values of resources from different machines
[size]	`xsd:integer`	File size, in bytes
[sha1]	`xsd:string`	SHA-1 hash value of file contents
[sha256]	`xsd:string`	SHA-256 hash value of file contents
[md5]	`xsd:string`	MD5 hash value of file contents
[text]	`xsd:string`	The text content of the file (empty if not a text file)
[encoding]	`xsd:anyURI`	The encoding of the text content of the file (empty if not a text file)
[octets]	`xsd:base64Binary`	The binary file content
[xmldoc]	`document-node()`	The result of parsing [text] into an XML document
[jsondoc-basex]	`document-node()`	The result of parsing [text] into a JSON document represented by a document-node in accordance with the rules defined by BaseX documentation
[jsondoc-w3c]	`document-node()`	The result of parsing [text] into a JSON document represented by a document-node in accordance with XPath function `fn:json-to-xml`
[htmldoc]	`document-node()`	The result of parsing [text] into an XML document represented by a document-node in accordance with the rules defined by TagSoup documentation
[csvdoc]	`document-node()`	The result of parsing [text] into an XML document represented by a document-node, as controlled by the CSV parsing parameter values derived from a file shape, in accordance with the rules defined by BaseX documentation

Property name	Value type	Description
[csvdocs]	Mapping: csv-parse-parameters => document-node()	The mapping result is a CSV document represented by a document-node as controlled by given CSV parsing parameter values, in accordance with the rules defined by BaseX documentation

A file has the following **derived resource properties**.

Table 5. Derived resource properties of a file resource, as currently evaluated by greenfox.

Property name	Value type	Description
[filepath]	`xsd:string`	The names of all ancestor folders and the folder itself, separated by a slash
[xmldoc.xpath]	Mapping: XPath expression string => XDM value	A mapping of XPath expressions to an XDM value, which is the value obtained by evaluating the expression in the context of [xmldoc]
[jsondoc-basex.xpath]	Mapping: XPath expression string => XDM value	A mapping of XPath expressions to an XDM value, which is the value obtained by evaluating the expression in the context of [jsondoc-basex]
[jsondoc-w3c.xpath]	Mapping: XPath expression string => XDM value	A mapping of XPath expressions to an XDM value, which is the value obtained by evaluating the expression in the context of [jsondoc-w3c]
[htmldoc.xpath]	Mapping: XPath expression string => XDM value	A mapping of XPath expressions to an XDM value, which is the value obtained by evaluating the expression in the context of [htmldoc]
[csvdoc.xpath]	Mapping: (csv-parse-parameters, XPath expression string) => XDM value	A mapping of CSV parsing parameter values and an XPath expression to an XDM value, which is the value obtained by evaluating the expression in the context of a document node from [csv-docs], obtained for the parsing parameter values
[foxpath-value]	Mapping: foxpath expression string => XDM value	A mapping of foxpath expressions to an XDM value, which is the value obtained by evaluating the expression in the context of [filepath]

Resource values are obtained by applying [foxpath-value] to the text of a foxpath expression, or one of the mappings [*-xpath] to the text of an XPath expression.

For information about CSV parsing parameters, see [x], section `#wiki/CSV_Module`.

Part 2: schema model

File system validation is a mapping of a file system tree and a greenfox schema to a set of greenfox validation results.

A **greenfox schema** is a set of shapes.

A **shape** is a resource shape or a value shape.

A **resource shape** is a set of constraints applicable to a file system resource. It has an optional target declaration.

A **target declaration** specifies the selection of a target.

A **target** is a set of focus resources.

A **focus resource** is a resource to be validated against a resource shape.

A resource shape is a **folder shape** or a **file shape**.

A **value shape** is a mapping of a focus resource to a resource value and a set of constraints.

A **constraint** maps a resource property or a resource value to a validation result.

A constraint is declared by a shape. It identifies a constraint component and assigns values to the constraint parameters.

A **constraint component** is a set of constraint parameter definitions and a validator.

A **constraint parameter** is defined by the specification of a constraint component. The definition includes a name, a type, the value semantics and whether the parameter is mandatory or optional.

A **validator** is a set of rules how a resource property or a resource value and the arguments bound to the constraint parameters are mapped to a validation result.

A **validation result** is a boolean value signaling conformance of a resource property or a resource value against a constraint, accompanied by additional information identifying the resource and the constraint, as well as further items of information with names and semantics as prescribed by the constraint component.

Part 3: validation model

File system validation is a mapping of a file system tree and a greenfox schema to a set of validation results, as defined in the following paragraphs.

Validation of a file system tree against a greenfox schema: Given a file system tree and a greenfox schema, the validation results are the union of results of the validation of the file system tree against all shapes in the greenfox schema.

Validation of a file system tree against a shape: Given a file system tree and a shape in the greenfox schema, the validation results are the union of the results of the validation of all focus resources that are in the target of the shape.

Validation of a resource against a shape: Given a focus resource in the file system tree and a shape in the greenfox schema, the validation results are the union of the results of the validation of the focus resource against all constraints declared by the shape, unless the shape has been deactivated, in which case the validation results are empty.

Validation of a focus resource against a constraint: Given a focus resource in the file system tree and a constraint of kind C in the greenfox schema, the validation results are defined by the validator of the constraint component C. The validator typically takes as input a resource property or a resource value of the focus resource and the arguments supplied to the constraint parameters.

Schema building blocks

This section summarizes the **building blocks** of a greenfox schema. Building blocks are the parts of which a serialized schema is composed. The serialized schema should be distinguished from the logical schema, which is independent of a serialization and can be defined as a set of logical components (as defined by the information model) and parameter bindings.

Each building block is represented by XML elements with a particular name. Note that there need not be a one-to-one correspondence between building blocks and logical components as defined by the information model. An Import declaration, for example, is a building block without corresponding logical component. Constraints, on the other hand, are logical components which in many cases are not represented by a separate building block, but by attributes attached to a building block. Note also that the information model includes logical components built into the greenfox language and without representation in any given schema (e.g. validators).

Table 6. The building blocks of a greenfox schema.

Building block	Role	XML representation
Import declaration	Declares the import of another greenfox schema so that its contents are included in the current schema	`gx:import`
Context declaration	Declares external schema variables, the values of which can be supplied by the agent launching the validation. Each variable is represented by a `gx:field` child element.	`gx:context`
Shapes library	A collection of shapes without target declaration, which can be referenced by other shapes	`gx:shapesLib`
Constraints library	A collection of constraint declaration nodes, which can be referenced by shapes	`gx:constraintsLib`
Constraint components library	A collection of constraint component definitions, for which constraints can be declared	`gx:contraint-ComponentsLib/*`
Constraint component definition	A user-defined constraint component. It declares the constraint parameters and provides a validator. Parameters are represented by `gx:param` child elements, the validator by a `gx:xpath` or `gx:foxpath` child element	`gx:constraint-Component`
Domain	A container element wrapping the shapes used for validating a particular file system tree, identified by its root folder	`gx:domain`
Resource shapes	A shape applicable to a file system folder or file	`gx:folder` `gx:file`
Value shape	A shape applicable to a resource value	`gx:xpath` `gx:foxpath`
Base shape declaration	References a shape so that its contents are included in the shape containing the reference	`gx:baseShape`

Constraint declaration node	An element representing one or several constraints declared by a shape. Constraint parameters are represented by attributes and/or child elements	`gx:targetSize`
		`gx:folderContent`
		`gx:mediaType`
		`gx:fileSize`
		`gx:lastModified`
		`gx:name`
		`gx:hashCode`

Conditional node	A set of building blocks associated with a condition, so that the building blocks are only used if the condition is met	`gx:ifMediatype`

Schema language extension

This section describes **user-defined constraint components**. Such components are defined within a greenfox schema by a `gx:constraintComponent` element. They can be used like built-in

constraint components, observing syntax rules implied by the details of the component defining element.

The extension model is based on the simple conceptual framework organizing greenfox:

- File system validation can be decomposed into smallest units which are the validation of a single resource against a single constraint
- A constraint resembles a function call, where the “function” is a constraint component and the “function parameters” are the constraint parameters
- A constraint is declared by an XML element whose name, attributes and content identify the constraint component and supply the parameter values (possibly along with the details of other constraints)
- The location of the constraint element implies the value input:
  - If the element is a value shape element, or a child of a value shape element, the test value is the resource value produced by the expression of the value shape
  - If it is a child element of a resource shape element, the test value is a resource property

These rules imply a simple model how to define **new constraint components**. The definition of a new constraint component comprises:

- *Specifies the signature* of the constraint component – component name, parameter names and types
- *Specifies the semantics* of the constraint component – the rules how the test value plus the parameter values are mapped to
  - A boolean result (signaling pass or failure)
  - Details of the validation result
- *Specifies the XML representation* of a constraint – element and attribute names and their mapping to constraint parameters
- *Provides an implementation* of the semantics

The XML representaton of a constraint is one of the following:

- a set of attributes which must be attached to a value shape
- an element which must be a child element of a specified set of shape elements (e.g. must be a child of a `<file>` element)

As an illustrative example, consider the creation of a new constraint component characterized as follows.

Constraint component IRI: `ex:grep`

Constraint parameters:

Name	Type	Meaning	Mandatory?	Default value
regex	`xsd:string`	A regular expression	+	-
flags	`xsd:string`	Evaluation flags	-	Zero-length string

XML syntax:

- Syntax style: constraint element
- element name: `ex:grep`
- parameter mapping:
  - parameter $regex - attribute @regex
  - parameter $flags – attribute @flags
- child elements: -

Semantics:

"A constraint is satisfied if the focus resource is a text file containing a line matching regular expression `$regex`, as controlled by the regex evaluation `flags` given by $flags (e.g. case-insensitively)"

The **implementation** may be provided by the following element, which must be a child element of `gx:constraintComponentsLib`:

```
<constraintComponent constraintElementName="ex:grep">
    <param name="pattern"></param>
    <param name="flags"></param>
    <xpath><![CDATA[
exists($this ! unparsed-text-lines(.)[matches(., $pattern, $flags)])
    ]]></xpath>
</constraintComponent>
```

Note in the expression text the use of a variable $this. It is always bound to the file path of the focus resource.

Validation results

This section describes the results produced by a greenfox validation.

Validation reports and representations

The primary result of a greenfox validation is an RDF graph called the **white validation report**. This is mapped to the **red validation report**, an RDF graph obtained by removing from a white report all triples not related to constraint violations. For red and white validation reports a **canonical XML representation** is defined. Apart from that there are **derived representations**, implementation-dependent reports which may use any data model and mediatype.

The **white validation report** is an RDF graph with exactly one instance of `gx:ValidationReport`. The instance has the following properties:

- `gx:conforms`, with an `xsd:boolean` value indicating conformance
- `gx:result`, with one value ...
  - for each constraint violation (“red and yellow values”)
  - for each constraint check which did not produce a violation (“green values”)
  - o for each observation, which is a result triggered by a value shape in order to record a resource value not related to constraint checking (“blue values”)

The **red validation report** is an RDF graph obtained by removing from the white validation report all green and blue result values. Note that the validation report defined by the SHACL language [x] corresponds to the red validation report defined by greenfox.

The **canonical XML representation** of a white or red validation report is an XML document with a `<gx:validationReport>` root element, which has for each `gx:result` value from the RDF graph one child element, which is a `<gx:red>`, `<gx:yellow>`, `<gx:green>` or `<gx:blue>` element, according to the `gx:result/gx:severity` property value being `gx:Violation`, `gx:Warning`, `gx:Info` or `gx:Observation`).

A **derived representation** is any kind of data structure, using any mediatype, representing information content from the white or red validation report in an implementation-defined way.

Validation result

A **validation result** is a unit of information which describes the outcome of validating a focus resource against a constraint: either constraint violation (“red” or “yellow” result), or conformance (“green” result).

A validation result is an RDF resource with several properties as described below. Key features of the result model:

- Every result is related to an individual file system resource (file or folder)

- Every result is related to an individual constraint (and, by implication, a shape)

This allows for meaningfull aggregation by resource, by constraint and by shape and, by implication, any combination of aggregated resources, constraints and shapes. Such aggregation may, for example, be useful for integrating validation results into a graphical representation of the file system and for analysis of impact.

A detailed description of the validation result model – RDF properties, SHACL equivalent and XML representation – is found in appendix A3.

Implementation

An implementation of a greenfox processor is available on github [x]. The processor is provided as a command line tool (`greenfox.bat`, `greenfox.sh`). Example call:

```
greenfox "val?gfox=/projects/greenfox/examples/gfox-system-s.xml,
      domain=/projects/greenfox/examples/system-s"
```

The implementation is written in XQuery and requires the use of the BaseX [x] XQuery processor.

Discussion

To be added.

A. Appendix A1: greenfox schema for system S

bla

B. Appendix A2: Alignment of key concepts between greenfox and SHACL

bla

C. Appendix A3: Validation result model

bla

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.