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 1
Getting started with greenfox 2
    The system – system S 2
    Building a greenfox schema "system S" 4
Basic principles 10
Information model 11
    Part 1: resource model 11
    Part 2: schema model 13
    Part 3: validation model 13
Schema building blocks 14
Schema language extension 15
Validation results 16
    Validation reports and representations 16
    Validation result 17
Implementation 17
Discussion 17
A. Greenfox schema "system S" 18
B. Alignment of key concepts between greenfox and SHACL 21
C. Validation result model 22
D. Note on the generation of resource values by expression chains 24
Bibliography 24

Introduction

How to validate data against expectations? Major options are visual inspection, programmatic 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 [7], 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) [7] and a *unified navigation model* (foxpath/XPath) [3] [4] [5], [9] [11] 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 (like a document type) 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. Concerning schemas, this document discusses only the XML representation. Concerning results, XML and RDF are dealt with.

Before providing an 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`, `output` 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 **message configuration file**; CSV file with three columns: request file name, response file name, expected return code
folder	.//test-*/input	Test case input folder, containing request messages
file	.//test-*/input/*	A **request message file**; name extension `.xml` or `.json`; mediatype corresponding to name extension
folder	.//test-*/output	Test case output folder, containing response messages
file	.//test-*/output/*	A **response message file**; 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.

Apart from these static constraints, the presence of some files implies the presence of other files:

- For every request message, there must be a response message with a name derived from the request file name (replacing substring `RQ` with `RS`).

Expectations are not limited to the presence of files and folders - they include details of file contents, in some cases relating the contents of different files with different mediatypes:

- 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 and JSON fields) with a particular name (e.g. `fooValue`) must be found in the appropriate XML codelist discovered in a set of codelist files
- Response message return codes (contained by XML and JSON documents) must be as configured by the corresponding row in a CSV table

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">

  <!-- *** System file tree *** -->
  <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 count="1"
                        countMsg="No XSD folder found"/>

            <file foxpath="*.xsd" id="xsdFileShape">
                <targetSize minCount="1"
                            minCountMsg="No XSDs found"/>
            </file>
        </folder>

        <!-- *** Codelist folder shape *** -->
        <folder foxpath=".\\resources\codelists" id="codelistFolderShape">
            <targetSize count="1"
                        countMsg="No codelist folder found"/>

            <file foxpath="*[is-xml(.)]" id="codelistFileShape">
                <targetSize minCount="1"
                            minCountMsg="No codelist files found"/>
            </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 [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 the @path of `<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] 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 here is

```
\tt\greenfox\resources\example-system\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 containing `<folder>` element's target declaration. So here we have a `file` element describing all files found at the relative path

```
*.xsd
```

evaluated in the context of any folder selected by

```
\tt\greenfox\resources\example-system\system-s\\resources\xsd
```

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. Every folder in the target is tested against every constraint in the shape. When a folder is tested against a constraint, it is said to be the **focus resource** of the constraint.

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 (called the **constraint component**) and a set of arguments passed to the **constraint parameters**. Every kind of constraint has a "signature", a characteristic set of mandatory and optional constraint parameters, defined in terms of name, type and cardinality. A *constraint component* can therefore be thought of as a library function, and a *constraint declaration* is like a function call, represented by elements and/or attributes. Here, we declare a `TargetMinCount` constraint, represented by a @minCount attribute on a `<targetSize>` element. When a resource is validated against a constraint, the imaginary function consumes the constraint parameter values, inspects the resource and returns a validation result. If the constraint is violated, the validation result is a `<gx:red>` element which contains an optional message (either supplied by an attribute or constructed by the processor), along with a set of information items identifying the violating resource (@filePath), the constraint (@constraintComp and @constraintID) and its parameter values (@minCount). In the case of a `TargetMinCount` constraint, the violating resource is the folder providing the context when evaluating the target declaration. 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"
```

```
    valueCount="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 minCount="1"
              minCountMsg="No testcase folders found">

  <!-- # Check - test folder content ok? -->
  <folderContent
       closed="true"
       closedMsg="Testcase member(s) other than input/output/config, log-*.">
    <memberFolders names="input, output, config"/>
    <memberFile name="log-*" count="*"/>
  </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 `TargetMinCount` constraint checks that the system contains at least one such folder. The contents of these testcase folders are subject to several constraints defined by the `<folderContent>` element. There must be three subfolders `input`, `output` and `config`, and there may be any number of `log-*` elements, but not any other members (`FolderContentClosed` constraint).

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 count="1"
              countMsg="Config file missing"/>
  ...
</file>
```

The `TargetCount` constraint makes this file mandatory, but 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 fine-grained validation.

We add to 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 an `XPathValueIn` 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 `XPathValueIn` constraint:

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

The second example would produce the following validation result, identify resource and constraint, describing the constraint and exposing the offending value:

```
<gx:red msg="Config file contains unknown return code"
    filePath="C:/tt/greenfox/resources/example-system/system-s/resources/xsd"
    constraintComp="ExprValueIn"
    constraintID="xpath_1-in"
    valueShapeID="xpath_1"
    exprLang="xpath"
    expr="//returnCode">
    <gx:value>ERROR-SYSTEM</gx:value>
</red>
```

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 - called a **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 the section called “Schema building blocks” for more information about value shapes.

Now we are going 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(.)"
      containsXPath=
           "$fileName ! replace(., '(.*)RQ(.*)$', '$1RS$2')"
```

```
      containsXPathMsg="Request without response"
  ...
<file>
```

A Foxpath value shape combines a foxpath expression (@expr) with a set of constraints. The expression maps the focus resource to a resource 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 foxpath 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 non-empty `<codelist>` elements with unique names:

```
<folder foxpath=".\\resources\codelists" id="codelistFolderShape">
    ...
    <!-- # Check - folder contains codelists? -->
    <foxpath expr=".\*.xml//codelist[entry]/@name"
             minCount="1"
             minCoutMsg="Codelist folder without codelists"
             itemsUnique="true"
             itemsUniqueMsg="Codelist names must be unique"/>
    ...
</folder>
```

Note the unified view of file system contents offered by the foxpath language: a single expression starts with file system navigation, visiting all `.xml` files in the current folder, enters their XML content and selects the @name attributes of non-empty `codelist` elements, which may occur at any depth inside the content trees.

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 with 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 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 with an `ExprValueInFoxpath` constraint, represented by the @inFoxpath attribute:

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

As always with an XPath value shape, the XPath expression (@expr) selects the content items to be checked. The `ExprValueInFoxpath` constraint works as follows: it evaluates the foxpath expression provided by constraint parameter @inFoxpath 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"
             eqXPath="//*:returnCode"
             eqXPathMsg="Return code not the configured value"
</file>
```

The complete schema is shown in Appendix A, *Greenfox schema "system S"*. 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 development 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 each resource validated against the shape.
- A resource which is validated against a shape is called a **focus resource**.
- A resource shape may have a **target declaration** which selects a set of focus resources.
- A target declaration of a resource shape can be a file path or a foxpath expression.
- A target declaration of a resource shape is resolved in the context of all resources obtained from the target declaration of the containing resource shape.
- Every violation of a constraint produces a **validation result** describing the violation and identifying the focus resource and the constraint.
- 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, or by a chain of expressions.
- A **value shape** combines an expression mapping the focus resource to a resource value, or a resource value to another resource value, and a set of constraints against which to validate the resource value obtained.
- 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 mapping a focus resource to a resource value is the file path of the focus resource. The foxpath context item used by a value shape mapping a preceding resource value to another resource value is a single item of the preceding resource value.
- The **XPath context item** used by a value shape mapping a focus resource to a resource value 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 cannot be constructed. The XPath context item used by a value shape mapping a preceding resource value to another resource value is a single item of the preceding resource value.

- **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 of a derived resource property, or of an item from another resource value.

Folder resources

The table below summarizes the **resource properties** of a folder resource, as currently evaluated by greenfox. More properties may be added in the future, e.g. representing access rights.

Table 2. Resource properties of a folder resource.

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.

Property name	Value type	Description
[filepath]	`xsd:string`	The names of all ancestor folders and the folder itself, separated by a slash

Resource values of a folder are obtained by evaluating a foxpath expression in the context of [filepath]. They can also be obtained by evaluating an XPath or a foxpath expression in the context of an item

taken from another resource value. See Appendix D, *Note on the generation of resource values by expression chains* for implications of this recursive definition.

File resources

A file has the following **resource properties**, as currently evaluated by greenfox.

Table 4. Resource properties of a file resource.

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 sequence if not a text file)
[encoding]	`xsd:string?`	The encoding of the text content of the file (empty sequence if not a text file)
[octets]	`xsd:base64-Binary`	The binary file content

A file has the following **derived resource properties**, as currently evaluated by greenfox.

Table 5. Derived resource properties of a file resource.

Property name	Value type	Description
[filepath]	`xsd:string`	The names of all ancestor folders and the folder itself, separated by a slash
[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

Resource values of a file are obtained by evaluating a foxpath expression in the context of [filepath], or evaluating an XPath expression in the context of a [*doc] or [*doc-*] property. They can also be

obtained by evaluating an XPath or a foxpath expression in the context of an item taken from another resource value. See Appendix D, *Note on the generation of resource values by expression chains* for implications of this recursive definition.

For information about CSV parsing parameters, see [1], 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 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, or a focus value.

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

A **focus value** is a resource value providing a context in which to evaluate value shapes (rather than in the context of a resource's file path or node tree representation). A focus value is typically a set of nodes selected from the resource's node tree representation.

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

A **value shape** is a mapping of a focus resource, or of a resource value, to a resource value and a set of constraints which apply to the value.

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

A constraint is defined by a **constraint declaration**.

A constraint declaration is provided 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 definition** specifies a name, a type, a cardinality range and value semantics.

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

A **validation result** describes the outcome of validating a resource against a constraint. It contains a boolean value signaling conformance, an identification of the resource and the constraint, constraint parameter values and optional details about the violation.

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 focus 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.

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 schema serialized as XML is composed. The serialized schema should be distinguished from the logical schema, which is independent of a serialization and can be described 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. There is not necessarily 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:shapes`
Constraints library	A collection of constraint declaration nodes, which can be referenced by shapes	`gx:constraints`
Constraint components library	A collection of constraint component definitions, for which constraints can be declared	`gx:constraint-Components`
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:validatorXPath` or `gx:validatorFoxpath` child element, or a @validatorXPath or @validatorFoxpath attribute	`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 shape	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`

Building block	Role	XML representation
Focus mapper	Maps a resource to a focus value, or the items of a focus value to another focus value; contains value shapes to be applied to the focus value; may contain other Focus mappers using the focus value items as input	`gx:focusNode`
Base shape declaration	References a shape so that its constraints 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:fileSize`
		`gx:folderContent`
		`gx:hashCode`
		`gx:lastModified`
		`gx:mediaType`
		`gx:resourceName`
		`gx:targetSize`
		`gx:xsdValid`

Conditional node	A set of building blocks associated with a condition, so that the building blocks are only used if the condition is satisfied	`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, which specifies the constraint component name, declares the constraint parameters and provides an implementation. The implementation is an XPath or a foxpath expression, which accesses the parameter values as pre-bound variables. User-defined constraint components are used like built-in components: a constraint is declared by an element with attributes (or child elements) providing the parameter values and optional messages.

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

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, added to the schema as a child element of `gx:constraintComponents`:

```
<constraintComponent constraintElementName="ex:grep">
    <param name="pattern" type="xs:string"/>
```

```
    <param name="flags" type="xs:string?"/>
    <validatorXPath>
        exists(unparsed-text-lines($this)[matches(., $pattern, $flags)])
    </validatorXPath>
</constraintComponent>
```

The *context item* supplied to the validator is assigned by the greenfox processor according to the following rules:

- If the constraint is used by a value shape: an item from the resource value
- If the constraint is used by a folder shape: the file path of the focus resource
- If the constraint is used by a file shape, the validator is an XPath expression and the file can be parsed into an XDM node tree: the root node of the node tree
- Otherwise the file path of the focus resource

Because of these rules, the example code uses the built-in variable `$this` which is always bound to the file path, rather than the context item (`.`) which may be the file path or a document node, dependent on the mediatype of the file.

The constraint can be used like this:

```
<file foxpath="...">
    ex:grep pattern="fbIx?" flags="i"
            msg="File does not contain string '$pattern'."
            msgOK="File contains string '$pattern'."/>
</file>
```

Note the variable references in the message text, which the greenfox processor replaces with the actual parameter values.

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 was successful (“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 (“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 SHACL [7] 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. It contains 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:Pass` 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 in Appendix C, *Validation result model*. Key features of the result model are:

- 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 meaningful aggregation by resource, by constraint and by shape, as well as any combination of aggregated resources, constraints and shapes. Such aggregation may be useful, e.g. for integrating validation results into a graphical representation of the file system, or for analysis of impact.

See Appendix C, *Validation result model* for a detailed description of the validation result model – RDF properties, SHACL equivalent and XML representation.

Implementation

An implementation of a greenfox processor is available on github [6]. The processor is provided as a command-line tool (`greenfox.bat`, `greenfox.sh`). Example call:

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

The implementation is written in XQuery and requires the use of the BaseX [1] XQuery processor.

Discussion

Due to the rigorous framework on which it is based, the functionality of greenfox can be extended easily. Any number of new constraint components can be added without increasing the complexity of the language, as the *usage* of any constraint component follows the same pattern: select the component and assign the parameter values. Validation *results* likewise retain their simplicity, as their structure is immutable: a collection of result objects, reporting the validation of a single resource against a single constraint, expressed in a small and stable core vocabulary. New constraint components can be enhancements of the core language or extensions defined by user-defined schemas. Library schemas may give access to domain-specific sets of constraint components.

Another aspect of extension concerns the reuse of existing constraints and shapes. Reuse should be facilitated by refining the syntax and semantics of parameterizing and extending existing components. The value gain is immediate and the purity of the conceptual framework is not endangered.

The remainder of this discussion deals with the possibility to extend greenfox beyond adding new constraint components and refining techniques of component reuse. Care must be taken to avoid a hodgepodge of features increasing complexity and reducing uniformity, making further extension increasingly difficult and risky. Ideally, the future development of the language should be guarded by an architectural style as defined by Roy Fielding [2] – a set of architectural constraints. A good starting point is an attempt to take an abstract and fundamental view of the language.

Greenfox is **tree-oriented**, as a tree-structured perception of a file system is natural: a folder contains folders and files, a file (often) contains tree-structured information (XML, JSON, HTML, CSV, …). The expressiveness of greenfox can in large parts be attributed to the expressiveness of tree navigation languages (XPath, XQuery, foxpath), in combination with the suitability of the XDM model [8] for turning different mediatypes into a unified substrate for those languages.

On the other hand, greenfox is based on a rigorous conceptual framework which has been defined by SHACL [7], a **validation language for graphs** – without any relationship to tree structures. This apparent contradiction is resolved by identifying the *fundamental* concepts shared by the SHACL and greenfox languages, distinguishing them from derived concepts accounting for all the outward differences. Such fundamental concepts are:

1. itemization of information
2. identification of a subset of items with resources
3. constraint check: resource + constraint parameters = true/false + details
4. itemization of validation: one resource against one constraint
5. itemization of validation results: one unit per pair of resource and constraint
6. resource interface model: resource properties and resource values
7. resource value model: a mapping of resource property or resource value to a value

The degree of abstraction makes it unnecessary to prescribe the data model (RDF / XDM), the alignment between items and resources (RDF-nodes / Files+Folders), the value mapping languages (SPARQL / XPath+foxpath). The conceptual foundation is equally well-suited for supporting an RDF or an XDM based view.

This perception can give guidance for the further development of greenfox. Greenfox differs from other validation languages in its main goal which is a **unified view on system validity**, integrating any resources which can be accommodated in a file system. Greenfox is intent on hiding outward heterogeneity (e.g. of mediatype) behind rigorous abstractions. In this field, RDF has *very much* to offer, as it separates information content from its representation in a most radical way. There is no reason not to also consider the use of RDF nodes as resource values, or to use RDF expressions as vehicles of mapping and navigation. The integration of graph and tree models, the combination of their complementary strengths, holds considerable promise for anyone interested in unified views of information. In spite of its deep commitment to a tree-oriented data model and expression languages built upon it, the greenfox language might in due time integrate with graph technology in order to offer yet more comprehensive answers to problems of validity.

A. Greenfox schema "system S"

This appendix lists the complete schema developed in the section called “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>

   <!-- *** System file tree *** -->
   <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 count="1"
                        countMsg="No XSD folder found"/>
            <file foxpath="*.xsd" id="xsdFileShape">
               <targetSize minCount="1"
```

```
                    minCountMsg="No XSDs found"/>
   </file>
</folder>

<!-- *** Codelist folder shape *** -->
   <folder foxpath=".\\resources\codelists"
           id="codelistFolderShape">
      <targetSize count="1"
                  countMsg="No codelist folder found"/>

      <!-- # Check - folder contains codelists? -->
      <foxpath
         expr="*.xml/codelist[entry]/@name"
         minCount="1"
         minCountMsg="Codelist folder without codelists"
         itemsUnique="true"
         itemsUniqueMsg="Codelist names must be unique"/>

      <file foxpath="*[is-xml(.)]" id="codelistFileShape">
         <targetSize minCount="1"
                     minCountMsg="No codelist files found"/>
      </file>
   </folder>

   <!-- *** Testcase folder shape *** -->
   <folder foxpath=".\\test-*[input][output][config]"
           id="testcaseFolderShape">
      <targetSize minCount="1"
                  minCountMsg="No testcase folders found"/>

      <!-- # Check - test folder content ok? -->
      <folderContent
         closed="true"
         closedMsg="Testcase contains member other than
                    input, output, config, log-*.">
         <memberFolders names="input, output, config"/>
         <memberFile name="log-*" count="*"/>
      </folderContent>

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

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

      <!-- *** Request file shape *** -->
      <file foxpath="input\(*.xml, *.json)"
```

```
                        id="requestFileShape">
                     <targetSize
                        minCount="1"
                        minCountMsg="Input folder without request msgs"/>

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

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

                     <!-- # Check - response fresh? -->
                     <lastModified ge="${lastModified}"
                                   geMsg="Stale output file"

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

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

                     </ifMediatype>

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

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

                   </file>
                </folder>
             </folder>
          </domain>
       </greenfox>
```

B. Alignment of key concepts between greenfox and SHACL

This appendix summarizes the conceptual alignment between greenfox and SHACL. The striking correspondence reflects 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 model (XDM), supporting powerful expression languages (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 B.1. Greenfox/SHACL alignment, part 1: validation model

Greenfox operation	**SHACL operaration**
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 resource 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|foxpath, applied to a resource	Property values = SPARQL property path, applied to a node

Table B.2. Greenfox/SHACL 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 decomposed into instances of validation of a single focus 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	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

Greenfox concept	SHACL	Remark
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 • 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 (“blue results”)
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

C. Validation result model

This appendix defines the validation result model.

In the table below, 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:blue>` element. Apart from the result properties shown in the table, individual constraint components may define additional properties.

Table C.1. The validation result model – RDF properties, description, corresponding SHACL result property and XML representation.

RDF property	Description	SHACL result property	XML representation
`gx:severity`	The possible values: • `gx:Violation` • `gx:Warning` • `gx:Pass` • `gx:Observation`	`sh:severity`	Local name of the result representing element: • `red = gx:Violation`

RDF property	Description	SHACL result property	XML representation
	While `gx:Observation` is a value not related to a constraint check, the other ones represent constraint violations or a successful check		• `yellow = gx:Warning` • `green = gx:Pass` • `blue = gx:Observation`
`gx:fileSystem`	Identifies the file system validated	An aspect of `sh:focusNode`	`ancestor::gx:validation-Report/@fileSystemURI`
`gx:focusResource`	File path of a file or folder resource	An aspect of `sh:focusNode`	`@filePath`
`gx:focusNode`	XPath of a node within an XDM node tree representing the contents of a file resource	`sh:focusNode`	`@nodePath`
`gx:xpath`	The XPath expression of a value shape	`sh:resultPath`	`@expr` or `./expr` + `@exprLang= "XPath"`
`gx:foxpath`	The foxpath expression of a value shape	`sh:resultPath`	`@expr` or `./expr` + `@exprLang= "foxpath"`
`gx:value`	A resource value, or single item of a resource value, causing a violation	`gx:value`	`@value` or `value` A value consisting of several items is represented by a sequence of `value` child elements
`gx:valueCount`	Number of resources in a target, or of resource value items, causing a violation	-	`@valueCount`
`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	`gx:sourceShape`	`@resourceShapeID`, or `@valueShapeID`
`gx:constraint-Component`	Identifies the kind of constraint	`sh:constraint-Component`	`@constraintComp`
`gx:message`	A message communicating details to humans; the value is the `@msg` or `@...Msg` attribute or `<msg>` or `<...Msg>` child	`sh:message`	`@msg` or `msg` + `msg/@xml:lang` A message with a language tag is represented by a child

RDF property	Description	SHACL result property	XML representation
	element value on the shape or constraint element in the schema, or a value assigned by the greenfox processor. In the above, … is a prefix identifying the constraint to which the message relates. Examples: `@minCountMsg`, `@exprValueEqMsg`.		element with language attribute.

D. Note on the generation of resource values by expression chains

The recursive definition of *resource values* allows the construction of resource values through **chains of expressions**. When a chain is used, each combination of items from all expressions except the last one is mapped to a distinct resource value, which itself may have zero, one or more items. As an example, consider a first expression mapping a folder to a sequence of files, a second expression mapping each file to all `<row>` elements found in its node tree representation, and a final expression mapping each `<row>` element to its `<col>` child elements. This chain generates one resource value for each combination of file and row, consisting of zero, one or more `<col>` elements. These values are resource values of the folder to which the expression chain was applied.

Bibliography

[1] *BaseX*. 2020. BaseX GmbH. http:// basex.org.

[2] *Architectural Styles and the Design of Network-based Software Architectures.*. 2000. Roy Fielding. https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[3] *FOXpath - an expression language for selecting files and folders.*. 2016. Hans-Juergen Rennau. http://www.balisage.net/Proceedings/vol17/html/ Rennau01/BalisageVol17-Rennau01.html.

[4] *FOXpath navigation of physical, virtual and literal file systems.*. 2016. Hans-Juergen Rennau. https://archive.xmlprague.cz/2017/files/xmlprague-2017-proceedings.pdf.

[5] *foxpath - an extended version of XPath 3.0 supporting file system navigation.*. Hans-Juergen Rennau. 2017. https://github.com/hrennau/shax.

[6] *Greenfox - a schema language for validating file system contents and, by implication, real-world systems.*. Hans-Juergen Rennau. 2020. https://github.com/hrennau/greenfox.

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

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

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