Analyse d'un problème de validation après configuration de profils SEDA

Logilab
104 boulevard Auguste Blanqui
FR-75013 PARIS
contact@logilab.fr

référence	BRDX20-211004-RAP-1
version	1.0
date	Octobre 2021
auteur	Logilab

Table des matières

1 Exposé du problème

L'outil de configuration de profils SEDA que Logilab a développé permet de configurer la structure attendue des fichiers SEDA en imposant l'existence de certains éléments et en en interdisant d'autres. Il permet aussi de fixer certaines valeurs. L'ensemble de ces contraintes est traduit dans un fichier au format relaxng [1] qui permet de valider les fichiers XML SEDA produits par les différents outils du processus de travail archivistique.

L'équipe configurant les profils SEDA a été amenée à décrire les fichiers pouvant être inclus dans un message. Cela se fait à l'aide d'un élément `DataObjectGroup` qui va contenir des éléments `BinaryDataObject`, lesquels décrivent chacun un fichier (URI du fichier, format du fichier, poids, description , etc.). L'équipe souhaite indiquer qu'on peut avoir des fichiers XML et des fichiers PDF, avec toujours au minimum un fichier XML.

Dans l'outil de configuration de profils SEDA, elle a donc inséré dans le bloc `DataObjectGroup` :

- un bloc `BinaryDataObject` dont le sous-élément `MimeType` a une valeur fixée à `application/xml` et dont la cardinalité est comprise entre 1 et l'infini (ce qui signifie qu'on a au minimum 1 élement et qu'on peut en avoir autant de nécessaire).
- un bloc `BinaryDataObject` dont le sous-élément `MimeType` a une valeur fixée à `application/pdf` et dont la cardinalité est comprise entre 0 et l'infini (ce qui signifie qu'on peut ne pas avoir d'élément et qu'on peut en avoir autant de nécessaire).

Ceci se traduit aujourd'hui de la façon suivante en *relaxng*

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <start>
    <element name="DataObjectGroup">
      <ref name="DataObjectGroupElt"/>
    </element>
  </start>

  <define name="DataObjectGroupElt">
    <ref name="BinaryDataObjectXml"/>
    <ref name="BinaryDataObjectPdf"/>
  </define>

  <define name="BinaryDataObjectXml">
    <oneOrMore>
      <element name="BinaryDataObject">
        <attribute name="DataObjectIdType">
          <text/>
        </attribute>
        <element name="Uri">
          <text/>
```

(suite sur la page suivante)

1. https://relaxng.org/

(suite de la page précédente)

```
        </element>
        <element name="FormatIdentification">
          <element name="MimeType">
            <choice>
              <value>application/xml</value>
            </choice>
          </element>
        </element>
      </element>
    </oneOrMore>
  </define>

  <define name="BinaryDataObjectPdf">
    <zeroOrMore>
      <element name="BinaryDataObject">
        <attribute name="DataObjectIdType">
          <text/>
        </attribute>
        <element name="Uri">
          <text/>
        </element>
        <element name="FormatIdentification">
          <element name="MimeType">
            <choice>
              <value>application/pdf</value>
            </choice>
          </element>
        </element>
      </element>
    </zeroOrMore>
  </define>

</grammar>
```

Une lecture attentive de ces règles montre qu'on s'attend à trouver d'abord tous les `BinaryDataObject` de type XML (avec obligatoirement 1 élément) et ensuite tous les `BinaryDataObject` de type PDF.

Le problème est que les outils du processus de travail archivistique vont générer des fichiers XML SEDA dans lesquels les `BinaryDataObject` ne sont pas ordonnés par type. Il en ressort que de nombreux fichiers XML SEDA sont rejetés par l'étape de validation par relaxng alors qu'on pourrait sans problème les utiliser (l'ordre des `BinaryDataObject` n'est pas important).

On voudrait donc pouvoir indiquer :

- que l'élément `DataObjectGroup` contient des éléments `BinaryDataObject`
- que certains éléments `BinaryDataObject` ont leur sous-élément `MimeType` fixé à la valeur `application/xml`
- que certains éléments `BinaryDataObject` ont leur sous-élément `MimeType` fixé à la valeur `application/pdf`
- qu'il y a au moins un élément `BinaryDataObject` de type `application/xml`

Il y a aussi d'autres règles mais nous avons simplifié le problème pour cette analyse.

2 Cadre d'analyse

Afin de mener cette analyse, nous avons préparé quelques fichiers XML d'exemple illustrant la plupart des situations pouvant survenir.

- Un fichier dans lequel les `BinaryDataObject` sont ordonnés par type (d'abord les XML puis les PDF). Ce fichier doit être valide.

  `exemple-valide-1.xml` :

```xml
<?xml version="1.0" encoding="UTF-8"?>

<DataObjectGroup>
  <BinaryDataObject DataObjectIdType="id01">
    <Uri>https://mon.exemple.fr/id01</Uri>
    <FormatIdentification>
      <MimeType>application/xml</MimeType>
    </FormatIdentification>
  </BinaryDataObject>
  <BinaryDataObject DataObjectIdType="id03">
    <Uri>https://mon.exemple.fr/id03</Uri>
    <FormatIdentification>
      <MimeType>application/xml</MimeType>
    </FormatIdentification>
  </BinaryDataObject>
  <BinaryDataObject DataObjectIdType="id02">
    <Uri>https://mon.exemple.fr/id02</Uri>
    <FormatIdentification>
      <MimeType>application/pdf</MimeType>
    </FormatIdentification>
  </BinaryDataObject>
</DataObjectGroup>
```

- Un fichier dans lequel les `BinaryDataObject` ne sont pas ordonnés par type. On y trouve un XML puis un PDF puis à nouveau un XML. Ce fichier doit être valide.

  `exemple-valide-2.xml` :

```xml
<?xml version="1.0" encoding="UTF-8"?>

<DataObjectGroup>
  <BinaryDataObject DataObjectIdType="id01">
    <Uri>https://mon.exemple.fr/id01</Uri>
    <FormatIdentification>
      <MimeType>application/xml</MimeType>
    </FormatIdentification>
  </BinaryDataObject>
  <BinaryDataObject DataObjectIdType="id02">
    <Uri>https://mon.exemple.fr/id02</Uri>
    <FormatIdentification>
      <MimeType>application/pdf</MimeType>
    </FormatIdentification>
  </BinaryDataObject>
  <BinaryDataObject DataObjectIdType="id03">
    <Uri>https://mon.exemple.fr/id03</Uri>
    <FormatIdentification>
      <MimeType>application/xml</MimeType>
    </FormatIdentification>
  </BinaryDataObject>
</DataObjectGroup>
```

- Un fichier dans lequel les `BinaryDataObject` sont ordonnés par type mais pas dans l'ordre spécifié dans l'outil. On y trouve un PDF puis deux XML. Ce fichier doit être valide.

  `exemple-valide-3.xml` :

```xml
<?xml version="1.0" encoding="UTF-8"?>

<DataObjectGroup>
  <BinaryDataObject DataObjectIdType="id02">
    <Uri>https://mon.exemple.fr/id02</Uri>
    <FormatIdentification>
      <MimeType>application/pdf</MimeType>
    </FormatIdentification>
  </BinaryDataObject>
  <BinaryDataObject DataObjectIdType="id01">
    <Uri>https://mon.exemple.fr/id01</Uri>
    <FormatIdentification>
      <MimeType>application/xml</MimeType>
    </FormatIdentification>
  </BinaryDataObject>
  <BinaryDataObject DataObjectIdType="id03">
    <Uri>https://mon.exemple.fr/id03</Uri>
    <FormatIdentification>
      <MimeType>application/xml</MimeType>
    </FormatIdentification>
  </BinaryDataObject>
</DataObjectGroup>
```

- Un fichier dans lequel il n'y a qu'un seul `BinaryDataObject` de type XML. Ce fichier doit être valide.
  `exemple-valide-4.xml` :

```xml
<?xml version="1.0" encoding="UTF-8"?>

<DataObjectGroup>
  <BinaryDataObject DataObjectIdType="id01">
    <Uri>https://mon.exemple.fr/id01</Uri>
    <FormatIdentification>
      <MimeType>application/xml</MimeType>
    </FormatIdentification>
  </BinaryDataObject>
</DataObjectGroup>
```

- Un fichier dans lequel il n'y a qu'un seul `BinaryDataObject` de type PDF. Ce fichier doit être invalide.
  `exemple-invalide-1.xml` :

```xml
<?xml version="1.0" encoding="UTF-8"?>

<DataObjectGroup>
  <BinaryDataObject DataObjectIdType="id02">
    <Uri>https://mon.exemple.fr/id02</Uri>
    <FormatIdentification>
      <MimeType>application/pdf</MimeType>
    </FormatIdentification>
  </BinaryDataObject>
</DataObjectGroup>
```

- Un fichier dans lequel il n'y a aucun `BinaryDataObject`. Ce fichier doit être invalide.
  `exemple-invalide-2.xml` :

```xml
<?xml version="1.0" encoding="UTF-8"?>

<DataObjectGroup>
</DataObjectGroup>
```

- Un fichier dans lequel un des `BinaryDataObject` n'a pas son attribut `DataObjectIdType` obligatoire. Ce fichier doit être invalide.
  `exemple-invalide-3.xml` :

```xml
<?xml version="1.0" encoding="UTF-8"?>

<DataObjectGroup>
  <BinaryDataObject DataObjectIdType="id01">
    <Uri>https://mon.exemple.fr/id01</Uri>
    <FormatIdentification>
      <MimeType>application/xml</MimeType>
    </FormatIdentification>
  </BinaryDataObject>
  <BinaryDataObject DataObjectIdType="id03">
    <Uri>https://mon.exemple.fr/id03</Uri>
    <FormatIdentification>
      <MimeType>application/xml</MimeType>
    </FormatIdentification>
  </BinaryDataObject>
  <BinaryDataObject>
    <Uri>https://mon.exemple.fr/id02</Uri>
    <FormatIdentification>
      <MimeType>application/pdf</MimeType>
    </FormatIdentification>
  </BinaryDataObject>
</DataObjectGroup>
```

Comme exposé précédemment, les règles relaxng produites par l'outil de configuration de profils SEDA ressemblent à ceci :

```xml
<?xml version="1.0" encoding="UTF-8"?>

<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <start>
    <element name="DataObjectGroup">
      <ref name="DataObjectGroupElt"/>
    </element>
  </start>

  <define name="DataObjectGroupElt">
    <ref name="BinaryDataObjectXml"/>
    <ref name="BinaryDataObjectPdf"/>
  </define>

  <define name="BinaryDataObjectXml">
    <oneOrMore>
      <element name="BinaryDataObject">
        <attribute name="DataObjectIdType">
          <text/>
        </attribute>
        <element name="Uri">
          <text/>
        </element>
        <element name="FormatIdentification">
          <element name="MimeType">
            <choice>
              <value>application/xml</value>
            </choice>
          </element>
        </element>
      </element>
    </oneOrMore>
  </define>

  <define name="BinaryDataObjectPdf">
    <zeroOrMore>
```

(suite sur la page suivante)

(suite de la page précédente)

```
      <element name="BinaryDataObject">
        <attribute name="DataObjectIdType">
          <text/>
        </attribute>
        <element name="Uri">
          <text/>
        </element>
        <element name="FormatIdentification">
          <element name="MimeType">
            <choice>
              <value>application/pdf</value>
            </choice>
          </element>
        </element>
      </element>
    </zeroOrMore>
  </define>

</grammar>
```

On peut utiliser la bibliothèque Python lxml[2] pour valider les différents fichiers XML présentés ci-avant avec ces règles relaxng. On obtient alors le résultat suivant :

```
$ python3 validate.py
RelaxNG: relaxng-rules-1.xml
  * exemple-invalide-1.xml => False
  * exemple-invalide-2.xml => False
  * exemple-invalide-3.xml => False
  * exemple-valide-1.xml => True
  * exemple-valide-2.xml => False
  * exemple-valide-3.xml => False
  * exemple-valide-4.xml => True
```

Comme on s'y attendait, le résultat de la validation des fichiers `exemple-valide-2.xml` et `exemple-valide-3.xml` renvoie la valeur `False` puisque tous les `BinaryDataObject` de type XML n'y sont pas définis avant les `BinaryDataObject` de type PDF.

3 Analyse et solutions possibles

Dans le fichier relaxng généré par l'outil de configuration de profils SEDA, le problème provient des lignes :

```
<define name="DataObjectGroupElt">
  <ref name="BinaryDataObjectXml"/>
  <ref name="BinaryDataObjectPdf"/>
</define>
```

Dans ce bloc qui définit le contenu d'un élément `DataObjectGroup`, on trouve successivement le bloc définissant l'élément `BinaryDataObject` avec le type XML puis le bloc définissant l'élément `BinaryDataObject` avec le type PDF. Ils sont donc contraints à être consécutifs.

Le bloc `BinaryDataObjectXml` a la structure suivante :

```
<define name="BinaryDataObjectXml">
  <oneOrMore>
    <element name="BinaryDataObject">
      <!-- ... -->
    </element>
```

(suite sur la page suivante)

2. https://lxml.de/

(suite de la page précédente)

```
  </oneOrMore>
</define>
```

Le bloc `BinaryDataObjectPdf` a la structure suivante :

```
<define name="BinaryDataObjectPdf">
  <zeroOrMore>
    <element name="BinaryDataObject">
      <!-- ... -->
    </element>
  </zeroOrMore>
</define>
```

Le premier impose d'avoir un élément `BinaryDataObject` puis permet d'en avoir un nombre quelconque. Le second permet d'avoir un nombre quelconque d'éléments `BinaryDataObject` (voire aucun). Le fait que ces deux blocs soient consécutifs (voir la définition `DataObjectGroupElt`) va donc aboutir à la contrainte, inutile dans le contexte applicatif, d'avoir d'abord tous les `BinaryDataObject` de type XML puis tous les `BinaryDataObject` de type PDF.

3.1 Utilisation de la structure `interleave`

En relaxng, il est possible d'utiliser une structure `interleave` pour indiquer qu'un élément va contenir des sous-éléments mais que l'ordre de ces sous-éléments n'a pas d'importance. En revanche, les cardinalités imposées sur les sous-éléments doivent être respectées (nombres minimum et maximum).

Il pourrait sembler judicieux dans le cas présent d'utiliser cette struture. Cependant, elle ne fonctionne que pour des sous-éléments portant des noms différents. Dans le cas présent, les sous-élements sont tous des `BinaryDataObject` ; il n'est donc malheureusement pas possible d'utiliser cette structure.

3.2 Utilisation d'une structure `zeroOrMore` au premier niveau

Afin de relâcher la contrainte sur la séquence de haut-niveau imposant la consécutivité des éléments `BinaryDataObject` de type XML puis des éléments `BinaryDataObject` de type PDF, on peut utiliser une structure `zeroOrMore` englobante. Ainsi, on pourra répéter autant de fois que nécessaire la séquence `BinaryDataObject` de type XML / `BinaryDataObject` de type PDF.

Le fichier de règles relaxng est donc désormais :

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <start>
    <element name="DataObjectGroup">
      <ref name="DataObjectGroupElt"/>
    </element>
  </start>

  <define name="DataObjectGroupElt">
    <zeroOrMore>
      <ref name="BinaryDataObjectXml"/>
      <ref name="BinaryDataObjectPdf"/>
    </zeroOrMore>
  </define>

  <define name="BinaryDataObjectXml">
    <oneOrMore>
      <element name="BinaryDataObject">
        <attribute name="DataObjectIdType">
```

(suite sur la page suivante)

(suite de la page précédente)

```
          <text/>
        </attribute>
        <element name="Uri">
          <text/>
        </element>
        <element name="FormatIdentification">
          <element name="MimeType">
            <choice>
              <value>application/xml</value>
            </choice>
          </element>
        </element>
      </element>
    </oneOrMore>
  </define>

  <define name="BinaryDataObjectPdf">
    <zeroOrMore>
      <element name="BinaryDataObject">
        <attribute name="DataObjectIdType">
          <text/>
        </attribute>
        <element name="Uri">
          <text/>
        </element>
        <element name="FormatIdentification">
          <element name="MimeType">
            <choice>
              <value>application/pdf</value>
            </choice>
          </element>
        </element>
      </element>
    </zeroOrMore>
  </define>

</grammar>
```

Avec ces nouvelles règles, la validation des fichiers XML exemple donne :

```
$ python3 validate.py
RelaxNG: relaxng-rules-2.xml
  * exemple-invalide-1.xml => False
  * exemple-invalide-2.xml => True
  * exemple-invalide-3.xml => False
  * exemple-valide-1.xml => True
  * exemple-valide-2.xml => True
  * exemple-valide-3.xml => False
  * exemple-valide-4.xml => True
```

Le résultat de la validation du fichier `exemple-valide-2.xml` renvoie maintenant la valeur `True`. En revanche, celle du fichier `exemple-valide-3.xml` renvoie toujours la valeur `False`. Cela provient du fait que les règles maintenant imposent qu'on ait toujours la séquence `BinaryDataObject` de type XML, avec au moins un élément, puis `BinaryDataObject` de type PDF mais que cette séquence puisse être répétée autant de fois qu'on veut. Dans le cas de `exemple-valide-3.xml`, on démarre par un `BinaryDataObject` de type PDF, sans aucun de type XML avant ; ceci n'est pas permis par les règles exposées ci-avant.

On remarque également que la validation du fichier `exemple-invalide-2.xml` renvoie la valeur `True`. Ce fichier ne contient aucun `BinaryDataObject`, ce qui est permis par les règles.

3.3 Utilisation d'une structure `zeroOrMore` + `choice` au premier niveau

Pour pallier le problème de la non-validation du fichier `exemple-valide-3.xml` vue ci-avant, on va utiliser une structure englobante `zeroOrMore` contenant une structure `choice`. Ainsi, on choisit soit des `BinaryDataObject` de type XML soit des `BinaryDataObject` de type PDF et on répète ce choix à loisir. La contrainte est désormais très faible au niveau des cardinalités : on peut n'avoir aucun `BinaryDataObject` ou que des `BinaryDataObject` de type PDF.

Le fichier de règles relaxng est maintenant :

```
<?xml version="1.0" encoding="UTF-8"?>

<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <start>
    <element name="DataObjectGroup">
      <ref name="DataObjectGroupElt"/>
    </element>
  </start>

  <define name="DataObjectGroupElt">
    <zeroOrMore>
      <choice>
        <ref name="BinaryDataObjectXml"/>
        <ref name="BinaryDataObjectPdf"/>
      </choice>
    </zeroOrMore>
  </define>

  <define name="BinaryDataObjectXml">
    <oneOrMore>
      <element name="BinaryDataObject">
        <attribute name="DataObjectIdType">
          <text/>
        </attribute>
        <element name="Uri">
          <text/>
        </element>
        <element name="FormatIdentification">
          <element name="MimeType">
            <choice>
              <value>application/xml</value>
            </choice>
          </element>
        </element>
      </element>
    </oneOrMore>
  </define>

  <define name="BinaryDataObjectPdf">
    <zeroOrMore>
      <element name="BinaryDataObject">
        <attribute name="DataObjectIdType">
          <text/>
        </attribute>
        <element name="Uri">
          <text/>
        </element>
        <element name="FormatIdentification">
          <element name="MimeType">
            <choice>
              <value>application/pdf</value>
            </choice>
```

(suite sur la page suivante)

(suite de la page précédente)

```
          </element>
        </element>
      </element>
    </zeroOrMore>
  </define>

</grammar>
```

Avec ces nouvelles règles, la validation des fichiers XML exemple donne :

```
$ python3 validate.py
RelaxNG: relaxng-rules-3.xml
  * exemple-invalide-1.xml => True
  * exemple-invalide-2.xml => True
  * exemple-invalide-3.xml => False
  * exemple-valide-1.xml => True
  * exemple-valide-2.xml => True
  * exemple-valide-3.xml => True
  * exemple-valide-4.xml => True
```

Tous les fichiers qu'on voudrait voir valides sont bien valides avec cette solution. En revanche, un certain nombre de fichiers invalides (problème de cardinalité dans les `BinaryDataObject`) sont aussi considérés comme valides.

Compte tenu des possibilités de relaxng, il n'est pas possible de faire mieux. En revanche, on peut utiliser un autre système de validation, par exemple Schematron .

3.4 Utilisation de règles Schematron

Schematron [3] est un système dans lequel des règles sont exprimées sous forme d'expressions XPath. Ces expressions sont appliquées au document XML qu'on veut valider et doivent renvoyer une valeur non fausse. Grâce aux expressions XPath, schematron peut donc vérifier que des valeurs sont présentes ou non dans certains éléments du document.

Dans le cas présent, on voudrait vérifier que, dans l'élément `DataObjectGroup`, il y a au moins un élément `BinaryDataObject` dont le sous-élément `MimeType` contient la valeur textuelle `application/xml`.

Ceci s'exprime de la façon suivante en Schematron :

```
<?xml version="1.0" encoding="UTF-8"?>

<schema xmlns="http://purl.oclc.org/dsdl/schematron">

  <pattern>
    <title>At least one BinaryDataObject with XML Mime type</title>
    <rule context="DataObjectGroup">
      <assert test="BinaryDataObject/FormatIdentification/MimeType[text() =
↪'application/xml']">Expect at least one &lt;BinaryDataObject&gt; element with a␣
↪MIME type equal to 'application/xml'</assert>
    </rule>
  </pattern>

</schema>
```

En utilisant la bibliothèque Python `lxml` pour effectuer une validation des fichiers XML exemple à partir de ces règles schematron, on obtient :

```
$ python3 validate.py
Schematron: schematron-rules-1.xml
  * exemple-invalide-1.xml => False
```

(suite sur la page suivante)

3. https://schematron.com/

(suite de la page précédente)

```
  * exemple-invalide-2.xml => False
  * exemple-invalide-3.xml => True
  * exemple-valide-1.xml => True
  * exemple-valide-2.xml => True
  * exemple-valide-3.xml => True
  * exemple-valide-4.xml => True
```

Les règles schematron se focalisent uniquement sur la contrainte « au moins un `BinaryDataObject` de type XML », il n'y a pas de vérification de la structure de l'arbre XML (cela est fait avec relaxng). Ceci explique pourquoi seuls les fichiers `exemple-invalide-1.xml` et `exemple-invalide-2.xml` ont une validation renvoyant la valeur `False`.

3.5 Utilisation combinée de règles Schematron et RelaxNG

Il est possible de combiner les règles schematron et relaxng au sein d'un même document XML. Certains outils sont d'ailleurs capables d'utiliser les deux en parallèle pour faire la validation.

Les règles combinées s'écrivent de la façon suivante :

```
<?xml version="1.0" encoding="UTF-8"?>

<rng:grammar
    xmlns:rng="http://relaxng.org/ns/structure/1.0"
    xmlns:sch="http://purl.oclc.org/dsdl/schematron">

  <rng:start>
    <rng:element name="DataObjectGroup">
      <rng:ref name="DataObjectGroupElt"/>
    </rng:element>
  </rng:start>

  <rng:define name="DataObjectGroupElt">
    <rng:zeroOrMore>
      <rng:choice>
        <rng:ref name="BinaryDataObjectXml"/>
        <rng:ref name="BinaryDataObjectPdf"/>
      </rng:choice>
    </rng:zeroOrMore>
  </rng:define>

  <rng:define name="BinaryDataObjectXml">
    <rng:oneOrMore>
      <rng:element name="BinaryDataObject">
        <rng:attribute name="DataObjectIdType">
          <rng:text/>
        </rng:attribute>
        <rng:element name="Uri">
          <rng:text/>
        </rng:element>
        <rng:element name="FormatIdentification">
          <rng:element name="MimeType">
            <rng:choice>
              <rng:value>application/xml</rng:value>
            </rng:choice>
          </rng:element>
        </rng:element>
      </rng:element>
    </rng:oneOrMore>
  </rng:define>
```

(suite sur la page suivante)

(suite de la page précédente)

```
  <rng:define name="BinaryDataObjectPdf">
    <rng:zeroOrMore>
      <rng:element name="BinaryDataObject">
        <rng:attribute name="DataObjectIdType">
          <rng:text/>
        </rng:attribute>
        <rng:element name="Uri">
          <rng:text/>
        </rng:element>
        <rng:element name="FormatIdentification">
          <rng:element name="MimeType">
            <rng:choice>
              <rng:value>application/pdf</rng:value>
            </rng:choice>
          </rng:element>
        </rng:element>
      </rng:element>
    </rng:zeroOrMore>
  </rng:define>

  <sch:pattern>
    <sch:title>At least one BinaryDataObject with XML Mime type</sch:title>
    <sch:rule context="DataObjectGroup">
      <sch:assert test="BinaryDataObject/FormatIdentification/MimeType[text() =
↪'application/xml']">In &lt;DataObjectGroup&gt; element, expect at least one &lt;
↪BinaryDataObject&gt; element with a MIME type equal to 'application/xml'</
↪sch:assert>
    </sch:rule>
  </sch:pattern>

</rng:grammar>
```

La bibliothèque Python `lxml` peut faire une validation relaxng et une validation schematron à partir de ce fichier unique. En les combinant, on obtient le résultat suivant pour les fichiers XML exemple :

```
$ python3 validate.py
Combined RelaxNG / Schematron: combined-relaxng-schematron-rules-1.xml
  * exemple-invalide-1.xml => False
  * exemple-invalide-2.xml => False
  * exemple-invalide-3.xml => False
  * exemple-valide-1.xml => True
  * exemple-valide-2.xml => True
  * exemple-valide-3.xml => True
  * exemple-valide-4.xml => True
```

En combinant les deux systèmes de validation, il est donc possible d'arriver au résultat souhaité.

4 Conclusion

Le système de validation RelaxNG ne permet pas de décrire l'ensemble des contraintes qu'on voudrait exprimer. Si on veut que les éléments `BinaryDataObject` de différents types puissent être placés dans n'importe quel ordre dans un élément `DataObjectGroup`, il n'est pas possible d'imposer des contraintes sur le nombre minimum ou maximum d'éléments `BinaryDataObject` de chacun des types.

Si on désire tout de même imposer des contraintes sur les cardinalités, il faut utiliser un système complémentaire de validation, comme par exemple Schematron. Cependant, cela complexifie le système de validation des fichiers XML SEDA produits lors du processus de travail archivistique, et il n'est peut-être pas possible de mettre en place une validation complémentaire. D'autre part, l'écriture automatique des règles Schematron à partir de l'outil de configuration

de profils SEDA n'est sans doute pas évidente à mettre ne œuvre. En effet, l'outil se concentre sur la structure des fichiers attendus et il faut prendre du recul pour pouvoir dégager les contraintes complémentaires.

Selon nous, aucun système de validation n'est parfait et il y a toujours des points qui ne peuvent pas être vérifiés. Il n'est donc ni étonnant ni gênant que le système relaxng seul ne puisse pas vérifier toutes les contraintes qu'on souhaite. Les outils en aval du valideur relaxng devront faire remonter des messages d'erreur explicite en cas de problème pour que les utilisateurs puissent comprendre leur erreur et apporter les corrections nécessaires.