XML - Semaine 3

Pierre Nerzic

février-mars 2018

Le cours de cette semaine présente deux mécanismes :

- RelaxNG encore un autre mécanisme de validation d'un document,
- XPath pour extraire des informations d'un document XML.

Pierre Nerzic

RelaxNG

Pierre Nerzic

Présentation

Nous revenons vers la validation de documents XML. RelaxNG (*Regular Language for XML Next Generation*) permet d'écrire des feuilles de validation de manière à la fois complète et agréable.

RelaxNG offre une syntaxe simple comme une DTD et des types aussi précis qu'un schéma.

En termes de performance, RelaxNG est bien supérieur aux Schémas XML, mais par contre, les messages d'erreur sont moins clairs. RelaxNG est adapté à la validation en masse de gros documents.

Exemple de document à valider

Voici un document modélisant des messages simples :

```
<?xml version="1.0" encoding="UTF-8"?>
<messages>
    <message numero="1" date="2018-01-01">
        <dest>promo2018</dest>
        <dest bcc="oui">Pierre Nerzic</dest>
        <contenu>Bonne année !</contenu>
    </message>
    <message numero="2">
        <dest>promo2018</dest>
        <contenu>Bonne rentrée !</contenu>
    </message>
    ...
</messages>
```

DTD du document

Sa DTD est :

```
<!ELEMENT messages ( message+ ) >

<!ELEMENT message ( dest+, contenu ) >
<!ATTLIST message numero CDATA #REQUIRED >
<!ATTLIST message date CDATA #IMPLIED >

<!ELEMENT dest ( #PCDATA ) >
<!ATTLIST dest bcc CDATA (oui|non) "non" >

<!ELEMENT contenu ( #PCDATA ) >
```

C'est très lisible mais les types sont trop simplistes.

Pierre Nerzic

XML Schema du document

Une partie de son schéma est :

```
<xsd:element name="messages" type="ElemMessages"/>

<xsd:complexType name="ElemMessages">
  <xsd:sequence>
    <xsd:element name="message" type="ElemMessage" maxOccurs="un
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ElemMessage">
  <xsd:sequence>
    <xsd:element name="dest" type="TypeDest" maxOccurs="unbounde
    <xsd:element name="contenu" type="xsd:string" />
  </xsd:sequence>
  <xsd:attribute name="numero" type="xsd:positiveInteger" use="r
  <xsd:attribute name="date" type="xsd:date" />
```


Pierre Nerzic

Feuille RelaxNG

Sa feuille de validation RelaxNG s'écrit en « syntaxe compacte » 

```
element messages {
  element message {
    attribute numero { xsd:positiveInteger },
    attribute date { xsd:date }?,
    element dest {
        attribute bcc { "oui"|"non" }?,
        text
    }+,
    element contenu { text }
  }*
}
```

Pour valider : `jing -c messages.rnc messages.xml`

La simplicité de la perfection.

Pierre Nerzic

Deux syntaxes pour RelaxNG

La syntaxe précédente s'appelle la *syntaxe compacte*, mais elle n'est pas au format XML. Alors il existe aussi une écriture XML : 

```
<?xml version="1.0" encoding="utf-8"?>
<rng:element name="messages"
    xmlns:rng="http://relaxng.org/ns/structure/1.0">
  <rng:oneOrMore>
    <rng:element name="message">
      <rng:attribute name="numero"><rng:data type="xsd:positiveI
      <rng:optional>
        <rng:attribute name="date"><rng:data type="xsd:date"/></
      </rng:optional>
      ...
      <rng:element name="contenu"><rng:text /></rng:element>
    </rng:element>
  </rng:oneOrMore>
</rng:element>
```


Pierre Nerzic

Principes de RelaxNG, syntaxe compacte

Les éléments sont définis par :

```
element NOM { CONTENU } REPETITIONS
```

- Le *contenu* peut être :
  - des attributs définis par : `attribute NOM { TYPE }`
  - des éléments enfants, voir le transparent suivant
  - du texte : mot-clé `text` (mais voir page 13)
  - rien : mot-clé `empty`
- Les *répétitions* sont spécifiées par un joker style egrep : `* ? +`

```
element message {
    attribute numero { text },
    attribute date { text } ?,
    element dest { text } +,
    element contenu { text }
}*
```

Ordonnancement des éléments enfants

- Il faut séparer les enfants successifs par des virgules
- Pour accepter les sous-éléments d'un élément dans un ordre quelconque, il faut les séparer par `&` :

  ```
  CONTENU1 & CONTENU2 ...
  ```

  Exemple :

  ```
  element personne {
      element nom { text } & element prenom { text }
  }
  ```

- Si, au contraire il y a des alternatives exclusives, il faut écrire :

  ```
  CONTENU1 | CONTENU2 ...
  ```

NB: en cas d'ambiguïté, il faut tout entourer de `(...)`

Exemple de successions dans le contenu

```
element annuaire {
  element contact {
    ( element personne {
            element nom { text } & element prenom { text }
      }
      |
      element entreprise { text }
    ),
    ( element email { text }+ | element telephone { text } )
  }*
}
```

Voir page 17 une autre syntaxe plus claire.

Types XSD et contraintes

Le type des données quelconques est `text`. On peut aussi employer un type XML Schemas plus précis. Il suffit seulement de les préfixer par `xsd:` : `xsd:integer`, `xsd:date`, `xsd:string`, etc.

Il est possible de rajouter des contraintes, les mêmes que les restrictions des schémas. On les écrit à la suite entre `{...}` :

```
element message {
    attribute numero { xsd:nonNegativeInteger },
    attribute date {
        xsd:date { minInclusive="2015-11-01" }
    } ?,
    element dest {
        xsd:string { pattern="[a-zA-Z_.]+@[a-zA-Z.]+" }
    } +,
    element contenu { xsd:string }
}*
```

Alternatives

Lorsque RelaxNG vérifie un document XML, il normalise les valeurs (suppression des espaces avant et après) avant de regarder si elles sont du bon type. Il n'y a pas de normalisation quand ce sont des alternatives :

```
attribute type { "portable" | "fixe" | "fax" }
```

Cette règle autorise seulement la première de ces deux lignes :

```
<telephone type="portable">...
<telephone type="     fixe   ">...
```

Types liste

Comme avec les schémas, on peut valider un élément contenant une séquence de valeurs séparées par des espaces :

```
element dimensions {
    list { xsd:float, xsd:float, xsd:float, ("mm"|"cm") }
}
element vecteur { list { xsd:float+ } }
```

Ces règles autorisent des choses telles que :

```
<dimensions>3.35 6.87 -1.57 mm</dimensions>
<vecteur>17.65 -98.2 374.2</vecteur>
```


Pierre Nerzic

Syntaxe compacte nommée

Au lieu d'imbriquer les définitions :

```
element NOM1 {
    element NOM2 { TYPE2 },
    element NOM3 { TYPE3 }
}
```

On peut écrire :

```
start = NOM1
NOM1 = element * { NOM2, NOM3 }
NOM2 = element * { TYPE2 }
NOM3 = element * { TYPE3 }
```

Ici l'étoile désigne l'élément courant. Ne pas confondre avec le joker de répétition.

Exemple

La feuille précédente se ré-écrit en (mais PB: jing plante) :

```
start = element annuaire {
    contact*
}
contact = element * {
    (personne | entreprise), (email+ | telephone)
}
personne = element * {
    nom & prenom          # bug avec jing
}
entreprise = element * { text }
email = element * { text }
telephone = element * { text }
nom = element * { text }
prenom = element * { text }
```

XPath

Pierre Nerzic

Présentation

XPath est un mécanisme (syntaxe + fonctions) permettant d'extraire des informations d'un document XML. Par exemple, dans le document `messages.xml`,

```
<messages>
    <message numero="1" date="2018-01-01">
        <dest>promo2018</dest>
        <dest bcc="oui">Pierre Nerzic</dest>
        <contenu>Bonne année !</contenu>
    </message>
    ...
```

extraire le contenu du message n°4 s'écrit ainsi en XPath :

```
/messages/message[@numero=4]/contenu
```

Parcours d'arbre

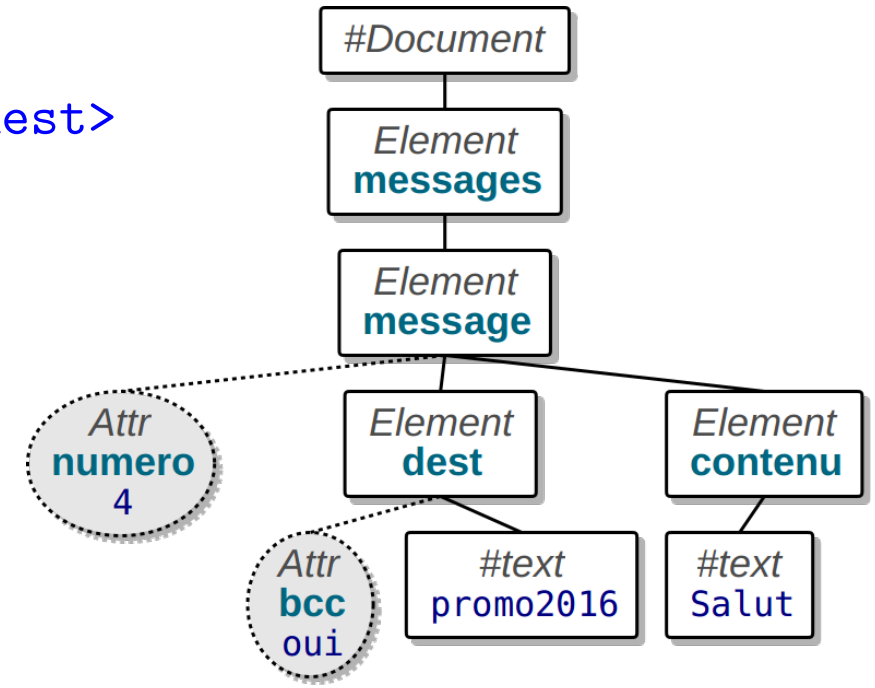
XPath sert à extraire des informations dans un arbre XML.

Soit ce document XML : 

```
<?xml version="1.0"?>
<messages>
  <message numero="4">
    <dest bcc="oui">promo2018</dest>
    <contenu>Salut</contenu>
  </message>
</messages>
```

NB: le document complet contient tous les messages.

Voici l'arbre XML correspondant :



Pierre Nerzic

Principe général

Le but d'XPath est d'aller chercher les informations voulues dans le document XML. Ex: quels sont les destinataires du message n°2 ?

Cela se fait à l'aide d'un chemin d'accès qui ressemble beaucoup à un nom complet Unix, mais avec des conditions écrites entre `[]`.

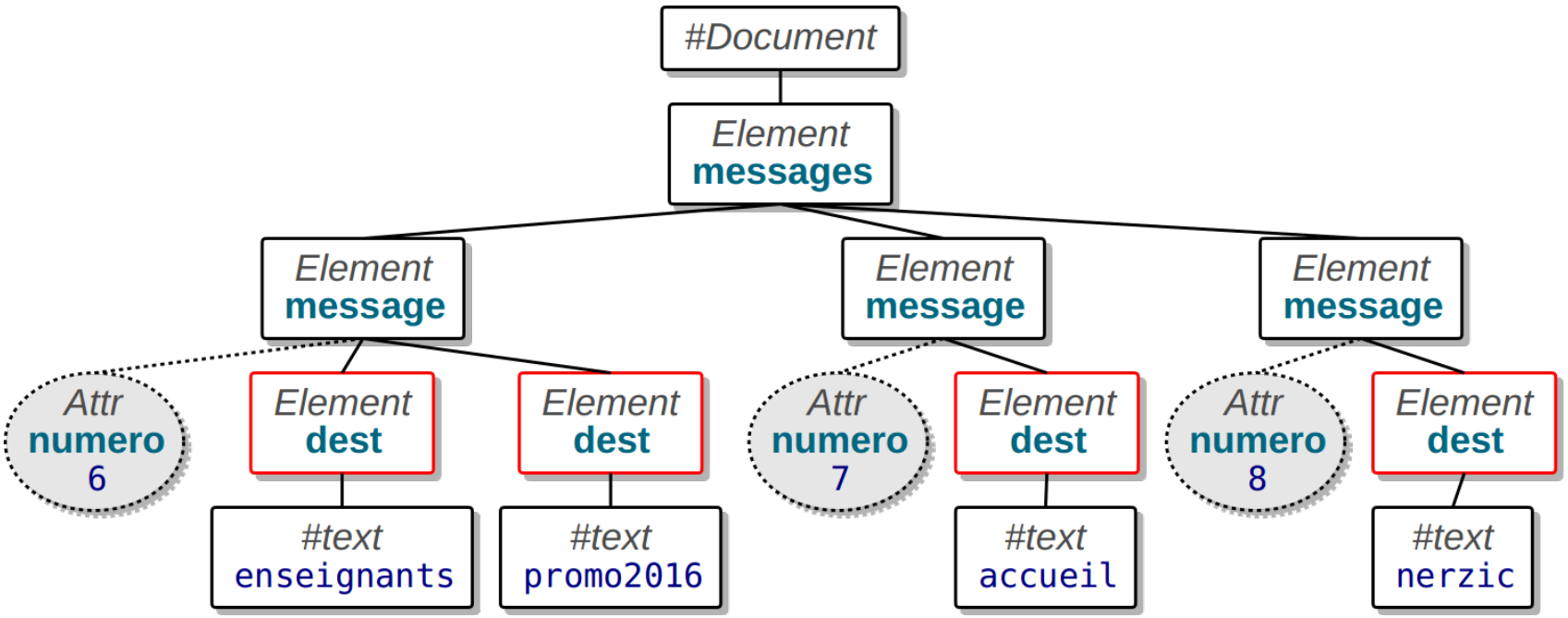
Exemple :

```
/messages/message[@numero="2"]/dest
```

C'est un peu comme un nom complet absolu dans Unix. Cependant, il y a énormément plus de possibilités pour écrire ces chemins et d'autre part, les chemins peuvent retourner plusieurs résultats.

Réponses multiples

Un point très important est qu'une expression XPath peut retourner plusieurs réponses. En effet, contrairement à Unix, un élément parent peut contenir plusieurs exemplaires du même élément enfant. Le chemin `/messages/message/dest` sélectionne 4 nœuds (rouge).



Évaluation d'une expression XPath

Pour évaluer une expression en ligne de commande, il y a :

- `xmlstarlet sel --template --value-of` *expression document.xml*
- `xmllint --xpath` *expression document.xml*
- `xmllint --shell` *document.xml* lance un mode interactif très intéressant :
  - `cat` *expression* : affiche les résultats de l'expression XPath
  - `grep` *mot* : cherche les occurrences du mot dans le document

La touche `F9` dans *XML Copy Editor* permet de saisir une expression XPath.

Pour les navigateurs, je vous fournis ce formulaire. Allez voir son source pour savoir comment évaluer du XPath en JavaScript.

XPath en JavaScript

```
var fichier = "messages.xml";
var xpath = "/messages/message/contenu/text()";
// demande du fichier au serveur
let requete = new XMLHttpRequest();
requete.onreadystatechange = function() {
  if (requete.readyState == 4 && requete.status == 200) {
    let xml = requete.responseXML;
    // évaluer l'expression XPath et afficher les résultats
    let nodes = xml.evaluate(xpath, xml, null,
                  XPathResult.ORDERED_NODE_ITERATOR_TYPE, null);
    while (node = nodes.iterateNext()) {
        document.write(node.nodeValue + "<br>");
    }
  }
}
requete.open("GET", fichier); requete.send();
```


Pierre Nerzic

Structure d'une expression XPath simple

Une expression XPath est une suite d'étapes séparées par des séparateurs : `[sep] étape1 sep étape2 sep étape3...` Les étapes sont les noms des éléments dans lesquels il faut aller successivement.

Cependant cela dépend du séparateur employé :

- `/` Ce séparateur se comporte comme dans Unix : s'il est mis au début du chemin, il représente le document entier ; s'il est mis entre les étapes, c'est un simple séparateur.
- `//` Ce séparateur signifie de sauter un nombre quelconque d'éléments quelconques. C'est un peu comme si on écrivait `/*/*/*.../` un nombre indéfini de fois, y compris 0. S'il est au début du chemin, cela signifie alors que la première étape est à chercher n'importe où dans l'arbre.

Exemples

Voici quelques exemples de chemins :

- `/messages/message/dest` sélectionne tous les éléments `<dest>` de tous les éléments `<message>` de la racine `<messages>`.
- `//dest` sélectionne tous les éléments `<dest>` où qu'ils soient dans l'arbre. Ça donne le même résultat que l'exemple précédent parce qu'il n'y a pas de `<dest>` ailleurs.
- `/messages//contenu` sélectionne tous les éléments `<contenu>` situés sous la racine `<messages>`.

Attributs des éléments

Pour désigner un attribut et non pas un sous-élément, on met un `@` devant le nom de l'attribut.

Exemples :

- `/messages/message/@numero` sélectionne tous les nœuds Attributs nommés `numero` des éléments `<message>`.
- `//@numero` sélectionne tous les nœuds attributs portant ce nom n'importe où dans l'arbre.

NB: XPath retourne les nœuds attribut sélectionnés sous la forme `nom="valeur"`. Pour avoir seulement la valeur de l'un d'entre eux, il faut écrire `string(chemin)`. Par exemple, `string(//messages[dest="promo2018"]/@numero)`. Attention `string()` exige qu'il n'y ait qu'un seul attribut sélectionné par l'expression XPath.

Autres étapes d'un chemin

D'autres étapes peuvent être employées :

- `.` désigne le nœud courant,
- `..` désigne le nœud parent,
- `*` désigne tous les éléments de ce niveau. C'est plus restreint que `//`.
- `|` regroupe les résultats de deux expressions XPath

Exemples :

- `/messages/*/@numero` sélectionne tous les nœuds Attributs nommés `numero` des éléments situés sous `<messages>`
- `//dest/@*` sélectionne tous les nœuds Attributs des éléments `<dest>` du document.
- `//message/dest|//message/contenu` sélectionne les éléments `<dest>` et `<contenu>` avec deux chemins complets.

Remarque sur l'opérateur d'alternative

Dans XPath 1.0, l'opérateur | permet seulement de réunir les résultats de deux expressions complètes et indépendantes. Contrairement au bon sens, l'expression suivante ne retourne pas les textes des <dest> et <contenu> trouvés dans les messages :

```
//message/dest|contenu/text()
```

Elle retourne seulement les éléments <dest> car la deuxième expression `contenu/text()` ne retourne rien.

- C'est seulement dans XPath 2.0 qu'on peut écrire :

  ```
  //message/(dest|contenu)/text()
  ```

- En XPath 1.0, il faut malheureusement écrire :

  ```
  //message/*[self::dest or self::contenu]/text()
  ```

Conditions sur les étapes

L'une des forces de XPath est de pouvoir rajouter des conditions appelées *prédicats* sur les étapes d'un chemin (éléments et attributs). Un prédicat se met entre `[...]` juste après l'élément dont il filtre l'un des enfants. On peut mettre plusieurs prédicats.

Exemple :

- `/messages/message[@numero=5]/contenu` sélectionne les `<contenu>` des messages dont l'attribut `numero` est 5.
- `//message[dest="promo2018"]/contenu` sélectionne la `<contenu>` des `<message>` ayant un sous-élément `<dest>` contenant la chaîne « promo2018 ».
- `//message[dest="iut"][@date="2018-01-01"]` sélectionne les `<message>` ayant un sous-élément `<dest>` contenant « iut » et un attribut date valant « 2018-01-01 ».

Syntaxe des conditions

Les conditions s'écrivent classiquement. On peut combiner des opérateurs logiques et d'autres conditions.

- Les comparaisons se font avec des expressions XPath qui portent sur le contenu du noeud courant,
- la notation [*index*] sélectionne l'élément ayant cet index (1 à n) dans la liste de son parent,
- le prédicat [*enfant*] est vrai si l'élément contient cet enfant.

Exemples :

- `//message[7]/contenu` sélectionne le `<contenu>` du 7e élément `<message>` du document.
- `//message[contenu and not(@date)]` sélectionne les `<message>` qui ont un `<contenu>` mais pas d'attribut date.

Opérateurs de comparaison

Pour écrire les prédicats, XPath propose ces opérateurs un peu différents de ceux du C :

- arithmétique : `+ - * div mod` (et non pas `/` et `%`)
- comparaisons : `< <= = != >= >` (et non pas `==`)
- logique : `and or not(condition)` (et non pas `&&`, `||` et `!`)

Exemples :

- `//message[not(@numero < 5 or @numero >= 9)]` sélectionne les `<message>` dont l'attribut `numero` est entre 5 et 8.
- `//message[@numero mod 5 = 0]` sélectionne les `<message>` dont l'attribut `numero` est un multiple de 5.

Fonctions XPath

XPath possède de très nombreuses fonctions, dont :

- Fonctions sur les éléments :
  - `string(s)` retourne le texte de l'expression `s`
  - `position()` retourne l'index de l'élément dans son parent (premier = n°1)
  - `last()` retourne le n° du dernier élément dans son parent

Exemples :

- `string(/messages/message[2]/contenu)` retourne le contenu du 2e message.
- `/messages/message[position()<=3]` sélectionne les 3 premiers éléments `<message>` du document.
- `//dest[position()>last()-3]` sélectionne les `<dest>` qui sont parmi les trois derniers enfants de leur parent.

Fonctions XPath (suite)

Une fonction est particulièrement utile : `count(expression)`. Elle compte le nombre de nœuds XML (élément, attributs, textes...) sélectionnés par l'expression. On l'utilise dans des conditions.

Attention, count() compte les nœuds sélectionnés, chacun dans son parent *séparément*. Ça conduit à faire des erreurs si on croit que `count` peut regrouper différents comptages.

Exemples :

- `//message[count(dest)>2]` retourne les éléments `<message>` ayant plus de deux enfants `<dest>`.
- `//message[count(//dest)>2]` retourne tous les éléments `<message>` s'il y a plus de deux éléments `<dest>` quelque part dans le document.

Fonctions XPath (suite)

- Fonctions sur les chaînes :
  - `string-length(s)` retourne la longueur de la chaîne `s`
  - `concat(s1, s2,...)` concatène les chaînes passées
  - `substring(s,deb,lng)` retourne `lng` caractères de `s` à partir du n°`deb` (premier = 1)
  - `contains(s1, s2)` vrai si `s1` contient `s2`
  - `starts-with(s1,s2)` et `ends-with(s1,s2)`
  - `matches(s, motif)` vrai si `s` correspond au motif

Exemple :

- `//message[string-length(contenu)<=15 and not(starts-with(dest, "promo"))]/@numero` retourne les numéros des messages dont le contenu ne fait pas plus de 15 caractères et aucun destinataire ne commence par « promo ».

Fonctions XPath (suite)

- Fonctions mathématiques :
  - `abs(nb)`, `ceiling(nb)`, `floor(nb)`, `round(nb)`
- Fonctions sur les dates et heures :
  - `year-from-dateTime(dt)`, `month-from-dateTime(dt)`, `day-from-dateTime(dt)`, `hours-from-dateTime(dt)`, `minutes-from-dateTime(dt)`, `seconds-from-dateTime(dt)`
  - `year-from-date(d)`, `month-from-date(d)`, `day-from-date(d)`
  - `hours-from-time(t)`, `minutes-from-time(t)`, `seconds-from-time(t)`

Retour sur les composants d'un chemin

Un chemin XPath est constitué de `[sep] étape1 sep étape2 sep étape3...` Chaque étape est soit le nom d'un élément, soit `@` et le nom d'un attribut ; chacune suivie éventuellement d'un prédicat entre crochets :

```
/racine/element1[filtre1]/.../@attribut[filtre3]
```

Les étapes sont appelées *sélecteurs*. On peut employer des sélecteurs spéciaux comme :

- `text()` sélectionne tous les nœuds texte sous l'élément courant, y compris tous ses descendants.
- `node()` sélectionne tous les nœuds enfants de l'élément.

Exemple :

- `/messages/message/contenu/text()`

Axes

XPath permet de rajouter encore une « décoration » sur chaque étape, la *direction* dans laquelle aller à partir de l'étape courante. Cette direction est appelée *axe*. Cela donne la syntaxe :

```
/racine/axe1::element1[filtre1]/axe2::element2[filtre2]/...
```

Par défaut, on descend toujours vers les enfants du nœud courant au niveau de chaque étape. Cet axe s'appelle `child`.

Exemple, ces deux syntaxes signifient la même chose :

- `/messages/message/@numero`
- `/messages/child::message/attribute::numero`

D'autres axes existent. Pour les comprendre, il faut étudier l'algorithme de XPath.

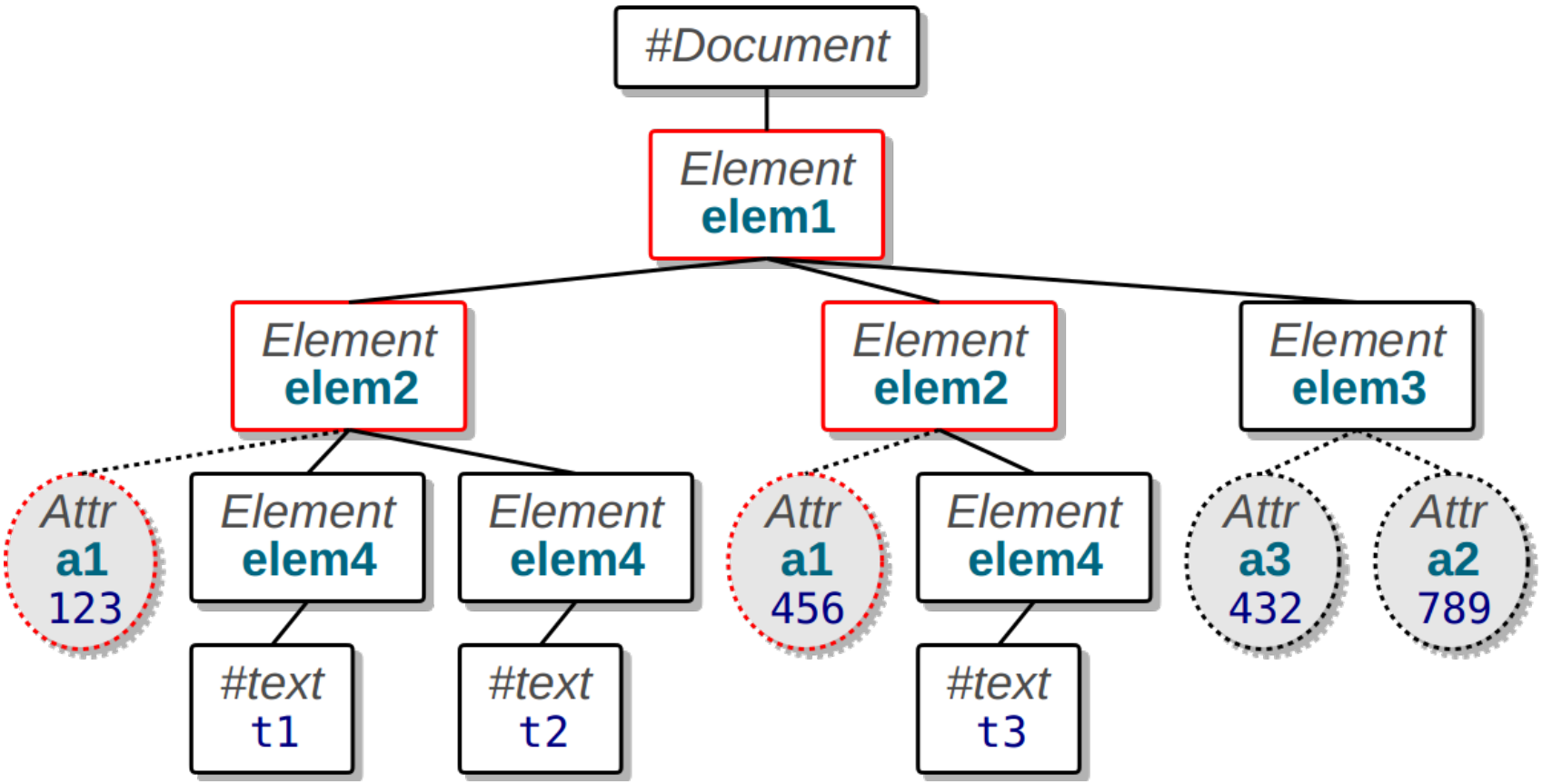
Algorithme de XPath

`XPath(nœud parent, chemin)` est un algorithme récursif. Il sélectionne une liste de nœuds par un chemin partant d'un nœud parent qu'on appelle *contexte d'évaluation*. Au départ, le contexte c'est le document entier.

1. Si le chemin est vide, alors ajouter le nœud parent dans la réponse à la requête XPath globale
2. Sinon extraire la première étape du chemin : `axe::nom[prédicat]`
3. Passer en revue tous les nœuds (éléments ou attributs) du nœud parent définis par l'*axe*. Par exemple pour l'axe `child::` ce sont tous les nœuds enfants, pour l'axe `attribute::` ce sont les nœuds attributs
4. Si le nœud correspond à l'étape (nom et prédicat), alors faire un appel récursif à XPath(nœud, reste du chemin)

Exemple

Soit un document XML représentant un arbre de nœuds. Par exemple, celui-ci pour évaluer `"/elem1/elem2/@a1"` :



Pierre Nerzic

Exemple (suite)

On part du nœud parent égal au document entier, indiqué par le / initial : `XPath(#Document, "elem1/elem2/@a1")`.

1. On commence par prendre l'étape `"elem1"` puis on fait une boucle sur tous les enfants du document. Il n'y en a qu'un, c'est `<elem1>` qui correspond à cette étape. Donc on fait un appel récursif `XPath(<elem1>, "elem2/@a1")`.
2. On prend l'étape `"elem2"` puis on passe les enfants du nœud `<elem1>` en revue : il y a `<elem2 a1="123">`, `<elem2 a1="456">` et `<elem3 ...>`. Les deux premiers correspondent à l'étape, donc, pour chacun d'eux, on fait un appel récursif :
   1. `XPath(<elem2 a1="123">, "@a1")`
   2. `XPath(<elem2 a1="456">, "@a1")`

Exemple (suite et fin)

Pour chacun des deux appels récursifs, il se passe la même chose :

3. On prend l'étape restante, `"@a1"`. Comme elle désigne un attribut, c'est une boucle sur les attributs du nœud parent qui est faite. L'attribut `a1` est présent et donc cela conduit à chaque fois à un nouvel appel récursif :
   1. `XPath(#Attr<a1="123">, "")`
4. Le chemin est vide donc on rajoute le nœud attribut `a1` dans la réponse finale.

Au final, la réponse contient les deux nœuds attribut `a1` du document.

Ce qu'il faut comprendre, c'est l'importance des boucles de parcours des nœuds et l'appel récursif qui en résulte quand l'étape correspond au nœud.

Axes

L'axe définit quels sont les nœuds explorés à chaque étape. Voici quelques axes utiles à connaître parmi ceux qui existent :

`child::` parcourir les nœuds enfants du contexte ; c'est l'axe utilisé par défaut.

`descendant::` parcourir tous les nœuds enfants et petit-enfants ; ça revient un peu à utiliser `//`.

`parent::` parcourir le nœud parent du contexte ; ça revient à utiliser `..` mais avec un test sur le parent voulu.

`ancestor::` parcourir tous les nœuds parent et grand-parents.

`preceding-sibling::` parcourir tous les nœuds frères précédents

`following-sibling::` parcourir tous les nœuds frères suivants

`attribute::` parcourir les nœuds attributs du contexte ; c'est l'axe par défaut pour une étape commençant par un `@`.

Exemples de chemins avec axes

- `/messages/message[last()]/child::contenu` retourne le contenu du dernier message du document.
- `/messages/message[@numero=7]/descendant::dest` sélectionne tous les nœuds situés sous le message n°7.
- `//message[@numero=5]/preceding-sibling::message` sélectionne les messages situés avant le n°5.
- `//dest[@bcc="oui"]/parent::node()` sélectionne le nœud parent d'un élément `<dest>` dont l'attribut `bcc` vaut oui.
- `//message[3]/attribute::numero` retourne l'attribut numéro du 3e message présent dans le document.