Greenfox – a schema language for validating file systems

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.

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 causing 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 [5], 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) [8] and a *unified navigation model* (foxpath/XPath) [2] [3] [4] [6] 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.

File or folder	**Name or pattern**	**Expectation**
folder	codelists	Contains one or more codelist files
folder	codelists/*	A codelist file; name not constrained; must be an XML document containing `<codelist>` elements with a @name attribute and `<entry>` children
folder	xsd	Contains one or more XSDs describing services messages
file	xsd/*	An XSD schema file; name not constrained
folder	test-*	A test case folder, containing `input`, `output` and `config` folders; apart from these only optional `log-*` files are allowed
folder	config	Test case config folder, containing file `msg.config.csv`
file	msg-config.csv	A CSV file with three columns: request file name, response file name, expected return code
folder	input	Test case input folder, containg request messages
file	input/*	A file representing a request message; name extension `.xml` or `.json`; mediatype corresponding to name extension
folder	output	A test case output folder, containing files representing response messages
file	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 with name `fooValue` must be found in the codelist with name `foo-article` (applies to XML and JSON responses alike)
- 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, which has a file path as specified by the @path attribute.

A <folder> element represents the set of folders matching the foxpath expression given by its @foxpath attribute, which is its *target declaration*. 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 preceded 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, which is …\system-s. Similarly, a <file> element represents the set of files selected by the foxpath expression given by its @foxpath attribute and resolved 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** which apply to a **target**, which is a (possibly empty) set of folders. When a <folder> has a @foxpath attribute, the target is the set of folders selected by the expression. The constraints are declared by child elements of the shape element.

Likewise, a <file> element represents a **file shape**, defining a set of constraints which apply to a target which is a set of files. When a <file> has a @foxpath attribute, the target is the set of files selected by the expression. 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 (called 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 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 optional constraint parameters. A constraint can be thought of as a function which consumes constraint parameter values and a resource value – a 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 specified by @msg on the constraint element, along with a set of information items identifying the violating resource (@folder), the constraint (@constraintComp and @constraintID) and its parameter values (@minCount). Example result:

```
<gx:red folder="C:/tt/greenfox/resources/example-system/system-s/resources/xsd"
        shapeID="xsdFolderShape"
        childShapeID="xsdFileShape"
        childShapeTargetFoxpath="*.xsd"
        constraintComp="TargetSize"
        constraintID="TargetSize_2"
        minCount="1"
        value="0"
        msg="No XSDs found"/>
```

A key principle of greenfox is that every constraint belongs to a resource shape and is applied to each resource in the target of that shape, referred to as the **focus resource**. In the common case, the focus resource is in the target of the nearest ancestor resource shape (`<file>` or `<folder>`). The `<targetSize>` constraint is an exception of the rule where the focus resource is *not* from the target of the containing `<folder>` or `<file>` shape, but the context folder used when evaluating the target declaration of that shape; usually this is a folder from the target of the "grand parent folder element", selected by the XPath `$targetSize/parent::*/parent::folder`).

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

```
<!-- *** Testcase folder shape *** -->
<folder foxpath=".\\test-*[input][output][config]" id="testcaseFolderShape">
    <targetSize msg="No testcase folders found" minCount="1"/>
    <folderContent msg="Testcase contains member other than input, output, config, log-*."
                   closed="true">
      <memberFolders names="input, output, config"/>
      <memberFiles names="log-*" occ="*"/>
    </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 of the containing `<folder>` shape – in other words, for each testcase folder. The constraint disallows any additional members except for *optional* files with a name matching `log-*` (of which any number is allowed, note the @occ attribute). The `folderContent` constraint is an example for a constraint component defining *complex* constraint parameters: for example, values supplied to the `memberFolders` parameter (which can accept any number of values) have a `names` and an (optional) `occ` field.

We proceed with a file shape whose target is 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>
```

As explained above, the `<targetSize>` constraint checks the focus resources from the target of the grandparent `<folder>`, which here are the testcase folders of system S. For any testcase folder which does not contain a file `config/msg-config.csv`, a constraint violation will be reported.

We want to be more specific: 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 subtle 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">
    <targetSize msg="Config file missing" count="1"/>

    <!-- Check - configured return codes ok? -->
    <xpath msg="Config file contains unknown return code" expr="//returnCode">
      <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 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 kinds of value shapes – **XPath value shape** <xpath> 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 obtained by replacing in the request file name the last substring "RQ" with "RS". This means a constraint which does not depend on file contents (as in the previous paragraph), but on the contents of the file system "around" the focus resource – a constraint whose 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">
   <targetSize msg="Input folder without request msgs" minCount="1"/>

   <!-- Check - request with response ? -->
  <foxpath msg="Request without response"
           expr="..\..\output\*\file-name(.)"
           containsXpath="$fileName ! replace(., '(.*)RQ(.*)$', '$1RS$2')"/>
```

A foxpath value shape combines a foxpath expression (@foxpath) with a set of constraints (represented by attributes and child elements of <foxpath>). 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 (which must be a file) 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. Note that the navigation path 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 apply this approach in order to constrain the codelists folder to contain <codelist> elements with a @name attribute and at least one non-empty <entry> child:

```
<!-- *** Codelist folder -->
<folder foxpath=".\\resources\codelists" id="codelistFolderShape">
    <targetSize msg="No codelist folder found" count="1"/>
    <foxpath expr=".\*.xml/codelist[entry/@code/string()]/@name" 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 (.*.xml), continuing within their collected content (…/codelist[…]/@name), arriving at @name attributes on <codelist> elements.

Now we turn to the files representing response messages. 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">
     <targetSize msg="Output folder without request msgs" minCount="1"/>

     <!-- *** Check - response fresh? *** -->
     <lastModified msg="Stale output file" ge="${lastModified}"/>

     <!-- *** Check - response non-empty? *** -->
     <fileSize msg="Empty output file" gt="0"/>
     ...
</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" value="2019-12-07"/>
  </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 a <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 points to 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 EqFoxpath constraint, represented by the @eqFoxpath attribute:

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

As always with an XPath value shape, the XPath expression (@expr) selects the content items to be checked. The EqFoxpath 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> in the response contains a string found in the codelist.

Note that this value shape works propertly 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 ok? *** -->
    <foxpath msg="Return code not the configured value"
             expr="..\..\config\msg-config.csv\csv-doc(., ',', 'yes')
                   //record[response eq $fileName]/returnCode"
             eqXPath="//*:returnCode"/>
```

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 in particular 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 resource, **folders** and **files**
- Resources are validated against **resource shapes**
- There are two kind 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, shape 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 typical target declaration is 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 has been mapped by an expression

- A resource value is obtained by two kinds of expression – **XPath expression** and **foxpath 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
- There are two kinds of value shapes, **XPath value shapes** and **foxpath value shapes**
- The **foxpath expression context** is usually the file path of the focus resource
- The **XPath expression context** is usually the root of an XDM node tree representing resource content, if the resource is a file and a node tree representation is available; otherwise in the context of the file path
- **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 in across file system contents) 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 resources – folders and files.

A 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 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 a resource property value, a derived resource property value, or the value of another mapping of a resource to an XDM value.

Folder resources

Table 1. Resource properties of a folder resource, as currently evaluated by greenfox. More properties may be added, e.g. representing access rights or a SHA-1 value.

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 identifying the parent is its file path
[children]	Folder and File resources	The XDM representation identifying the children are their file paths
[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 2. Derived resource properties of a folder resource, as currently evaluated by greenfox.

Property name	Value type	Description
[filepath]	`xsd:string`	The names of all ancdestor 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

File resources

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

Table 3. Resource properties of a file resource, as currently evaluated by greenfox. More properties may be added, e.g. representing access rights or a SHA-1 value.

Property name	Value type	Description or remark
[name]	`xsd:string`	Optional – the file system root folder does not have a name
[parent]	Folder resource	The XDM representation identifying the parent is its file path
[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 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

[csvdocs]	Mapping:
	csv-parse-parameters =>
	`document-node()`
		The mapping result is a CSV document represented by a document-node as controlled by the parsing parameters, in accordance with the BaseX documentation,

[last-modified]	`xsd:dateTime`	May be out of sync when comparing values of resources from different machines
[size]	`xsd:integer`	File size, in bytes

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

Table 4. Derived resource properties of a file resource, as currently evaluated by greenfox. More properties may be added.

Property name	**Value type**	**Description**
[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-parse-parameters 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 csv-parse-parameters
[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]

Mapping [text] to CSV documents is controlled by csv-parse-params, see [1], “documentation”, “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 and an optional target declaration.

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

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

A **constraint** maps a resource value or a particular resource property to a boolean value.

A constraint identifies a constraint component and assigns values to the constraint component parameters.

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

A **parameter declaration** specifies the name and type of a mandatory or optional parameter used by a validator.

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

Part 3: validation model

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

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 each resource which is in the target of the shape of the schema.

Validation of a resource against a shape: Given a 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 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 resource in the file system tree and a constraint of kind C in the greenfox schema, the validation results are defined by the validators and observers of the constraint component C. These validators and observers typically take as input a resource value (e.g. [xmldoc]) of the resource and the arguments supplied to the constraint component parameters.

Schema building blocks

This section gives an overview of the basic schema building blocks:

- Import declarations
- Context declaration
- Resource shapes
- Value shapes
- Constraints
- Constraint components

[- to-be-added -]

Schema language extension

This section describes features enabling an extension of the schema language. Extension 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 is like a function call, where the "function" is the constraint component and the "function parameters" are the constraint parameters
- A constraint is declared by an XML construct - typically an element whose name, attributes and content identify the constraint component and convey the parameter values
- The location of the constraint element implies the value input:
    - If it is a child of a value shape element, the test value is the value produced by the expression of the value shape
    - If it is a child of a resource shape element, the test value is the file path of the resource

It follows that the schema language can be extended by defining **new constraint components**. This requires …

- *To define the signature* of the constraint component – parameter names and types
- *To define the XML representation* of a constraint – element name, kinds and names of the parameter nodes (which default to "attribute with a name equal to the parameter name")
- *To define the semantics* – how the test value plus the parameter values is mapped to
    - A boolean result (constraint passed or violated?)
    - Details of the validation result
- *To provide an implementation*

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

Name: grep

Parameter:	fileName	– a file name pattern
Parameter:	deep	– flag indicating if folder traversal is deep
Parameter:	glob	– a glob pattern
Parameter:	regex	– a regular expression
Parameter:	flags	– flags for the evaluation of a regular expression or a glob pattern

XML syntax: element name: `ex:grep`
parameter nodes: attributes with name equal parameter name

Semantics:

- If child of a `<file>` shape: a constraint is satisfied if the focus file is a text file containing a line with a substring `$plain`, or matching glob pattern `$glob`, or matching regular expression `$regex`, evaluating the glob pattern or regular expression as indicated by the matching flags given by `$flags` (e.g. case-insensitively)
- If child of a `<folder>` shape: a constraint is satisfied if the folder contains a file with a name matching name pattern `$fileName` and with text content satisfying the constraint as described above; if `$deep` is "true", consider also name matching files in descendant folders
- If child of a value shape: a constraint is satisfied if the test value supplied by the shape expression satisfies the constraint as described above

The implementation is either schema-based or plugin-based.

Schema-based extension is implemented by adding to the greenfox schema an element <gx:constraintComponent> declaring the signature and providing the implementation:

```
<gx:constraintComponent constraintElementName="ex:grep">
   <gx:parameter name="fileName" type="xs:string" occ="?"/>
   <gx:parameter name="deep" type="xs:boolean" occ="?"/>
   <gx:parameter name="plain" type="xs:boolean" occ="?"/>
   <gx:parameter name="pattern" type="xs:string" occ="?"/>
   <gx:parameter name="regex" type="xs:string" occ="?"/>
   <gx:parameter name="flags" type="xs:string" occ="?"/>
   <gx:xquery>><![CDATA[
      error(QName((), 'NOT_YET_IMPLEMENTED'),
            'Not yet implemented - constraint component 'greb')
      (: ***
       Replace with XQuery code implementing the constraint, using pre-bound variables
       $fileName, $deep, $plain, $pattern, $regex, $flags
         *** :)
      ]]></gx:xquery>
    </gx:constraintComponent>
```

Plugin-based extension is implemented by dropping into the bin folder of the greenfox installation an XQuery module providing the implementation code, conforming to a set of conventions described in the product documentation.

[- to-be-added: description of an alternative extension mechanism using command-line invocations of external tools -]

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")
    - for each observation, which is a result triggered by a value shape in order to record a resource value not related to constraint checking ("white values")

The **red validation report** is an RDF graph obtained by removing from the white validation report all green and white result values. Note that the validation report defined by the SHACL language [5] 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:white>

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 either describes a constraint violation ("red" or "yellow" result) or a successful validation of a focus resource against a constraint ("green" result).

A validation result is an RDF resource with several properties as described below. A key feature of the result model is that …

- 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 [xxx]. 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 [1] XQuery processor.

Discussion

[- to-be-added -]

Bibliography

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

[2] foxpath – an extended version of XPath 3.0 supporting file system navigation. Hans-Juergen Rennau. 2016. https://github.com/hrennau/foxpath

[3] FOXpath - an expression language for selecting files and folders. Rennau, Hans-Jürgen. Presented at Balisage: The Markup Conference 2016, Washington, DC, August 2 - 5, 2016. In *Proceedings of Balisage: The Markup Conference 2016*. Balisage Series on Markup Technologies, vol. 17 (2016). https://doi.org/10.4242/BalisageVol17.Rennau01.

[4] FOXpath navigation of physical, virtual and literal file systems. Hans-Juergen Rennau. 2017. https://archive.xmlprague.cz/2017/files/xmlprague-2017-proceedings.pdf p. 161-180

[5] Shapes Constraint Language (SHACL) 20.7.2017. World Wide Web Consortium (W3C). https://www.w3.org/TR/shacl

[6] XML Path Language (XPath) 3.1.2017. World Wide Web Consortium (W3C). https://www.w3.org/TR/xpath-31/

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

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

Appendix A1: greenfox schema for system S

This appendix lists the complete schema developed in section “Getting started with greenfox”.

```
<?xml version="1.0" encoding="UTF-8"?>
<greenfox greenfoxURI="http://www.greenfox.org/ns/schema-examples/system-s"
          xmlns="http://www.greenfox.org/ns/schema">

  <!-- *** External context *** -->
  <context>
      <field name="lastModified" value="2019-12-01"/>
  </context>

  <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"/>
        <foxpath expr="*.xml/codelist[entry/@code/string()]/@name" minCount="1"/>
        <file foxpath="*[is-xml(.)]">
          <targetSize msg="No codelist files found" minCount="1"/>
        </file>
      </folder>

      <!-- *** Testcase folder shape *** -->
      <folder foxpath=".\\test-*[input][output][config]" id="testcaseFolderShape">
```

```
        <targetSize msg="No testcase folders found" minCount="1"/>
        <folderContent msg="Testcase contains member other than input, output, config, log-*."
                       closed="true">
          <memberFolders names="input, output, config"/>
          <memberFiles names="log-" occ="*"/>
        </folderContent>

        <!-- *** msg config shape -->
        <file foxpath="config\msg-config.csv" mediatype="csv" csv.separator=","
              csv.withHeader="yes" id="msgConfigFileShape">
          <targetSize msg="Config file missing" count="1"/>

          <!-- Check - configured return codes ok? -->
          <xpath msg="Config file contains unknown return code" expr="//returnCode">
            <in>
              <eq>OK</eq>
              <eq>NOFIND</eq>
              <like>ERROR_*</like>
            </in>
          </xpath>
        </file>

        <!-- *** Request file shape *** -->
        <file foxpath="input\(*.xml, *.json)" id="requestFileShape">
          <targetSize msg="Input folder without request msgs" minCount="1"/>

          <!-- Check - request with response ? -->
          <foxpath msg="Request without response"
                   expr="..\..\output\*\file-name(.)"
                   containsXPath="$fileName ! replace(., '(.*)RQ(.*)$', '$1RS$2')"/>
        </file>

        <!-- *** Response file shape *** -->
        <file foxpath="output\(*.xml, *.json)" mediatype="xml-or-json">
          <targetSize msg="Output folder without request msgs" minCount="1"/>

          <!-- *** Check - response fresh? *** -->
          <lastModified msg="Stale output file" ge="${lastModified}"/>

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

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

          <!-- *** Check - known article number? -->
          <xpath msg="Unknown foo article number"
                 expr="//*:fooValue"
                 eqFoxpath="$domain\\codelists\*.xml/codelist[@name eq 'foo-article']
                            /entry/@code"/>

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

Appendix A2: Alignment of key concepts between greenfox and SHACL

This appendix summarizes the conceptual alignment between greenfox and SHACL. The striking correspondence is a consequence of our decision to use SHACL as a blueprint for the conceptual framework underlying the greenfox language. Greenfox can be thought of as a combination of SHACL's abstract validation model with a view of the file system through the prism of a unified value and expression model (XDM, XPath/XQuery + foxpath).

The alignment is described in two tables. The first table provides an aligned definition of the validation process as a decomposable operation as defined by greenfox and SHACL. The second table is an aligned enumeration of some building blocks of the conceptual framework underlying greenfox and SHACL.

Table 5. Alignment, part 1: validation model

Greenfox operation	SHACL operation
Validation of a file system against a greenfox schema	Validation of a data graph against a shapes graph
= Union of the results of the validation of the file system against all shapes	= Union of the results of the validation of the data graph against all shapes
Validation of a file system against a shape	Validation of a data graph against a shape
= Union of the results of all focus resources in the target of the shape	= Union of the results of all focus nodes in the target of the shape
Validation of a focus resource against a shape = Union of the results of the validation of the focus resource against all constraints declared by the shape	Validation of a focus node against a shape = Union of the results of the validation of the focus node against all constraints declared by the shape
Validation of a focus node against a constraint = function(constraint parameters , focus resource, resource values?	Validation of a focus node against a constraint = function(constraint parameters , focus node, property values?
Resource values = XPath(resource) | foxpath (resource)	Property values = SPARQL property path (node)

Table 6: Alignment, part 2: conceptual building blocks

Greenfox concept	**SHACL**	**Remark**
Resource shape: • Folder shape • File shape	Node shape	Common key concept: shape = set of constraints for a set of resources
Focus resource	Focus node	Common view: validation can be partioned into validation of a single resource against a single shape
Target declaration • Foxpath expression • Literal file system path	Target declaration • Class members • Subjects of predicate IRI • Objects of predicate IRI • Literal IRI (node target)	Difference: in greenfox a target declaration is essentially a navigation result, in SHACL it tends to be derived from class membership (ontological)
Resource value	Value node	Common view: non-trivial validation requires mapping resources to values
Mapping resource to value: • XPath expression • Foxpath expression	Mapping resource to property: • SPARQL property path	Common view: the mapping of a resource to a value is an expression
Value shape: • XPath shape • foxpath shape	Property shape	Common view: usefulness of an entity combining a single mapping of the focus resource to a value with a set of constraints for that value
Constraint declaration • Constraint component • Constraint parameters	Constraint declaration • Constraint component • Constraint parameters	Common view: a constraint declaration can be thought of as a function call
Constraint component • Signature • Mapping semantic	Constraint component • Signature • Mapping semantic	Common view: a constraint component can be thought of as a library function
Validation report • Constraint violations • Constraint passes • Observations	Validation report • Only constraint violations	Common view: a result is an RDF resource; difference: in greenfox also successful constraint checks produce results (“green results”); difference: in greenfox also observations can be produced, results unrelated to constraint checking (“white results”)
Constraint parameter • atomic • structured	Constraint parameter • atomic	Difference: in greenfox constraint parameter may have any degree of complexity
Extension language: • XPath/XQuery expression • foxpath expression	Extension language: • SPARQL SELECT queries • SPARQL ASK queries	Common view: extension of functionality is based on an expression language for mapping resources to values and values to a result

Mediatype integration: • Common data model • Common navigation model	-	Difference: in contrast to SHACL, greenfox faces a heterogeneous collection of validation targets, calling for integration concepts

Appendix A3. Validation result model

This appendix gives a detailed account of the validation result model.

Table 7. The validation result model – RDF properties, SHACL equivalent and XML representation. The XML representation is rendered as an XPath expression to be evaluated in the context of the XML element representing the result, which is a `<gx:red>`, `<gx:yellow>`, `<gx:green>` or `<gx:white>` element. Apart from the values shown in the table, individual constraint components may define additional values.

Property	Description	SHACL result property	XML representation

`gx:severity`	The possible values:
	`gx:Violation`
	`gx:Warning`
	`gx:Info`
	`gx:Pass`
	`gx:Observation`
	While `gx:Observation` is a value not related to a constraint check, the other ones represent constraint violations or a successful check
		`sh:severity`	Local name of the result representing element:
			`red    - gx:Violation`
			`yellow - gx:Warning`
			`green  - gx:Pass`
			`white  - gx:Observation`

`gx:fileSystem`	Identifies the file system validated	An aspect of `sh:focusNode`	`ancestor::gx:validationReport /@fileSystemURI`
`gx:focusFile`	File path of a file resource	An aspect of `sh:focusNode`	`@file`
`gx:focusFolder`	File path of a folder resource	An aspect of `sh:focusNode`	`@folder`
`gx:xpath`	The XPath expression of a value shape	`sh:resultPath`	`@xpath` or `./xpath`
`gx:foxpath`	The foxpath expression of a value shape	`sh:resultPath`	`@foxpath` or `./foxpath`
`gx:value`	A resource value, or single item of a resource value, causing a violation	`sh:value`	`@value` or `./value` A value consisting of several items is represented by a sequence of `<value>` child elements
`gx:sourceShape`	The value shape or resource shape defining the constraint; the value is the @id value on the shape element in the schema if present, or a value assigned by the greenfox processor otherwise	`sh:sourceShape`	`@shapeID`

gx:constraintComponent	Identifies the kind of constraint	sh:constraintComsponent	@constraintComponent
gx:message	A message communicating details to humans; The value is the @msg or <msg> value on the shape or constraint element in the schema, or a value assigned by the greenfox processor	sh:message	@msg or ./msg with ./msg/@xml:lang