XML - Semaine 2

Pierre Nerzic

février-mars 2022

La validation permet de vérifier la structure et le contenu d'un document XML avant de commencer à le traiter : les bons éléments avec les bons attributs présents aux bons endroits.

Nous allons voir deux techniques :

- La plus simple basée sur les *Document Type Definitions* (DTD) venant de la norme SGML,
- La plus complète basée sur des *XML Schema*.

Une autre norme, RelaxNG sera présentée la semaine prochaine.

Validité d'un document

Pierre Nerzic

Introduction

Il y a deux niveaux de correction pour un document XML :

- Un document XML **bien formé** (*well formed*) respecte les règles syntaxiques d'écriture XML : écriture des balises, imbrication des éléments, entités, etc. C'est la plus facile des vérifications.
- Un document **valide** respecte des règles supplémentaires sur les noms, attributs et organisation des éléments.

La validation est cruciale pour une entreprise, p.ex. une banque, qui gère des transactions représentées en XML. S'il y a des erreurs dans les documents, cela peut compromettre l'entreprise. Il vaut mieux être capable de refuser un document invalide plutôt qu'essayer de le traiter et pâtir des erreurs qu'il contient.

Processus de validation

D'abord, il faut disposer d'un fichier contenant des règles. Il existe plusieurs langages pour faire cela : DTD, XML Schemas, RelaxNG et Schematron. Ces langages modélisent des règles de validité plus ou moins précises et d'une manière plus ou moins lisible.

Chaque document XML est comparé à ce fichier de règles, à l'aide d'un outil de validation : `xmlstarlet`, `xmllint`, `rnv`...

La validation indique :

- soit le document est valide, conforme aux règles,
- soit il contient des erreurs comme : tel attribut de tel élément contient une valeur interdite par telle contrainte, il manque tel sous-élément ou attribut dans tel élément, etc.

On va commencer avec les DTD qui sont les plus simples.

Document Type Definitions (DTD)

Pierre Nerzic

Présentation

Un *Document Type Definitions* est une liste de règles définies au début d'un document XML pour permettre sa validation pendant sa lecture. Elle est déclarée par un élément spécial `DOCTYPE` juste après le prologue et avant la racine :

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE itineraire ... >
<itineraire nom="essai">
    <etape distance="0km">départ</etape>
    <etape distance="1km">tourner à droite</etape>
</itineraire>
```

NB: les DTD sont issues de la norme SGML et n'ont pas la syntaxe XML.

Intégration d'une DTD

Une DTD peut être :

- interne, intégrée au document. C'est signalé par un couple `[ ]` :

```
<!DOCTYPE itineraire [
    ...
]>
```

- **externe**, dans un autre fichier, signalé par `SYSTEM` suivi de l'URL du fichier :

```
<!DOCTYPE itineraire SYSTEM "itineraire.dtd">
```

- mixte, il y a à la fois un fichier et des définitions locales :

```
<!DOCTYPE itineraire SYSTEM "itineraire.dtd" [
    ...
]>
```

Le plus souvent, c'est une DTD externe car on utilise la même pour tous les documents à traiter.

Standalone ?

Vous trouverez parfois un attribut `standalone` valant `yes` ou `no` dans les prologues XML. Il est optionnel et présent uniquement quand il y a une DTD interne.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
```

Par défaut il vaut `no`, donc on ne le met que lorsqu'il vaut `yes`.

- `"no"` : la DTD interne n'est pas seulement présente pour valider mais aussi pour fournir les valeurs par défaut et les entités et il peut y avoir des définitions externes (`SYSTEM`).
- `"yes"` : la DTD interne ne sert que pour la validation et ne peut pas employer d'entités ou de règles externes. Le document XML et avec sa DTD interne est totalement indépendant, complet en lui-même.

Outils de validation d'un document avec DTD

Deux commandes Unix permettent de valider un document : `xmlstarlet` et `xmllint`.

- Pour vérifier la syntaxe d'un document XML :
  - `xmlstarlet val --well-formed -e document.xml`
  - `xmllint --noout document.xml`
- Pour valider un document par rapport à une DTD interne :
  - `xmlstarlet val --embed -e document.xml`
  - `xmllint --valid --noout document.xml`
- Pour valider un document par rapport à une DTD externe :
  - `xmlstarlet val --dtd document.dtd -e document.xml`
  - `xmllint --dtdvalid document.dtd --noout document.xml`

Les éditeurs spécialisés XML intègrent la validation.

Contenu d'une DTD

Une DTD contient des règles comme celles-ci :

```
<!ELEMENT itineraire (etape+)>
<!ATTLIST itineraire nom CDATA #IMPLIED>

<!ELEMENT etape (#PCDATA)>
<!ATTLIST etape distance CDATA #REQUIRED>
```

Ce sont des règles qui définissent :

- des éléments (`ELEMENT`) : leur nom et le contenu autorisé,
- des attributs (`ATTLIST`) : leur nom et options.

Voici maintenant les explications détaillées.

Racine du document

Le nom présent après le mot-clé DOCTYPE indique la racine du document. C'est un élément qui est défini dans la DTD.

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<!DOCTYPE itineraire [
    <!ELEMENT itineraire (etape+)>
    <!ATTLIST itineraire nom CDATA #IMPLIED>
    <!ELEMENT etape (#PCDATA)>
    <!ATTLIST etape distance CDATA #REQUIRED>
]>
<itineraire nom="essai">
    <etape distance="0km">départ</etape>
    <etape distance="1km">tourner à droite</etape>
</itineraire>
```

`<!DOCTYPE itineraire ...` définit la racine `<itineraire>`

Définition d'un élément

La règle `<!ELEMENT nom contenu>` permet de définir un élément : son nom et ce qu'il peut y avoir entre ses balises ouvrante et fermante. La définition du *contenu* peut prendre différentes formes :

- `EMPTY` : signifie que l'élément doit être vide,
- `ANY` : signifie que l'élément peut contenir n'importe quels éléments (définis dans la DTD) et textes (leur ordre d'apparition et leur nombre ne seront pas testés),
- `(#PCDATA)` : signifie que l'élément ne contient que du texte,
- (*définitions de sous-éléments*) : spécifie les sous-éléments qui peuvent être employés, voir plus loin.

Les DTD ne sont pas strictes sur les types des données.

Exemple de contenus

Voici un exemple d'éléments simples :

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<!DOCTYPE itineraire [
    <!ELEMENT itineraire ANY>
    <!ELEMENT boucle EMPTY>
    <!ELEMENT etape (#PCDATA)>
]>
<itineraire>
  <boucle/>
  <etape>départ</etape>
  <etape>tourner à droite</etape>
  <boucle></boucle>
  <itineraire/> <!-- c'est possible avec ANY -->
  <etape>tourner à gauche</etape>
</itineraire>
```

Définition de sous-éléments

On arrive à la définition de sous-éléments. C'est une liste ordonnée dans laquelle chaque sous-élément peut être suivi d'un joker parmi * + ? identiques à ceux de egrep pour indiquer des répétitions.

```
<!ELEMENT itineraire (boucle?, etape+, variante*)>
```

Se lit ainsi : l'élément <boucle> est en option, il doit être suivi d'un ou plusieurs <etape> puis d'aucun, un ou plusieurs éléments <variante>

La liste peut contenir des alternatives exclusives notées (*contenu1* | *contenu2* | ...) comme avec egrep :

```
<!ELEMENT informations (topoguide | carte)>
```

signifie que l'élément <information> peut contenir soit un enfant <topoguide>, soit un enfant <carte>.

Contenus alternatifs

On peut grouper plusieurs séquences avec des parenthèses pour spécifier ce qu'on désire :

```
<!ELEMENT personne (titre?, (nom,prenom+) | (prenom+,nom))>
```

Pour finir sur les contenus, on peut aussi indiquer qu'ils peuvent contenir du texte ou des sous-éléments :

```
<!ELEMENT etape (#PCDATA | waypoint)* >
```

Cela permet de valider ces éléments :

```
<etape>avancer tout droit</etape>
<etape><waypoint lon="3.1" lat="48.2"/></etape>
<etape><waypoint lon="3.2" lat="48.1"/>aller au phare</etape>
```

Définition des attributs

Om emploie la règle `<!ATTLIST elem attr type valeur ...>` : le nom de l'élément concerné, le nom de l'attribut, son type (voir plus loin) et sa valeur. Le mot clé `#REQUIRED` en tant que valeur indique que l'attribut est obligatoire, `#IMPLIED` qu'il est optionnel, et si on fournit une chaîne `"..."`, c'est la valeur par défaut.

```
<!ELEMENT waypoint EMPTY>
<!ATTLIST waypoint
    lon       CDATA  #REQUIRED
    lat       CDATA  #REQUIRED
    ele       CDATA  #IMPLIED
    precision CDATA  "50m"
    source (gps|utilisateur|carte) "carte">
```

NB: le type de l'attribut n'est pas représenté finement par les DTD.

Types d'attributs

Il y a plusieurs types possibles, voici les plus utiles (voir cette page) :

`CDATA` l'attribut peut prendre n'importe quelle valeur texte. Il n'y a malheureusement pas de vérification sur ce qui est fourni. Ne pas confondre avec `#PCDATA`.

`(mot1|mot2|...)` Cela force l'attribut à avoir l'une des valeurs de l'énumération.

`ID` l'attribut est un identifiant XML, sa valeur doit être une chaîne (pas un nombre) unique parmi tous les autres attributs de type `ID` du document.

`IDREF` l'attribut doit être égal, dans le document XML, à l'identifiant d'un autre élément.

Définition d'entités (rappel et précisions)

Les entités sont des symboles qui représentent des morceaux d'arbre XML ou des textes.

```
    ...
    <!ENTITY copyright "© IUT Lannion 2022">
    <!ENTITY depart "<etape>Point de départ</etape>">
    <!ENTITY equipement SYSTEM "equipement.xml">
]>
<itineraire>
    <auteur>&copyright;</auteur>
    &equipement;
    &depart;
</itineraire>
```

Pour valider et visualiser un document avec ses entités remplacées :

```
xmllint --valid --noent document.xml
```

XML Schemas

Pierre Nerzic

Présentation

Les Schémas XML sont une norme W3C pour spécifier le contenu d'un document XML. Ils sont écrits en XML et permettent d'indiquer les conditions de validité beaucoup plus finement.

Voici un exemple de schéma pour se faire une idée : 

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="reference" type="ElemReference" />
  <xsd:complexType name="ElemReference">
    <xsd:sequence>
      <xsd:element name="titre" type="xsd:string" />
      <xsd:element name="auteur" type="xsd:string" />
      <xsd:element name="ISBN" type="xsd:string" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```


Pierre Nerzic

Association entre un document et un schéma local

Pour attribuer un schéma de validation local à un document XML, on peut ajouter un attribut situé dans un *namespace* spécifique

```
<?xml version="1.0"?>
<reference
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="reference.xsd">
  <titre>Comprendre XSLT</titre>
  <auteur>Bernd Amann et Philippe Rigaux</auteur>
  <ISBN>2-84177-148-2</ISBN>
</reference>
```

On valide le document par une de ces commandes Unix :

- `xmllint --schema schema.xsd --noout document.xml`
- `xmlstarlet val --xsd schema.xsd -e document.xml`

Association entre un document et un schéma public

Lorsque le schéma est public, mis sur un serveur, c'est un peu différent car il faut définir un *namespace* et l'URL d'accès :

```
<?xml version="1.0"?>
<reference
    xmlns="http://www.iut-lannion.fr"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.iut-lannion.fr reference.xsd"
  <titre>Comprendre XSLT</titre>
  <auteur>Bernd Amann et Philippe Rigaux</auteur>
  <ISBN>2-84177-148-2</ISBN>
</reference>
```

Et il faut ajouter les attributs `xmlns` et `targetNamespace` valant le même *namespace* à la racine du schéma, voir cet exemple.

Principes généraux des Schémas XML

Comme une DTD, un schéma permet de définir des éléments, leurs attributs et leurs contenus. Mais il y a une notion de typage beaucoup plus forte qu'avec une DTD. Avec un schéma, il faut définir les types de données très précisément :

- la nature des données : chaîne, nombre, date, etc.
- les contraintes qui portent dessus : domaine de définition, expression régulière, etc.

Avec ces types, on définit les éléments :

- noms et types des attributs
- sous-éléments possibles avec leurs répétitions, les alternatives, etc.

C'est tout cela qui complique beaucoup la lecture d'un schéma.

Structure générale d'un schéma

Un schéma est contenu dans un arbre XML de racine <xsd:schema>. Le contenu du schéma définit les éléments qu'on peut trouver dans le document. Voici un squelette de schéma :

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="itineraire" type="ElemItineraire" />
    ... définition du type ElemItineraire ...
</xsd:schema>
```

Il valide le document partiel suivant :

```
<?xml version="1.0"?>
<itineraire>
    ...
</itineraire>
```


Pierre Nerzic

Remarque importante

On peut aussi écrire le schéma précédent ainsi :

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="itineraire">
        ... définition de l'élément ...
    </xsd:element>
</xsd:schema>
```

Cette écriture ne dissocie pas les définitions de l'élément et du type. On met la définition en tant que contenu de l'élément.

C'est acceptable pour de tout petits documents, mais ça peut vite devenir illisible.

Définition d'éléments

Un élément <*nom*>*contenu*</*nom*> du document est défini par un élément `<xsd:element name="`*nom*`" type="`*TypeContenu*`">` dans le schéma.

Dans l'exemple suivant, le type est `xsd:string`, c'est du texte quelconque (équivalent à `#PCDATA` dans une DTD) :

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="message" type="xsd:string"/>
</xsd:schema>
```

Ce schéma valide le document suivant :

```
<?xml version="1.0"?>
<message>Tout va bien !</message>
```


Pierre Nerzic

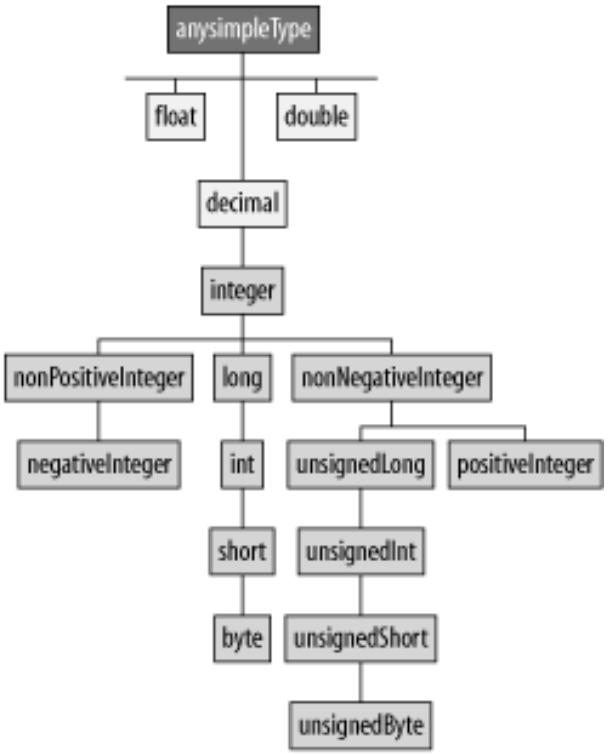
Définition de types de données

L'exemple précédent indique que l'élément `<message>` doit avoir un contenu de type `xsd:string`, c'est à dire du texte. Ce type est un « type simple ». Il y a de nombreux types simples prédéfinis, dont :

- chaîne :
  - `xsd:string` est le type le plus général
  - `xsd:token` vérifie que c'est une chaîne nettoyée des sauts de lignes et espaces d'indentation
- date et heure :
  - `xsd:date` correspond à une chaîne au format `AAAA-MM-JJ`
  - `xsd:time` correspond à `HH:MM:SS.s`
  - `xsd:datetime` valide `AAAA-MM-JJTHH:MM:SS`, on doit mettre un `T` entre la date et l'heure.

Types de données (suite)

- nombres :
  - `xsd:float`, `xsd:decimal` valident des nombres réels
  - `xsd:integer` valide des entiers
  - il y a de nombreuses variantes comme `xsd:nonNegativeInteger`, `xsd:positiveInteger`...



Pierre Nerzic

Types de données (suite)

- autres :
  - `xsd:ID` pour une chaîne identifiante, `xsd:IDREF` pour une référence à une telle chaîne
  - `xsd:boolean` permet de n'accepter que `true`, `false`, `1` et `0` comme valeurs dans le document.
  - `xsd:base64Binary` et `xsd:hexBinary` pour des données binaires.
  - `xsd:anyURI` pour valider des URI (URL ou URN).

Restrictions sur les types

Lorsque les types ne sont pas suffisamment contraints et risquent de laisser passer des données fausses, on peut rajouter des contraintes. Elles sont appelées *facettes* (*facets*).

Dans ce cas, on doit définir un type `simpleType` et lui ajouter des restrictions. Voici un exemple :

```
<xsd:element name="temperature" type="TypeTemperature" />

<xsd:simpleType name="TypeTemperature">
  <xsd:restriction base="xsd:decimal">
    <xsd:minInclusive value="-30"/>
    <xsd:maxInclusive value="+40.0"/>
  </xsd:restriction>
</xsd:simpleType>
```

Définition de restrictions

La structure d'une restriction est :

```
<xsd:restriction base="type de base">
    <xsd:CONTRAINTE value="PARAMETRE"/>
    ...
</xsd:restriction>
```

```
<xsd:simpleType name="TypeNumeroSecu">
  <xsd:restriction base="xsd:string">
    <xsd:whiteSpace value="collapse"/>
    <xsd:pattern value="[12][0-9]{12}( [0-9]{2})?"/>
  </xsd:restriction>
</xsd:simpleType>
```

Les contraintes qu'on peut mettre dépendent du type de données. Il y a une hiérarchie entre les types qui fait que par exemple le type nombre hérite des restrictions possibles sur les chaînes.

Restriction communes à tous les types

Ces restrictions (*facettes*) sont communes à tous les types :

- longueur de la donnée : `xsd:length`, `xsd:maxLength`, `xsd:minLength`. Ces contraintes vérifient que le texte présent dans le document a la bonne longueur.
- expression régulière étendue (egrep) : `xsd:pattern`. C'est une contrainte très utile pour vérifier toutes sortes de données :

```
<xsd:simpleType name="TypeTemperature">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[-+]?[1-9][0-9]?°C"/>
    <xsd:whiteSpace value="collapse"/>
  </xsd:restriction>
</xsd:simpleType>
```

Restrictions communes (suite)

- `xsd:whiteSpace` indique ce qu'on doit faire avec les caractères espaces, tabulation et retour à la ligne éventuellement présents dans les données à vérifier :
  - `value="preserve"` : on les garde tels quels
  - `value="replace"` : on les remplace par des espaces
  - `value="collapse"` : on les supprime tous (à utiliser avec un motif).
- énumération de valeurs possibles : 

```
<xsd:simpleType name="TypeFreinsVélo">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="disque"/>
    <xsd:enumeration value="patins"/>
    <xsd:enumeration value="rétropédalage"/>
  </xsd:restriction>
</xsd:simpleType>
```

Restrictions sur les dates et nombres

Les dates et nombres possèdent quelques contraintes sur la valeur exprimée :

- bornes inférieure et supérieure :
  - `xsd:minExclusive` et `xsd:minInclusive`
  - `xsd:maxExclusive` et `xsd:maxInclusive`
- nombre de chiffres :
  - `xsd:totalDigits` : vérifie le nombre de chiffres total (partie entière et fractionnaire, sans compter le point décimal)
  - `xsd:fractionDigits` : vérifie le nombre de chiffres dans la partie fractionnaire.

Types à alternatives

Comment valider une donnée qui pourrait être de plusieurs types possibles, par exemple, valider les deux premiers éléments et refuser le troisième :

```
<couleur>Chartreuse</couleur>
<couleur>#7FFF00</couleur>
<couleur>02 96 46 93 00</couleur>
```

Si on déclare l'élément <couleur> comme acceptant n'importe quelle valeur chaîne, on ne pourra rien vérifier :

```
<xsd:element name="couleur" type="xsd:string"/>
```

Types à alternatives (suite)

Alors on crée un « type à alternatives » qui est équivalent à plusieurs possibilités, par exemple `xsd:token` ou `xsd:hexBinary`. Attention, ce n'est pas comme déclarer une énumération de *valeurs possibles*. Ici, on parle de *types possibles*.

Pour exprimer qu'un type peut correspondre à plusieurs autres types, il faut le définir en tant que `<xsd:union>` et mettre les différents types possibles dans l'attribut `memberTypes` :

```
<xsd:simpleType name="TYPE_ALTERNATIF">
    <xsd:union memberTypes="TYPE1 TYPE2 ..."/>
</xsd:simpleType>
```

Les types possibles sont séparés par un espace.

Exemple de type à alternatives

Voici un exemple pour les couleurs :

```
<xsd:simpleType name="TypeCouleurs">
  <xsd:union memberTypes="TypeCouleursNom TypeCouleursHex"/>
</xsd:simpleType>

<xsd:simpleType name="TypeCouleursNom">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[A-Z][a-z]+"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="TypeCouleursHex">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="#[0-9A-F]{6}"/>
  </xsd:restriction>
</xsd:simpleType>
```


Pierre Nerzic

Données de type liste

Comment contraindre un élément à contenir des données sous forme de liste (séparées par des espaces), par exemple 

```
<departements>22 29 35 44 56</departements>
```

C'est facile à l'aide d'un « type liste » basé sur un type simple, ici `xsd:integer`. C'est une construction en deux temps :

- il faut le type de base. Dans l'exemple, c'est un type qui définit ce qu'est un numéro de département correct, une restriction d'entier positif à deux chiffres.
- on l'emploie dans une définition de type `<xsd:list>` :

```
<xsd:simpleType name="TYPE_LISTE">
    <xsd:list itemType="TYPE_BASE"/>
</xsd:simpleType>
```

Exemple de liste

Voici la feuille XSD complète :

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="departements" type="TypeLstDepartements"/>

  <xsd:simpleType name="TypeLstDepartements">
    <xsd:list itemType="TypeDepartement"/>
  </xsd:simpleType>

  <xsd:simpleType name="TypeDepartement">
    <xsd:restriction base="xsd:positiveInteger">
      <xsd:totalDigits value="2"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```


Pierre Nerzic

Contenu d'éléments

On revient maintenant sur les éléments. Nous avons vu comment définir un élément contenant un texte, un nombre, etc. :

```
<xsd:element name="NOM" type="TYPE"/>
```

Ça définit une balise <NOM> pouvant contenir des données du type indiqué par TYPE :

```
<?xml version="1.0"?>
<NOM>données correspondant à TYPE</NOM>
```

Comment définir un élément dont le contenu peut être d'autres éléments, ainsi que des attributs ? En fait, c'est la même chose que les xsd:simpleType, sauf que le type est « complexe ». Un type complexe peut contenir des sous-éléments et des attributs.

Exemple de type complexe

Pour modéliser un élément <personne> ayant deux éléments enfants <prénom> et <nom>, il suffit d'écrire ceci :

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="personne" type="ElemPersonne"/>
    <xsd:complexType name="ElemPersonne">
        <xsd:all>
            <xsd:element name="prénom" type="xsd:string"/>
            <xsd:element name="nom" type="xsd:string"/>
        </xsd:all>
    </xsd:complexType>
</xsd:schema>
```

La structure <xsd:all> contient une liste d'éléments qui doivent se trouver dans le document à valider. Il y a d'autres structures.

Contenu d'un type complexe

Un <xsd:complexType> peut contenir trois sortes d'enfants :

```
<xsd:complexType name="ElemPersonne">
    <xsd:sequence> ou <xsd:choice> ou <xsd:all>...
</xsd:complexType>
```

Les enfants peuvent être :

- <xsd:sequence>éléments...</xsd:sequence> : ces éléments doivent arriver dans cet ordre
- <xsd:choice>éléments...</xsd:choice> : le document à valider doit contenir l'un des éléments
- <xsd:all>éléments...</xsd:all> : le document à valider doit contenir ces éléments, pas plus d'une fois chacun, et dans n'importe quel ordre.

Exemple de séquence

Dans cet exemple, les éléments <personne>, <numero>, <rue>, <cpostal> et <ville> doivent se suivre dans cet ordre :

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="adresse" type="ElemAdresse"/>
    <xsd:complexType name="ElemAdresse">
        <xsd:sequence>
            <xsd:element name="personne" type="ElemPersonne"/>
            <xsd:element name="numero" type="xsd:integer"/>
            <xsd:element name="rue" type="xsd:string"/>
            <xsd:element name="cpostal" type="xsd:integer"/>
            <xsd:element name="ville" type="xsd:string"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```


Pierre Nerzic

Exemple de choix

Pour représenter une limite temporelle, par exemple la date de fin d'une garantie, soit on mettra un élément <date_fin> soit un élément <durée> :

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="limite" type="ElemLimiteTemps"/>
  <xsd:complexType name="ElemLimiteTemps">
    <xsd:choice>
      <xsd:element name="date_fin" type="xsd:date"/>
      <xsd:element name="durée" type="xsd:positiveInteger"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:schema>
```


Pierre Nerzic

Exemple de all

Ici les éléments <nom>, <prenom> peuvent être dans n'importe quel ordre, <nom> doit impérativement y être, mais pas <prenom> :

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="personne" type="ElemPersonne"/>
    <xsd:complexType name="ElemPersonne">
        <xsd:all>
            <xsd:element name="prénom" type="xsd:string"
                    xsd:minOccurs="0"/>
            <xsd:element name="nom" type="xsd:string"
                    minOccurs="1"/>
        </xsd:all>
    </xsd:complexType>
</xsd:schema>
```

L'attribut `minOccurs` donne le nombre minimal d'occurrences.

Imbrication de structures

On peut imbriquer plusieurs structures pour définir des éléments à suivre et en option :

```
<xsd:complexType name="ElemPersonne">
    <xsd:sequence>
        <xsd:element name="prénom" type="xsd:string"/>
        <xsd:element name="nom" type="xsd:string"/>
        <xsd:choice>
            <xsd:element name="age" type="xsd:string"/>
            <xsd:element name="date_naiss" type="xsd:date"/>
        </xsd:choice>
    </xsd:sequence>
</xsd:complexType>
```

Par contre, on ne peut pas faire de mélange avec `<xsd:all>`.

Nombre de répétitions

Dans le cas de la structure `<xsd:sequence>`, il est possible de spécifier un nombre de répétition pour chaque sous-élément.

```
<xsd:complexType name="ElemPersonne">
    <xsd:sequence>
        <xsd:element name="prénom" type="xsd:string"
                minOccurs="1" maxOccurs="2"/>
        <xsd:element name="nom" type="xsd:string"
                minOccurs="0" maxOccurs="1"/>
    </xsd:sequence>
</xsd:complexType>
```

Par défaut, les nombres de répétitions min et max sont 1. Pour enlever une limite sur le nombre maximal, il faut écrire `maxOccurs="unbounded"`.

Définition d'attributs

Les attributs se déclarent dans un `<xsd:complexType>` :

```
<xsd:complexType name="ElemPersonne">
    ...
    <xsd:attribute name="NOM" type="TYPE" [OPTIONS]/>
</xsd:complexType>
```

nom	le nom de l'attribut
type	le type de l'attribut, ex: `xsd:string` pour un attribut quelconque
options	mettre `use="required"` si l'attribut est obligatoire, mettre `default="valeur"` s'il y a une valeur par défaut.

Cas spéciaux

Plusieurs situations sont assez particulières et peuvent sembler très compliquées :

- éléments vides sans ou avec attributs
- éléments textes sans ou avec attributs
- éléments avec enfants sans ou avec attributs
- éléments avec textes et enfants sans ou avec attributs

Voici comment elles sont modélisées en XML Schémas.

Pierre Nerzic

Élément vide sans attribut

C'est le cas le plus simple :

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:element name="test" type="ElemTest"/>

    <xsd:complexType name="ElemTest"/>

</xsd:schema>
```

Appliqué au document entier, ça permet de ne valider que :

```
<?xml version="1.0" encoding="utf-8"?>
<test/>
```


Pierre Nerzic

Élément vide avec attribut

On rajoute un attribut obligatoire :

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="test" type="ElemTest"/>

  <xsd:complexType name="ElemTest">
    <xsd:attribute name="att" type="xsd:string" use="required"/>
  </xsd:complexType>

</xsd:schema>
```

Appliqué au document entier, ça permet de ne valider que :

```
<?xml version="1.0" encoding="utf-8"?>
<test att="ok"/>
```

Élément texte sans attribut

Il suffit de définir l'élément en tant que `simpleType` avec un `<xsd:restriction>` pour définir son contenu :

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="test" type="ElemTest"/>

  <xsd:simpleType name="ElemTest">
    <xsd:restriction base="xsd:integer"/>
  </xsd:simpleType>
</xsd:schema>
```

Appliqué au document entier, ça permet de ne valider que :

```
<?xml version="1.0" encoding="utf-8"?>
<test>123</test>
```

Élément texte avec attribut

On doit faire ainsi :

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="test" type="ElemTest"/>
  <xsd:complexType name="ElemTest">
    <xsd:simpleContent>
      <xsd:extension base="xsd:integer">
        <xsd:attribute name="att" type="xsd:string"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:schema>
```

Appliqué au document entier, ça permet de ne valider que :

```
<?xml version="1.0" encoding="utf-8"?>
<test att="ok">456</test>
```

Éléments enfants sans attribut

C'est comme précédemment, par exemple une séquence : 

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="test" type="ElemTest"/>
    <xsd:complexType name="ElemTest">
        <xsd:sequence>
            <xsd:element name="test1"/>
            <xsd:element name="test2" type="xsd:string"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

Appliqué au document entier, ça permet de ne valider que : 

```
<?xml version="1.0" encoding="utf-8"?>
<test><test1/><test2>texte</test2></test>
```


Pierre Nerzic

Éléments enfants avec attribut

Pour valider des attributs sur l'élément parent : 

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="test" type="ElemTest"/>
    <xsd:complexType name="ElemTest">
        <xsd:sequence>
            <xsd:element name="test1"/>
            <xsd:element name="test2" type="xsd:string"/>
        </xsd:sequence>
        <xsd:attribute name="att" type="xsd:string"/>
    </xsd:complexType>
</xsd:schema>
```

Appliqué au document entier, ça permet de ne valider que : 

```
<?xml version="1.0" encoding="utf-8"?>
<test att="ok"><test1/><test2>texte</test2></test>
```


Pierre Nerzic

Éléments enfants avec texte mélangé

Rajouter l'attribut `mixed="true"` à `<xsd:complexType>` : 

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="test" type="ElemTest"/>
    <xsd:complexType name="ElemTest" mixed="true">
        <xsd:sequence>
            <xsd:element name="test1"/>
            <xsd:element name="test2" type="xsd:string"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

Appliqué au document entier, ça permet de ne valider que : 

```
<?xml version="1.0" encoding="utf-8"?>
<test>texte<test1/>texte<test2>texte2</test2>texte</test>
```

