GESTION DE L'INFORMATION SUR INTERNET
Autour de XML

Bernd Amann et Philippe Rigaux

2 septembre 2003

Table des matières

1 Avant-propos

Ce polycopié propose une introduction au langage XML et à quelques-uns des principaux langages de traitement et de manipulation de documents XML. Il est basé sur le livre *Comprendre XSLT*, de Bernd Amann et Philippe Rigaux, paru aux Éditions O'Reilly en 2001. Comme son nom l'indique, le livre est consacré à XSLT. De nombreuses parties ont donc été supprimées dans le présent polycopié, et d'autres – notamment celles consacrées à DOM et SAX – ont été ajoutées.

Tous les exemples et documentation complémentaires sont récupérables sur le site du cours :

http://www.lri.fr/ rigaux/GII

Vous pouvez y trouver en particulier des explications pour installer quelques-uns des logiciels évoqués, un texte d'introduction à XML et XSLT, et une référence des éléments et fonctions XSLT.

L'enseignement comprend des cours magistraux, des TD/TP et un projet. Les cours sont basés sur le polycopié mais s'effectuent à un rythme assez rapide et ne couvrent pas tous les détails. Une bonne méthode de travail consiste sans doute à suivre les cours, et à compléter/réviser ses connaissances ensuite avec le polycopié.

Les TD s'effectuent sur machine et sont principalement destinés à vous permettre de prendre en main les langages et API XML, et ce dans le cadre des outils proposés. Dès que vous avez effectué assez d'exercices pour vous sentir autonome, vous pouvez vous embarquer dans le projet. Notez que vous êtes encouragés à faire preuve d'autonomie et à prendre des initiatives pour compléter et améliorer les exercices et projets qui vous sont proposés. Attention cependant à rester dans les clous et à ne pas partir dans n'importe quelle direction. En cas de doute : demandez.

2 Bases de données et XML

Plaçons-nous pour l'instant dans la situation où des informations stockées dans une base de données doivent être converties en document(s) XML. Nous effectuons un rappel des principales caractéristiques du modèle relationnel avant de décrire le processus de conversion.

2.1 Quelques rappels sur les BD relationnelles

Prenons l'exemple (très simplifié) de la base de données d'un organisme de voyage. Cet organisme propose des séjours (sportifs, culturels, etc) se déroulant dans des stations de vacances. Chaque station propose un ensemble d'activités (ski, voile, tourisme). Enfin l'organisme souhaite gérer une liste des clients (avec le solde de leur compte !) et des séjours auxquels ils ont participé avec leurs dates de début et de fin.

La première chose à faire est d'établir un *schéma* décrivant comment les informations nécessaires vont être stockées dans la base. La conception d'un schéma relationnel n'entre pas dans le cadre de ce livre. Il suffira de dire qu'il est constitué d'un ensemble de schémas de tables (ou *relation*), chaque table étant décrite par un ensemble d'attributs. Voici une représentation concise du schéma pour notre agence de voyages.

– Station (**nomStation**, capacité, lieu, région, tarif)

id	nom	prénom	ville	région	solde
10	Fogg	Phileas	Londres	Europe	12465
20	Pascal	Blaise	Paris	Europe	6763
30	Kerouac	Jack	New York	Amérique	9812

La table *Client*

idClient	station	début	nbPlaces
10	Passac	2001-07-01	2
30	Santalba	2001-08-14	5
20	Santalba	2001-08-03	4
30	Passac	2001-08-15	3
30	Venusa	2001-08-03	3
20	Venusa	2001-08-03	6
30	Farniente	2002-06-24	5
10	Farniente	2002-09-05	3

La table *Séjour*

FIG. 1 – *Les clients et leurs séjours*

– Activité (***nomStation***, **libellé**, prix)

– Client (**id**, nom, prénom, ville, région, solde)

– Séjour (***idClient***, **station**, **début**, nbPlaces)

On trouve donc quatre tables, chacune ayant un nom distinct. Elles correspondent respectivement à la description des stations, des activités proposées dans chaque station, des clients et des séjours effectués par les clients dans les stations. Pour chaque table on a donné la liste des attributs, sans entrer pour l'instant dans le détail des types de données. Parmi ces attributs certains présentent une importance particulière.

La *clé primaire* est le (ou les) attribut(s) dont la valeur va permettre d'identifier de manière unique une ligne dans la table. Les clés primaires sont représentées **en gras**. Pour le schéma ci-dessus, cela signifie qu'il ne peut pas y avoir deux stations avec le même nom. Le choix des clés relève de choix de conception (souvent assez délicats) que nous ne commenterons pas ici. Pour la table *Client*, on a visiblement considéré que le nom n'était pas suffisant pour identifier une ligne de la table, et on a créé un attribut artificiel `id` pour numéroter les clients au fur et à mesure de leur insertion.

Certaines clés peuvent être constituées de plusieurs attributs. Chaque ligne de la table *Activité* par exemple est identifiée par le nom de la station, et le libellé de l'activité. Il peut donc y avoir plusieurs activités de libellés différents dans une même station, et plusieurs activités de mêmes libellés dans des stations différentes. Enfin la table *Séjour* est identifiée par trois attributs : un client peut revenir plusieurs fois dans la même station, mais pas à la même date.

Une *clé étrangère* est un (ou plusieurs) attribut(s) dans une table A dont la valeur fait référence à une valeur de clé primaire dans une table B. Prenons l'exemple des tables de la figure 1. L'attribut *idClient* dans la table *Séjour* contient une valeur de clé de la table *Client*. De même l'attribut *station* contient une valeur de clé de la table *Station*. Les clés étrangères constituent des références entre tables, avec deux objectifs :

– définir des *contraintes d'intégrité référentielle* : on ne devrait pas par exemple pouvoir référencer dans la table *Séjour* un client qui n'existe pas dans la table *Client* ;

– permettre, par des opérations dites de *jointure*, le regroupement de données réparties dans plusieurs tables : à partir de la première ligne de la table *Séjour*, on sait par exemple que le client s'appelle Phileas Fogg en prenant `idClient` et en recherchant la ligne correspondante dans la table *Client*.

Les clés primaire et étrangère sont les caractéristiques essentielles d'un schéma, et définissent les principales contraintes liant les tables. D'autres contraintes, internes à chaque table, peuvent être exprimées.

nomStation	capacité	lieu	région	tarif
Venusa	350	Guadeloupe	Antilles	1200
Farniente	200	Sicile	Europe	1500
Santalba		Martinique	Antilles	2000
Passac	400	Alpes	Europe	1000

La table *Station*

nomStation	libellé	prix
Venusa	Voile	150
Venusa	Plongée	
Farniente	Plongée	130
Passac	Ski	200
Passac	Piscine	20
Santalba	Kayac	50

La table *Activité*

FIG. 2 – *Les stations et leurs activités*

Prenons par exemple la figure 2. On a :

1. des contraintes de types : certains attributs sont numériques, d'autres alphanumériques, avec ou sans décimale, etc ;
2. des contraintes d'existence : la capacité de la station Santalba, ou le tarif de la plongée à Venusa, sont inconnus, mais tous les autres attributs ont une valeur ;
3. des contraintes d'appartenance à un ensemble énuméré : on pourrait par exemple définir une codification des régions (« Océan Indien », « Antilles », « Europe », « Amérique », « Asie », etc).

Toutes ces contraintes sont exprimables avec le langage SQL (ou plus exactement la partie de la norme SQL ANSI relative à la définition de données). Ainsi on peut déclarer que certains attributs doivent *toujours* être connus (contrainte `NOT NULL`), qu'il sont des instances de types SQL, ou que leur valeur appartient à une liste fixée.

2.2 Transformation d'une base de données en XML

Nous étudions maintenant de manière systématique la transformation de tout ou partie d'une base de données relationnelle en document XML.

Éléments ou attributs ?

La première solution, immédiate, consiste à conserver la structure « plate » de la base relationnelle, et à transcrire chaque table par un élément ayant le nom de la table ou un dérivé (par exemple `<Stations>`). Cet élément contient lui-même un élément pour chaque ligne, ayant pour nom un autre dérivé du nom de la table (par exemple « `Station` », sans « s »), enfin chaque attribut de la table est représenté par un élément, constituant ainsi un troisième niveau dans l'arbre.

Il existe (au moins) une autre possibilité. Au lieu de représenter les attributs de la table par des éléments, on peut les représenter par des attributs XML de l'élément représentant la ligne. Voici ce que cela donnerait pour la table *Station*.

Exemple 2.1 *StationAttrs.xml* : *Représentation de Station avec des attributs*

```
<?xml version='1.0' encoding='ISO-8859-1?>

<Stations>
```

```
<Station nomStation='Venusa'
capacite='350'
lieu='Guadeloupe'
region='Antilles'
tarif='1200.00'
/>
<Station nomStation='Farniente'
capacite='200'
lieu='Seychelles'
region='Océan Indien'
tarif='1500.00'
/>
<Station nomStation='Santalba'
capacite='null'
lieu='Martinique'
region='Antilles'
tarif='1200.00'
/>
<Station nomStation='Passac'
capacite='400'
lieu='Alpes'
region='Europe'
tarif='1000.00'
/>

</Stations>
```

Cette méthode présente quelques avantages. Tout d'abord elle est assez proche, conceptuellement, de la représentation relationnelle. Chaque ligne d'une table devient un élément XML, chaque attribut de la ligne devient un attribut XML de l'élément. La structure est plus fidèlement retranscrite, et notamment le fait qu'une ligne d'une table forme un tout, manipulé solidairement par les langages. En SQL par exemple, on n'accède jamais à un attribut sans être d'abord passé par la ligne de la table.

Techniquement, l'absence d'ordre (significatif) sur les attributs XML correspond à l'absence d'ordre significatif sur les colonnes d'une table. On s'affranchit donc en partie des problèmes potentiels évoqués précédemment. Du point de vue du typage, l'utilisation des attributs permet également d'être plus précis et plus proche de la représentation relationnelle :

- on ne peut pas avoir deux fois le même attribut pour un élément, de même qu'on ne peut pas avoir deux colonnes avec le même nom dans une table (notez que ce n'est pas vrai si on représente les colonnes par des éléments XML) ;
- on peut indiquer, dans une DTD, la liste des valeurs que prend un attribut, ce qui renforce un peu les contrôles sur le document.

Enfin l'utilisation des attributs aboutit à un document moins volumineux. Nous utiliserons donc plutôt cette méthode de conversion par la suite, bien qu'il soit juste de souligner que la représentation des colonnes par des éléments a aussi ses défenseurs. Comme pour beaucoup d'autres problèmes sans solution tranchée, le choix dépend en fait beaucoup de l'application et de l'utilisation qui est faite des informations.

Représentation des associations entre tables

Passons maintenant à la représentation XML des liens entre les tables. En relationnel, nous avons vu que les liens sont définis par une correspondance entre la clé primaire dans une table, et une clé étrangère dans une autre table. L'opération (relationnelle) qui va ensuite s'appuyer sur cette correspondance pour rapprocher les lignes de deux tables est la *jointure*. Dans la figure 2, page 4, on voit par exemple que les tables *Station* et *Activité* sont indépendantes l'une de l'autre (elles peuvent être stockées en des endroits

différents), mais que l'on peut, *par calcul*, reconstituer l'association en prenant, pour chaque ligne décrivant une station, les lignes de *Activité* ayant le nom de la station.

En d'autres termes, la condition nécessaire et suffisante pour qu'il soit possible de reconstituer l'information est l'existence d'un critère de rapprochement. Il est tout à fait possible d'appliquer le même principe en XML. Voici par exemple un document où figurent des éléments de type `Station` et de type `Activite` (nous n'avons gardé que deux stations pour simplifier). Ces éléments sont indépendants les uns des autres (ici cela signifie que des informations apparentées ne sont pas liées dans la structure hiérarchique du document), mais on a conservé le critère de rapprochement.

Exemple 2.2 *StationActivite.xml* : *Stations et activités*

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<Stations>
<Station nomStation='Venusa'
capacite='350'
lieu='Guadeloupe'
region='Antilles'
tarif='1200.00'
/>

<Station nomStation='Farniente'
capacite='200'
lieu='Seychelles'
region='Océan Indien'
tarif='1500.00'
/>

<Activite nomStation='Venusa'
libelle='Voile'
prix='150.00'
/>

<Activite nomStation='Venusa'
libelle='Plongee'
/>

<Activite nomStation='Farniente'
libelle='Plongée'
prix='130.00'
/>

</Stations>
```

Maintenant, comme dans le cas du relationnel, il est possible de déterminer *par calcul*, la correspondance entre une activité et une station. Supposons que l'on exécute un programme XSLT. Si le nœud courant est de type `Station`, il est possible de sélectionner toutes les activités avec l'expression XPath :

```
/Activite[@nomStation=current()/@nomStation]
```

De même, si le nœud courant est de type `Activite`, voici l'expression XPath qui désigne la station :

```
/Station[@nomStation=current()/@nomStation]
```

Cette recherche s'apparente à la jointure de SQL, même si on procède assez différemment ici : il faut d'abord se positionner sur un nœud, puis exprimer avec XPath le chemin qui mène aux nœuds associés.

Cette représentation est possible, mais elle n'est pas naturelle en XML et mène à quelques difficultés. Elle n'est pas naturelle parce qu'une activité est *toujours* dans une et une seule station : il est donc inutile de la représenter séparément. Elle présente des difficultés parce qu'il va être très coûteux de parcourir des chemins compliqués dans l'arbre pour relier des nœuds.

La bonne représentation dans ce cas consiste à *imbriquer* les éléments de type `Activite` dans l'élément de type `Station`. On peut du même coup s'épargner la peine de conserver le nom de la station dans les éléments <Activite> puisque la correspondance Station/Activité est maintenant représentée par la structure, et pas par un lien de navigation basé sur des valeurs communes. Voici ce que cela donne.

Exemple 2.3 *StationImbriquee.xml* : *Représentation avec imbrication*

```
<?xml version='1.0' encoding='ISO-8859-1?>

<Stations>

<Station nomStation='Venusa'
  capacite='350'
  lieu='Guadeloupe'
  region='Antilles'
  tarif='1200.00'>

  <Activite libelle='Voile' prix='150.00'/>
  <Activite libelle='Plongee'/>
</Station>

<Station nomStation='Farniente'
  capacite='200'
  lieu='Seychelles'
  region='Océan Indien'
  tarif='1500.00'>

  <Activite libelle='Plongée' prix='130.00'/>
</Station>

</Stations>
```

Cette représentation est bien meilleure. Il est maintenant possible, pour un nœud courant de type `Station`, d'accéder à toutes les activités avec l'expression XPath `Activite`. Inversement l'expression « `..` » donne la station pour une activité donnée. Ces expressions s'évaluent très efficacement puisque les informations apparentées sont localisées dans la même partie de l'arbre XML. Notons au passage qu'il n'est pas possible d'utiliser d'imbrication dans le relationnel classique (les SGBD « relationnel/objet » récents savent le faire). Tout y est représenté « à plat », ce qui mène à multiplier le nombre des tables, souvent inutilement.

La méthode ci-dessus ne pose pas de problème dans le cas où l'un des éléments (ici <Activite>) est lié à un et un seul autre (ici <Station>). On parle *d'association de un à plusieurs* en modélisation base de données.

En revanche, dans le cas d'associations « plusieurs à plusieurs », l'imbrication ne va pas de soi. Prenons par exemple l'association entre les stations et les clients. Pour une station il peut y avoir plusieurs clients, et réciproquement. Dans le schéma relationnel on a créé une table intermédiaire *Séjour* pour représenter cette association.

Il n'est pas évident de choisir l'ordre d'imbrication des éléments. Tout dépend de l'ordre de navigation qui va être employé. Si on suppose par exemple que les accès se feront par les stations, on peut choisir l'imbrication représentée dans l'exemple suivant (toujours en nous limitant à deux stations) :

Exemple 2.4 *StationSejour.xml* : *Les stations et leurs séjours imbriqués, avec les clients à part*

```
<?xml version='1.0' encoding='ISO-8859-1?>

<Stations>

<Station nomStation='Venusa'
    capacite='350'
```

```
    lieu='Guadeloupe'
    region='Antilles'
    tarif='1200.00'>

  <Sejour idClient='30'
     debut='2001-08-03'
     nbPlaces='3'/>

   <Sejour idClient='20'
     debut='2001-08-03'
     nbPlaces='6'/>
</Station>

<Station nomStation='Farniente'
    capacite='200'
    lieu='Seychelles'
    region='Océan Indien'
    tarif='1500.00'>

   <Sejour idClient='30'
       debut='2002-06-24'
       nbPlaces='5'/>

    <Sejour idClient='10'
       debut='2002-09-05'
       nbPlaces='3'/>
</Station>

<Client id='10'
nom='Fogg'
prenom='Phileas'
ville='Londres'
region='Europe'
solde='12465.00'
/>
<Client id='20'
nom='Pascal'
prenom='Blaise'
ville='Paris'
region='Europe'
solde='6763.00'
/>
<Client id='30'
nom='Kerouac'
prenom='Jack'
ville='New York'
region='Amérique'
solde='9812.00'
/>

</Stations>
```

Un séjour est lié à une seule station, donc on a appliqué la méthode d'imbrication. Il est très facile, si le nœud courant est de type `Station`, d'accéder aux séjours de la station avec l'expression XPath `Sejour`. Le problème se pose maintenant pour les clients qui ont été laissés à part. Si, étant sur un nœud client, on recherche tous les séjours d'un client, l'expression correspondante est :

```
/Station/Sejour[@idClient=current()/id]
```

En introduisant une hiérarchie Station/Séjour, on a donc privilégié un chemin d'accès aux données. Dès que, travaillant sur l'information relative aux stations et à leurs séjours, on voudra obtenir celle relative aux clients, il faudra suivre des liens de navigation et accéder à une partie éloignée de l'arbre, ce qui est difficile à exprimer, et difficile à évaluer pour le processeur.

On peut compléter la hiérarchie en y introduisant les clients, ce qui permet de supprimer le lien de navigation basé sur l'identifiant du client. On obtient alors le document suivant.

Exemple 2.5 *StationSejourClient.xml* : *Stations, séjours et clients*

```
<?xml version='1.0' encoding='ISO-8859-1?>

<Stations>

<Station nomStation='Venusa'
    capacite='350'
    lieu='Guadeloupe'
    region='Antilles'
    tarif='1200.00'>

  <Sejour debut='2001-08-03'
     nbPlaces='3'>
      <Client id='30' nom='Kerouac' prenom='Jack'
         ville='New York' region='Amérique' solde='9812.00'/>
  </Sejour>

   <Sejour debut='2001-08-03'
     nbPlaces='6'>
      <Client id='20' nom='Pascal' prenom='Blaise'
        ville='Paris' region='Europe' solde='6763.00'/>
  </Sejour>
</Station>

<Station nomStation='Farniente'
    capacite='200'
    lieu='Seychelles'
    region='Océan Indien'
    tarif='1500.00'>

   <Sejour  debut='2002-06-24'
       nbPlaces='5'>
      <Client id='30' nom='Kerouac' prenom='Jack'
         ville='New York' region='Amérique' solde='9812.00'/>
  </Sejour>

    <Sejour debut='2002-09-05'
       nbPlaces='3'>
       <Client id='10' nom='Fogg' prenom='Phileas'
         ville='Londres' region='Europe' solde='12465.00'/>
  </Sejour>
</Station>

</Stations>
```

Cette fois, en supposant que le point d'accès est toujours une station, on a toutes les informations situées dans le même sous-arbre, ce qui va permettre d'y accéder efficacement et simplement. On voit en revanche que si on souhaite prendre comme point d'accès un client, les informations utiles sont réparties un peu partout dans l'arbre, et que leur reconstitution sera plus difficile.

Une autre conséquence de la structure hiérarchique propre aux documents XML est qu'il est difficile d'éviter la redondance d'information. Le client Jack Kerouac, avec toutes ses caractéristiques, apparaît

par exemple plusieurs fois dans le document *StationSejourClient.xml*. Cela peut s'avérer un inconvénient important si on souhaite par exemple obtenir une liste des clients sans doublons car les opérations de groupement et d'élimination de doublons sont très difficiles à réaliser en XSLT. Un autre inconvénient est la taille plus importante du document par rapport à celui obtenu en mettant un seul exemplaire de chaque élément, et en les associant par des liens. Finalement, si on considère le cas où le document peut être modifié, la duplication des informations sur un client obligera l'auteur de modifier par exemple le nom du client à plusieurs endroits du document (*anomalie de mise à jour*).

La base de données que nous utilisons dans nos exemples est très simple. Il est clair que pour des bases réalistes présentant quelques dizaines de tables, la conception d'un schéma XML d'exportation doit faire des compromis entre l'imbrication des données et la conservation des correspondances clé primaire/clé étrangère sous forme de lien de navigation dans le document XML. Tout dépend alors des besoins de l'application, de la partie de la base qu'il faut exporter, et des chemins d'accès privilégiés aux informations qui seront utilisés dans l'exploitation du document.

2.3 Création de la DTD

Quand on crée un outil exportant une base de données (ou une partie d'une base de données) dans un document XML, on peut, en plus du document représentant le contenu de la base, créer la DTD. L'intérêt est de pouvoir fournir aux utilisateurs une description de la structure qui ne dépende pas d'une instance particulière de la base. La création d'une DTD permet également de reconstituer en partie les contraintes du schéma relationnel,

Nous continuons à prendre l'exemple de notre base « Agence de voyages », en supposant les choix suivants :

1. les colonnes sont représentées par des attributs XML ;
2. le chemin d'accès principal est la station ;
3. pour chaque station on trouve, imbriqués, les séjours de la station, et dans chaque séjour les clients qui ont séjourné dans la station ;
4. pour les besoins de la présentation, on va supposer que les activités de la station sont représentées par des éléments indépendants, avec un lien de navigation.

Un exemple de document conforme à ce schéma est *StationSejourClient.xml*, page 9, auquel on pourrait ajouter des éléments `<Activite>` comme fils de l'élément racine.

Racine du document

Les documents auront un élément racine de type `Stations`, constitué de 0 ou plusieurs éléments de type `Station` et de 0 ou plusieurs éléments de type `Activite`.

```
<!ELEMENT Stations (Station*,Activite*)>
```

Description de la table *Station*

Rappelons le schéma relationnel de la table *Station*. Nous allons essayer de nous y conformer dans la DTD.

```
CREATE TABLE Station (nomStation VARCHAR (30) NOT NULL,
                      capacite   INTEGER,
                      lieu       VARCHAR (30) NOT NULL,
                      region     VARCHAR (30) NOT NULL,
                      tarif      DECIMAL (10,2),
                      PRIMARY KEY (nomStation),
                      CONSTRAINT nom_region
```

```
                    CHECK (region IN ('Océan Indien', 'Antilles',
                           'Europe', 'Amérique', 'Asie','Afrique'))
                );
```

Chaque colonne de la table devient un attribut de l'élément. Nous ne pouvons pas reproduire le type, donc tous sont déclarés comme `CDATA`. En ce qui concerne la contrainte d'existence `NOT NULL`, elle a une équivalence directe dans la DTD.

1. si la colonne est à `NOT NULL`, l'attribut XML est qualifié avec `#REQUIRED` (attribut obligatoire) ;
2. sinon l'attribut XML est qualifié avec `#IMPLIED` (attribut optionnel).

On choisit donc de ne *pas* représenter un attribut à `NULL`, ce qui est le plus proche équivalent en XML du fait que la valeur est inconnue. La valeur `NULL` correspond vraiment à une absence de valeur, ce qui est très différent d'une valeur à 0 ou de la chaîne vide, cette dernière pouvant être représentée par un élément ou un attribut vide en XML.

Les autres contraintes sur le contenu de la table sont d'une part la clé primaire, d'autre part l'ensemble de valeurs énumérées pour la colonne `region`. Quand la clé primaire est constituée d'une seule colonne, elle correspond à un attribut de type `ID` d'une DTD XML :

```
nomStation ID #REQUIRED
```

Si le parseur est validant (autrement dit s'il vérifie que le document est conforme à la DTD), il contrôlera qu'il n'y a pas deux éléments <`Station`> avec le même attribut `nomStation`. En fait la portée d'une déclaration d'identifiant couvre tous les éléments d'un même document, quel que soit leur type, alors qu'elle est restreinte à une table en relationnel. Si on déclare un `ID` pour un autre élément correspondant à une autre table, il faut envisager qu'il puisse exister des valeurs de clé identiques dans les deux tables (dans ce cas on peut par exemple préfixer les clés par le nom de la table, même si ce n'est pas une garantie totale). Dans le cas – fréquent – où la clé primaire comprend plusieurs colonnes de la table, il n'existe pas d'équivalent dans la notation des DTD.

Pour la valeur énumérée, on peut aussi l'indiquer dans la DTD. Notez que, par chance (!), il n'y a pas de valeur contenant des blancs dans notre liste des régions, ce qui serait possible dans le schéma relationnel mais ne pourrait pas être retranscrit dans la DTD.

Finalement il reste à indiquer qu'un élément <`Station`> peut avoir comme fils 0 ou plusieurs éléments <`Sejour`>. Voici la partie de la DTD correspondant aux lignes de la table *Station* avec les séjours :

```
<!ELEMENT Station (Sejour*)>
<!ATTLIST Station
          nomStation ID #REQUIRED
          capacite   CDATA #IMPLIED
          lieu       CDATA #REQUIRED
          tarif      CDATA #REQUIRED
          region     (Océan_Indien|Antilles|Europe|Amérique|Asie) #REQUIRED
>
```

Tables *Séjour* et *Client*

Voici la commande de création de la table *Séjour*. La clé est constituée de plusieurs attributs, et on ne peut donc pas la représenter avec la DTD.

```
CREATE TABLE Sejour  (idClient INTEGER NOT NULL,
                      station VARCHAR (30) NOT NULL,
                      debut DATE NOT NULL,
                      nbPlaces  INTEGER NOT NULL,
                       PRIMARY KEY (idClient, station, debut),
                       FOREIGN KEY (idClient) REFERENCES Client,
                       FOREIGN KEY (station) REFERENCES Station);
```

Cette table comprend deux clés étrangères : chaque séjour fait référence à une (une seule) station et un (un seul) client. Comme nous avons choisi de représenter par une imbrication les liens entre ces trois entités, la reprise de ces clés étrangères dans la DTD est inutile. Il suffit de dire qu'un élément de type `Sejour` est composé d'un (un seul) élément de type `Client` :

```
<!ELEMENT Sejour (Client)>
<!ATTLIST Sejour
          debut    CDATA #REQUIRED
          nbPlaces CDATA #REQUIRED
>
```

Enfin la table *Client* est retranscrite en reprenant simplement les colonnes de la tables comme attributs dans la DTD. Notez que l'on ne peut plus dire que l'attribut `id` est unique dans le document puisqu'un client sera dupliqué pour chaque séjour auquel il a participé (voir document *StationSejourClient.xml*, page 9). On doit quand même conserver l'attribut `id` dans l'élément, pour pouvoir reconnaître les éléments correspondant au même client.

La table *Activite*

Pour les besoins de la cause, nous plaçons les éléments de type `Activite` indépendamment de la station à laquelle ils se rattachent, en conservant un lien identique à celui de la base relationnelle. Voici la commande de création de cette table.

```
CREATE TABLE Activite (nomStation   VARCHAR (30) NOT NULL,
                       libelle      VARCHAR (30) NOT NULL,
                       prix         DECIMAL (10,2),
                       PRIMARY KEY (nomStation, libelle),
                       FOREIGN KEY (nomStation) REFERENCES Station
                       );
```

On ne peut pas déclarer d'identifiant puisqu'il est composé de deux colonnes. En revanche il est possible de transcrire partiellement la clé étrangère par une référence `IDREF`.

```
<!ELEMENT Activite  EMPTY>
<!ATTLIST Activite
          nomStation IDREF #REQUIRED
          libelle   CDATA #REQUIRED
          prix      CDATA #IMPLIED
>
```

Un processeur validant vérifiera, sur notre exemple, que chaque attribut `nomStation` d'un élément `<Activite>` a une valeur égale à celle de l'attribut correspondant dans un élément `<Station>`. Contrairement au schéma de la table relationnelle, rien n'indique que `nomStation` dans `<Activite>` fait référence à `nomStation` dans `<Station>` puisque les identifiants sont globaux au document.

Voici pour conclure la DTD complète.

Exemple 2.6 *BaseStation.dtd* : *La DTD de la base*

```
<!-- DTD des documents exportés de la base
     "Agence de voyages"   -->

<!ELEMENT Stations (Station*,Activite*)>

<!ELEMENT Station (Sejour*)>
<!ATTLIST Station
          nomStation ID #REQUIRED
          capacite   CDATA #IMPLIED
          lieu       CDATA #REQUIRED
```

```
          tarif      CDATA #REQUIRED
          region     (Océan_Indien|Antilles|Europe|Amérique|Asie) #REQUIRED
>

<!ELEMENT Sejour (Client)>
<!ATTLIST Sejour
          debut    CDATA #REQUIRED
          nbPlaces CDATA #REQUIRED
>

<!ELEMENT Client  EMPTY>
<!ATTLIST Client
          id      ID    #REQUIRED
          nom     CDATA #REQUIRED
          prenom  CDATA #REQUIRED
          ville   CDATA #REQUIRED
          region  CDATA #REQUIRED
          solde  CDATA #REQUIRED
>

<!ELEMENT Activite  EMPTY>
<!ATTLIST Activite
          nomStation IDREF #REQUIRED
          libelle   CDATA #REQUIRED
          prix      CDATA #IMPLIED
>
```

2.4 XSQL

L'utilitaire XSQL fait partie du *XML Development Kit* (XDK) d'Oracle, un ensemble d'outils de traitement de documents XML comprenant les parseurs DOM et SAX, un processeur XSLT, et de nombreux *packages* Java. Contrairement à XSP qui est un langage généraliste destiné à permettre l'inclusion de n'importe quel code Java dans des pages XML, XSQL se concentre sur l'extraction de données d'une base Oracle avec le langage SQL (quelque peu étendu), sur l'inclusion au format XML du résultat de la requête dans un document, suivie enfin d'une transformation XSLT. XSQL est donc de ce point de vue plus limité que XSP, mais s'avère en contrepartie beaucoup plus simple à utiliser.

XSQL fournit la plupart des fonctionnalités que nous étudions depuis le début de ce chapitre et constitue donc un excellent exemple d'une réalisation pratique d'un environnement de publication web intégrant une base de données. On y trouve :

1. le paramétrage du format de sortie XML, semblable à ce qu'offre notre petit outil *ExportXML.java* ;
2. la possibilité de paramétrer les requêtes avec des variables provenant par exemple d'une requête HTTP ;
3. un enchaînement du processus d'intégration des données issues de la base avec une transformation XSLT ;
4. enfin une cloisonnement (presque) complet entre les couches SQL, XML et XSLT.

Nous commençons par quelques exemples simples de documents XSQL avant de décrire les principales options fournies par cet outil. Pour un présentation complète nous vous recommandons le livre *Building Oracle XML Applications* de Steve Muench, aux Éditions O'Reilly.

Exemples de documents XSQL

Voici le plus simple exemple de document XSQL.

Exemple 2.7 *XSQL.xsql* : *Un exemple de document XSQL*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsql:query
           connection="connexionVisiteur"
           xmlns:xsql="urn:oracle-xsql">
     SELECT * FROM Station WHERE nomStation='Passac'
</xsql:query>
```

On retrouve les principes déjà étudiés pour XSP (et, plus globalement, pour des langages de production de HTML dynamique comme les JSP). Ce document est un document XML, certaines balises étant associées à un espace de nom particulier (ici `xsql`). Le processeur qui traite ce document reconnaît ces balises comme des instructions, déclenche un traitement et crée un nouveau document dans lequel les balises « dynamiques » sont remplacées par le résultat de ce traitement. L'élément racine (ici `<xsql:query>`) doit contenir les deux attributs suivants :

- `connection` qui indique le nom de la connexion à utiliser pour accéder à Oracle ; ce nom de connexion est associé, dans un fichier de paramétrage, au compte utilisateur ainsi qu'à la base à utiliser ;
- la déclaration de l'espace de noms `xsql`.

`<xsql:query>` est le principal élément de XSQL. Son contenu est une requête SQL (ici l'interrogation de la table *Station*), et il peut prendre un ensemble d'attributs (voir plus loin) pour paramétrer notamment la manière dont le résultat est mis en forme XML. En l'absence d'indication sur la mise en forme souhaitée, XSQL produit un document XML de la forme suivante :

Exemple 2.8 *Station.xsql* : *Le document XSQL résultat de la requête sur Passac*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<ROWSET>
 <ROW num="1">
   <NOMSTATION>Passac</NOMSTATION>
   <CAPACITE>400</CAPACITE>
   <LIEU>Alpes</LIEU>
   <REGION>Europe</REGION>
   <TARIF>1000</TARIF>
 </ROW>
</ROWSET>
```

L'élément racine du document est `<ROWSET>` (ensemble de lignes). Chaque ligne de la table est représentée par un élément de type `ROW`, automatiquement identifié par un attribut `num` dont la valeur, par défaut, est simplement un compteur sur les lignes du résultat. Enfin chaque colonne est également représentée par un élément. Le document ci-dessus renvoie le résultat au format XML, mais on peut bien entendu appliquer un programme de transformation XSLT sur le serveur avant la transmission au client.

Ce document peut être traité dans plusieurs des architectures présentées précédemment : avec un programme lancé depuis la ligne de commande, avec une *servlet*, ou dans une page JSP. Au moment où le document est constitué, le résultat est intégré au document principal.

Pour appliquer un programme XSLT on utilise l'instruction de traitement `<?xml-stylesheet>`. Reprenons notre document *Promotion.xml* (page **??**) dans lequel nous souhaitons toujours intégrer dynamiquement des données issues de la base. Voici le document XSQL pour atteindre ce but :

Exemple 2.9 *Promotion.xsql* : *Le document XSQL avec la promotion pour Passac*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<?xml-stylesheet href="Promotion.xsl" type="text/xsl"?>

<Promotion auteur="Jules"
           connection="connexionVisiteur"
           xmlns:xsql="urn:oracle-xsql">

  <Description>
   Nous proposons une réduction de <Reduction>25</Reduction>
   pourcent, restreinte à la période du
   <Periode> <Debut>Septembre 2001</Debut>
   au <Fin>Octobre 2001</Fin>  </Periode> pour tous
   les séjours dans certaines stations de vacances.
   L'automne est une saison <Important>merveilleuse</Important>
    pour reprendre des forces à la montagne. Vous pourrez
     profiter du calme,
    d'une nature aux couleurs chatoyantes, et d'un contact
    privilégié avec l'autochtone. <Important>Attention
    le nombre de places offertes est limité.</Important>
  </Description>

  <xsql:query
       rowset-element="Stations"
       row-element="Station">
     SELECT nomStation, capacite, lieu, region, tarif
     FROM Station WHERE nomStation='Passac'
  </xsql:query>

 </Promotion>
```

Cet exemple illustre l'intégration de `<xsql:query>` dans un document XML standard contenant d'autres éléments « en dur ». Notez que l'attribut `connection` reste associé à la racine du document. Des paramètres de mise en forme du résultat issu de la base sont maintenant associés à `<xsql:query>` :

- `rowset-element` définit le nom de l'élément racine du fragment produit par `<xsql:query>`, sa valeur par défaut étant `ROWSET` comme nous l'avons vu ;
- `row-element` définit le nom de l'élément pour chaque ligne du résultat de la requête SQL, sa valeur par défaut étant `ROW`.

On peut également utiliser des paramètres provenant d'une requête HTTP. Nous reprenons notre exemple d'un serveur web permettant d'interroger la base *Station* en donnant trois paramètres (voir notre *servlet* page **??** et son formulaire associé) : le nom de la station à afficher, le nom de l'élément racine du document, et le nom de l'élément pour chaque ligne. Voici ce service d'interrogation avec XSQL :

Exemple 2.10 *RequeteXSQL.xsql* : *Document XSQL avec paramètres*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

  <xsql:query connection="connexionVisiteur"
              xmlns:xsql="urn:oracle-xsql"
              rowset-element="{@racine}"
              row-element="{@element}">

    SELECT * FROM Station
    WHERE nomStation='{@nomStation}'
  </xsql:query>
```

Les paramètres sont représentés par la syntaxe *@nomParam*. On peut les intégrer dans les attributs de <xsql:query>, ou dans la requête SQL elle-même.

XSQL fournit donc une interface très simple entre une base de données et XML. Il existe de nombreux autres éléments et options pour choisir par exemple la mise en forme XML, le mode de sérialisation du document résultat, ou, à l'inverse, le stockage d'un document XML dans Oracle. Nous complétons cette présentation en décrivant les possibilités de mise en forme du document XML produit par une requête XSQL.

Options de sorties XML

XSQL permet de contrôler les options de sorties XML à plusieurs niveaux. Tout d'abord l'élément <xsql:query> utilise un ensemble d'attributs présentés dans le tableau 1 et dont les plus importants ont été illustrés dans les exemples qui précèdent.

Nom de l'attribut	Description
rowset-element	Nom de l'élément racine du résultat
row-element	Nom de l'élément pour chaque ligne du résultat
max-rows	Nombre maximal de lignes ramenées par la requête
skip-rows	Nombre de lignes à ignorer dans le résultat
id-attribute	Nom de l'attribut XML identifiant une ligne (id) par défaut
id-attribute-column	Nom de la colonne constituant la clé du résultat (par défaut XSQL crée un compteur de lignes)
null-indicator	Si yes, inclut les valeurs à NULL avec un indicateur (par défaut une valeur à NULL n'engendre pas d'élément).
tag-case	Permet d'indiquer si les noms d'éléments sont en majuscules (upper), minuscules (lower), ou respectent la casse utilisée dans la requête (valeur par défaut).
bind-params	Définit une liste de paramètres

TAB. 1 – *Les attributs de l'élément* ***xsql:query***

Au niveau de la requête SQL elle-même, on peut agir sur le format du résultat en demandant notamment une *imbrication* des éléments. En principe SQL est un langage qui agit sur des tables et produit une table. Dans une table relationnelle, on trouve des lignes constituées d'attributs avec des valeurs « atomiques » (des entiers, chaînes de caractères, ...). On ne peut pas en théorie dire qu'une valeur est un graphe, une autre table, ou toute structure un tant soit peu complexe. Il n'existe donc, à aucun niveau, la possibilité de manipuler une structure hiérarchique.

Oracle fournit deux types d'extensions pour le modèle relationnel : au niveau du modèle, et au niveau du langage d'interrogation. Au niveau du modèle, le SGBD d'Oracle est, depuis la version 8, « relationnel-objet », et permet d'imbriquer des types de données complexes dans des tables relationnelles. La mise en forme XML d'une table de ce type reprend exactement l'imbrication du modèle Oracle.

La seconde possibilité apparaît au niveau du langage lui-même, avec l'opérateur CURSOR qui, utilisé dans la clause SELECT, permet de constituer une résultat de requête qui comprend des attributs simples et des tables imbriquées. Voici un exemple de requête SQL avec CURSOR :

```
SELECT nomStation, capacite,
       CURSOR(SELECT libelle, prix FROM Activite
              WHERE A.nomStation = S.nomStation) AS activites
FROM  Station S
```

Dans la clause SELECT, nomStation et capacite sont des valeurs atomiques (des chaînes de caractères), mais l'évaluation de CURSOR est une table. Ce troisième attribut est nommé activites avec la clause AS. On obtient donc une table imbriquée dont voici la représentation XML (en nous limitant à la station Passac).

```
<Stations>
```

```
  <Station num='1'>
    <nomStation>Passac</nomStation>
    <capacite>400</capacite>
    <activites>
     <activites_row num='1'>
      <libelle>Ski</libelle>
      <prix>200.00</prix>
     </activites_row>
     <activites_row num='2'>
       <libelle>Piscine</libelle>
       <prix>20.00</prix>
     </activites_row >
    </activites>
  </Station>
</Stations>
```

On peut utiliser plusieurs curseurs, avec plusieurs niveaux d'imbrication, ce qui permet d'obtenir des documents hiérarchiques XML arbitrairement complexes.

3 Le langage XPath

3.1 Expressions XPath

Une *expression XPath* désigne un ou plusieurs nœuds en exprimant des *chemins* dans un arbre XML. L'*évaluation* d'une expression donne un résultat qui peut être soit une valeur numérique ou alphanumérique, soit un sous-ensemble des nœuds de l'arbre. Enfin cette évaluation tient compte d'un *context général* qui détermine le résultat.

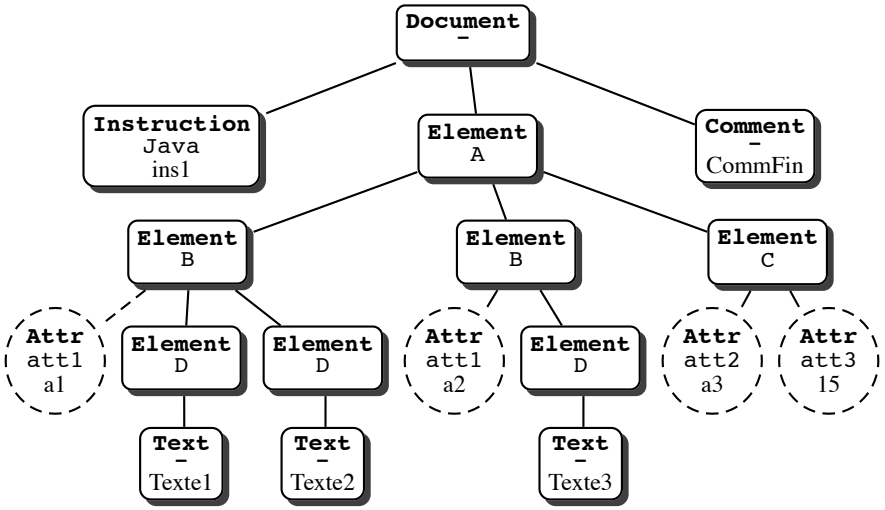

FIG. 3 – *Exemple d'un arbre XPath*

Nous allons développer chacun de ces aspects ci-dessous, en prenant pour exemple principal l'arbre de la figure 3. Comme on peut le constater cet arbre présente à peu près tous les types de nœuds manipulables par XPath, avec plusieurs configurations. Rappelons que dans nos représentations chaque nœud est caractérisé par les trois composants suivants :

1. le *type* du nœud, toujours présent, en gras ;

2. le *nom* pour les nœuds de type **Element** ou **Attr**, ou la cible pour le type **ProcessingInstruction** ; pour tous les autres types de nœud il n'y a pas de nom ;

3. la *valeur*, valable pour tous les types de nœuds, y compris les attributs, à l'exception du type **Element** et du type **Document**.

Cet arbre est obtenu à partir du document sérialisé suivant, que vous pouvez récupérer sur notre site pour tester les expressions XPath (les outils permettant d'effectuer ces tests sont présentés dans l'annexe **??**).

Exemple 3.1 *ExArbreXPath.xml* : *Le fichier XML pour les exemples XPath*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<?java ins1?>
<A>
  <B att1='a1'>
   <D>Texte1</D>
   <D>Texte2</D>
  </B>
  <B att1='a2'>
   <D>Texte3</D>
  </B>
  <C att2='a3' att3='15'/>
</A>
<!-- CommFin -->
```

Chemins absolus et chemins relatifs

Un chemin XPath peut être *absolu*, et prendre son origine à la racine du document, notée « / ». Par exemple l'expression :

```
/A/B/@att1
```

correspond à un ou plusieurs chemins qui partent de la racine du document (« / »), puis passent par l'élément racine si le type de cet élément est `A`, parcourent les éléments de type `B` fils de cet élément racine, et enfin désignent les attributs `att1` de ces éléments `B`. La figure 4 montre les deux nœuds, de type **Attr**, désignés par deux chemins différents.

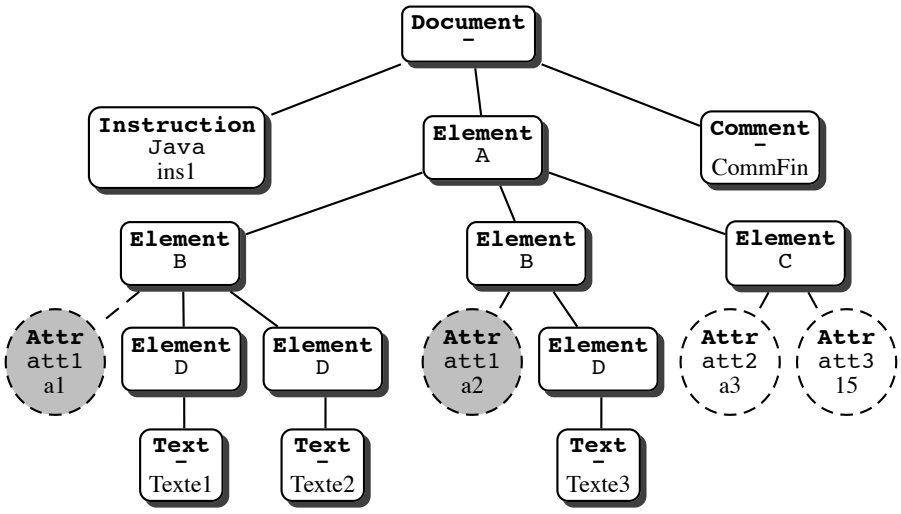

FIG. 4 – *Chemins vers les attributs* `att1` *des nœuds B*

Un chemin peut également être *relatif* et prendre pour origine un nœud du document, dit *nœud contexte*. En pratique l'utilisation de chemins relatifs est la plus répandue avec XSLT puisque le principe de base consiste à déclencher des règles à partir de nœuds considérés comme les nœuds contextes des chemins XPath. L'expression (noter l'absence du « / » initial) :

```
A/B/@att1
```

désigne un ou plusieurs chemins qui *partant d'un nœud contexte de type quelconque*, passent successivement par les nœuds de type `A`, `B` et aboutissent aux attributs de nom `att1`. Ce type d'expression est en principe plus puissant puisqu'il s'applique à tous les types de documents dans lesquels on retrouve un motif de ce type, quel que soit par ailleurs la position relative de ce motif par rapport à la racine du document. On peut considérer un chemin absolu comme une forme particulière de chemin relatif dans laquelle le nœud contexte est systématiquement la racine du document.

Syntaxe XPath

Une expression XPath est constituée d'une suite d'étapes, séparées par des « / ». La forme générale est donc :

$$[/]\text{étape}_1/\text{étape}_2/.../\text{étape}_n$$

le « / » initial étant optionnel, et distinguant les chemins relatifs des chemins absolus, comme nous l'avons expliqué précédemment.

Chaque étape peut elle-même se subdiviser en trois composants :

1. *l'axe* qui définit le sens de parours des nœuds à partir du nœud contexte ;
2. *le filtre* qui indique le type des nœuds qui seront retenus dans l'évaluation de l'expression ; la notion de type recouvre ici les types de nœuds dans un arbre XML (commentaire, texte, instruction de traitement,...) ainsi que, pour les éléments et les attributs, le nom de la balise ou de l'attribut ;
3. *le (ou les) prédicat(s)* qui exprime(nt) des propriétés que doivent satisfaire les nœuds retenus à l'issue du filtrage pour être finalement inclus dans le résultat.

La forme générale d'une étape est donc :

```
axe::filtre[prédicat1][prédicat2]...
```

Signalons dès maintenant qu'il est possible, quand on utilise XPath avec XSLT, d'effectuer une *union d'expressions* avec l'opérateur « | ». Par exemple l'expression `//A | B/@att1` désigne l'union des ensembles désignés par les expressions `//A` et `B/@att1`.

Les expressions XPath que nous avons utilisées jusqu'ici correspondaient aux formes les plus simples du langage XPath. En particulier, comme nous l'avons souligné dans le chapitre **??**, nous n'avons jamais eu recours aux prédicats qui sont optionnels et nous avons toujours choisi d'utiliser comme axes de parours les *fils* ou les *attributs* du nœuds contexte. Le premier axe, noté `child` dans XPath, est également considéré comme celui par défaut. Le deuxième axe, noté `attribute`, est distingué par le symbole « @ » avant le nom de l'attribut. On peut noter que les attributs sont traités par un axe spécifique.

Évaluation d'une expression XPath

L'évaluation d'une étape peut être soit une valeur, soit un *ensemble de nœuds* (*node-set* en Anglais). Nous laissons de côté, jusqu'à la page **??**, le cas des valeurs pour nous concentrer sur le cas où l'évaluation revient à désigner un ensemble de nœuds du document source.

Il est bon de noter qu'un nœud n'est jamais extrait du document ni placé hors de son contexte (à savoir sa position dans l'arbre). Un *node-set* doit donc plutôt être considéré comme un ensemble de *références* vers des nœuds de l'arbre XML. Notons également qu'un même nœud ne peut être référencé qu'une seule fois dans un même ensemble.

Quand une expression est constituée de plusieurs étapes, on se trouve en général dans la situation où l'évaluation de chaque étape (sauf éventuellement la dernière) donne un ensemble de nœuds. L'évaluation de l'expression complète est alors basée sur les principes suivants :

- à partir du nœud contexte, on évalue l'étape 1 ; on obtient un ensemble de nœuds ;
- on prend alors, un par un, les nœuds de cet ensemble, *et on les considère chacun à leur tour comme nœud contexte pour l'évaluation de l'étape 2* ;
- la règle précédente se généralise comme suit : à chaque étape, on prend successivement comme nœud contexte chacun des nœuds faisant partie du résultat de l'étape précédente.

Reprenons un des exemples donnés précédemment pour clarifier cette évaluation :

```
/A/B/@att1
```

L'expression ci-dessus désigne des chemins absolus, constitués de trois étapes, dont le nœud contexte initial est donc la racine du document (figure 5). L'axe de recherche n'est jamais indiqué, et on suit donc celui par défaut, `child`, consistant à parcourir les fils du nœud contexte.

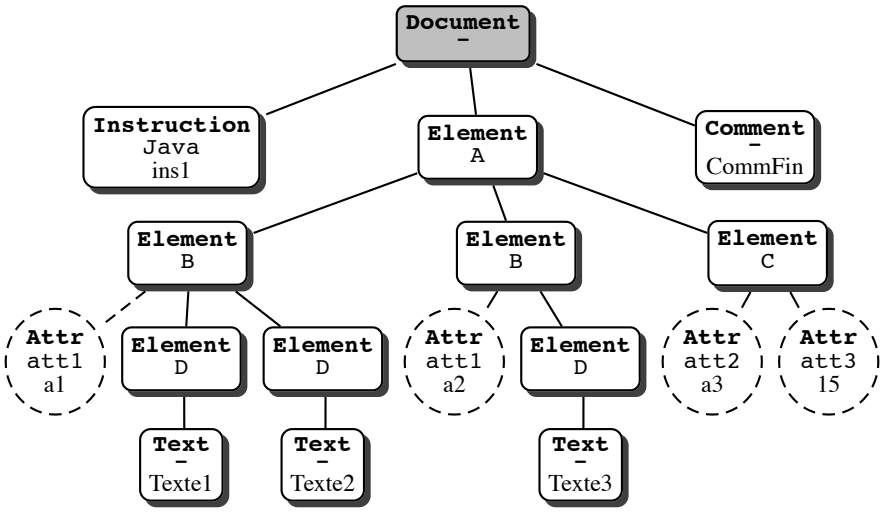

FIG. 5 – *Le nœud contexte initial*

À partir de la racine, on évalue donc l'étape 1. Le filtre est `A` : on recherche un nœud fils, dont le type est **Element**, et le nom de balise `A`. Si on le trouve, il ne peut y en avoir qu'un seul car c'est l'élément racine du document : le résultat de l'étape 1 est donc un ensemble réduit à un seul nœud (figure 6).

Maintenant on prend ce nœud comme contexte, et on évalue l'étape 2 dont le filtre est `B`. Le résultat de cette évaluation va être l'ensemble des nœuds dont le type est **Element**, et le nom de balise `B` (figure 7).

Enfin, on va prendre chaque nœud de type `B` comme nœud contexte, et on cherchera pour chacun un attribut `att1` : le résultat final est un ensemble d'attributs, autrement dit de nœuds dont le type est **Attr**. On obtient le résultat de la figure 4, page 18.

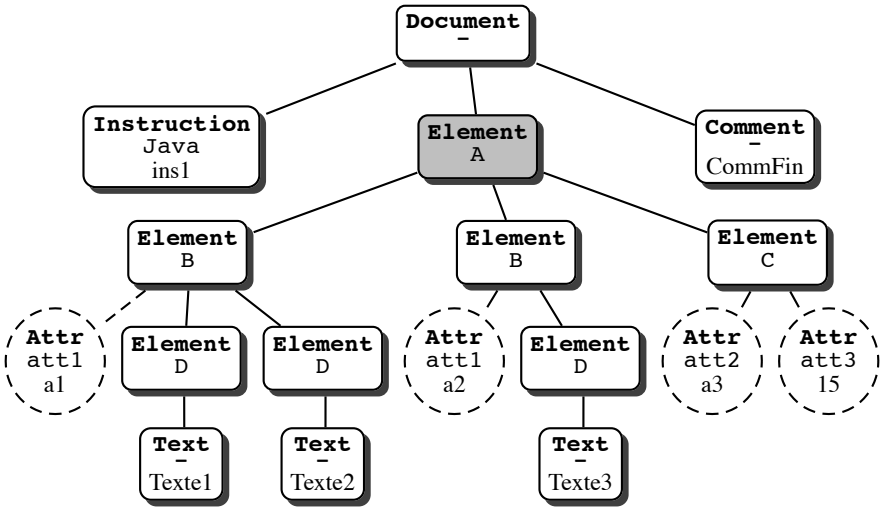

FIG. 6 – *Première étape de l'évaluation de l'expression* `/A/B/@att1`

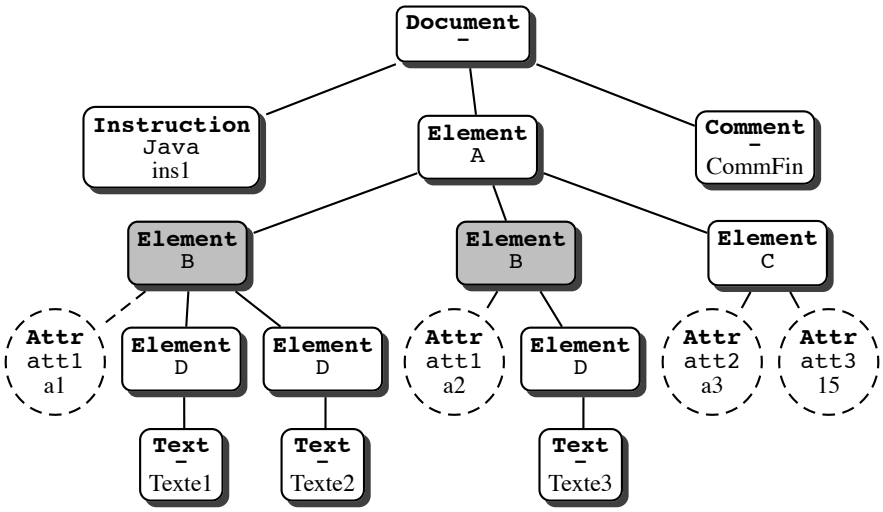

FIG. 7 – *Deuxième étape de l'évaluation de l'expression* `/A/B/@att1`

Contexte d'évaluation

Chaque étape d'une expression XPath est évaluée en fonction d'un ensemble d'informations qui constituent le *context* de l'évaluation. Il est très important d'être conscient que le context change à chaque étape, d'une part pour comprendre le résultat d'une évaluation XPath, d'autre part pour construire correctement des expressions.

Ce context comprend en premier lieu le nœud que nous avons appelé *nœud context* jusqu'à présent, à partir duquel sont construits les chemins des étapes suivantes. Mais le context d'évaluation d'une étape comprend également *l'ensemble de nœuds* obtenu par l'évaluation de l'étape précédente, constituant le context commun d'évaluation pour chaque nœud pris individuellement.

Reprenons l'exemple précédent pour clarifier cette notion de context. L'expression XPath est

```
/A/B/@att1
```

avec trois étapes. À l'issue de la deuxième étape, correspondant au filtre `B`, on obtient un ensemble comportant deux nœuds, représenté dans la figure 7. Cet ensemble va constituer le *contexte commun* à l'évaluation de l'étape suivante, `@att1`.

Les nœuds de cet ensemble sont alors considérés chacun leur tour en tant que nœud context pour l'évaluation de l'étape `@att1`. En pratique, on pourra connaître la *position* d'un nœud dans son context (fonction *position*(), voir page **??**) ou savoir si un nœud est le dernier de son context (fonction *last*()). Ici il y a deux positions, 1 et 2, caractérisant les deux occurrences de nœuds de type `B`.

3.2 Les axes

Nous présentons maintenant de manière exhaustive les options des trois composants d'une étape dans une expression XPath (axes, filtres et prédicats), en commençant par les axes. Rappelons que tout axe s'interprète par rapport à un nœud contexte. Un axe XPath recouvre alors les deux notions suivantes :

1. un sous-ensemble des nœuds de l'arbre relatif au nœud contexte ;
2. l'ordre de parours de ces nœuds à partir du nœud contexte.

Le tableau 2 donne la liste des axes XPath. Tous ces axes désignent des parours de recherche de nœuds dans l'arbre XML, à l'exception de l'axe `attribute` qui s'applique à l'arbre des attributs et `namespace` qui s'applique à la recherche d'espaces de noms, ce dernier aspect étant laissé de côté jusqu'au chapitre **??**, page **??**.

Axe	**Description**
`child`	les fils du nœud contexte (axe par défaut)
`attribute`	les attributs du nœud contexte
`parent`	le père du nœud contexte
`descendant`	tous les descendants du nœud contexte
`ancestor`	tous les ancêtres du nœud contexte
`self`	le nœud contexte lui-même
`preceding-sibling`	tous les frères gauches du nœud contexte
`following-sibling`	tous les frères droits du nœud contexte
`preceding`	les nœuds précédant le nœud contexte dans l'ordre de parcours du document
`following`	les nœuds suivant le nœud contexte dans l'ordre de parcours du document
`descendant-or-self`	les descendants du nœud contexte, et le nœud contexte lui-même
`ancestor-or-self`	les ancêtres du nœud contexte, et le nœud contexte lui-même
`namespace`	s'applique aux espace de noms : voir chapitre **??**

TAB. 2 – *Les axes XPath*

Il est toujours possible de spécifier une étape XPath en donnant pour axe un des noms de la première colonne du tableau 2. Certaines facilités d'écriture sont cependant permises :

– quand l'axe n'est pas indiqué dans une étape, c'est `child` qui est pris par défaut ;

– certains axes peuvent être notés de manière abrégée, comme par exemple « @ » qui remplace `attribute`.

Nous décrivons maintenant chaque axe, en prenant comme exemple de référence l'arbre de la figure 8, avec comme nœud contexte le second nœud de type `B`, comme illustré sur la figure.

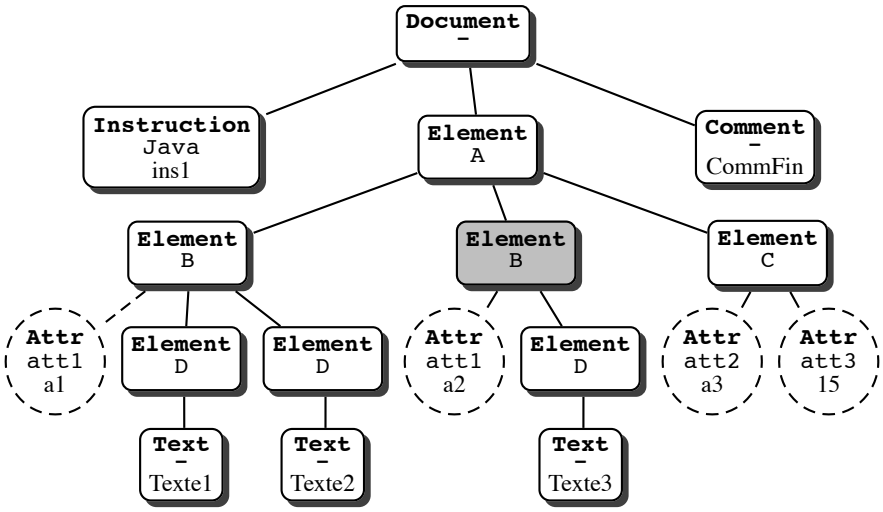

FIG. 8 – *Exemple de référence pour les axes XPath*

L'axe `child`

L'axe `child` désigne les fils du nœud contexte. Cet axe ne s'applique pas aux attributs, mais prend en compte en revanche tous les autres types de nœuds. La figure 9 montre l'unique nœud désigné par cet axe à partir de notre nœud contexte.

L'axe peut être utilisé explicitement, mais le plus souvent on omettra l'indication `child` pour se reposer sur l'interprétation par défaut. Les deux expressions ci-dessous sont donc équivalentes et, appliquées à notre arbre-exemple, donnent *tous* les nœuds de type `D`. :

```
/child::A/child::B/child::D
```

est équivalent à

```
/A/B/D
```

L'axe `attribute`

Cet axe constitue une exception car il est le seul à s'appliquer à l'arbre des attributs. Cet arbre est lui-même très particulier puisqu'il a toujours pour racine un nœud de type **Element**, et un seul niveau constitué de feuilles, *non ordonnées* mais identifiées par leur nom, les attributs.

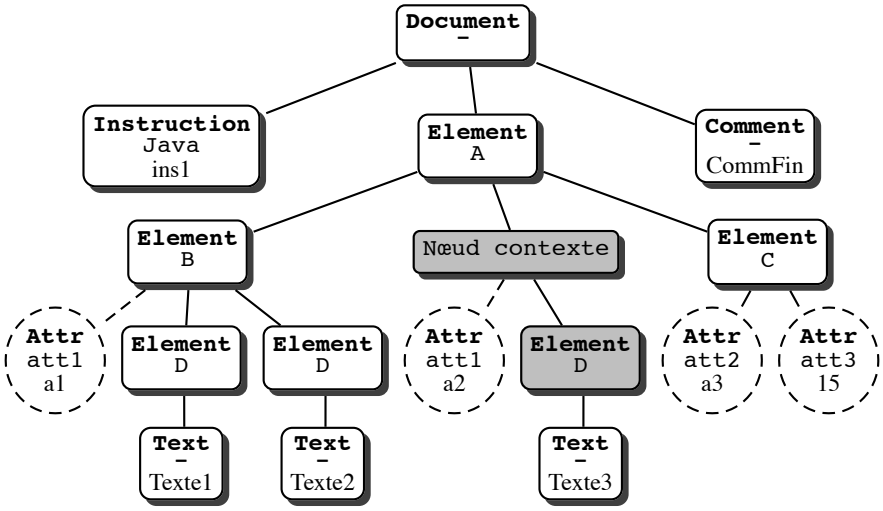

FIG. 9 – *L'axe* `child`

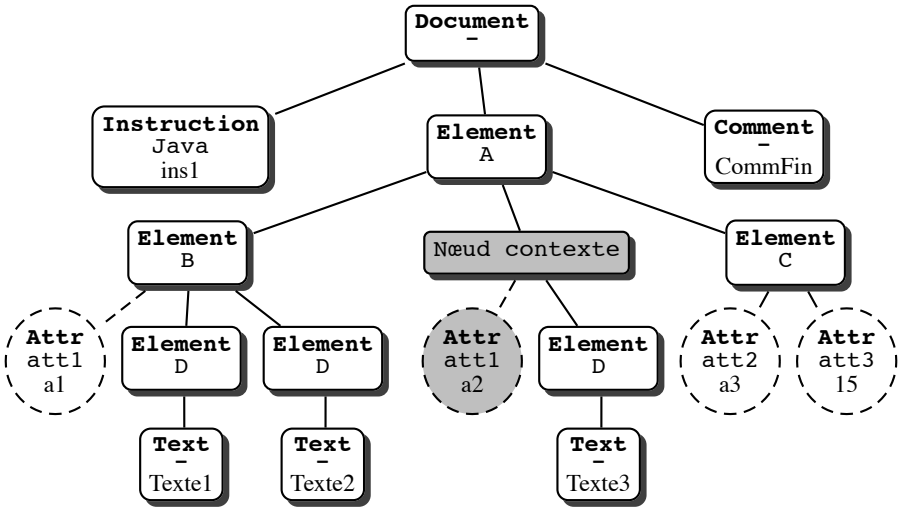

FIG. 10 – *Résultat de* `attribute::att1`

En conséquence, l'axe sera toujours utilisé dans la dernière étape d'une expression XPath, et l'attribut sera désigné par son nom. L'étape permettant de sélectionner l'attribut de nom `att1` à partir de notre nœud contexte (figure 10) est donc :

```
attribute::att1
```

On utilise le plus souvent l'abréviation *`@nom`* pour remplacer *`attribute::nom`* (remarquez la disparition des ::). On notera donc l'étape précédente de manière équivalente :

```
@att1
```

L'ensemble des attributs de nom `att1` associés à un élément de type `B` lui-même fils d'un élément-racine de type `A` est donc désigné par l'expression `/A/B/@att1`.

L'axe `parent`

Cet axe désigne le nœud père du nœud contexte (figure 11). L'expression `parent::A` permet donc de « remonter » d'un niveau dans l'arbre à partir de notre nœud de type `B`.

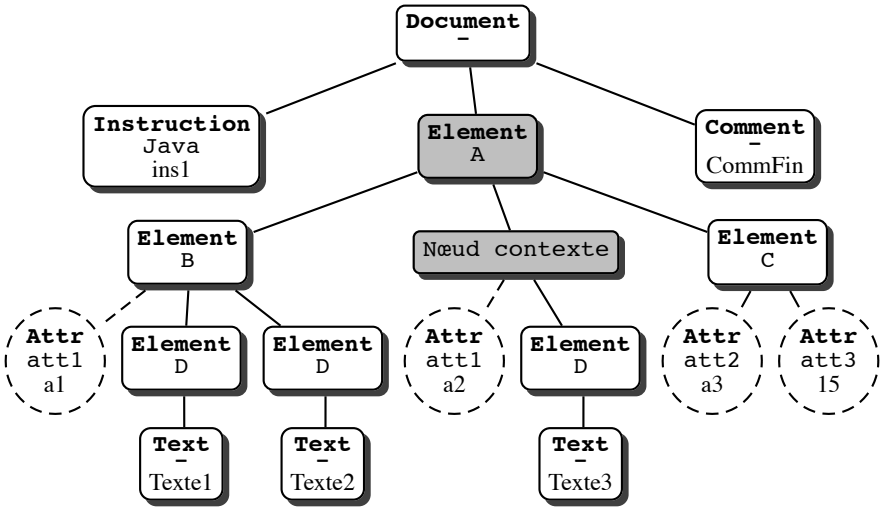

FIG. 11 – *Résultat de* `parent::A`

L'abréviation pour cet axe est « .. », conformément à une convention largement utilisée dans les systèmes de fichiers pour désigner le parent d'un répertoire. En fait cette abréviation n'est pas strictement équivalente à `parent::A` qui utilise le filtre `A` alors que « .. » désigne le nœud père *quel que soit son type*. L'équivalent de « .. » en notation développée est :

```
parent::node()
```

où `node()` est un filtre qui sélectionne tous les types de nœuds de l'arbre XML (à l'exception des attributs). On peut composer cet axe pour remonter d'un nombre donné de niveaux dans l'arbre. Par exemple l'expression suivante :

```
../..
```

désigne, à partir de notre nœud contexte de référence, la racine du document. On pourrait à partir de là redescendre sur l'un des trois fils de la racine, mais on rencontre alors un problème de nommage puisque seul l'un de ces fils est de type **Element** avec un nom. Nous verrons page 28 comment utiliser des modes de référencement autres que le nommage des nœuds. On peut utiliser le filtre `node()` pour éviter ce problème de nommage :

```
../../node()
```

Cette expression désigne donc les trois fils de la racine du document, qui sont de trois types différents. Notez bien que cette expression est une abréviation de

```
parent::node()/parent::node()/child::node()
```

où le dernier axe `child` indique bien que les attributs – s'il y en avait – ne sont pas concernés.

L'axe `descendant`

Cet axe permet de désigner tous les nœuds de l'arbre XML principal (à l'exception donc des attributs et des espaces de noms) qui sont descendants du nœud contexte. Voici l'expression donnant les descendants de notre nœud contexte, tous types confondus (figure 12) :

```
descendant::node()
```

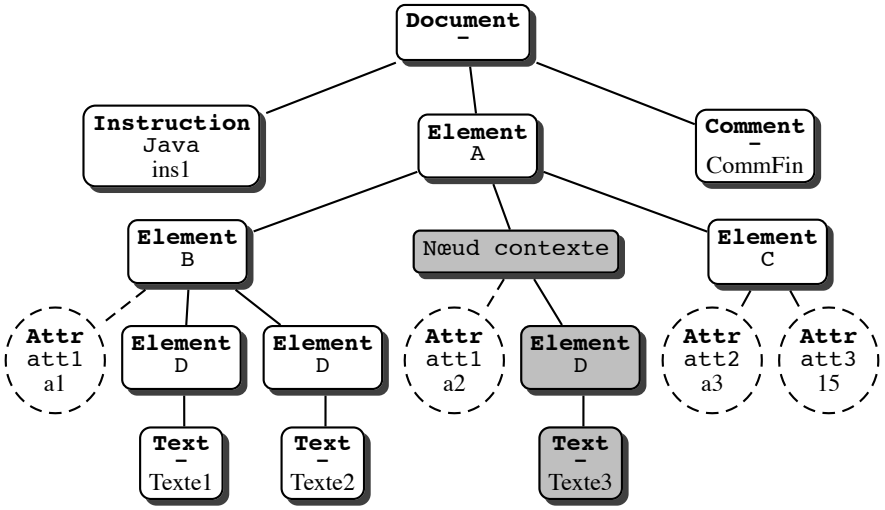

FIG. 12 – *Résultat de* `descendant::node()`

Le filtre `node()` est ici important pour désigner à la fois un nœud de type **Element** et un nœud de type **Text**. Le premier, seul, serait obtenu par l'expression `descendant::D` et le second par l'expression `descendant::text()`.

L'axe `ancestor`

L'axe `ancestor` est réciproque de `descendant` : il désigne tous les ancêtres du nœud contexte, et ce quel que soit leur type (toujours à l'exception du type **Attr**). La figure 13 montre, sur notre exemple de référence, les nœuds désignés par

```
ancestor::node()
```

L'axe `self`

Cet axe désigne le nœud contexte lui-même. Le résultat dépend ensuite du filtre qui est appliqué. Sur notre exemple, l'expression `self::B` revient bien à sélectionner le nœud de type `B` qui constitue notre nœud contexte. En revanche `self::A` renverrait, toujours sur notre exemple, un ensemble vide puisque le nœud contexte n'est pas de type `A`.

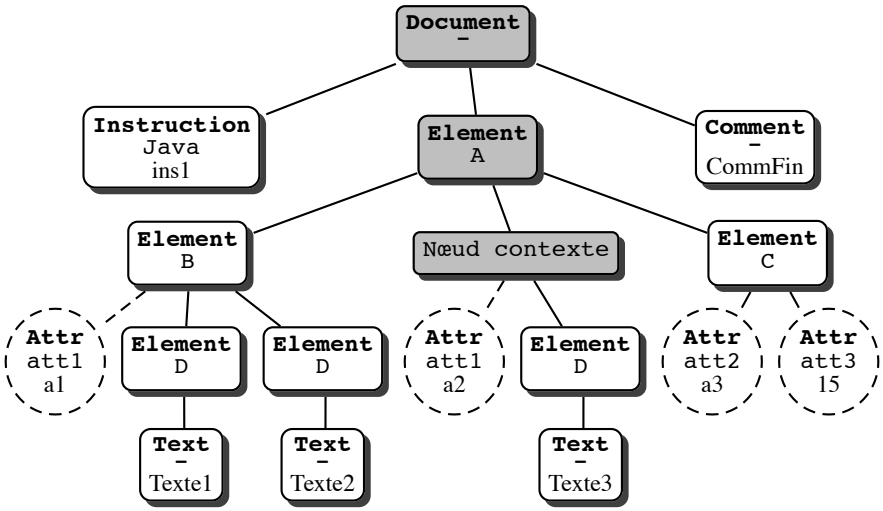

FIG. 13 – *Résultat* `ancestor::node()`

Afin de pouvoir désigner le nœud contexte sans avoir à s'embarrasser de donner son type, on utilise souvent l'expression

```
self::node()
```

qui utilise le filtre *node*(), valide pour tout type de nœud. Il existe une notation abrégée, « `.` », pour `self::node()`, très fréquemment utilisée. L'expression suivante utilise donc des notations abrégées pour désigner l'attribut `att1` du nœud contexte :

```
./@att1
```

Sans utiliser la notation abrégée, cette expression s'écrirait :

```
self::node()/attribute::att1
```

Les axes `preceding-sibling` et `following-sibling`

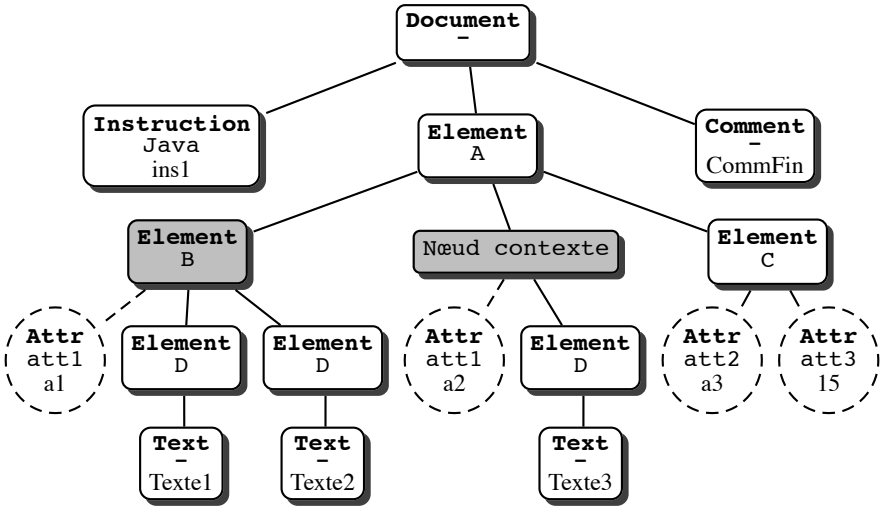

FIG. 14 – *Résultat de* `preceding-sibling::node()`

Ces deux axes désignent respectivement les frères à gauche et à droite du nœud contexte. La figure 14 montre l'axe `preceding-sibling` appliqué à notre exemple, avec un filtre `node()`.

On obtient évidemment *en l'occurrence* le même résultat avec l'expression `preceding-sibling::B`, dont le filtre limite la sélection des nœuds aux éléments de type `B`.

Les axes `preceding` et `following`

Ces deux axes désignent respectivement l'ensemble des nœuds qui précèdent ou qui suivent le nœud contexte *dans l'ordre de parours du document*. Rappelons que cet ordre correspond, pour la représentation arborescente, à un parours gauche en profondeur d'abord, et plus simplement, pour la représentation sérialisée, à un parours séquentiel du document.

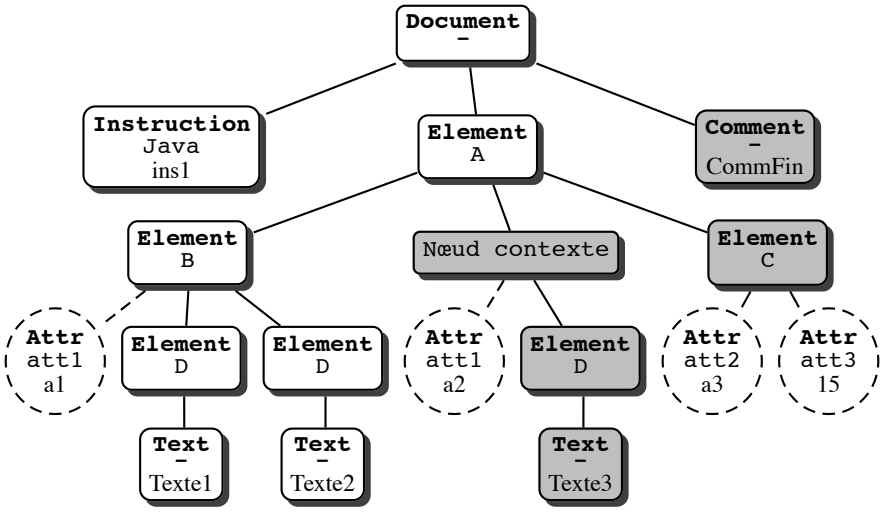

FIG. 15 – *Résultat de* `following::node()`

La figure 15 montre sur notre exemple les nœuds sélectionnés par l'expression `following::node()`. On peut vérifier dans le document *ExArbreXPath.xml*, page 18 que les nœuds séléctionnés correspondent au contenu qui suit la balise ouvrante de l'élément contexte (deuxième élément de type `B`).

Les axes `descendant-or-self` et `ancestor-or-self`

Comme l'indiquent leurs noms, ces axes désignent respectivement l'union des axes `descendant` et `self`, et l'union des axes `ancestor` et `self`, Par exemple l'expression `descendant-or-self::node()` désigne les mêmes nœuds que ceux déjà montrés dans la figure 12, avec le nœud contexte en plus.

Le tableau 3 récapitule les abréviations qu'on peut utiliser dans les axes XPath. Il faut être prudent dans l'utilisation de ces abréviations, et toujours se ramener à la forme développée quand on n'est pas sûr de la signification de la forme abrégée.

3.3 Les filtres

Un filtre permet d'éliminer un certain nombre de nœuds parmi ceux sélectionnés par un axe. Il existe essentiellement deux types de filtrage pour sélectionner des nœuds : *par leur type* et *par leur nom*. Ces deux critères de sélection sont les plus utilisés dans les expressions XPath. Nous verrons ensuite l'utilisation des prédicats pour construire des critères beaucoup plus complexes.

Abréviation	Notation étendue équivalente
.	`self::node()`
..	`parent::node()`
@	`attribute::`
@att	`attribute::att`
@*	`attribute::*`
A/@*	`child::A/attribute::*`
Axe par défaut	`child::`
/A	`/child::A`
A/*	`child::A/child::*`
Étape par défaut	`descendant-or-self::node()`
//	`/descendant-or-self::node()/`
.//	`self::node()/descendant-or-self::node()/`
A//B	`child::A/descendant-or-self::node()/child::B`

TAB. 3 – *Les abréviations XPath*

Filtrage par nom

Le filtrage par nom ne s'applique qu'aux nœuds pour lesquels cette notion de nom est pertinente, à savoir ceux de type **Element**, de type **ProcessingInstruction** et de type **Attr**. Il s'effectue simplement en indiquant le nom, ou une partie du nom.

La plus grande part des exemples d'expression XPath que nous avons vus jusqu'à présent comprenaient un filtrage par nom. Par exemple l'expression `/A/B/D` désigne, en partant de la racine du document, l'élément de nom `A`, puis les éléments de nom `B` fils de `A`, et enfin les éléments de nom `D` fils de `B`. Les chemins obtenus sur notre exemple sont montrés dans la figure 16.

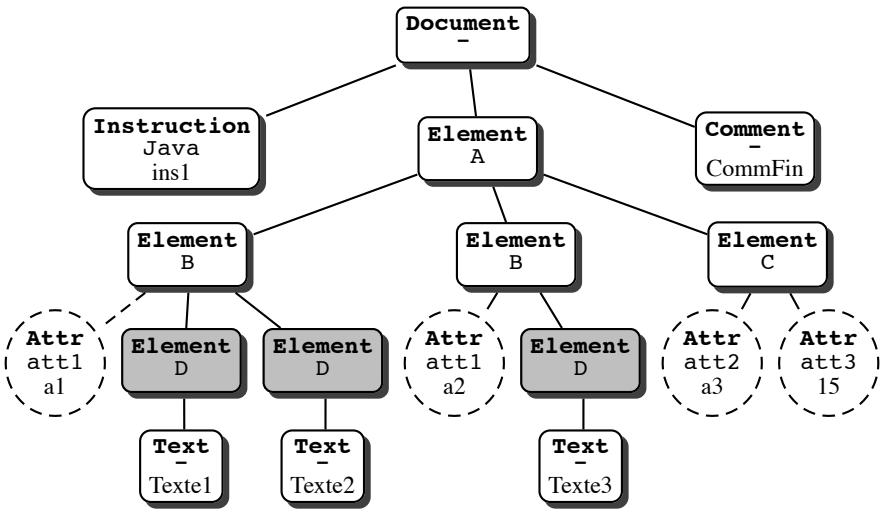

FIG. 16 – *Filtre avec noms d'éléments (*`/A/B/D`*)*

Un filtrage avec nom d'attributs ne peut intervenir que dans la dernière étape d'une expression XPath (puisqu'un attribut est toujours un nœud feuille). Voici par exemple une expression qui désigne tous les attributs nommés `att2`, quelle que soit par ailleurs leur position dans l'arbre :

```
/descendant::node()/@att2
```

On combine une étape `/descendant::node()` qui, à partir de la racine, va sélectionner tous les nœuds à l'exception des attributs, puis pour chacun de ces nœuds on cherche l'attribut `att2`. Le résultat est montré dans la figure 17.

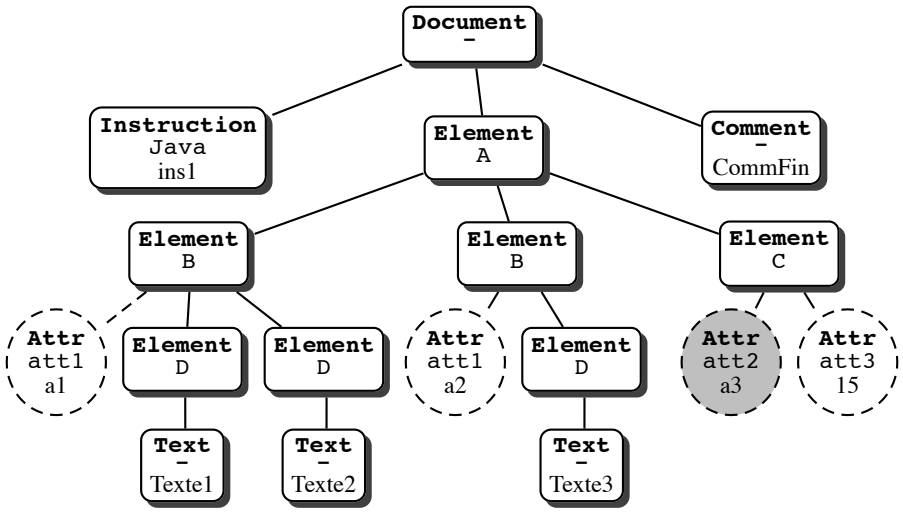

FIG. 17 – *Filtre avec noms d'attributs (*`/descendant::node()/@att2`*)*

Filtrage par type de nœud

Les filtres par nommage ne concernent que les attributs ou les nœuds de type **Element**. Il est possible de désigner *tous* les nœuds de type **Attr** (pour l'axe `attribute`) ou de type **Element** (pour tous les autres axes) avec le caractère « * ». L'expression suivante va donc sélectionner tous les attributs de l'arbre, quel que soit leur nom :

```
/descendant::node()/@*
```

De même l'expression suivante sélectionne tous les éléments fils de l'élément racine <A>, quel que soit leur nom :

```
/A/*
```

Le résultat est donné dans la figure 18.

Notez que seuls les éléments de type **Element** sont concernés, et pas ceux de type **Comment**, **Text** ou **ProcessingInstruction**. Des filtres spécifiques existent aussi pour ces types. Ils sont montrés dans le tableau 4.

Filtre	**Nœuds désignés**
`text()`	tous les nœuds de type **Text**
`comment()`	tous les nœuds de type **Comment**
`processing-instruction()`	tous les nœuds de type **ProcessingInstruction**
`*`	tous les nœuds de type **Element** (avec tous les axes sauf `attribute`) ou de type **Attr** (avec l'axe `attribute`)
`node()`	tous les nœuds (de n'importe quel type)

TAB. 4 – *Les filtres sur les types de nœud*

Voici quelques exemples pour illustrer l'utilisation de ces filtres. L'expression

```
/processing-instruction()
```

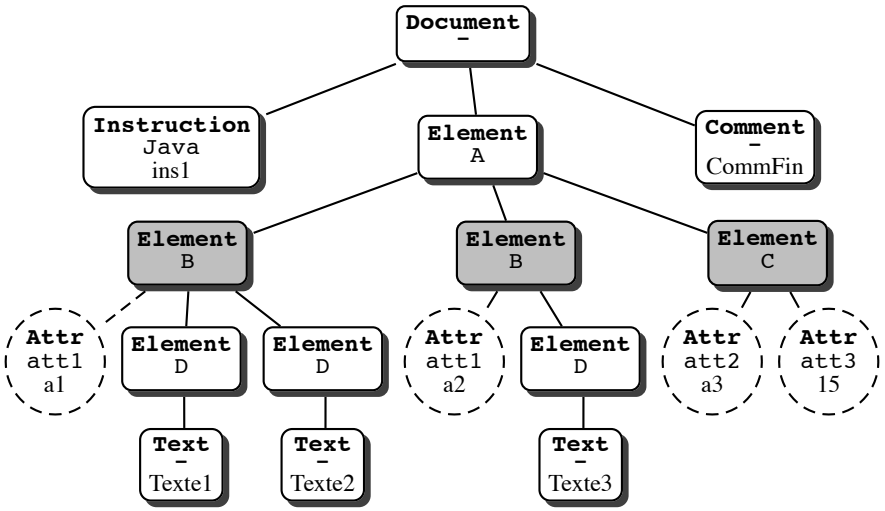

FIG. 18 – *Résultat de* `/A/*`

désigne les nœuds fils de la racine du document de type **ProcessingInstruction**. En prenant notre exemple on obtient le premier fils de la racine du document. On peut faire référence de manière plus précise à une instruction de traitement en utilisant son nom dans la fonction de filtrage. Par exemple, le filtre

```
processing-instruction('java')
```

sélectionne uniquement les instructions de traitement dont la cible est `java` (filtrage par nom).

Pour désigner *tous* les commentaires fils de la racine on utiliserait :

```
/comment()
```

Les commentaires n'ayant pas de nom, il n'est évidemment pas possible d'y appliquer un filtrage par nom. Enfin le filtre `text()` désigne les nœuds de type **Text**. Voici l'exemple d'une expression désignant tous les nœuds de texte situés sous un élément <B>

```
/A/B//text()
```

Notez l'utilisation de la notation abrégée « // ». En notation non abrégée on aurait :

```
/child::A/child::B/descendant-or-self::node()/text()
```

Le résultat de la dernière expression appliquée à notre exemple de référence est donné dans la figure 19.

3.4 Prédicats

La troisième partie d'une étape est un *prédicat*, autrement dit une expression booléenne constituée d'un ou plusieurs tests, composés avec les connecteurs logiques habituels `and` et `or`. La négation est fournie sous la forme d'une fonction *not*().

Un test est toute expression donnant un résultat booléen (`true` ou `false`). En fait, à peu près n'importe quelle expression XPath peut être convertie en un booléen avec XPath, ce qui donne beaucoup de liberté, au prix parfois d'une certaine obscurité.

Un ensemble de nœuds par exemple est interprété comme `true` s'il n'est pas vide, et comme `false` sinon. L'expression

```
/A/B[@att1]
```

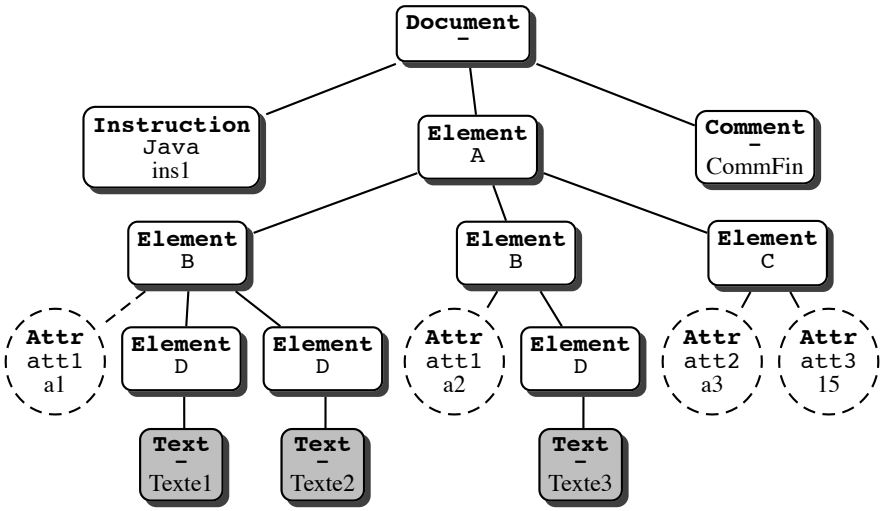

FIG. 19 – *Résultat de* `/A/B//text()`

a donc la signification suivante : l'expression `@att1` dans le prédicat retourne l'ensemble des attributs de nom `att1` des éléments de type `B`. S'il n'y a pas d'attribut de ce nom, l'ensemble est vide et la valeur booléenne du prédicat est `false`. Pour dire les choses simplement, cette expression sélectionne les éléments de type `B` qui ont un attribut `att1`. Nous verrons page **??** comment on traduit des chaînes de caractères ou des nombres en valeur booléennes.

Les tests peuvent également consister à exprimer des conditions sur les valeurs de certains nœuds ou variables, et à appeler des *fonctions* parmi celles proposées par XPath. Les fonctions XPath les plus utilisées sont probablement *position*(), qui donne la position du nœud au sein du contexte, et *last*(), qui donne la position du dernier nœud dans le contexte (autrement dit le nombre de nœuds dans le contexte puisque les positions sont numérotées à partir de 1). Voici un premier exemple d'une expression XPath avec prédicat.

```
/A/B/descendant::text()[position()=1]
```

Le résultat est celui de la figure 20 : on sélectionne les premiers nœuds parmi les nœuds de type **Text** qui descendent des éléments de type `B` désignés par `/A/B`.

Un prédicat `[position()=`*n*`]` peut s'abréger simplement en `[`*n*`]`, le chiffre *n* étant implicitement considéré comme faisant référence à la position. Ainsi, l'expression précédente peut également être écrite

```
/A/B/descendant::text()[1]
```

Il est très important de noter que le prédicat d'une étape prend en compte le *contexte* constitué de l'ensemble de nœuds obtenu par évaluation de l'axe et du filtre de cette même étape. Ici l'étape est évaluée deux fois, pour chacun des nœuds de type `B`, et à chaque fois le prédicat est appliqué aux nœuds obtenus par évaluation de `descendant::text()`. On obtient donc le premier fils de type **Text** de chaque nœud `B`.

Revenons sur les fonctions *position*() et *last*(). Elles s'appuient sur une numérotation des nœuds dans un ensemble, elles-même fonction de deux critères :

1. l'ordre des nœuds du document XML ;
2. l'axe utilisé dans l'étape XPath : la plupart des axes respectent l'ordre du document (*forward-axes*), mais d'autres, dits *axes inverses* (*reverse-axes*), adoptent l'ordre inverse. La règle générale est que tous les axes qui peuvent désigner des nœuds situés *avant* le nœud contexte (`ancestor`, `ancestor-or-self`, `preceding-sibling` et `preceding`) sont des axes inverses.

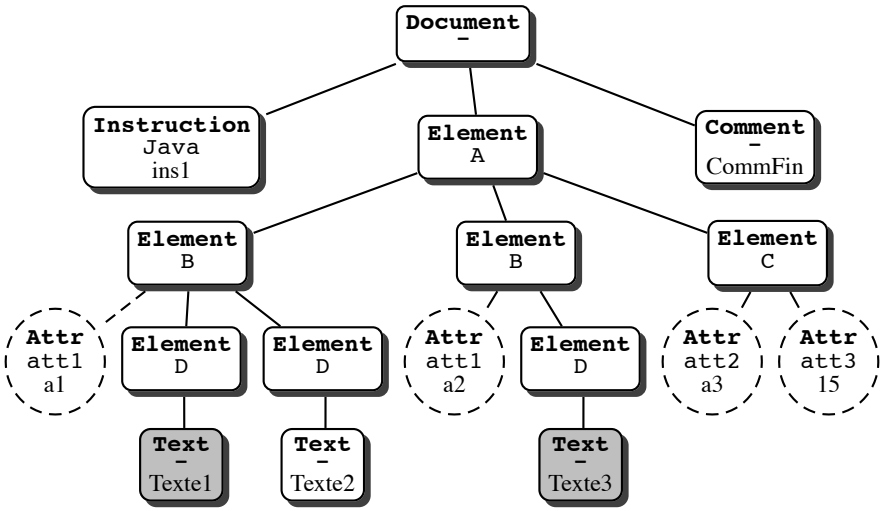

FIG. 20 – *Résultat de* `/A/B/descendant::text()[position()=1]`

Il est possible de tester les valeurs de certains nœuds en faisant référence à ces nœuds par des expressions XPath. Voici un premier exemple simple : une expression qui sélectionne tous les éléments de nom `B` ayant un attribut `att1` dont la valeur est `a1`.

```
/A/B[@att1='a1']
```

Les prédicats peuvent ainsi référencer des nœuds très éloignés du nœud contexte. Voici par exemple une expression qui sélectionne tous les nœuds de nom `B` si et seulement s'il existe un nœud fils de l'élément racine donc le type est `C` avec un attribut `att3` ayant la valeur `15`.

```
/A/B[/A/C/@att3=15]
```

On peut s'attendre à ce que cette expression soit très coûteuse à évaluer puisqu'il faudra, pour chaque nœud `B`, tester le prédicat en suivant les chemins absolus décrits par l'expression `/A/C/@att3` dans l'arbre XML (à moins d'utiliser un processeur XPath assez malin pour s'apercevoir que le résultat du prédicat est indépendant du contexte et peut donc s'évaluer une seule fois).

On peut effectuer des compositions booléennes des conditions élémentaires avec les connecteurs `and` et `or` (la négation est fournie par la fonction *not*() dans XPath). Voici une expression qui prend le dernier élément `B` fils de l'élément racine `A`, et ce seulement si ce dernier élément a un attribut `att1` dont la valeur est `a1`.

```
/A/B[@att1='a1' and position()=last()]
```

Composition de prédicats

L'utilisation d'une succession de prédicats marqués par les crochets [ ] dénote une *composition*. La différence essentielle d'une telle composition, par rapport à l'utilisation de connecteurs logiques `and` et `or`, est que l'ensemble de nœuds issus de l'évaluation du premier prédicat est pris comme contexte pour l'évaluation du second, et ainsi de suite. Reprenons l'exemple précédent pour clarifier les choses.

```
/A/B[@att1='a1' and position()=last()]
```

On pourrait très bien dans cet exemple inverser les deux termes du `and` sans changer la signification de l'expression : le prédicat s'évalue dans les deux cas en prenant pour contexte l'ensemble des nœuds obtenu à l'issue de `/A/B`. Prenons maintenant l'expression suivante :

```
/A/B[@att1='a1'][position()=last()]
```

Ici on va prendre tous les éléments `B` petit-fils de la racine du document qui ont un attribut `att1` valant `a1` (premier prédicat). Puis on évalue le second prédicat pour ne garder que le dernier parmi les nœuds qui ont satisfait le premier prédicat. Si on inverse les deux prédicats on a l'expression :

```
/A/B[position()=last()][@att1='a1']
```

Ici on commence par prendre le dernier nœud de type `B`, et on ne le garde que s'il a un attribut `att1` de type `a1`. On obtient donc un résultat différent.

Dans ce chapitre nous commençons notre étude approfondie de XSLT par les aspects relatifs à la *programmation*. Nous rassemblons sous ce terme toutes les instructions qui permettent de spécifier une *transformation* d'un *document source* vers un *document résultat*. Il n'est sans doute pas inutile d'expliciter les parties de XSLT qui ne relèvent pas directement de cette notion de programmation, et qui seront abordées plus loin dans ce livre :

1. nous ne nous intéressons pas au *style* de programmation XSLT ;
2. les spécificités du langage de sortie sont pour l'instant ignorées : les exemples seront basés sur la production de documents HTML simples ;
3. enfin nous délaissons pour l'instant les instructions qui sont orientées vers la mise en forme du document résultat.

Si vous avez lu les chapitres qui précèdent – ce qui est recommandé – vous devriez déjà avoir une compréhension intuitive d'une part des principes de base de XSLT, et d'autre part de la structuration d'un document XML. Nous nous plaçons donc directement dans la situation où un programme, le *processeur XSLT*, dispose en entrée de deux documents (figure 21) :

– le *document XML source*

– le *programme XSLT*

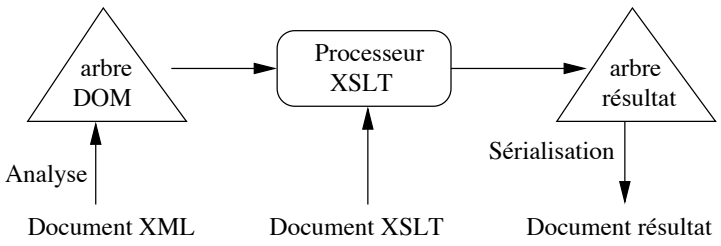

FIG. 21 – *Schéma du traitement d'une transformation XSLT*

Nous supposons que le document source a été analysé et se trouve disponible pour le processeur sous forme d'un arbre XML. Notez bien qu'il s'agit d'une vue « conceptuelle » et que, en pratique, les processeurs XSLT sont libres de s'appuyer sur une représentation interne de leur choix. Sur la base des informations trouvées dans le document source et des instructions du programme XSLT, le processeur produit un document résultat, qui peut également être vu comme un arbre construit au fur et à mesure de l'évaluation (nous donnerons un exemple d'une telle évaluation page 59). Une fois la transformation terminée, l'arbre résultat peut être sérialisé si besoin est pour être stocké, transmis sur le réseau, etc.

Nous prendrons comme exemple principal dans ce chapitre le document suivant, décrivant un cours (ou l'ébauche d'un cours…) basé sur le présent livre.

Exemple 3.2 *CoursXML.xml* : *Document XML pour le cours associé à ce livre*

```
<?xml version='1.0' encoding="ISO-8859-1"?>
```

```
<COURS CODE="TC234">
 <SUJET>Publication XSLT</SUJET>
  <ENSEIGNANTS>
    <!-- Enseignant responsable -->
    <NOM>Amann</NOM>
    <NOM>Rigaux</NOM>
  </ENSEIGNANTS>
  <PROGRAMME>
    <SEANCE ID="1">Documents XML</SEANCE>
    <SEANCE ID="2">Programmation XSLT</SEANCE>
    <ANNEE>2003</ANNEE>
  </PROGRAMME>
</COURS>
```

Nous commençons par décrire de manière globale le contenu d'un programme XSLT et des différents types d'éléments que l'on peut y trouver. Le chapitre consiste en une étude approfondie des *règles* (*templates*), puis des *instructions de contrôle* qui permettent d'effectuer des itérations et tests.

Nous donnons ensuite un exemple détaillé de l'évaluation d'un programme produisant un document HTML à partir du document *CoursXML.xml*. Cet exemple explique, étape par étape, comment un processeur XSLT construit le document résultat par choix de *nœuds courants* et application de *règles* à ces nœuds. Enfin nous concluons ce chapitre par quelques exemples de techniques de programmation avancée avec XSLT.

4 Programmes XSLT

Un programme XSLT *est* un document XML. Ce document doit bien entendu être bien formé (fermeture des balises, non-imbrication des balises ouvrante et fermante, etc). Les instructions spécifiques à XSLT doivent aussi respecter une certaine structure – relativement simple – et se limiter à des types d'éléments prédéfinis.

4.1 Structure d'un programme XSLT

Un programme XSLT doit respecter les règles syntaxiques XML. Il faut bien être conscient que cela s'applique non seulement aux *éléments XSLT* eux-mêmes, mais également aux éléments qui sont destinés à être insérés dans le résultat, que nous appellerons *élément littéraux* par la suite. Le document XSLT suivant par exemple n'est pas bien formé.

Exemple 4.1 *ExXSLT1.xsl* : *Un document XSLT mal formé*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

 <xsl:template match="COURS">
  <html>
    <head><title>Fiche du cours</title></head>
    <body bgcolor="white">
      <p>
       <h1>
         <i><xsl:value-of select="SUJET"/></i>
       </h1>
       <hr>
       <xsl:apply-templates/>
    </body>
```

```
  </html>
 </xsl:template>
</xsl:stylesheet>
```

Les éléments XSLT ne posent pas de problème, mais l'unique règle tente de produire le résultat suivant :

```
<html>
 <head><title>Fiche du cours</title></head>
 <body bgcolor="white">
   <p>
     <h1>
        <i>Publication XSLT</i>
     </h1>
     <hr>
 </body>
</html>
```

Bien qu'il s'agisse d'un document HTML correct (en tout cas conforme à la norme HTML 4.0), les éléments littéraux `<p>` et `<hr>` ne sont pas refermés. La présence de ce fragment HTML n'est donc pas acceptée dans le programme XSLT.

Cela illustre bien le fait que XSLT est avant tout destiné à transformer un document XML en un autre document XML, par production de fragments bien formés. La sortie d'un document « texte » qui ne serait pas du XML bien formé est possible mais implique des options spéciales qui seront étudiées dans le chapitre **??**.

L'élément `xsl:stylesheet`

L'élément racine d'un document XSLT est de type `xsl:stylesheet`. La forme habituelle de cet élément est :

```
<xsl:stylesheet
      version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

On trouve donc en général deux attributs :

1. L'attribut `version` est obligatoire et correspond au numéro de version de la « recommandation » XSLT publiée par le W3C. La seule recommandation à l'heure où ce livre est écrit est la 1.0, la version 2.0 étant à l'état de *Working Draft*.

2. L'attribut `xsl` définit l'espace de noms (*namespace*) `xsl` et l'associe à l'URI
   http://www.w3.org/1999/XSL/Transform

   Ce mécanisme permet de qualifier tous les éléments XSLT par le préfixe `xsl:`, et de considérer tous les autres éléments comme des éléments littéraux à inclure dans le document résultat. Les espaces de noms sont étudiés dans le chapitre **??**.

Il existe d'autres attributs possibles pour `xsl:stylesheet`. Ils seront présentés au fur et à mesure. Notez qu'il existe également un type d'élément `xsl:transform` qui est l'exact synonyme de `xsl:stylesheet`.

Remarque : Dans ce chapitre et les suivants nous avons choisi de ne pas donner une présentation exhaustive des attributs et des règles syntaxiques relatives à chaque élément XSLT, afin de rendre la lecture plus facile et de privilégier la « mise en contexte » de ces éléments. L'annexe **??** en revanche contient une référence XSLT avec la syntaxe de chaque élément : l'élément `xsl:stylesheet` par exemple est décrit page **??**.

Sous l'élément racine `xsl:stylesheet` on trouve le contenu du programme au sein duquel on distingue communément deux types d'éléments. Tout d'abord les *éléments de premier niveau* (*top-level elements*) sont fils de l'élément racine `xsl:stylesheet`. Le principal élément de ce type est `xsl:template`

qui est utilisé pour définir les règles XSLT. Les *instructions* constituent le second type d'élément : on les trouve essentiellement dans les *corps de règles*.

Éléments de premier niveau

Les types d'élément qui peuvent être fils de l'élément racine `<xsl:stylesheet>` sont dit « éléments de premier niveau » (*top-level éléments*). Ils sont tous (du moins ceux de la version 1.0) rassemblés dans le tableau 5, avec une courte description et la page où vous pouvez trouver la référence pour l'élément.

Type d'élément	Description
`xsl:attribute-set`	Définit un groupe d'attributs (page **??**)
`xsl:decimal-format`	Définit un format d'affichage pour les numériques (page **??**)
`xsl:import`	Import d'un programme XSLT (page **??**)
`xsl:include`	Inclusion d'un programme XSLT (page **??**)
`xsl:key`	Définit une clé pour un groupe de nœuds (page **??**)
`xsl:namespace-alias`	Définit des *alias* pour certains espaces de nom (page **??**)
`xsl:output`	Indique le format de sortie (HTML, XML, texte) (page **??**)
`xsl:param`	Définit un paramètre (page **??**)
`xsl:preserve-space`	Conserve les nœuds blancs (nœuds constitués uniquement d'espaces) pour certains éléments (page **??**)
`xsl:strip-space`	Supprime les nœuds blancspour certains éléments (page **??**)
`xsl:template`	Définit une règle XSLT (page **??**)
`xsl:variable`	Définit une variable XSLT (page **??**)

TAB. 5 – *Éléments XSLT de premier niveau*

Tous ces types d'élément ne sont pas de la même importance, et tous ne couvrent pas le même type de fonctionnalités. Même s'il est difficile d'établir une classification qui ne soit pas, au moins en partie, arbitraire, on peut grossièrement distinguer :

– les éléments qui affectent la production du document résultat ;

– les éléments qui s'appliquent à la transformation du document source.

Dans ce chapitre nous décrivons les éléments qui relèvent de la seconde catégorie : `xsl:template`, `xsl:param` et `xsl:variable`. Les éléments relatifs à la production du résultat sont pour l'essentiel présentés dans le prochain chapitre.

L'ordre des éléments de premier niveau n'a pas d'impact sur le comportement du programme XSLT (à l'exception de `xsl:import` : voir ci-dessous), et on peut les arranger dans l'ordre qui semble le plus convenable pour la lecture. Cette particularité constitue l'un des aspects du caractère « déclaratif » de XSLT : on indique les actions à effectuer en présence de certains événements, à charge pour le processeur de déclencher ces actions de manière appropriée.

Quels sont les autres fils possibles de `xsl:stylesheet` ? Il est interdit d'avoir des nœuds de type **Text**, autrement dit du texte placé immédiatement sous l'élément racine. En revanche on peut trouver des commentaires et des instructions de traitement, les premiers étant ignorés, et le second pris en compte seulement s'ils contiennent des instructions reconnues par le processeur.

Dans le même esprit, on peut trouver au premier niveau des éléments dont le type est spécifique au processeur XSLT. En général ces éléments se distinguent par un préfixe comme `xalan:`, `msxml3:` ou `saxon:`. L'introduction de ce type d'élément non normalisé risque de poser des problèmes de portabilité si on souhaite qu'un programme XSLT puisse être traité par n'importe quel processeur (voir instruction `xsl:fallback`, page **??**).

Enfin le concepteur d'un programme peut lui-même définir ses propres éléments de premier niveau, en les caractérisant par un préfixe. Cela n'a d'intérêt que pour introduire du contenu (par exemple : des codifications, des messages en plusieurs langues, des paramètres…) au sein d'un programme, en plus des instructions de ce programme. Ces éléments sont ignorés par le processeur, et leur utilité est douteuse. On peut en tout cas toujours s'en passer et utiliser d'autres mécanismes.

En résumé les éléments du tableau 5 constituent la référence des types d'éléments, fils de `xsl:stylesheet`, reconnus par n'importe quel processeur XSLT conforme à la recommandation XSLT 1.0. Un des ajouts importants prévus dans la version 1.1 était un élément `xsl:script` qui permet d'introduire des parties programmées (en Java ou JavaScript) dans un programme XSLT. Cette extension risque de soulever des problèmes de portabilité et a fait l'objet d'un débat au sein du groupe de normalisation qui a décidé de la supprimer dans la version 2.0 de XSLT (laquelle est toujours au stade de *Working Draft* au moment de l'impression de ce livre).

Instructions XSLT

L'exécution d'un programme XSLT consiste à déclencher (ou *instancier*) des règles définies par des éléments de premier niveau `xsl:template`. L'instanciation d'une règle consiste à produire un fragment du document résultat en fonction d'éléments littéraux, de texte et d'instructions XSLT.

Toute combinaison de tels constituants est nommée un *corps de règle*. Alors que les deux premières catégories (éléments littéraux et texte) sont insérés dans le document résultat, les *instructions XSLT* sont interprétées par le processeur XSLT. Voici par exemple un corps de règle que nous avons rencontré dans le chapitre introductif : il crée une liste HTML avec les séances d'une salle de cinéma :

```
<h3>Séances</h3>
<ol>
  <xsl:for-each select="SEANCES/SEANCE">
    <li><xsl:value-of select="."/></li>
  </xsl:for-each>
</ol>
```

On trouve donc des éléments littéraux (balises `<h3>`, `<ol>`, `<li>` et leurs fermetures), des nœuds de type **Text** (par exemple le nœud fils de `<h3>` dont le contenu est `Séances`), et enfin des instructions, éléments dont le nom est préfixé par `xsl:`. La notion de corps de règle est assez large et recouvre toute partie du programme destinée à produire un fragment du document résultat.

Le tableau 6 donne la liste de toutes les instructions. Toutes ces instructions font partie de la recommandation XSLT 1.0 à l'exception de `xsl:result-document` qui est prévu dans XSLT 2.0, mais déjà largement présent dans les implantations actuelles. Dans la version 2.0 de XSLT cette instruction ajoute un attribut `format` qui permet de spécifier le format de sérialisation du fragment généré (`xml`, `html`, `xhtml`, `text`). Comme pour les éléments de premier niveau, nous donnons une brève description, ainsi que la page de l'annexe **??** où vous pouvez trouver la référence de l'instruction.

On peut noter que `xsl:variable` et `xsl:param` sont les seuls types d'élément à apparaître à la fois comme élément de premier niveau et comme instruction. L'emplacement de l'élément définit la *portée* de la variable et du paramètre : voir page **??**.

Si l'on s'en tient à la classification déjà faite précédemment, on constate que les instructions relatives à la transformation du document source plutôt qu'à la production du résultat sont `xsl:apply-templates` et `xsl:call-template` qui toutes deux déclenchent des règles, `xsl:choose`, `xsl:for-each`, `xsl:if`, `xsl:message` et `xsl:variable` qui correspondent à peu près aux tests, itérations, sorties à l'écran et définitions de variables que l'on trouve dans la programmation plus classique. Ces « instructions de contrôle » sont présentées page **??**.

4.2 Modularité : `xsl:import` et `xsl:include`

Jusqu'à présent nous avons toujours considéré qu'un programme XSLT était contenu dans un seul fichier. Heureusement il est possible, comme dans tout langage de programmation digne de ce nom, d'intro-

Type d'élément	Description
`xsl:apply-imports`	Permet d'appliquer une règle importée, tout en la complétant par un nouveau corps (page **??**)
`xsl:apply-templates`	Déclenche l'application de règles (page **??**)
`xsl:attribute`	Insère un attribut dans un élément du document résultat (page **??**)
`xsl:call-template`	Appelle une règle par son nom (page **??**)
`xsl:choose`	Structure de test équivalente au `switch` d'un langage comme Java ou C++ (page **??**)
`xsl:comment`	Insère un nœud **Comment** dans le document résultat (page **??**)
`xsl:copy`	Copie un nœud du document source dans le document résultat (page **??**)
`xsl:copy-of`	Copie un nœud, ainsi que tous ses descendants (page **??**)
`xsl:result-document`	Permet de créer plusieurs documents résultats : ajouté dans XSLT 2.0, mais reconnu par la plupart de processeurs XSLT actuels, éventuellement sous un autre nom (page **??**)
`xsl:element`	Insère un nœud **Element** dans le document résultat (page **??**)
`xsl:fallback`	Règle déclenchée si le processeur ne reconnaît pas une instruction (page **??**)
`xsl:for-each`	Pour effectuer des itérations (page **??**)
`xsl:if`	Pour effectuer un branchement conditionnel (page **??**)
`xsl:message`	Pour produire un message pendant le traitement XSLT (page **??**)
`xsl:number`	Permet de numéroter les nœuds du document résultat (page **??**)
`xsl:variable`	Permet de définir un paramètre (page **??**)
`xsl:processing-instruction`	Insère un nœud **ProcessingInstruction** dans le document résultat (page **??**)
`xsl:text`	Insère un nœud **Text** dans le document résultat (page **??**)
`xsl:value-of`	Évalue une expression XPath et insère le résultat (page **??**)
`xsl:variable`	Permet de définir une variable (page **??**)

TAB. 6 – *Instructions XSLT*

duire une certaine modularité afin de répartir les instructions dans plusieurs fichiers d'une taille raisonnable, et surtout de pouvoir *réutiliser* une règle dans plusieurs programmes.

XSLT fournit deux éléments de premier niveau pour intégrer des fichiers afin de constituer des programmes : ce sont `xsl:import` et `xsl:include`. Signalons tout de suite que les notions de « librairie » constituée de « modules » qui ne peuvent être utilisés indépendamment, et de « programme principal » constitué par assemblage de librairies, n'existent pas en XSLT. Rien ne distingue, du point de vue syntaxique, un programme destiné à être inclus d'un programme incluant ou important d'autres fichiers. Tous doivent être des programmes XSLT conformes, avec un élément racine `xsl:stylesheet`.

L'assemblage de plusieurs programmes peut créer des *conflits*. Au moment de l'évaluation plusieurs règles peuvent s'appliquer aux mêmes nœuds. Afin de déterminer la règle à appliquer, XSLT utilise un système de *priorités* que nous décrivons page 52. La distinction entre `xsl:include` et `xsl:import` est justement relative à la méthode appliquée pour choisir une règle en cas de conflit :

1. dans le cas de `xsl:import` le processeur affecte aux règles importées une *préséance* [1]. inférieure à celle du programme importateur ;
2. dans le cas de `xsl:include`, le processeur traite les règles du programme inclus sans les distinguer, en terme de préséance, de celles du programme principal : tout se passe comme si `xsl:include` était simplement remplacé, avant l'évaluation, par le contenu du programme référencé.

Une autre différence entre `xsl:import` et `xsl:include` est que `xsl:import` doit apparaître avant tout *autre* élément de premier niveau dans un programme XSLT. On peut trouver en revanche plusieurs `xsl:import` puisque rien n'empêche d'importer plusieurs fichiers dans un programme.

À l'exception des différences ci-dessus, `xsl:include` et `xsl:import` sont très proches. Dans les deux cas un fichier inclus/importé peut lui-même inclure ou importer d'autres fichiers, tant qu'il n'existe pas de cycle qui mènerait un fichier à s'inclure lui-même. Les deux types d'élément partagent également la même syntaxe : le nom ou l'URI du fichier inclus est référencé dans l'attribut `href` :

```
<xsl:import href="Programme.xsl"/>
```

Prenons un premier exemple pour illustrer l'intérêt du mécanisme. Le fichier *ExXSLTImport.xsl* contient une première règle qui s'applique à un élément racine de type `COURS`, produit un « squelette » de document HTML avec le sujet du cours dans la balise `<TITRE>`, et enfin déclenche un appel de règles avec `xsl:apply-templates`.

Exemple 4.2 *ExXSLTImport.xsl* : *Un programme à importer*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

 <xsl:strip-space elements="*"/>

 <xsl:template match="/">
   <html>
     <head>
      <title>
        <xsl:value-of select="COURS/SUJET"/>
      </title>
     </head>
     <body bgcolor="white">
          <xsl:apply-templates/>
     </body>
   </html>
 </xsl:template>
```

1. Nous utilisons le terme (un peu désuet) de préséance pour traduire l'anglais *precedence*, et distinguer cette notion de celle de *priorité* d'application des règles

```
 <xsl:template
      match="SUJET | ENSEIGNANTS | PROGRAMME"/>
</xsl:stylesheet>
```

La seconde règle s'applique aux fils de <COURS> et ne fait rien. Le programme est donc très « neutre » vis-à-vis du contenu du document source. Il se contente de produire le « cadre » de présentation HTML ci-dessous.

Exemple 4.3 *ExXSLTImport.html* : *Le document HTML produit par ExXSLTImport.xsl*

```
<html>
<head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Publication XSLT</title>
</head>
<body bgcolor="white"></body>
</html>
```

On peut imaginer que cette règle pourrait produire un squelette HTML beaucoup plus sophistiqué, avec menus, *frames*, contrôles JavaScript, et tout un attirail graphique rendant la présentation beaucoup plus attrayante. L'intérêt est qu'une fois cette mise en page réalisée et insérée dans l'unique règle, on peut la *réutiliser* en l'important dans d'autres programmes.

Voici un programme qui affiche la liste des enseignants du cours. Il commence par importer le fichier *ExXSLTImport.xsl*, ce qui dispense de définir une règle s'appliquant à la racine du document. Ensuite on *redéfinit* les règles s'appliquant au sous-arbre des enseignants pour produire une liste avec les noms. Ces règles ont une préséance supérieure à celles du fichier importé qui ne produisaient aucun résultat.

Exemple 4.4 *ExXSLTPrincipal.xsl* : *Un programme importateur de ExXSLTImport.xsl*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:import href="ExXSLTImport.xsl"/>

  <xsl:template match="COURS/ENSEIGNANTS">
    <ol><xsl:apply-templates select="NOM"/></ol>
  </xsl:template>

  <xsl:template match="NOM">
    <li><xsl:value-of select="."/></li>
  </xsl:template>

</xsl:stylesheet>
```

La préséance doit être distinguée du niveau de *priorité* d'une règle, notion que nous exposerons page 52. La préséance s'applique à l'importation de programme, et elle est toujours inférieure pour les règles du programme importé par rapport aux règles du programme importateur.

Cette règle se généralise assez simplement à l'importation de plusieurs documents de la manière suivante :

- si un document D_1 est importé *avant* un document D_2, alors *toutes* les règles de D_2 ont une préséance supérieure à celles de D_1 ;
- si une règle r_1 a une préséance supérieure à r_2, elle-même supérieure à r_3, alors r_1 a une préséance supérieure à r_3 (transitivité).

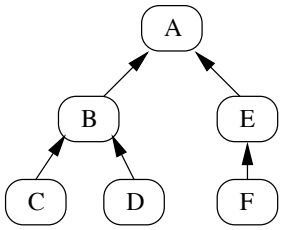

FIG. 22 – *Exemple de plusieurs imports*

Prenons un exemple simple, illustré dans la figure 22. le document principal `A` importe d'abord `B`, qui lui-même importe `C` et `D`, puis `A` importe `E` qui importe `F`.

Les règles de `C` ont une préséance inférieure à celles de `B`, et donc inférieures à celles de `A`. C'est vrai aussi des règles de `D`, mais elles ont une préséance supérieure à celles de `C` puisque `D` est importé après `C`. Enfin les règles de `F` ont une préséance inférieure à celles de `E`, mais supérieures à toutes celles issues de `B`, `C` ou `D` puisque `F` a été importé après. En résumé l'ordre (descendant) de préséance pour ces documents est `A`, `E`, `F`, `B`, `D` et `C`.

Remarque : On peut voir dans cet arbre que l'ordre descendant de préséance pour ces documents correspond à un parours en profondeur de droite à gauche.

L'ordre de préséance est pris en compte au moment où le processeur recherche les règles applicables à un nœud. Si plusieurs règles sont candidates, seules celles ayant l'ordre de préséance le plus élevé sont conservées et prises en compte. Au sein de ces règles, c'est alors l'ordre de *priorité* qui s'applique.

Le mécanisme d'importation s'apparente à la surcharge de méthode en programmation orienté-objet. Dans les deux cas on *réutilise* les parties génériques et on *redéfinit* les parties spécifiques à une situation donnée. Un inconvénient possible est que la totalité de la règle importée est remplacée par la règle du programme local, alors que l'on peut souhaiter parfois exécuter cette règle importée tout en l'enrichissant de quelques instructions complémentaires. C'est ce que permet l'instruction XSLT `xsl:apply-imports`, présentée page **??**.

4.3 Application d'un programme XSLT

Un programme XSLT s'applique à un document XML pour en effectuer une transformation. Comment et quand déclencher cette transformation ? Si on prend comme application de base la publication d'informations sur le Web, on peut envisager plusieurs situations :

1. *serveur statique* : les transformations sont effectuées en *batch* afin de produire les pages HTML à partir d'un ou plusieurs documents XML ; les pages HTML statiques sont alors transférées sur le serveur ;

2. *serveur dynamique* : cette fois la transformation s'effectue à la demande en fonction des requêtes HTTP ;

3. *client dynamique* : le document XML et le programme XSLT sont transmis au client qui effectue la transformation.

La première solution est simple et ne pose pas de problème de performance puisque les pages HTML sont directement disponibles. Elle peut souffrir d'un certain manque de réactivité, le contenu étant figé entre deux transformations.

La plupart des processeurs permettent des transformations « statiques » à partir de la ligne de commande. Voici par exemple une transformation avec Xalan, le processeur XSLT de la fondation Apache

dont l'installation est décrite dans l'annexe **??**. La commande transforme le document *Alien.xml* avec le programme *Film.xsl*, et produit la page HTML *Alien.html*.

```
java org.apache.xalan.xslt.Process -in Alien.xml
     -xsl Film.xsl -out Alien.html
```

La seconde solution permet une adaptation aux évolutions en temps réel, mais risque de soulever des problèmes de performances si de très nombreuses transformations doivent être effectuées sur des documents volumineux. Plusieurs environnements de publication basée sur ce mécanisme existent : Cocoon par exemple, également produit par la fondation Apache, se base sur le processeur Xalan pour déclencher des transformations dynamiquement en fonction de la requête HTTP, du type de navigateur utilisé, etc (voir annexe **??**).

Enfin la dernière solution consiste à effectuer la transformation sur le client. À l'heure actuelle seuls les navigateurs Mozilla et Internet Explorer 5 (IE5) disposent d'un processeur XSLT intégré. Outre les problèmes de compatibilité avec le navigateur, cette solution présente l'inconvénient de transmettre toutes les informations du document XML au client, ce qui n'est pas forcément souhaitable.

L'instruction `<?xml-stylesheet?>`

Le moyen le plus courant pour associer un document XML à un programme XSLT est une instruction de traitement `<?xml-stylesheet?>` qui doit apparaître dans le prologue du document à transformer. Voici par exemple comment indiquer, dans le document *Alien.xml*, que le programme associé est *Film.xsl*.

```
<?xml-stylesheet href="Film.xsl" type="text/xsl"?>
```

C'est une instruction de traitement, dont le nom est `xml-stylesheet`, et le contenu une liste d'attributs[2]. Ces attributs sont :

- `href`, qui donne l'URI du programme XSLT (ce peut être un fichier local, ou un fichier accessible sur le Web) ;
- `type`, qui donne le type MIME du programme XSLT : `text/xml` ou `text/xsl` sont des choix reconnus, le premier étant celui officiellement préconisé par le W3C ;
- `title`, une chaîne de caractères qui peut être utilisée pour permettre le choix du programme à appliquer ;
- `media`, qui indique le format du document produit par la transformation ;
- `alternate`, qui vaut `no` si le programme XSLT référencé doit être utilisé en priorité, ou `yes` sinon.

Les deux premiers attributs sont obligatoires, les autres servant essentiellement à effectuer un choix quand plusieurs instructions `<?xml-stylesheet?>` sont présentes. L'interprétation de ces attributs et les valeurs qu'ils peuvent prendre sont en parties dépendantes du processeur ou de l'environnement. Voici par exemple comment on peut indiquer à l'environnement de publication Cocoon une transformation par défaut avec le programme *Film.xsl*, et, seulement dans le cas d'un dialogue avec un terminal WML, le programme *FilmWML.xsl*.

```
<?xml-stylesheet href="Film.xsl" type="text/xsl"?>
<?xml-stylesheet href="FilmWML.xsl" type="text/xsl" media="wap"?>
```

L'utilisation de cette instruction n'est pas toujours souhaitable car elle va dans une certaine mesure à l'encontre d'une séparation stricte du document XML et des programmes de transformation. Chaque processeur définit ses propres modes d'application de programme XSLT. Signalons également que la recommandation XSLT prévoit la possibilité d'inclure directement le programme XSLT dans le document XML à transformer (*embedded stylesheets*). Dans ce cas il n'y a plus de document XSLT indépendant, ce qui empêche de réutiliser un programme pour d'autres documents XML de même structure.

2. Une instruction de traitement n'étant pas un élément, on ne peut pas dire qu'elle a des « attributs » au sens strict du terme. Il ne s'agit ici que d'un choix de présentation afin de clarifier le contenu de l'instruction, mais celui-ci pourrait être constitué de texte non structuré.

5 Les règles XSLT

Nous décrivons dans cette section les *règles* XSLT et leur utilisation. Il s'agit d'un des aspects essentiels de XSLT puisque dans beaucoup de cas on peut écrire un programme uniquement sous la forme de règles transformant certains nœuds et produisant une partie du résultat. La définition d'une règle s'effectue avec l'élément `xsl:template`, et le déclenchement avec `xsl:apply-templates` ou `xsl:call-template`.

Une règle peut être déclenchée ou *instanciée* soit par son nom, soit en donnant la catégorie des nœuds du document source auxquels elle s'applique. Cette catégorie est spécifiée par une sous-classe des expressions XPath désignée par le terme *pattern* dans la recommandation XSLT. Nous commençons par présenter ces *patterns* avant de revenir en détail sur les règles et leur instanciation.

Important : L'instanciation d'une règle (ainsi que des instructions contenues dans les règles) est *toujours* associée à l'un des nœuds du document source que nous désignons par le terme *nœud courant* dans tout ce qui suit. Attention, il ne faut pas confondre le *nœud courant* d'une règle ou instruction XSLT avec le *nœud contexte* d'une étape XPath. On trouve dans une règle des expressions XPath, absolues ou relatives. Dans le cas d'une expression relative, le premier nœud contexte est le nœud courant de la règle, mais ce nœud contexte change ensuite à chaque étape, tandis que le nœud courant reste toujours celui associé à la règle. On peut d'ailleurs y faire référence dans l'expression XPath avec la fonction *current*(), décrite dans l'annexe, page **??**.

5.1 Les *patterns*

Le déclenchement des règles basé sur les *patterns* n'est pas la partie la plus évidente de XSLT. Nous prenons un exemple simple pour commencer.

Un exemple

Le programme suivant recherche et affiche les noms des enseignants, les attributs `ID` et l'intitulé des séances de cours.

Exemple 5.1 *Pattern1.xsl* : *Exemple pour illustrer les patterns*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" encoding="ISO-8859-1"/>

  <xsl:template match="/">
    <xsl:apply-templates select="//NOM"/>
    <xsl:apply-templates select="//SEANCE/@ID"/>
    <xsl:apply-templates select="//SEANCE"/>
  </xsl:template>

  <xsl:template match="NOM">
    <NOM><xsl:value-of select="."/></NOM>
  </xsl:template>

  <xsl:template match="@ID">
    <IDSEANCE><xsl:value-of select="."/></IDSEANCE>
  </xsl:template>

  <xsl:template match="PROGRAMME/SEANCE">
    <SEANCE><xsl:value-of select="."/></SEANCE>
  </xsl:template>

</xsl:stylesheet>
```

Appliqué à notre document *CoursXML.xml* (page 34), on obtient le document suivant :

Exemple 5.2 *Pattern1.xml* : *Le résultat du programme*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<NOM>Amann</NOM><NOM>Rigaux</NOM>
<IDSEANCE>1</IDSEANCE>
<IDSEANCE>2</IDSEANCE>
<SEANCE>Documents XML</SEANCE>
<SEANCE>Programmation XSLT</SEANCE>
```

On trouve dans le programme *Pattern1.xsl* plusieurs expressions XPath. Certaines apparaissent comme valeur de l'attribut `match` de l'élément `xsl:template`, d'autres comme valeur de l'attribut `select` de `xsl:apply-templates`. Elles jouent un rôle différent :

1. l'expression dans l'élément `xsl:apply-templates` sert à désigner un ensemble de nœuds, et constitue donc une utilisation classique de XPath ;
2. l'expression dans l'élément `xsl:template` exprime en revanche une condition sur les nœuds qui vont permettre de déclencher la règle.

Les deux éléments fonctionnent solidairement : `xsl:apply-templates` permet de constituer un ensemble de nœuds issus du document source. Puis, pour chacun de ces nœuds, le processeur XSLT va chercher la règle (on suppose qu'il n'y en a qu'une pour l'instant) telle que le nœud « satisfait » la condition exprimée par l'attribut `match`.

Il reste à définir la satisfaction d'une condition exprimée par l'expression de l'attribut `match` – appelons cette expression *pattern* à partir de maintenant. On peut l'exprimer assez simplement (!) de la manière suivante : un nœud N *satisfait un pattern* s'il existe quelque part dans l'arbre du document source un nœud C tel qu'en évaluant le *pattern* avec C comme nœud contexte, on obtienne un ensemble qui contient N.

Reprenons notre programme *Pattern1.xsl*. Le premier ensemble de nœuds considéré par le processeur XSLT est l'ensemble qui contient uniquement la racine du document. On cherche donc la (ou les) règle(s) dont le *pattern* est satisfait par ce nœud racine. Il n'en existe qu'une, c'est la première règle. Le *pattern* spécifié dans l'attribut `match` est l'expression « / », désignant un chemin absolu qui retourne toujours la racine du document.

Le corps de la règle déclenchée consiste en trois éléments `xsl:apply-templates`. Pour chacun des nœuds choisis par ces instructions, le processeur XSLT va chercher la (ou les) règles dont le nœud satisfait le *pattern* :

– Le premier <`xsl:apply-templates`> constitue, par évaluation de l'expression XPath `//NOM`, l'ensemble de *tous* les nœuds de type `NOM` dans le document. Seule la deuxième règle convient. En prenant en effet comme nœud contexte le nœud de type `ENSEIGNANTS` et en évaluant le *pattern* `NOM`, on obtient bien tous les nœuds de type `NOM` (rappelons que le *pattern* utilisé est une abréviation de `child::NOM`).

  Remarquez que cette règle se déclenche pour *tous* les éléments de type `NOM`, quelle que soit leur position dans l'arbre, puisqu'il suffit de prendre le père de ces nœuds (quel qu'il soit) et d'évaluer le *pattern* pour que la condition soit satisfaite.

– La deuxième instruction <`xsl:apply-templates`> du corps de la première règle constitue, par évaluation de l'expression XPath `//SEANCE/@ID`, l'ensemble de tous les attributs `ID` des nœuds de type `SEANCE`. Cette fois, c'est la troisième règle qui est déclenchée. En prenant en effet le père <`SEANCE`> de chaque attribut et en évaluant le *pattern* `@ID`, on obtient bien l'attribut lui-même. Comme précédemment, remarquez que cette règle se déclenche pour *tous* les attributs `ID`, quel que soit le type de l'élément auquel ils appartiennent.

– Le dernier élément <`xsl:apply-templates`> de la première règle constitue, par évaluation de l'expression XPath `//SEANCE`, l'ensemble de tous les nœuds de type `SEANCE`. Cette fois-ci, c'est la dernière règle qui va être déclenchée. Prenons en effet le premier nœud <`SEANCE`>

comme nœud contexte. En considérant le père de ce nœud, c'est-à-dire l'élément <PROGRAMME>, comme nœud contexte et en évaluant le *pattern* `PROGRAMME/SEANCE`, on obtient un ensemble vide. On ne peut donc pas dire, à ce stade, que la condition est satisfaite. En prenant maintenant le grand-père, le nœud <COURS>, et en évaluant `PROGRAMME/SEANCE`, qui est un raccourci de `child::PROGRAMME/child::SEANCE`, on obtient bien un ensemble qui contient notre premier nœud <SEANCE>, et le *pattern* de la règle est satisfait.

On peut noter cette fois que la règle ne s'applique qu'aux éléments de type `SEANCE` ayant pour père un élément de type `PROGRAMME`. Cette règle est donc moins générale que les précédentes qui ne dépendent pas du type du père.

En conclusion un *pattern* dans une règle définit la « situation » des nœuds de l'arbre qui vont pouvoir déclencher la règle. Cette situation peut être très générale : un *pattern* `NOM` définit une règle applicable à tous les éléments de type `NOM`, quel que soit leur position, et quel que soit le type de leurs parents. Un *pattern* comme `PROGRAMME/SEANCE` est déjà plus restrictif. À l'extrême, un *pattern* peut être un chemin absolu XPath, et la règle ne s'applique plus alors qu'à des sous-arbres très particuliers.

Expressions XPath et *patterns*

Pourquoi distinguer les *patterns* et les expressions XPath en général ? La réponse se situe dans l'interprétation des *patterns* qui consiste à *chercher un nœud contexte* à partir duquel le résultat du *pattern* contient le nœud courant. Ceci a deux conséquences :

1. l'expression doit toujours retourner un ensemble de nœuds (`node-set`) ;
2. la recherche du nœud contexte peut devenir très complexe si on acceptait n'importe quelle expression XPath donnant un ensemble de nœuds : il faudrait, dans certains cas, pour vérifier si un nœud satisfait une expression, chercher un nœud contexte adapté parmi *tous* les nœuds du document source. Nous verrons plus loin que la définition des *patterns* permet de limiter cet espace de recherche et d'implantation avec un algorithme plus efficace.

La règle suivante par exemple n'est pas valide : l'attribut `match` est une expression XPath correcte mais dont le résultat n'est pas un ensemble de nœuds. La condition de déclenchement de la règle n'a donc pas de sens.

```
<!-- Pas bon                        -->
<xsl:template match="1">
      ...
</xsl:template>
```

La règle suivante est interdite car elle utilise une expression XPath qui n'est pas un *pattern* :

```
<!-- Pas bon                        -->
<xsl:template match="B/preceding::node()">
      ...
</xsl:template>
```

Pour déterminer si cette règle doit être déclenchée quand on rencontre un nœud N, il faut vérifier si on trouve un nœud de type `B` qui est un successeur de N. Pour l'élément racine par exemple, on peut être obligé de parcourir tout l'arbre XML avant de trouver un tel nœud.

Il est clair qu'autoriser toutes les expressions XPath engendrerait des problèmes de performance, ainsi que des difficultés de développement des processeurs XSLT. Les seuls axes autorisés dans les *pattern* sont donc `child` et `attribute` ainsi que leur notation abrégée. De plus, il est possible d'utiliser la notation `//` de `/descendant-or-self::node()/`. Attention, il n'est pas possible d'utiliser l'axe `descendant-or-self` explicitement dans un *pattern*.

Cette restriction permet de décrire l'évaluation d'un *pattern* P par rapport à un nœud courant [3] N selon l'algorithme suivant. Cet algorithme fait d'abord une distinction entre le cas d'un *pattern* absolu et le cas d'un *pattern* relatif :

1. Si P est absolu, prendre la racine comme nœud contexte et évaluer P ; si N fait partie du résultat, alors N satisfait P ;
2. Sinon
   (a) prendre N comme nœud contexte et évaluer P ; si N fait partie du résultat, alors N satisfait P ;
   (b) sinon prendre le père de N, et recommencer l'évaluation ; si N fait partie du résultat, alors N satisfait P ;
   (c) sinon recommencer en parcourant les ancêtres de N jusqu'à la racine du document.

Les *patterns* absolus sont généralement plus rares que les *patterns* relatifs. Évidemment il est possible dans la plupart des cas d'éviter le parours de tous les ancêtres du nœud courant. Il faut cependant être conscient que l'évaluation d'un *pattern* doit être faite pour tous les nœuds désignés par `xsl:apply-templates`, et qu'elle peut être longue si l'expression est complexe.

Quelques exemples

Voici quelques exemples de *patterns* avec leur signification. On peut constater qu'il n'existe pas de restriction sur les filtres ou les prédicats utilisés dans un *pattern*.

– le *pattern* absolu `/COURS/ENSEIGNANTS` sera satisfait par tous les nœuds de type `ENSEIGNANTS` fils d'un élément racine de type `COURS` ;
– `//SEANCE[@ID=2]` ou `SEANCE[@ID=2]` seront satisfait par tout nœud de type `SEANCE` ayant un attribut `ID` valant 2 (vérifiez l'équivalence des deux expressions) ;
– `NOM[position()=2]` sera satisfait par tout nœud qui est le deuxième fils de type `NOM` de son père ;
– `*[position()=2][name()="NOM"]` sera satisfait par tout nœud de type `NOM`, second fils de son père ;
– `/COURS/@CODE[.="TC234"]` sera satisfait par l'attribut `CODE`, fils de l'élément racine <`COURS`>, et dont la valeur est `TC234`.

5.2 Règles

Une règle est définie par un élément de premier niveau `xsl:template`. Cet élément comprend des attributs qui décrivent les conditions de déclenchement de la règle, et un contenu ou *corps de règle* décrivant le texte à produire quand la règle est déclenchée (ou « instanciée » pour utiliser le vocabulaire XSLT). L'instanciation d'une règle s'effectue toujours pour un nœud du document source désigné par le terme de *nœud courant*. Toute expression *relative* XPath utilisée dans le corps de la règle, que ce soit pour effectuer des tests, extraire des informations ou appeler d'autres règles, prendra ce nœud courant comme contexte pour la première étape.

3. Rappelons nous appelons *nœud courant* celui pour lequel la règle a été déclenchée, et *nœud contexte* celui qui sert de point de départ à l'évaluation d'une expression XPath.

L'élément `xsl:template`

Les attributs d'une règle sont au nombre de quatre [4] :

1. `match` est le *pattern* désignant les nœuds de l'arbre XML pour lesquels la règle peut se déclencher ;
2. `name` définit une *règle nommée* qui pourra être appelée directement par son nom ;
3. `mode` permet de définir des *catégories* de règles, à appeler dans des circonstances particulières ;
4. enfin `priority` donne une priorité explicite à la règle.

Tous les attributs sont optionnels, mais soit `name`, soit `match` doit être défini. Dans le premier cas, la règle sera appelée par `xsl:call-template`. Dans le second cas, c'est au processeur de déterminer, en fonction du nœud courant et du *pattern* (ainsi que de la résolution des conflits éventuels), si la règle doit être appelée. Ces deux types de déclenchement correspondent à deux styles de programmation différents. Le premier cas (appel de règle) s'apparente plutôt à une programmation par appel de fonction, tandis qu'avec le second, plus « déclaratif », on se contente de « déclarer » ce qu'on veut obtenir, à charge pour le processeur de faire le choix de la règle appropriée.

Règles par défaut

XSLT définit un ensemble de règles par défaut qui sont appliquées quand aucune règle du programme n'est sélectionnée. La première règle par défaut s'applique à la racine du document et à tous les éléments. Elle se contente de déclencher un appel de règle pour tous les fils du nœud courant.

```
<xsl:template match="* | /">
  <xsl:apply-templates/>
</xsl:template>
```

Rappelons qu'un appel à `xsl:apply-templates` sans attribut `select` est équivalent à la sélection de tous les fils du nœud courant. Cette règle est utilisée pour tous les `xsl:apply-templates` qui ne trouvent pas de règle à déclencher, y compris quand un mode est indiqué (voit page 49).

La seconde règle par défaut s'applique aux nœuds de texte et aux attributs. Elle insère le contenu textuel de ces nœuds dans le document résultat.

```
<xsl:template match="text() | @*">
  <xsl:value-of select="."/>
</xsl:template>
```

Enfin la dernière règle par défaut s'applique aux commentaires et aux instructions de traitement. Le comportement par défaut est de les ignorer. La règle ne fait donc rien :

```
<xsl:template match="processing-instruction() | comment()"/>
```

Voici le programme XSLT minimal : il ne contient aucune règle, à part les règles par défaut qui sont implicites.

Exemple 5.3 *Defaut.xsl* : *Un programme XSLT minimal*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform"/>
```

4. La syntaxe complète de l'élément (ainsi que de tous ceux que nous présentons par la suite) est donnée dans l'annexe **??**, page **??**.

L'application de ce programme à notre document *CoursXML.xml* donne le résultat suivant :

Exemple 5.4 *Defaut.xml* : *Le résultat des règles par défaut*

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

 Publication XSLT


    Amann
    Rigaux


    Introduction
    Documents XML
    Programmation XSLT
```

Les règles par défaut se sont donc appliquées. Elles ont permis de parcourir tous les éléments du document, en produisant le contenu des nœuds de type **Text**. Notez que les attributs ne sont pas sélectionnés par les expressions XPath dans les `xsl:apply-templates` des règles par défaut, et que leur contenu n'apparaît donc pas.

5.3 Déclenchement de règles avec `xsl:apply-templates`

L'élément `xsl:apply-templates` désigne un ensemble de nœuds avec une expression XPath, et demande l'application d'une règle pour chaque nœud. L'expression (quand elle n'est pas absolue) est toujours évaluée en prenant comme nœud contexte le nœud courant pour lequel la règle contenant `xsl:apply-templates` a été instanciée.

Cet élément a deux attributs, tous deux optionnels :

1. `select` contient l'expression XPath désignant les nœuds à traiter ;
2. `mode` est la catégorie des règles à considérer.

L'expression de l'attribut `select` doit toujours ramener un ensemble de nœuds. Sa valeur par défaut est `child::node()`, autrement dit tous les fils du nœud courant, quel que soit leur type, à l'exception comme d'habitude des attributs (voir chapitre **??**).

Instanciation des règles

Pour chaque nœud de l'ensemble désigné par l'expression XPath, le processeur va rechercher la règle à appliquer. Il n'y a pas de raison *a priori* pour que la même règle soit appliquée à tous les nœuds. Prenons l'exemple du programme suivant, appliqué à notre exemple *CoursXML.xml*.

Exemple 5.5 *ApplyTemplates.xsl* : *Exemple de* `xsl:apply-templates`

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" encoding="ISO-8859-1"/>

  <xsl:template match="/">
    <xsl:apply-templates select="//ENSEIGNANTS"/>
  </xsl:template>
```

```
  <xsl:template match="ENSEIGNANTS">
    <xsl:comment>
         Application de la règle ENSEIGNANTS
    </xsl:comment>
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="NOM">
    <xsl:value-of select="position()"/> : Noeud NOM
  </xsl:template>

  <xsl:template match="text()">
    <xsl:value-of select="position()"/> : Noeud de texte
  </xsl:template>

  <xsl:template match="comment()">
    <xsl:value-of select="position()"/> : Noeud de commentaire
  </xsl:template>

</xsl:stylesheet>
```

Il contient plusieurs règles. La première déclenche un `xsl:apply-templates` pour tous les nœuds de type `ENSEIGNANTS`. Il n'existe qu'un nœud de ce type, et une règle associée qui est instanciée.

Cette règle produit un commentaire dans le document résultat avec l'instruction `xsl:comment`, puis déclenche à son tour un `xsl:apply-templates` sans donner de cible, ce qui revient à sélectionner tous les fils du nœud <`ENSEIGNANTS`>. On obtient le résultat suivant :

Exemple 5.6 *ApplyTemplates.xml* : *Résultat du programme précédent*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--
         Application de la règle ENSEIGNANTS
    -->
  1 : Noeud de texte
  2 : Noeud de commentaire
  3 : Noeud de texte
  4 : Noeud NOM
  5 : Noeud de texte
  6 : Noeud NOM
  7 : Noeud de texte
```

Que s'est-il passé ? On peut reconnaître l'exécution de la seconde règle au commentaire qu'elle a produit. Ensuite le processeur a pris tous les nœuds fils de <`ENSEIGNANTS`> et a cherché la règle à instancier. On peut constater qu'il y a sept nœuds, dont quatre sont des nœuds blancs. Nous avons vu que ces nœuds étaient présents dans l'arbre XML, et ils ne sont pas éliminés par le processeur XSLT, à moins de le demander explicitement avec un `xsl:strip-space` : voir le chapitre **??**, page **??**.

En général ces nœuds blancs sont traités « silencieusement » par la règle par défaut pour les nœuds de type **Text** qui consiste à insérer le contenu dans le résultat, mais comme ce contenu est constitué d'espaces, cela passe en général inaperçu.

Ici il existe une règle spécifique pour les nœuds textes, qui affiche notamment leur position. C'est cette règle que le processeur XSLT a instanciée. On peut faire la même remarque pour le nœud de type **Comment** pour lequel une règle spécifique existe dans notre programme.

Un élément `xsl:apply-templates` est associé à une expression XPath qui sélectionne un ensemble de nœuds. Comme nous l'avons vu dans le chapitre **??**, pour chaque nœud de cet ensemble, on connaît la position avec la fonction *position*() et le nombre de nœuds avec la fonction *last*().

Ces informations servent de *contexte* à l'instanciation des différentes règles. Pour chaque règle, il existe un nœud courant, celui qui a déterminé la sélection de la règle. On peut de plus faire référence à la position ou à la taille du contexte : en l'occurrence, cela nous a permis de numéroter les nœuds avec *position*().

5.4 Sélection des règles

Regardons maintenant comment une règle est sélectionnée quand plusieurs possibilités existent. Pour chaque nœud sélectionné par un `xsl:apply-templates`, le processeur va regarder toutes les règles du programme, et tester le *pattern* comme indiqué précédemment (voir page 44) pour déterminer si le nœud satisfait la règle.

Quand aucune règle n'est trouvée, c'est la règle par défaut qui est instanciée. Si une seule règle est trouvée, elle s'applique, ce qui signifie bien qu'elle est *prioritaire* par rapport à la règle par défaut. Cette même notion de priorité se généralise à la situation où plusieurs règles sont candidates. Le processeur doit alors choisir la règle à instancier en fonction d'un système de priorités.

Un autre facteur intervenant dans le choix d'une règle est le *mode*. Enfin nous avons vu page 38 que l'importation de programme définit une préséance (à ne pas confondre avec la priorité).

Nous décrivons ci-dessous le rôle des attributs `priority` et `mode` de l'élément `xsl:template`, avant de récapituler les principes de choix d'une règle. Le document XML utilisé pour nos exemples dans cette section est une liste de trois films.

Exemple 5.7 *ListeFilms.xml* : *Une liste de films*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<FILMS>
<FILM>
<TITRE>Vertigo</TITRE>
<ANNEE>1958</ANNEE><GENRE>Drame</GENRE>
<MES>Alfred Hitchcock</MES>
<RESUME>Scottie Ferguson, ancien inspecteur de police,
 est sujet au vertige depuis qu'il a vu mourir son
 collègue. Elster, son ami, le charge de surveiller
 sa femme, Madeleine, ayant des tendances
 suicidaires. Amoureux de la jeune femme
 Scottie ne remarque pas le piège qui se trame autour
 de lui et dont il va être la victime...
</RESUME>
</FILM>
<FILM>
<TITRE>Alien</TITRE>
<ANNEE>1979</ANNEE><GENRE>Science-fiction</GENRE>
<MES>Ridley Scott</MES>
<RESUME>Près d'un vaisseau spatial échoué sur une
 lointaine planète, des Terriens en mission découvrent
 de bien étranges "oeufs". Ils en ramènent un à bord,
 ignorant qu'ils viennent d'introduire parmi
 eux un huitième passager particulièrement
 féroce et meurtrier.
</RESUME>
</FILM>
<FILM>
<TITRE>Titanic</TITRE>
<ANNEE>1997</ANNEE><GENRE>Drame</GENRE>
<MES>James Cameron</MES>
<RESUME>Conduite par Brock Lovett, une expédition
 américaine fouillant l'épave du Titanic remonte à la
 surface le croquis d'une femme nue. Alertée par les
 médias la dame en question, Rose DeWitt
 Bukater, aujourd'hui centenaire, rejoint les lieux du
 naufrage, d'où elle entreprend de conter le
 récit de son fascinant, étrange et tragique voyage...
</RESUME>
</FILM>
</FILMS>
```

Priorités

La priorité d'une règle peut être soit indiquée explicitement avec l'attribut `priority`, soit calculée implicitement par le processeur.

Prenons tout d'abord le cas où on indique explicitement une priorité. Supposons que l'on veuille recopier tout le document *ListeFilms.xml*, à l'exception du résumé des films. On peut envisager de définir une règle pour tous les types de nœuds, en recopiant leur contenu, balises comprises, sauf dans le cas des éléments de types `RESUME` pour lesquels on ne fait rien.

Une solution plus simple et plus générale consiste à utiliser l'instruction `xsl:copy` qui recopie un nœud du document source vers le document résultat. Dans notre cas `xsl:copy` doit être utilisé pour tous les nœuds, sauf pour <`RESUME`>. Deux règles suffisent :

- une règle pour les éléments de type `RESUME`, qui ne fait rien ;
- une règle pour *tous* les nœuds qui effectue la copie du nœud pour lequel elle est instanciée, et qui déclenche un `xsl:apply-templates` pour les fils de ce nœud.

Les deux règles peuvent être sélectionnées pour un nœud de type `RESUME`. Afin d'indiquer que la première est prioritaire, il suffit de donner à la seconde une priorité faible, par exemple -1. Voici le programme XSLT qui copie tous les films sauf leur résumé.

Exemple 5.8 *Priority.xsl* : *Utilisation de l'attribut* `priority`

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
>
 <xsl:output method="xml" encoding="ISO-8859-1"/>

  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="RESUME"/>

   <xsl:template match="@*|node()" priority="-1">
     <xsl:copy>
       <xsl:apply-templates select="@*|node()"/>
     </xsl:copy>
   </xsl:template>

</xsl:stylesheet>
```

La troisième règle avec l'attribut `match="@*|node()"` et avec la priorité -1 s'applique à tous les types de nœuds. Elle commence par copier le nœud courant, puis sélectionne avec `xsl:apply-templates` tous ses fils, y compris les attributs. Pour chacun de ses fils la même règle s'applique récursivement, jusqu'à ce que la totalité du document soit parcourue et recopiée.

Les éléments <`RESUME`> constituent la seule exception : la règle appelée en priorité est alors celle qui est spécifique à ce type d'élément, et elle ne produit aucun résultat. On obtient donc le document suivant :

Exemple 5.9 *Priority.xml* : *Résultat du programme Priority.xsl*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<FILMS>
<FILM>
<TITRE>Vertigo</TITRE>
<ANNEE>1958</ANNEE><GENRE>Drame</GENRE>
<MES>Alfred Hitchcock</MES>
```

```
</FILM>
<FILM>
<TITRE>Alien</TITRE>
<ANNEE>1979</ANNEE><GENRE>Science-fiction</GENRE>
<MES>Ridley Scott</MES>

</FILM>
<FILM>
<TITRE>Titanic</TITRE>
<ANNEE>1997</ANNEE><GENRE>Drame</GENRE>
<MES>James Cameron</MES>

</FILM>
</FILMS>
```

Quand une priorité explicite n'est pas indiquée, le système en calcule une en se basant sur le *pattern* de l'attribut `match` :

1. Tous les *patterns* constitués d'une seule étape XPath, avec un nom d'élément ou d'attribut et sans prédicat ont une priorité égale à 0.

   Exemple : `FILM`, `@CODE`.

2. Le *pattern* `processing-instruction('nom')` a lui aussi une priorité égale à 0.

3. Tous les *patterns* constitués d'une seule étape XPath, avec un filtre sur tous les éléments ou attributs d'un espace de nom ont une priorité égale à -0,25.

   Exemple : `xbook:*`.

4. Les filtres sans espace de noms et autres qu'un nom d'élément ou d'attribut ont une priorité égale à -0,5.

   Exemple : `node()`, `processing-instruction()`, `*`

Enfin tous les *patterns* qui n'appartiennent pas à une des catégories ci-dessus ont une priorité égale à 0,5. C'est le cas par exemple des *patterns* avec prédicat, ou des *patterns* constitués de plusieurs étapes.

Il reste un cas à considérer, celui d'un *pattern* constitué d'une union avec l'opérateur « | ». Par exemple :

```
<xsl:template match="SUJET|node()">
  corps de la règle
</xsl:template>
```

Dans ce cas le processeur constitue simplement deux règles indépendantes, et calcule la priorité pour chacune. On se retrouve donc avec :

```
<xsl:template match="SUJET">
  corps de la règle
</xsl:template>
<xsl:template match="node()">
  corps de la règle
</xsl:template>
```

La première règle a une priorité égale à 0, la seconde une priorité égale à -0,5.

Si on reprend l'exemple du programme *Priority.xsl*, page 52, on s'aperçoit maintenant qu'il était inutile de donner une priorité puisque celles déterminées par défaut sont les suivantes :

– `RESUME` a une priorité de 0 ;

– la dernière règle se divise en deux :

  – `@*` a une priorité de -0,5 ;
  – `node()` a une priorité de -0,5 ;

Donc la dernière règle est moins prioritaire, ce qui correspond à l'intuition qu'elle est moins « spécifique », autrement dit qu'elle s'applique à une plus grande variété de nœuds. C'est la même intuition qui mène à affecter une priorité plus grande à un *pattern* comme `FILM/TITRE` (priorité 0,5) qu'au *pattern* `TITRE` (priorité 0).

Modes

Les modes constituent la dernière manière de guider le processeur dans le choix d'une règle. L'attribut `mode` est optionnel : en son absence la règle sera concernée par tout `xsl:apply-templates` qui lui non plus n'utilise pas son attribut `mode`. Sinon le mode de la règle et le mode de `xsl:apply-templates` doivent coïncider.

Les modes sont principalement utiles quand on souhaite placer la même information issue du document source en plusieurs endroits du document résultat. Nous allons prendre un exemple qui suffira à illustrer l'idée. Supposons que l'on veuille créer un document HTML avec une présentation de tous les films présents dans *ListeFilms.xml* (voir page 51). La page HTML affichée dans le navigateur aura une taille respectable. Si on veut consulter un film particulier, il faudra faire défiler la page dans la fenêtre.

Pour faciliter la tâche de l'utilisateur, on peut introduire une liste d'ancres HTML au début de la page. Le résultat de la transformation est illustré dans la figure 23. En cliquant sur les ancres placées au début du document, on se positionne directement sur le film correspondant.

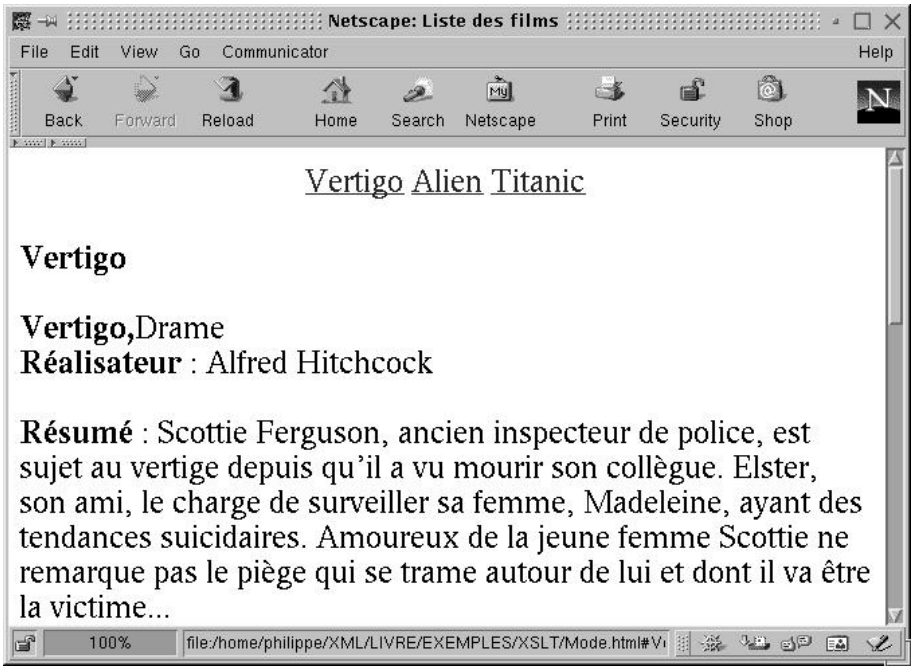

FIG. 23 – *Affichage des films avec Netscape*

La balise `<A>` peut être utilisée pour définir une position interne à un document en plaçant des « marques ». Par exemple :

```
<a name='Alien'/>
```

définit une position de nom `Alien`. On peut alors utiliser une ancre donnant directement accès à cette position :

```
<a href='#Alien'>Lien vers le film Alien</A>
```

Le programme XSLT qui suit effectue deux passages sur les nœuds de type `FILM`, avec deux règles utilisant le même *pattern* mais deux modes différents.

Les règles sont appelées avec deux `xsl:apply-templates`, avec les modes correspondant.

```
<xsl:apply-templates select="FILM" mode ="Ancres"/>
<xsl:apply-templates select="FILM"/>
```

Au cours du premier passage, la règle avec `mode='Ancres'` est choisie et crée un tableau HTML avec une seule ligne contenant une ancre pour chaque film. Le second passage crée une représentation HTML complète de chaque film. Voici le programme complet :

Exemple 5.10 *Mode.xsl* : *Utilisation de l'attribut* `mode`

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html"  encoding="ISO-8859-1"/>

  <xsl:template match="FILMS">
       <html>
      <head><title>Liste des films</title></head>
      <body bgcolor="white">
          <center><table>
           <xsl:apply-templates select="FILM"
                          mode ="Ancres"/>
          </table></center>
          <xsl:apply-templates select="FILM"/>
      </body>
     </html>
   </xsl:template>

   <xsl:template match="FILM" mode="Ancres">
     <td><a href="#{TITRE}">
       <xsl:value-of select="TITRE"/>
     </a></td>
   </xsl:template>

   <xsl:template match="FILM">
     <a name="{TITRE}"/>
     <h1><xsl:value-of select="TITRE"/></h1>
     <b><xsl:value-of select="TITRE"/>,</b>
          <xsl:value-of select="GENRE"/>
      <br/>
      <b>Réalisateur</b> : <xsl:value-of select="MES"/>
      <p>
       <b>Résumé</b> : <xsl:value-of select="RESUME"/>
       </p>
   </xsl:template>

</xsl:stylesheet>
```

Notez la possibilité d'introduire des expressions XPath pour *calculer* la valeur de l'attribut d'un élément littéral, comme dans :

```
<a name="{TITRE}"/>
```

Ce mécanisme est décrit dans le chapitre qui suit, page **??** (voir également dans la référence XSLT, page **??**).

Algorithme de sélection d'une règle

Voici, en résumé, comment le processeur XSLT choisit une règle à instancier pour un nœud sélectionné par `xsl:apply-templates`. Au départ, la liste des règles candidates est constituée de toutes celles qui ont un attribut `match`.

1. On restreint la liste à toutes les règles qui ont le même `mode` que l'élément `xsl:apply-templates`, ou pas d'attribut `mode` si `xsl:apply-templates` lui-même n'en a pas.
2. On teste le *pattern* de l'attribut `match` pour déterminer si le nœud satisfait la règle.
3. Si on a trouvé plusieurs règles, on ne garde que celle(s) qui a (ont) la plus grande préséance d'importation (voir page 38).
4. S'il reste encore plusieurs règles, on prend celle qui a la plus grande priorité.

Si cet algorithme ne permet pas de départager deux règles, le processeur XSLT peut s'arrêter ou en choisir une (en principe la dernière dans l'ordre du programme). Il y a clairement dans ce cas un problème de conception du programme. Si, au contraire, aucune règle n'est trouvée, alors la règle par défaut s'applique (voir page 48).

5.5 Appel de règle avec `xsl:call-template`

Quand une règle utilise l'attribut `name` au lieu de l'attribut `match`, on peut l'instancier explicitement avec un `xsl:call-template`. L'utilisation de `xsl:call-template` s'apparente aux fonctions dans un langage de programmation classique. Comme les fonctions, la règle appelée doit permettre de factoriser des instructions utilisées en plusieurs endroits du programme.

Reprenons l'exemple affichant les nœuds du document *CoursXML.xml* en les numérotant. Le programme *ApplyTemplates.xml* (voir page 50) contenait une règle pour chaque type de nœud rencontré, et toutes ces règles produisaient à peu près le même résultat : la position du nœud courant, et un texte.

Le programme ci-dessous remplace le corps de ces règles par un appel `xsl:call-template` à une règle nommée `Afficher` qui se charge de produire le résultat.

Exemple 5.11 *CallTemplate.xsl* : *Exemple de* `xsl:call-template`

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="xml" encoding="ISO-8859-1"/>

  <xsl:template match="/">
    <xsl:apply-templates select="//ENSEIGNANTS"/>
  </xsl:template>

  <xsl:template match="ENSEIGNANTS">
    <xsl:comment>
         Application de la règle ENSEIGNANTS
    </xsl:comment>
```

```
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template name="Afficher">
    <xsl:value-of select="position()"/> : <xsl:value-of select="."/>
  </xsl:template>

  <xsl:template match="NOM">
    <xsl:call-template name="Afficher"/>
  </xsl:template>

  <xsl:template match="text()">
    <xsl:call-template name="Afficher"/>
  </xsl:template>

  <xsl:template match="comment()">
    <xsl:call-template name="Afficher"/>
  </xsl:template>

</xsl:stylesheet>
```

L'amélioration n'est pas encore complète, et nous verrons qu'il est possible de faire mieux en utilisant `xsl:for-each` ou un passage de paramètre. Le progrès notable est qu'il devient maintenant possible de modifier l'ensemble de la sortie du programme en ne touchant qu'au corps de la règle `Afficher`. Le résultat du programme est donné ci-dessous.

Exemple 5.12 *CallTemplate.xml* : *Résultat du programme précédent*

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--    Application de la règle ENSEIGNANTS -->
1 :
2 :  Enseignant responsable
3 :
4 : Amann
5 :
6 : Rigaux
7 :
```

Il faut souligner que, contrairement à `xsl:apply-templates`, un appel avec `xsl:call-template` ne change pas le contexte : le nœud courant (et donc son contenu), sa position et la taille du contexte restent connus dans le corps de la règle `Afficher`.

La comparaison entre `xsl:call-template` et un appel de fonction reste limitée. En particulier, dans ce que nous avons vu jusqu'ici,

– une instanciation de règle ne « renvoie » pas de valeur à la règle qui l'a déclenchée ;

– une instanciation de règle ne prend pas d'argument autre que le nœud auquel elle s'applique ;

Nous verrons dans la section consacrée aux variables, page **??**, comment stocker dans une structure temporaire le texte produit par une règle au lieu de l'insérer directement dans le résultat.

En ce qui concerne la deuxième limitation, elle est en partie levée par la possibilité de passer des paramètres aux règles.

5.6 Paramètres

L'exemple de `xsl:call-template` que nous avons donné précédemment offrait assez peu d'intérêt puisqu'il n'était pas possible de faire varier le texte produit par `Afficher` en fonction du type de nœud.

Nous allons améliorer la règle en lui passant en paramètre le texte à afficher après la position du nœud courant.

Le passage de paramètres s'effectue en deux étapes. Tout d'abord on indique au niveau de la définition de la règle, avec des éléments `xsl:param`, le nom du paramètre et sa valeur par défaut. Il existe deux possibilités de passer :

`<xsl:param name=`*`nom`* `select=`*`expression`*`>/`

L'attribut *`name`* donne le nom du paramètre, et l'attribut *`select`* (optionnel) est une expression XPath qui, convertie en chaîne de caractères, donne la valeur par défaut.

Voici la nouvelle version de la règle `Afficher`, avec un paramètre nommé `texte` ayant pour valeur par défaut `string(inconnu)`.

```
<xsl:template name="Afficher">
  <xsl:param name="texte" select="string(inconnu)"/>
  <xsl:value-of select="position()"/> :
            <xsl:value-of select="$texte"/>
</xsl:template>
```

Attention aux chaînes de caractères constantes dans XPath. Si on avait mis :

```
<xsl:param name="texte" select="inconnu"/>
```

le processeur XSLT aurait évalué la valeur par défaut en recherchant un élément `<inconnu>` fils de l'élément courant. La fonction *string*() lève l'ambiguïté. Une autre possibilité est d'encadrer la chaîne par des apostrophes simples :

```
<xsl:param name="texte" select="'inconnu'"/>
```

Si l'attribut `select` est absent, il est possible d'indiquer la valeur par défaut par le contenu de l'élément `xsl:param`. Voici une version équivalente de la règle précédente :

```
<xsl:template name="Afficher">
  <xsl:param name="texte">inconnu</xsl:param>
  <xsl:value-of select="position()"/> :
            <xsl:value-of select="$texte"/>
</xsl:template>
```

Maintenant on peut, avant d'appeler la règle, définir les paramètres avec `xsl:with-param`. Voici par exemple l'appel à `Afficher` en donnant au préalable au paramètre la valeur « `texte vide` ».

```
<xsl:template match="text()">
  <xsl:call-template name="Afficher">
    <xsl:with-param name="texte" select="'texte vide'"/>
  </xsl:call-template>
</xsl:template>
```

ou

```
<xsl:template match="text()">
  <xsl:call-template name="Afficher">
    <xsl:with-param name="texte">texte vide</xsl:with-param>
  </xsl:call-template>
</xsl:template>
```

Le positionnement de `xsl:with-param` suit des règles précises :

1. il peut apparaître dans le contenu de l'élément `xsl:call-template`, comme ci-dessus ; c'est d'ailleurs le seul contenu possible pour `xsl:call-template` ;

2. on peut aussi passer des paramètres à `xsl:apply-templates`, en les plaçant immédiatement après la balise ouvrante de cet élément.

Le passage des paramètres avec XSLT est d'une grande souplesse. On peut passer à une règle des paramètres qu'elle n'attend pas, ou au contraire ne pas lui passer de paramètre. Dans le premier cas le paramètre est tout simplement ignoré. Dans le second cas la règle prend en compte la valeur par défaut du paramètre, si cette valeur existe.

Il est possible d'utiliser `xsl:param` comme un élément de premier niveau. Dans ce cas, comme il n'y a pas eu de déclenchement de règle avec `xsl:with-param` pour affecter une valeur au paramètre, c'est au processeur de déterminer quelle est la valeur du paramètre. Il existe plusieurs possibilités, dont les deux suivantes :

1. quand le processeur est invoqué depuis la ligne de commande, une option permet en général de définir le paramètre : c'est par exemple l'option *-PARAM* avec le processeur XALAN (voir l'annexe **??**, page **??**) ;
2. quand le processeur est intégré à un serveur web (cas de Cocoon), les paramètres sont ceux du protocole HTTP (variables `GET` ou `POST`).

6 Évaluation d'un programme XSLT

Voici maintenant une vue d'ensemble de *l'évaluation* d'un programme XSLT. L'objectif est de récapituler les principaux concepts d'une transformation XSLT, essentiels pour la compréhension du langage : *nœud courant* et *nœud contexte*, *choix et instanciation de la règle* appliquée au nœud courant, *production du résultat*. Cet exemple est probablement à lire plusieurs fois pour bien réaliser la signification de ces notions qui peuvent paraître abstraites (ou confuses) au moment d'un premier contact avec XSLT. Une première lecture peut certainement se contenter d'une compréhension intuitive, et les suivantes s'attacher aux subtilités.

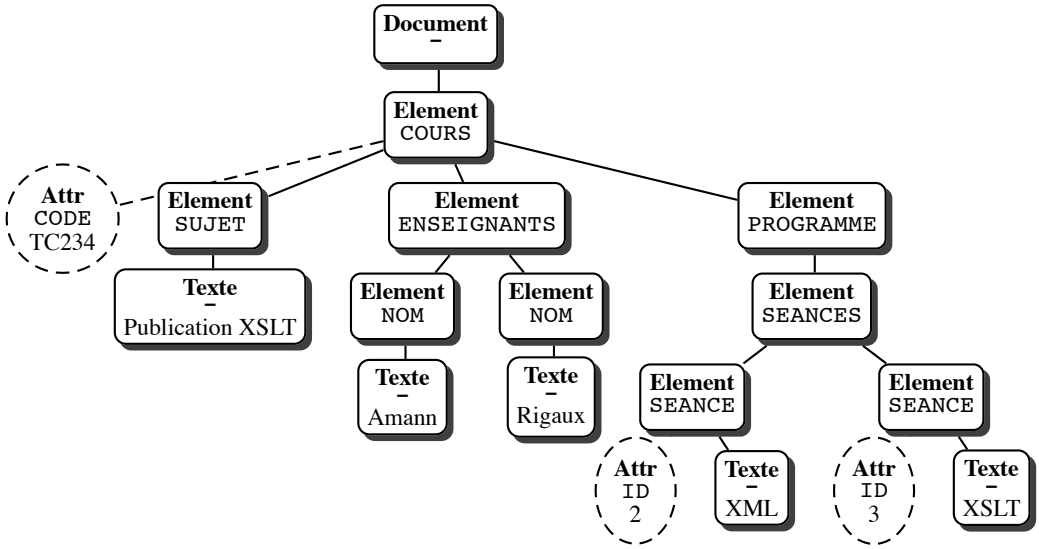

FIG. 24 – *Exemple de référence pour XSLT*

Nous prenons l'exemple d'une transformation du document *CoursXML.xml*, page 34. La figure 24 illustre la représentation arborescente de ce document. La transformation construit une page HTML. Voici

le programme.

Exemple 6.1 *Recap.xsl* : *Le programme à évaluer*

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:output method="html" encoding="ISO-8859-1"/>

  <xsl:template match="/">
    <html>
      <head>
        <title><xsl:value-of select="COURS/SUJET"/></title>
      </head>
      <body bgcolor="white">

        <xsl:apply-templates/>

      </body>
     </html>
  </xsl:template>

  <xsl:template match="SUJET">
    <h1><center><xsl:value-of select="."/></center></h1>
  </xsl:template>

  <xsl:template match="ENSEIGNANTS">
    <h1>Enseignants</h1>
    <ol>
      <xsl:apply-templates select="NOM"/>
    </ol>
  </xsl:template>

  <xsl:template match="PROGRAMME">
    <h1>Programme</h1>
    <ul>
      <xsl:for-each select="SEANCE">
         <xsl:call-template name="AfficheSeance"/>
      </xsl:for-each>
    </ul>
  </xsl:template>

  <xsl:template match="ENSEIGNANTS/NOM">
    <li><xsl:value-of select="." /></li>
  </xsl:template>

  <xsl:template name="AfficheSeance">
      <li> Séance <xsl:value-of select="concat(
                                position(), '/', last(), ' : ', . )"/>
       </li>
  </xsl:template>
</xsl:stylesheet>
```

6.1 Initialisation de l'environnement

Après l'analyse des deux documents, le processeur va commencer par créer le nœud racine du document résultat (figure 25), et par se positionner sur la racine du document source qui devient le *nœud courant*.

FIG. 25 – *Début de l'évaluation : le document résultat*

Étant donné un nœud courant, le processeur recherche parmi la liste des règles du programme, celles qui sont applicables à ce nœud, conformément à l'algorithme présenté page 51. La seule règle qui convient pour la racine du document est la première, car son attribut `match` contient l'expression XPath « / » qui désigne le nœud racine.

6.2 Instanciation de règles

La première règle est donc *instanciée*. Son contenu est copié sous la racine du document résultat, ce qui donne l'arbre provisoire du résultat montré dans la figure 26.

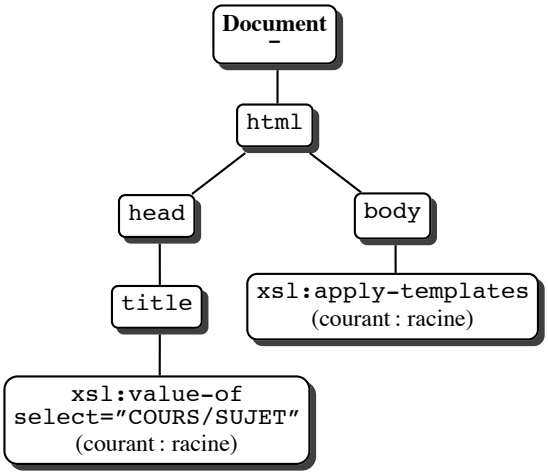

FIG. 26 – *Après instanciation de la première règle*

Cet arbre contient deux éléments de l'espace de noms XSLT désigné par le préfixe `xsl`. Ils doivent donc être évalués par le processeur et remplacés par le résultat obtenu. Chacun reste associé au nœud courant du document source dans le contexte duquel il a été instancié : ici c'est l'élément racine. Quand le processeur traitera un de ces éléments, les expressions XPath relatives qui lui sont associées (par exemple dans un attribut `select`) seront donc évaluées en fonction de ce nœud courant.

On voit bien dès ce stade que l'évaluation d'un programme XSLT n'est pas un processus linéaire dans lequel les balises ouvrante et fermante des éléments sont produites séquentiellement. Un arbre est construit par instanciation de nœuds complets, et à ces nœuds sont rattachés de nouveaux fils au fur et à mesure que les instructions XSLT restantes sont évaluées. Ici encore, il vaut donc mieux raisonner, comme nous le faisons depuis le début de ce livre, sur une représentation arborescente des documents que sur une version sérialisée avec balises.

Une particularité de ce type d'évaluation est que le processeur a le choix de la prochaine instruction à évaluer. On constate sur la figure 26 que les deux instructions `xsl:value-of` et `xsl:apply-templates`

peuvent être évaluées séparément, dans un ordre quelconque, voire en parallèle. Il suffit que le processeur conserve l'information sur le *nœud courant* associé à chacune de ces instructions pour pouvoir y revenir et les traiter à tout moment.

6.3 Insertion de valeurs : `xsl:value-of`

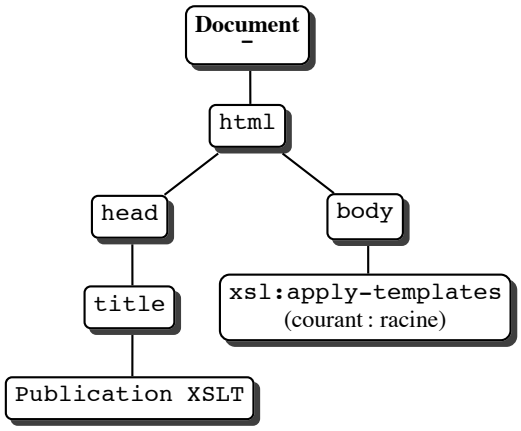

FIG. 27 – *Après évaluation de* `xsl:value-of`

Si on choisit d'évaluer d'abord `xsl:value-of`, on obtient l'arbre temporaire de la figure 27. À partir du nœud courant qui est la racine du document source, on a évalué l'expression XPath `COURS/SUJET`, et on a remplacé l'instruction XSLT par le résultat textuel de cette évaluation (voir fonction *string*(), chapitre **??**, page **??**), la chaîne de caractères `'Publication XSLT'`.

6.4 Évaluation de règles

Passons maintenant à l'instruction `xsl:apply-templates`. L'attribut `select` est absent, donc le processeur applique le choix par défaut qui consiste à sélectionner les fils du nœud courant, soit en l'occurrence <COURS>, l'élément racine du document. C'est cet élément qui devient le nouveau nœud courant.

La recherche dans le programme d'une règle applicable échoue, car on peut vérifier qu'aucune expression `match` ne désigne ce nœud. Dans ce cas le processeur applique une *règle par défaut* (voir page 48). Elle consiste à sélectionner les fils du nœud courant, <COURS>, et à leur appliquer une règle. On a donc maintenant un ensemble de trois éléments, <SUJET>, <ENSEIGNANTS> et <PROGRAMME>.

Le processeur va prendre ces trois éléments tour à tour comme nœud courant, rechercher la règle pour chacun, et l'instancier. Il existe trois règles distinctes dans le programme, qui s'appliquent chacune à un de ces trois éléments. Le remplacement du nœud `xsl:apply-templates` par le corps de ces trois règles donne l'arbre résultat temporaire de la figure 28.

Le processeur continue à prendre dans l'arbre résultat en cours de constitution les éléments dont l'espace de noms est `xsl`. Pour chacun de ces éléments il existe un nœud courant dans l'arbre source, lui-même faisant partie d'un ensemble de nœuds. Pour l'élément `xsl:value-of` dans la figure 28 par exemple, le nœud courant dans l'arbre source est <SUJET>, et l'ensemble de nœuds (le contexte) est constitué des trois fils de <COURS>. À partir de la situation de la figure 28, le processeur va donc (en supposant que les instructions sont traitées dans l'ordre du document) :

1. évaluer `xsl:value-of`, avec l'attribut `select` valant « . », le nœud courant étant <SUJET> : le résultat est `Publication XSLT` ;

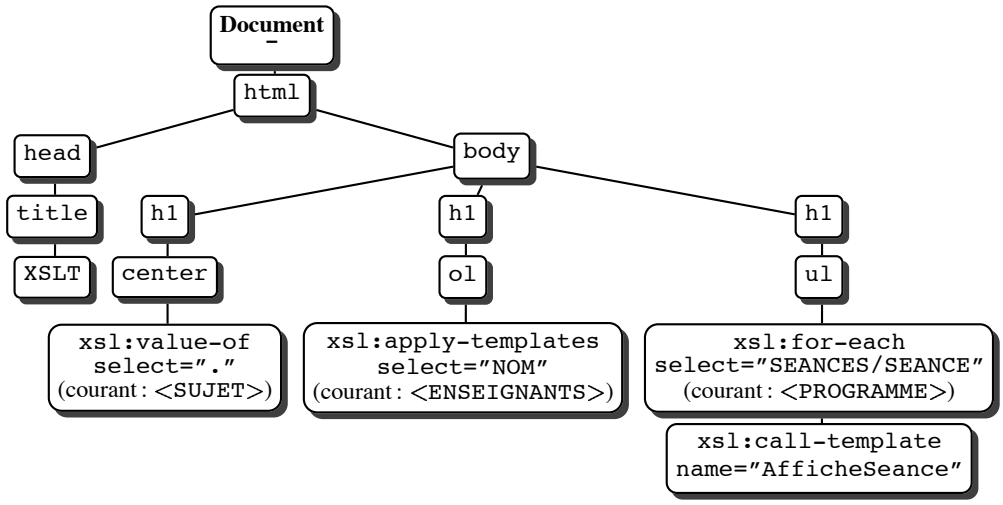

FIG. 28 – *Après évaluation de* `xsl:apply-templates`

2. évaluer l'expression `NOM` de l'attribut `select` du `xsl:apply-templates`, le nœud courant étant `<ENSEIGNANTS>` : on obtient deux nœuds pour lesquels la seule règle qualifiée est celle dont le *pattern* est `ENSEIGNANTS/NOM` ;

3. enfin l'expression `SEANCE` de l'attribut `select` du `xsl:for-each` est évaluée, ce qui donne – le nœud courant étant `<PROGRAMME>` – les deux éléments de type `SEANCE` du document source.

La figure 29 montre la situation de l'arbre résultat intermédiaire après évaluation du `xsl:for-each`. Il reste au processeur à évaluer deux appels `xsl:call-template` à la règle nommée `AfficheSeance`. En apparence ces deux appels sont identiques, mais ce qui les distingue c'est leur nœud courant dans le document source.

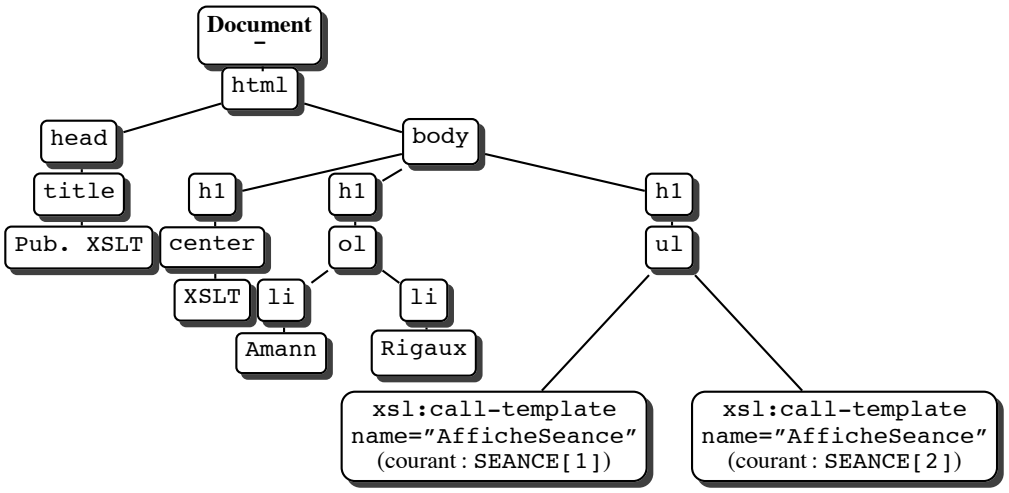

FIG. 29 – *Après évaluation du* `xsl:for-each`

L'instruction `xsl:for-each` a pour effet de changer le nœud courant pour chaque instanciation de son contenu (voir page **??**). L'ensemble de nœuds, lui, est constitué par l'évaluation de l'expression du `xsl:for-each`, et se trouve donc commun aux deux appels de règles.

Contrairement à `xsl:apply-templates`, un appel à `xsl:call-template` ne change ni le nœud courant (une des deux séances), ni l'ensemble des nœuds formant le contexte (les deux séances). L'instanciation de la règle nommée donnera donc le résultat final de la figure 30.

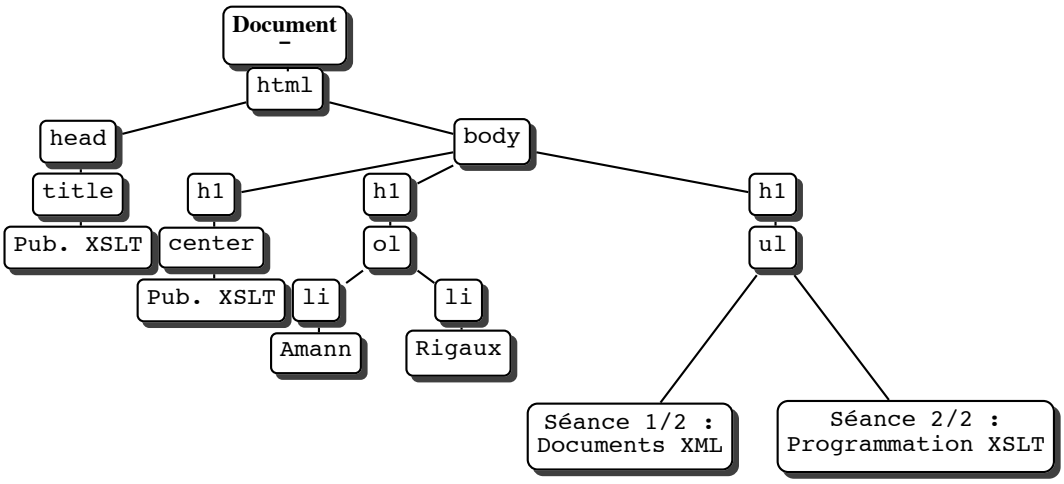

FIG. 30 – *Le résultat final*

6.5 Sérialisation du résultat

L'évaluation est terminée puisque le document résultat ne contient plus aucune instruction XSLT. Il reste à *sérialiser* le document obtenu, ce qui donne le résultat ci-dessous [5].

Exemple 6.2 *Recap.html* : *le document HTML sérialisé*

```
<html>
<head>
<META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Publication XSLT</title>
</head>
<body bgcolor="white">

<h1>
<center>Publication XSLT</center>
</h1>

<h1>Enseignants</h1>
<ol>
<li>Amann</li>
<li>Rigaux</li>
</ol>

<h1>Programme</h1>
<ul>
<li> S&eacute;ance 1/2 : Documents XML</li>
<li> S&eacute;ance 2/2 : Programmation XSLT</li>
</ul>
```

5. Dans le cas d'un document HTML, certaines transformations peuvent intervenir, comme la transformation d'un <hr/> en <hr>. Elles sont décrites dans le chapitre suivant.

```
</body>
</html>
```

Cet exemple a montré, sur un cas simple mais représentatif, les principales caractéristiques de l'évaluation d'un programme XSLT. Récapitulons les points les plus importants :

1. le processeur construit un arbre, et pas une version sérialisée du document ;
2. chaque instruction est placée dans le document résultat, puis évaluée ; elle reste toujours associée à un nœud courant dans le document source, lui-même faisant partie d'un ensemble de de nœuds : ces deux informations constituent le contexte d'évaluation des expressions XPath placées dans l'instruction ;
3. on ne peut pas préjuger de l'ordre d'exécution des instructions d'un programme XSLT : le langage lui-même est conçu pour que le résultat ne dépende pas de cet ordre.

Avant d'être transmis à un traitement, quel qu'il soit, un document XML (sérialisé) est traité par un *processeur XML* ou *parseur* qui va analyser son contenu et déterminer sa structure. Dans le cas d'un parseur DOM le résultat de cet analyse est un arbre DOM.

DOM fournit une spécification orientée-objet basée sur des méthodes d'investigation et d'extraction de données. Bien que cette spécification définisse de manière précise les opérations sur un arbre XML, sa mise en œuvre en pratique nécessite le recours à un environnement de programmation comme Java.

7 Le modèle DOM

Le *Document Object Model*, DOM [6] est une des normes de description d'une structure XML. La modélisation DOM d'un document XML consiste à décrire ce document comme un graphe composé *d'objets* appartenant à un ensemble déterminé de *types* de nœuds, la composition des objets étant régie par des règles qui définissent la grammaire de XML.

DOM est une spécification de haut niveau, indépendante d'un langage particulier. Une connaissance des concepts orientés-objet de base est donc suffisante. Nous donnons une première présentation des aspects essentiels de cette spécification, avant de la compléter par des exemples.

7.1 Types de nœuds DOM

Dans sa représentation DOM, un document XML est essentiellement un arbre constitué de *nœuds* de différents types. Les types de nœuds sont définis sous forme d'*interfaces*. Ainsi, un document XML devient un objet qui contient d'autres objets avec des propriétés et méthodes qui peuvent être utilisées pour écrire des applications XML [7]. Le tableau 7 résume tous les types de nœuds fournis par DOM.

La plupart de ces interfaces (onze sur treize) correspondent aux différents catégories syntaxiques XML que nous avons décrites page **??**. Les seules exceptions sont **DocumentFragment** et **Notation** :

– L'interface **DocumentFragment** est essentiellement destinée à faciliter la programmation et correspond à une version allégée de l'interface **Document**. Elle peut être ignorée sans dommage dans le contexte XSLT.

– L'interface **Notation** correspond à une notion faisant partie de la DTD du document et elle permet à une application XML de choisir les actions pour traiter des données non-XML dans le contenu du document.

Tous ces interfaces sont considérées par DOM comme des spécialisations d'une interface **Node** dont ils héritent toutes les propriétés et méthodes. La figure 31 montre cette hiérarchie de spécialisation (l'interface **DocumentFragment** n'apparaît pas dans la figure). Les interfaces *TreeNode*, *Leaf* et *Container* montrées

6. La présentation qui suit correspond au *DOM (Core) Level 1*.

7. Comme Java, DOM utilise la notion d'*interface* pour souligner l'indépendance d'un langage de programmation et d'une implantation spécifique. Nous allons utiliser dans la suite le terme *type DOM* pour souligner les aspects structuraux et *interface DOM* pour les aspects liés à la programmation.

Type de Nœuds	Catégorie syntaxique XML
Document	document XML (racine)
DocumentType	type du document (DTD)
ProcessingInstruction	instruction de traitement
Element	élément XML
Attribute	attribut XML
Entity	déclaration d'entité
EntityReference	référence vers entité
Comment	commentaire
CharacterData	commentaire et section de texte
Text	section de texte
CDataSection	section CDATA
DocumentFragment	fragment de document XML
Notation	notation (fait partie de la DTD)

TAB. 7 – *Types de nœuds DOM*

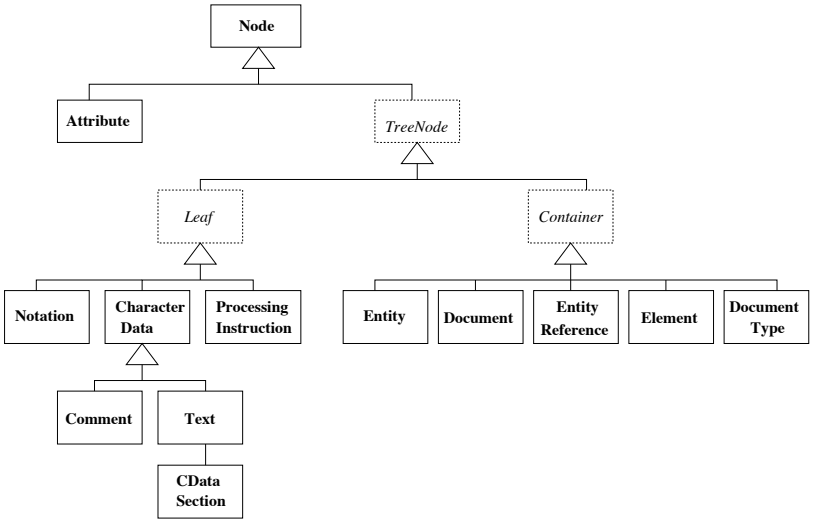

FIG. 31 – *Hiérarchie de spécialisation des interfaces DOM*

dans la figure 31 ne font pas partie du modèle DOM, mais nous les avons ajoutées pour clarifier le rôle des différents interfaces dans une arborescence DOM :

– l'interface *TreeNode* permet de différentier les attributs XML qui correspondent à des nœuds de type **Attr** des autres nœuds d'une arborescence DOM : nous revenons sur cette question ci-dessous ;

– parmi les nœuds de type *TreeNode* nous distinguons ceux qui acceptent des fils (type *Container*), et ceux qui sont forcément des feuilles de l'arbre (type *Leaf*).

Le modèle DOM spécifie une interface de programmation qui permet soit de modifier un document, soit de l'inspecter en vue d'en extraire des informations, point de vue que nous privilégions. Toutes ces opérations sont disponibles sous forme de *propriétés*[8], et de *méthodes* propres à chaque type de nœud.

Avant d'y revenir en détail, nous introduisons trois interfaces supplémentaires qui définissent quelques structures de données nécessaires à représentation complète d'une arborescence DOM. Ces interfaces ne sont pas représentées dans la hiérarchie de la figure 31.

7.2 Interface DOMString

Dans le modèle DOM, comme dans la syntaxe XML, une chaîne est une séquence de caractères Unicode. Ce système d'encodage est largement répandu dans les applications informatiques et aussi utilisé, par exemple, dans le langage Java pour la représentation des chaînes de caractères (le type **DOMString** peut être directement implanté par le type *String* de Java). Le standard Unicode définit chaque caractère par un entier sur deux (16-bit Unicode) ou quatre octets (16-bit Unicode étendu). Plus précisément, DOM (et la recommandation XML) distingue entre

– les caractères blancs : *x9* (tabulation), *xA* (saut à la ligne), *xD* (retour chariot) et *x20*. Généralement, on utilise une représentation héxadécimale qui permet de représenter une valeur par deux symboles dans l'ensemble $\{0,\ldots,9,1,\ldots,F\}$. Ainsi, les entiers *x9*, *xA*, *xD* et *x20* correspondent aux entiers 9, 10, 13 et 32 dans le système décimal pour la représentation des entiers ;

– les caractères 16-bit Unicode dont les valeurs héxadécimales sont comprises entre *x21* et *xD7FF*, et entre *xE000* et *xFFFD* ;

– les caractères 16-bit Unicode étendu, dont les valeurs héxadécimales sont comprises entre *x10000* et *x10FFFF* et entre *xE000* et *xFFFD*. Bien que la représentation interne des caractères « étendus » soit deux fois plus longue (4 octets) que la représentation des autres caractères (2 octets), tous sont considérés comme des caractères simples (une position) dans les fonctions qui calculent la taille d'une chaîne de caractères ou la position d'un caractère.

Dans DOM, la comparaison de deux chaînes de caractères s'effectue sur la représentation Unicode de leurs caractères et prend donc en compte ainsi la casse des chaînes comparées.

7.3 Interfaces NodeList et NamedNodeMap

DOM définit deux interfaces supplémentaires permettant de constituer des groupes de nœuds. Ces deux interfaces sont utilisées pour décrire les résultats de certaines méthodes.

Le type **NodeList** correspond à la notion de tableau indicé. Il représente une liste (ordonnée) de nœuds. Ces nœuds ne sont pas tous forcément de même type. **NodeList** peut permettre par exemple la manipulation de l'ensemble des fils d'un nœud de l'arbre, parmi lesquels on peut trouver des nœuds de type **Text**, **Element**, **CDataSection**, etc.

Le type **NamedNodeMap** correspond à la notion de tableau associatif (ou table de hachage). Il sert à représenter un ensemble (donc sans ordre) de nœuds identifiés par un nom. Ce type est essentiellement destiné à gérer une liste d'attributs XML.

8. Nous utilisons le mot *propriété* au lieu de celui, plus couramment employé, *d'attribut* afin de ne pas introduire de confusion avec les attributs XML.

7.4 Interface Node

Le type **Node** est la racine de toutes les autres interfaces proposées par DOM. Il définit les propriétés essentielles et fournit la plus grande part des opérations importantes dans DOM.

Les propriétés spécifiées par l'interface **Node** sont récapitulées dans le tableau 8. Les trois premières permettent de déterminer les informations « locales » concernant un nœud, c'est-à-dire son type, son nom et sa valeur sous forme d'un objet de type **DOMString**. Le nom et la valeur ne constituent pas des notions pertinentes pour tous les types de nœuds : un nœud de type **Text** par exemple n'a pas de nom, et un nœud de type **Element** n'a pas de valeur.

Propriété	**Type**
nodeType	**unsigned short**
nodeName	**DOMString**
nodeValue	**DOMString**
parentNode	**Node**
firstChild	**Node**
lastChild	**Node**
childNodes	**NodeList**
previousSibling	**Node**
nextSibling	**Node**
attributes	**NamedNodeMap**

TAB. 8 – *Propriétés du type **Node***

On peut remarquer que dans une spécification objet plus rigoureuse l'information sur le nom ou la valeur ne devrait pas apparaître à un niveau global **Node** mais au niveau d'une interface spécialisée spécifique aux sous-types de **Node** ayant soit un nom, soit une valeur, soit les deux. DOM n'adopte pas une telle approche, d'une part parce que XML s'y prête assez difficilement, d'autre part pour des raisons d'efficacité de l'implantation. On peut donc accéder à toutes les opérations nécessaires au niveau du type **Node**, à charge pour le programmeur de tester le type d'un objet appartenant à l'interface générique **Node** pour savoir quelles sont les opérations appropriées (technique dite de *casting*, à éviter en principe dans une approche objet). La propriété *NodeType* est justement destinée à effectuer ce test.

Cette conception constitue une limite de DOM, modèle qui se trouve à mi-chemin entre une modélisation conceptuelle rigoureuse et indépendante de l'implantation, et des compromis techniques imposées par des critères d'efficacité ou de facilité de développement. L'inconvénient est qu'il faut connaître l'interprétation des propriétés en fonction du type de nœud, ce que présente le tableau 9.

Type de nœud	**nodeName**	**nodeValue**
CDATASection	`#cdata-section`	contenu de la section CDATA
Comment	`#comment`	contenu du commentaire
Document	`#document`	`NULL`
DocumentType	nom de la DTD	`NULL`
Element	nom de l'élément	`NULL`
ProcessingInstruction	nom de la cible	le contenu (moins la cible)
Text	`#text`	contenu du nœud **Text**
Notation	nom de notation (voir page **??**)	`NULL`
Entity	nom de l'entité	`NULL`
EntityReference	nom de l'entité référencée	`NULL`
Attr	nom de l'attribut	valeur de l'attribut

TAB. 9 – *Interprétation des propriétés nodeName et nodeValue*

La figure 32 montre une instance d'un document DOM, avec des choix de représentation que nous adopterons dans toute la suite de ce livre et qu'il est bon de souligner dès maintenant.

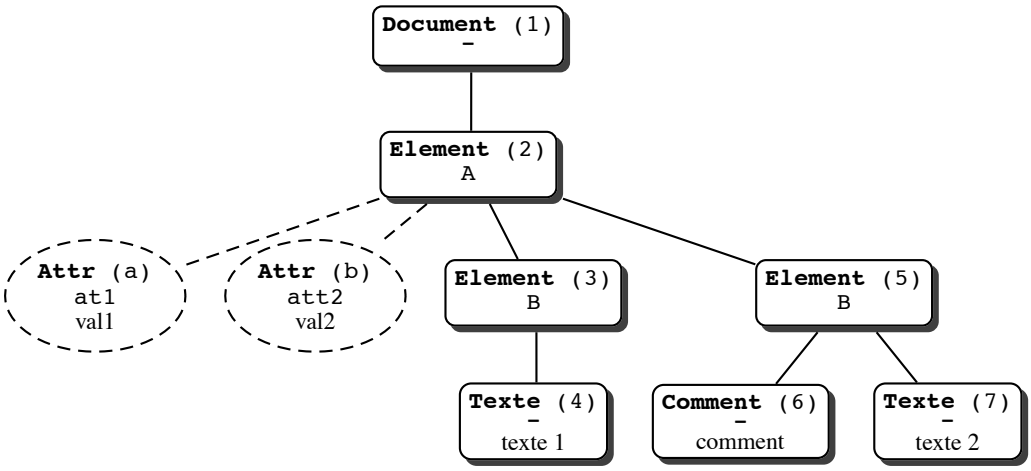

FIG. 32 – *Instance d'un arbre DOM*

Dans chaque nœud figurent trois informations : le type DOM (première ligne en gras), son nom (deuxième ligne) et la valeur (troisième ligne). Comme souligné précédemment, le nom et la valeur ne sont pertinents que pour certains types de nœuds. Par exemple les nœuds DOM de type **Text** ont une valeur mais pas de nom, ce que nous représentons par un symbole « – » dans la deuxième ligne, et, à l'inverse, les nœuds DOM de type **Element** ont un nom, mais pas de valeur. Dans ce cas la troisième ligne est absente.

L'arbre DOM de la figure 32 contient neuf nœuds dont deux sont des attributs. Pour simplifier la description de cet arbre, nous identifions chaque nœud par un entier ou un caractère (s'il s'agit d'un attribut) écrit entre parenthèses à côté du type du nœud. La racine (nœud 1) du document est de type **Document** et contient un fils unique (2) de type **Element**. L'élément racine a deux attributs (notés a et b) et deux sous-éléments (3 et 5) de type **Element**. Le premier élément, de type `B` (il ne faut pas confondre le type d'un nœud DOM avec le type d'un élément) et contient un seul nœud de type **Text** ; le deuxième élément contient un commentaire et du texte. Si on parcourt cet arbre à partir de la racine, en allant de gauche à droite et en profondeur d'abord, on obtient la séquence de nœuds 1 2 3 4 5 6 7 (on remarque que les attributs n'apparaissent pas dans ce parcours).

Propriétés structurelles de Node

La structure arborescente et les liens qui permettent de se déplacer dans cette structure sont représentés par les autres propriétés (sauf *attributs*) de l'interface **Node** (tableau 8, page 68).

Parmi les propriétés structurelles, la plus importante est *childNodes* qui est de type **NodeList** et donne la liste des fils d'un nœud de type *Container*. La structure générale d'un arbre DOM est montrée dans la figure 33. Pour chaque type de nœud on indique les enfants possibles par un arc. Tous les arcs sont étiquetés par *childNodes[]*, le symbole [] indiquant que les fils sont stockés sous forme d'une liste ordonnée.

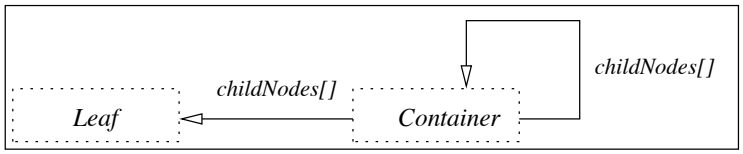

FIG. 33 – *Structure générale d'un arbre DOM*

Voici les autres propriétés :

1. *parentNode* référence le père d'un nœud ;
2. *firstChild* et *lastChild* référencent respectivement le premier et le dernier fils ;

3. *previousSibling* et *nextSibling* référencent respectivement le frère gauche et le frère droit [9]. Deux nœuds sont *sibling* s'ils ont le même père, mais pas s'ils sont au même niveau dans l'arbre mais avec des pères différents.

Le tableau 10 montre la « valeur » de ces propriétés pour quelques nœuds de l'arbre DOM présenté dans la figure 32 : pour les propriétés *parentNode*, *firstChild*, *lastChild previousSibling* et *nextSibling* la valeur correspond à une référence vers un nœud représenté par son identificateur. La propriété *childNodes* retourne une liste de nœuds (**NodeList**) qui est représentée par une séquence d'entiers entre crochets. Finalement, la propriétés *attributes* retourne un ensemble d'attributs (**NamedNodeMap**) que nous représentons sous forme d'un ensemble d'identificateurs d'attributs (un ensemble est noté {} pour le différentier d'une liste).

Bien entendu, tout ou partie de ces valeurs peuvent être à `NULL` (non définies) : il ne peut pas y avoir de père pour un nœud de type **Document**, pas de fils, de frère précédent ou suivant pour tous les nœuds de type *Leaf*.

Toutes les propriétés structurelles précédentes s'appliquent à *tous* les nœuds DOM (même les attributs) et contiennent comme valeur pour les nœuds qui ne sont pas de type *Container* des listes vides (par exemple pour *childNodes*) ou la valeur `NULL`. Il faut noter que la propriété *parentNode* n'est pas définie pour les nœuds de type **Attribute**.

Propriété	**Nœud**	**Valeur**
parentNode	1	`NULL`
	2	1
firstChild	5	6
	6	`NULL`
lastChild	5	7
	6	`NULL`
childNodes	2	[3, 5]
	4	`NULL`
previousSibling	3	`NULL`
	5	3
nextSibling	2	`NULL`
	3	5
attributes	2	{a, b}
	4	`NULL`

TAB. 10 – *Propriétés structurelles pour l'arbre DOM de la figure 32*

Des propriétés comme *firstChild* ou *lastChild* ne font *jamais* référence à des attributs. Le dernière propriété de l'interface **Node** que nous n'avons pas détaillé est *attributes*. Cette propriété, seulement définie pour les éléments, est de type **NamedNodeMap** qui fournit un ensemble d'opérations pour inspecter et mettre à jour les attributs d'un élément.

Remarque : Bien que l'interface **NamedNodeMap** permette de choisir un attribut par son nom (méthode `getNamedItem()`) *et* par sa position (méthode `item()`), rappelons que les attributs d'un élément ne sont pas ordonnés. En d'autres termes, si deux éléments se distinguent seulement par l'ordre des attributs dans la propriété *attributes*, ils sont considérés comme identiques.

Opérations du type Node

Passons maintenant aux opérations du type **Node**, données dans le tableau 11. Elles permettent de modifier un document en ajoutant, remplaçant et supprimant les fils d'un nœud. Les méthodes `hasChildNodes()` et `hasAttributes()` permettent de vérifier l'existence de fils ou d'attributs et la méthode

9. Le mot *sibling* en anglais désigne indifféremment le frère ou la sœur, et n'a malheureusement pas d'équivalent en français. Nous utiliserions volontiers le mot « sœur » si « nœud » était du genre féminin...

cloneNode() permet de créer une copie d'un nœud (ainsi que de tous ses descendants si la valeur du paramètre booléen *prof* est `true`).

Certaines de ces opérations ne sont pas autorisées sur certains types de nœuds, comme par exemple l'insertion d'un fils pour un nœud de type **Text**. Il est donc de la responsabilité de celui/celle qui programme un traitement de s'assurer qu'aucune opération interdite ne pourra être déclenchée par le traitement, ce qui là encore montre que DOM ne s'appuie que partiellement sur les concepts orientés-objet.

Résultat	Méthode	Paramètres	Explication
Node	*insertBefore()*	**Node** *nouv*, **Node** *fils*	Insertion du nœud *nouv* avant le fils *fils*; retourne *nouv*
Node	*replaceChild()*	**Node** *nouv*, **Node** *anc*	Remplacement du fils *anc* par le nœud *nouv*; retourne *anc*
Node	*removeChild()*	**Node** *fils*	Suppression de *fils* dans la liste des fils; retourne *fils*
Node	*appendChild()*	**Node** *nouv*	Ajout du nœud *nouv* à la fin de la liste des fils; retourne *nouv*
boolean	*hasChildNodes()*		Retourne `vrai` si le nœud à des fils et `faux` sinon
Node	*cloneNode()*	**boolean** *prof*	Retourne une copie du nœud

TAB. 11 – *Opérations du type* ***Node***

Exemple: parcours d'un arbre DOM

Les propriétés et les méthodes de l'interface **Node** peuvent être utilisées pour créer des programmes. Par exemple, la fonction suivante réalise un parcours d'un arbre DOM en *pré-ordre*, c'est-à-dire en profondeur à gauche :

```
fonction parcours_préordre(Node nœud) {
     print nœud→id;
     print ' ';
     si nœud→hasChildNodes()
          pour tous les nœuds fils dans nœud→childNodes
               parcours_préordre(fils);
}
```

La fonction *parcours_préordre()* prend un nœud DOM comme argument et utilise les propriétés *nodeName* et *childNodes* et la méthode booléenne *hasChildNodes()* de l'interface **Node**. La propriété `id` ne fait pas partie de l'interface **Node** et elle donne les identificateurs des nœuds que nous avons introduit plus haut.

L'application de cette fonction à un nœud imprime d'abord l'identificateur du nœud suivi d'un espace, vérifie ensuite si le nœud a des fils et, si c'est le cas, effectue un parcours pour chacun des fils dans l'ordre de la liste *childNodes*. Le résultat affiché par la fonction *parcours_préordre()* pour tous les nœuds dans l'arbre de la figure 32 est donné dans le tableau 12.

Pour les feuilles (4, 6 et 7), la fonction affiche uniquement leurs identificateurs. Pour les nœuds 3 et 5, elle affiche l'identificateur du nœud suivi de ses enfants (qui sont des feuilles). En remontant dans l'arborescence on arrive à la racine qui produit l'affichage de tous les nœuds du documents (sauf les attributs) dans l'ordre.

Comme nous l'avons déjà remarqué, l'ordre des nœuds est significatif (toujours en excluant les attributs XML). Deux documents ayant les mêmes nœuds mais pas dans le même ordre seront donc considérés comme différents. Cette notion d'ordre est reflétée par le type de la propriété *childNodes* qui est une *liste* de nœuds et permet de distinguer les fils d'un élément par leur position (le premier fils, le deuxième,

Appel de fonction	Paramètre	Résultat affiché
parcours_préordre()	1	1 2 3 4 5 6 7
	2	2 3 4 5 6 7
	3	3 4
	4	4
	5	5 6 7
	6	6
	7	7

TAB. 12 – *Parcours en pré-ordre de l'arbre DOM de la figure 32*

etc…). Ceci signifie, en pratique, qu'un traitement qui parcourt la liste des fils d'un élément ne donnera pas forcément le même résultat si l'ordre vien à changer.

Reprenons, par exemple, l'arbre de la figure 32, mais en changeant l'ordre des fils de la racine et de ceux du nœud 5 (la figure 34).

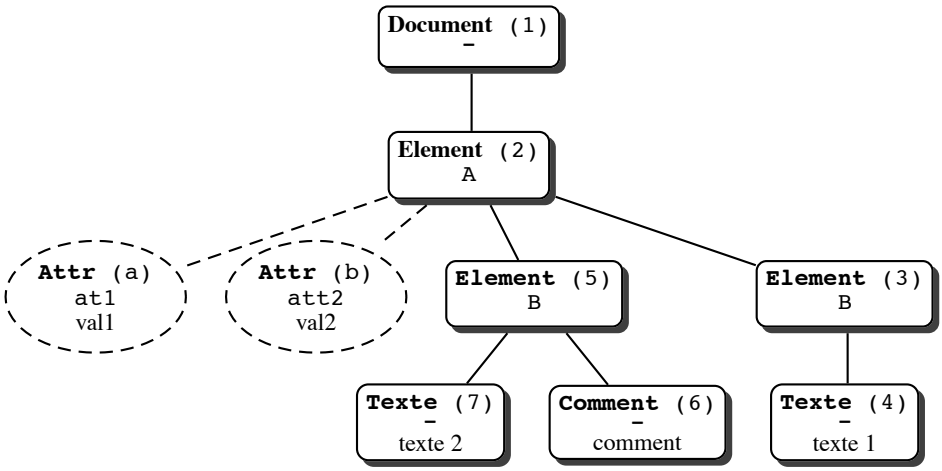

FIG. 34 – *Instance d'un arbre DOM avec un ordre différent*

Maintenant, si on applique la fonction *parcours_préordre()* aux nœuds de l'arbre de la figure 34, on constate que le résultat dans le tableau 13 est différent du résultat du parcours de l'arbre d'origine.

Appel de fonction	Paramètre	Résultat affiché
parcours_préordre()	1	1 2 5 7 6 3 4
	2	2 5 7 6 3 4
	3	3 4
	4	4
	5	5 7 6
	6	6
	7	7

TAB. 13 – *Parcours en pré-ordre de l'arbre DOM de la figure 34*

7.5 Interface Document

La structure d'un document XML bien formé doit respecter des contraintes précises. Ainsi, chaque arbre DOM qui correspond à un document XML a une seule racine de type **Document** qui représente la

racine d'un document XML (à distinguer de *l'élément racine* du document). Dans un arbre DOM, tout accès au contenu d'un document passe donc initialement par ce nœud.

Le nœud **Document** doit obéir aux règles suivantes :

– il a une liste de fils comprenant un nombre quelconque de nœuds de type **ProcessingInstruction**, **Comment** et un – un seul – nœud de type **Element**. Ce nœud est *l'élément racine du document* ;

– il peut avoir (ce n'est pas obligatoire) un fils de type **DocumentType** qui doit apparaître avant l'élément racine et qui correspond à la définition du type du document (voir page **??**).

Les nœuds qui précèdent l'élément racine constituent le *prologue* du document XML, et les nœuds qui suivent *l'épilogue*.

Nom	Type
doctype	**DocumentType**
implementation	**DocumentImpl**
documentElement	**Element**

TAB. 14 – *Propriétés du type* ***Document***

Le tableau 14 donne les propriétés du type **Document**. La propriété *doctype* est de type **DocumentType** et contient la DTD du document XML. Elle est à `NULL` si la DTD est absente. La propriété *implementation* est utilisée pour référencer l'objet « créateur » du document de type **DocumentImpl**. Souvent, mais pas obligatoirement, cet objet correspond au parseur qui génère une représentation DOM à partir d'un document XML. Finalement, la propriété *documentElement* référence l'élément racine du document.

Un nœud de type **Document** fournit des opérations permettant de *créer* tous les autres nœuds d'un arbre DOM. Ce type d'objet agit donc comme une « fabrique » des différents types de nœuds d'une arborescence DOM. Le tableau 15 donne quelques-unes de ces opérations :

Résultat	Méthode	Paramètres
Element	`createElement()`	**DOMString** *nom*
Text	`createTextNode()`	**DOMString** *texte*
Comment	`createComment()`	**DOMString** *texte*
CDATASection	`createCDATASection()`	**DOMString** *texte*
ProcessingInstruction	`createProcessingInstruction()`	**DOMString** *cible*, *texte*
Attr	`createAttribute()`	**DOMString** *nom*
EntityReference	`createEntityReference()`	**DOMString** *nom*

TAB. 15 – *Opérations du type* ***Document***

Chaque méthode `createX()` crée un objet de type *X* à partir des paramètres qui sont fournis sous forme de chaînes de caractères de type **DOMString** (voir section 7.2). Nous avons vu dans la section **??** que ces chaînes de caractères doivent respecter quelques contraintes quand il s'agit d'un nom d'élément, d'un nom d'attribut ou d'une cible dans une instruction de traitement (**ProcessingInstruction**). En particulier, ils ne doivent pas contenir d'espaces.

7.6 Interfaces Element et Attr

Un élément XML est un arbre et peut contenir des commentaires, du texte, des références vers les entités et d'autres éléments. De plus, un élément peut avoir zéro, un ou plusieurs attributs. Nous avons déjà souligné qu'un attribut est de type **Node** mais n'est pas traité de la même façon que les autres nœuds d'un arborescence DOM. Il existe plusieurs raisons à cette distinction :

– les attributs ne peuvent apparaître que comme fils d'un élément ;

– les attributs d'un élément ne sont pas ordonnés ;

– un élément ne peut pas avoir plusieurs attributs de même nom.

Pour toutes ces raisons, on peut considérer chaque élément à la foi comme un nœud dans un arbre DOM, et comme la racine d'un arbre non-ordonné dont les fils sont des attributs. Ces deux rôles d'un élément sont indépendants : on ne mélange jamais les fils « DOM » d'un élément et ses fils de type **Attr**. Cette distinction est illustrée dans la figure 35 par la représentation de deux compositions différentes.

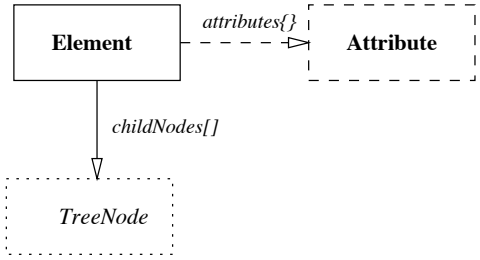

FIG. 35 – *Structure d'un élément*

Les opérations définies pour les éléments sont données dans le tableau 16.

Résultat	Méthode	Paramètres	Explication
DOMString	`getAttribute()`	**DOMString** `name`	Retourne la valeur de l'attribut `name`
DOMString	`setAttribute()`	**DOMString** `name`, **DOMString** `value`	Affecte la valeur `value` à l'attribut `name` ; retourne `value`
void	`removeAttribute()`	**DOMString** `name`	Supprime l'attribut avec le nom `name` de la liste d'attributs
Attr	`removeAttributeNode()`	**Attr** `oldAttr`	Supprime l'attribut `oldAttr` et retourne l'attribut supprimé
Attr	`getAttributeNode()`	**DOMString** `name`	Retourne la valeur de l'attribut `name`
Attr	`setAttributeNode()`	**Attr** `newAttr`	Insère l'attribut `newAttr` dans la liste des attributs (si un attribut du même nom existe déjà il est remplacé); retourne l'attribut remplacé ou `NULL`
NodeList	`getElementsByTagName()`	**DOMString** `name`	Retourne la liste des éléments descendants dont le type est égal à `name`; le résultat est trié dans l'ordre d'un parcours en pré-ordre
void	`normalise()`		*normalise* le contenu textuel d'un élément : un élément normalisé ne contient pas de nœuds de type **Text** adjacent (tous sont fusionnés en un seul nœud).

TAB. 16 – *Opérations du type **Element***

L'opération de normalisation est utile si on a ajoute plusieurs nœuds de type **Text** consécutifs comme fils d'un élément : dans la version sérialisée du même élément, ces fils consécutifs ne peuvent plus être distingués l'un de l'autre. Ainsi, l'application de la méthode `normalise()` à tous les éléments d'un arbre DOM donne un nouvel arbre qui correspond exactement à l'arbre qu'on obtient par une sérialisation

suivie d'une analyse par un parseur DOM. Cela peut être important dans le cas où on utilise des opérations qui dépendent du nombre ou de la position des fils d'un élément.

7.7 Interfaces DocumentType, Entity et EntityReference

La *Document Type Definition* (DTD) d'un document XML est représentée par un nœud de type **DocumentType**. Il est optionnel et ne peut apparaître qu'une seule fois dans un arbre DOM, comme propriété du nœud **Document**. Dans DOM, la seule information qui est accessible à partir de cette interface est la liste d'entités et de notations déclarés dans la DTD.

La notion d'entité permet de décomposer un document en plusieurs entités *physique* qui sont référencées dans le un document. Les interfaces **Entity** et **EntityReference** sont surtout importantes pour le passage d'une représentation sérialisée à une représentation DOM. Elles permettent de rassembler des informations distribuées pour créer un nouveau document et représentent précisément la structure *physique* d'un document XML.

Une fois la représentation DOM créée, la structure physique initiale devient peu importante et on peut se concentrer sur la vision *logique* du document. Nous n'allons pas nous appesantir sur ces notions plus que nécessaire, et considérerons que les références aux entités sont résolues (autrement dit la référence est remplacée par la valeur) au moment de l'application d'un programme XSLT. Ainsi, avec la disparition des entités, la DTD d'un document devient une boîte noire qui ne peut plus être examinée par un programme XSLT.

8 Du document sérialisé à l'arbre DOM

Nous allons maintenant comparer la version « sérialisée » d'un document XML avec sa représentation sous forme d'un arbre DOM. Cette comparaison permet de comprendre le passage d'une représentation à l'autre : dans le sens XML –> DOM, ce passage correspond à l'*analyse* d'un document texte par un parseur (processeur XML) qui construit une arborescence DOM pour le traitement du document par un programme. Le sens DOM –>XML correspond à la sérialisation d'un arbre DOM, sous forme de document texte.

Voici un document XML contenant des éléments avec attributs.

Exemple 8.1 *ExXML2DOM1.xml* : *Fichier XML avec attributs*

```
<?xml version='1.0' encoding="ISO-8859-1"?>
<A at1="val1">
  <B>du texte</B>
  <C>
    <D at2="0">autre texte</D>
    <D at2="1">texte</D>
  </C>
</A>
```

Le premier attribut est défini dans l'élément <A>. Son nom est `at1` est sa valeur est `val1`. Cet élément a deux fils <B> et <C> dont le premier contient un texte et le deuxième contient de nouveau deux fils avec un attribut `at2` chacun. L'arbre DOM correspondant est montré dans la figure 36.

Les attributs sont également représentés sous forme de nœuds dans l'arbre DOM (figure 36). Nous leur affectons une représentation différente à cause de leurs caractéristiques spécifiques.

Voici un autre exemple avec une instruction de traitement et un commentaire :

Exemple 8.2 *ExXML2DOM2.xml* : *Document XML avec instructions de traitement et commentaires*

```
<?xml version='1.0' encoding="ISO-8859-1"?>
<?xml-stylesheet href="feuille.xsl" type="text/xsl"?>
<A>
```

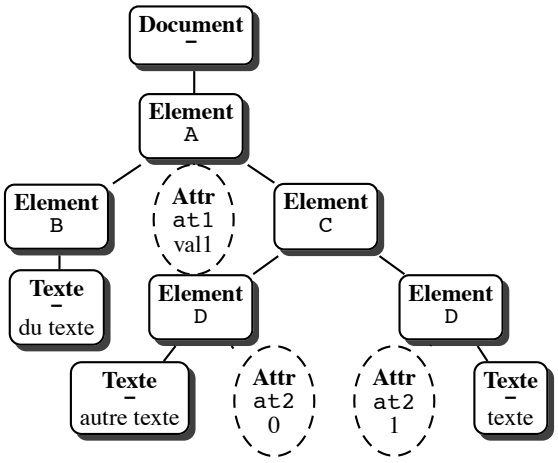

FIG. 36 – *Représentation DOM du fichier XML*

```
  <C>
    <!-- un commentaire -->
    <D at1="1">un &lt; et un &gt;</D>
  </C>
</A>
```

Ce document contient une instruction de traitement dans la deuxième ligne – bien que syntaxiquement conforme à la définition, la première ligne du document n'est pas une instruction de traitement. La cible de cette instruction est `xml-stylesheet` et elle indique au processeur XSLT l'URL du programme et le type du transformation à appliquer. Enfin le commentaire dans la cinquième ligne est une description courte de l'élément qui suit. Les instructions de traitement et les commentaires sont également représentés sous forme de nœuds dans l'arbre DOM (figure 37). On peut également constater que les références `<` et `>` on été remplacées par leurs valeurs respectives dans le fils de type **Texte** de l'élément `D`.

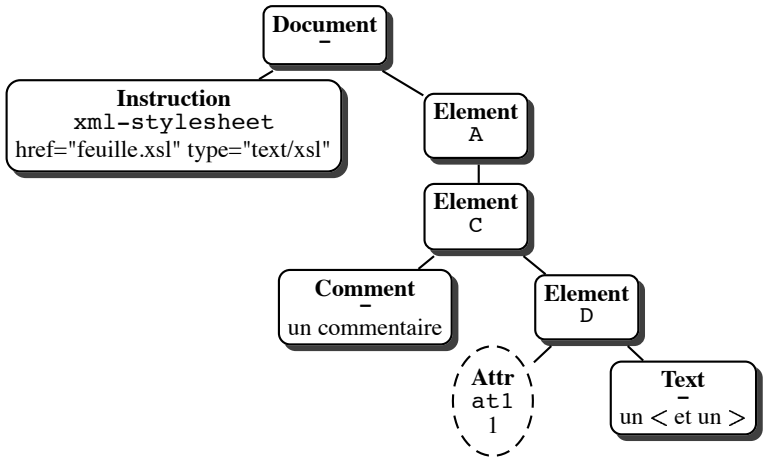

FIG. 37 – *Représentation DOM avec commentaires et instructions de traitement*

Au moment de l'analyse, le processeur se base sur des caractères réservés comme <, > et & pour structurer le document. En contrepartie le contenu textuel ne peut pas contenir de balises, ni même de

caractères comme < ou > puisque ceux-ci vont être interprétés comme faisant partie du marquage. Les sections **CDATA** permettent de prévenir cette interprétation.

Exemple 8.3 *ExXML2DOM3.xml* : *Document XML avec une section* ***CDATA***

```
<?xml version='1.0' ?>
<A>
&lt;B&gt;
<![CDATA[
<?xml version='1.0' ?>
<A>
 &lt;B&gt;
 &lt;/B&gt;
</A>
]]>
&lt;/B&gt;
</A>
```

Dans l'exemple ci-dessus, le contenu textuel du document XML comprend le contenu d'un nœud de type **Text** (« Voici l'exemple d'un document XML ») et le contenu d'une section **CDATA**. La figure 38 montre l'arbre DOM correspondant.

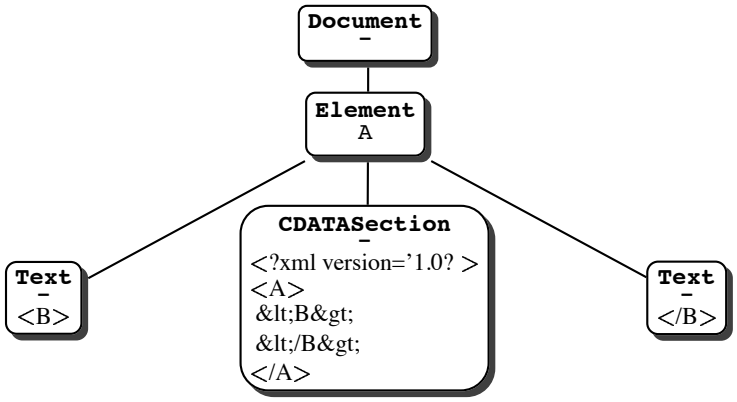

FIG. 38 – *Représentation DOM avec un nœud de type* ***CDATASection***

8.1 Construction d'un arbre DOM

Nous allons maintenant étudier plus en détail le schéma de construction d'un arbre DOM à partir d'un document XML sérialisé. Voici un fichier XML simple avec quatre éléments :

Exemple 8.4 *ExXML2DOMa.xml* : *Fichier XML*

```
<?xml version='1.0' encoding="ISO-8859-1"?>
<A>
 <B>texte simple</B>
 <C>
  <D> </D>
 </C>
</A>
```

Le document XML est représentée linéairement sous la forme d'une chaîne de caractères, et structuré à l'aide de balises qui peuvent être distinguées des informations textuelles par les symboles < et > qui les

entourent. La construction de l'arbre DOM consiste à parcourir cette chaîne séquentiellement en appliquant l'algorithme ci-dessous, décrit à l'aide des opérations DOM qui ont été présentées dans la section 7.

```
Document fonction parseDoc(Stream fic){
     Document document = new Document;
     appeler parse(fic, document, document);
     retourner document; }

fonction parse(Stream fic, Document document, Node père) {
     Node fils;
     DOMString fragmentXML;
     DOMString filsType;
     boolean finNode = false;
     tant que le flux fic n'est pas vide et not finNode {
          fragmentXML = lireToken(fic);
          si fragmentXML est une balise fermante
               finNode = true;
          sinon {
               filsType = getType(fragmentXML);
               fils = document→create(filsType);
               père→appendNode(fils);
               si filsType est un sous-type de Container
                    appeler parse(fic, document, fils);
               sinon
                    fils→value = extractValue(fragmentXML);
          }
     }
}
```

Voici quelques explications :

– La fonction *parseDoc()* peut être considérée comme une implantation simplifiée d'un parseur DOM : elle prend comme argument un fichier XML *fic* et retourne un document de type **Document** qui est le résultat de l'analyse. Elle crée tout d'abord un nœud de type `Document` qui est ensuite envoyé comme paramètre à la fonction *parse()*.

– La fonction *parse()* effectue l'analyse du fichier *fic* passé en paramètre. En plus des paramètres, elle définit quatre variables qui représentent respectivement un fils du nœud à construire (*fils*), son type (*filsType*), un fragment du document XML (*fragmentXML*) à analyser et une variable booléenne qui est égale à `true` si l'analyse du nœud *père* est terminée.

  Ensuite, la fonction entre dans une boucle qui lit successivement le fichier à analyser et s'arrête quand le fichier est consommé ou l'analyse du nœud actuel est terminé. Chaque étape de cette boucle effectue les tâches suivantes :

  – La fonction *lireToken()* (code non spécifié) sépare les segments du flux de caractères par rapport à leur type DOM. Ainsi cette fonction doit reconnaître toutes les balises (ouvrantes et fermantes), mais également distinguer les noms d'attributs et leur valeurs, et ainsi de suite.

  – Si le fragment retourné par la fonction *lireToken()* est une balise fermante, on sort de la boucle et de la fonction *parse()* en mettant *finNode* à `true`.

  – Sinon la fonction *getType()* (code non spécifié) détermine le type du fragment XML par sa balise (un fragment sans balise est reconnu comme un nœud de type **Text**). Ce type permet ensuite de créer un nœud du même type qui sera ajouté comme fils du nœud *père* :

`père→appendNode(fils)`
Si le fils peut avoir lui-même des fils, c'est-à-dire s'il est de type *Container*, on fait un appel récursif de la fonction `parse()` avec le fils comme nouveau père. Sinon le fils est une feuille et sa valeur est le fragment XML retourné par la fonction `extractValue()`.

Fragment lu	**Code exécuté**
<A>	b=a→createElement(A); a→appendNode(b); parse(fic,a,b);
#xA#x20	c=a→createTextNode(#xA#x20); b→appendNode(c);
<B>	d=a→createElement(B); g→appendNode(d); parse(fic,a,d);
texte simple	e=a→createTextNode(texte simple); d→appendNode(e);
</B>	retour (d)
#xA#x20	f=a→createTextNode(#xA#x20); b→appendNode(c);
<C>	g=a→createElement(C); b→appendNode(d); parse(fic,a,g);
#xA#x20#x20	h=a→createTextNode(#xA#x20#x20); b→appendNode(h);
<D>	i=a→createElement(D); g→appendNode(i); parse(fic,a,i);
#x20	j=a→createTextNode(#x20); i→appendNode(j);
</D>	retour (i)
#xA#x20	k=a→createTextNode(#xA#x20); g→appendNode(k);
</C>	retour (g)
#xA	l=a→createTextNode(#xA); b→appendNode(l);
</A>	retour (b)

TAB. 17 – *Fragmentation du document XML*

Ce parseur est très simplifié et, par exemple, ne prend pas en compte la la DTD du document et les entités. Néanmoins il devrait permettre une meilleure compréhension du passage XML –>DOM.

L'application de cet algorithme au document précédent est illustrée dans le tableau 17. Pendant l'analyse d'un document XML, les espaces (`#20`) et retours à la ligne (`#xA`) sont considérés comme des caractères. En effet, la norme du W3C préconise que tous ces « blancs » doivent être transmis sans modification par le parseur à l'application. L'arbre DOM résultant est représenté dans la figure 39.

8.2 Traitement des espaces pendant la construction

En étudiant cet arbre DOM on peut observer qu'il existe six nœuds de type **Text** – identifiés par les entiers 3, 6, 8, 10, 11 et 12 – qui ne contiennent que des espaces (*white-space node*). Tous ces nœuds

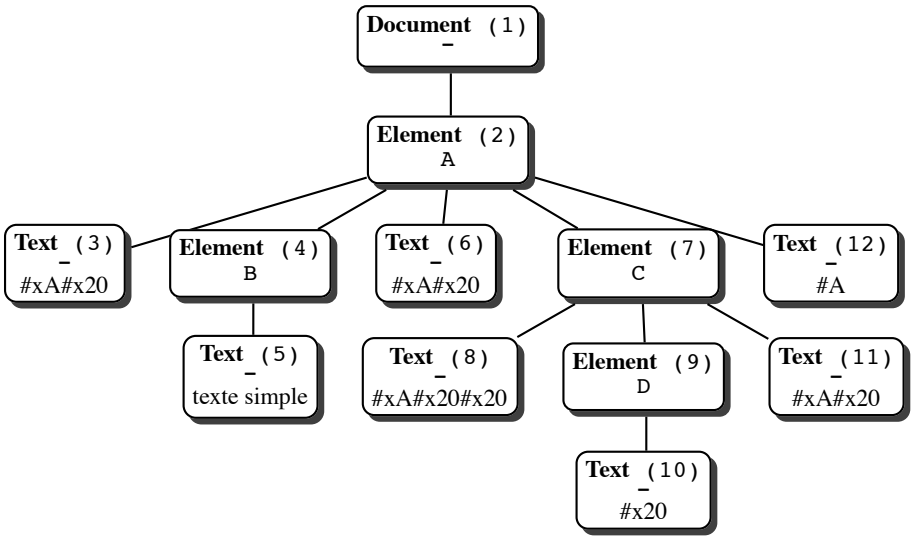

FIG. 39 – *Représentation DOM du fichier XML*

(*white-space node*) sauf le nœud 10 apparaissent comme nœud « séparateurs » entre deux éléments. Le nœud 10 est le seul fils d'un élément de type `D`.

Souvent les espaces entre deux éléments ne sont pas porteurs d'une information significative et servent essentiellement à améliorer la lisibilité d'un document sérialisé par une hiérarchisation du contenu. Aussi, la recommandation XML a décidé de distinguer entre les nœuds-espace significatifs et non-significatifs : cette distinction est seulement possible si le document a une DTD et si on utilise un parseur XML *validant* qui vérifie la conformité du document par rapport à sa DTD.

XSLT permet également de distinguer espaces significatifs et non-significatifs. Néanmoins, les créateurs de XSLT ont décidé que cette séparation ne doit pas dépendre du parseur utilisé avant le traitement XSLT, mais uniquement du contenu source du document et du programme XSLT. Ainsi, un programme XSLT suppose par défaut que tous les espaces dans un document (plus précisément dans l'élément racine de ce document) sont transmis au programme, à l'exception des espaces dans les valeurs d'attributs qui subissent un traitement spécifiques, et des fins de lignes qui sont normalisées et représentées par le caractère `#xA`.

Nous n'avons pas représenté, pour des raisons de lisibilité; tous les espaces dans les arbres DOM jusqu'ici. Nous continuerons à le faire quand les espaces ne sont pas cruciaux pour la représentation du contenu d'un document ou pour la compréhension du comportement d'un programme.

8.3 Deux fonctions de navigation

Pour conclure cette partie sur la syntaxe XML et le modèle DOM, voici deux exemples de fonctions qui permettent d'extraire des éléments dans un document XML. Elles prennent comme argument un nœud DOM et une chaîne de caractères qui correspond au type des éléments fils recherchés.

```
boolean fonction testNode(Node node, DOMString elType) {
     retourner (node→nodeType == ELEMENT_NODE) et
          (elType == "*" ou node→nodeName == elType);
}


NodeList fonction child(Node node, DOMString elType) {
     Node fils;
```

```
     DocumentFragment résultat = new DocumentFragment;
     integer i=0;
     tant que i < node→childNodes()→length() {
          fils = node→childNodes()→item(i);
          si testNode(fils, elType);
               résultat = résultat→appendChild(fils)
          i = i+1;
     }
     retourner résultat→childNodes();
}
```

La fonction *child(node, elType)* parcourt tous les fils du nœuds *node* et vérifie avec *testNode()* que le nœud correspond à un élément de type *elType* (cette condition est également vérifiée si *elType* est égal à « * »).

L'évaluation de cette fonction avec différents nœuds de l'arbre DOM correspondant au fichier *ExXML2DOM.xml* est illustrée dans le tableau 18. Nous utilisons toujours nos identificateurs pour distinguer entre les différents nœuds de l'arbre.

Appel de fonction	Résultat
child(1,"")*	[2]
child(1,"A")	[2]
child(1,"B")	[]
child(2,"")*	[4,7]
child(2,"B")	[4]
child(2,"C")	[7]
child(2,"D")	[]

TAB. 18 – *Application de la fonction* `child()` *à ExXML2DOMa.xml*

La fonction suivante cherche les éléments parmi les *descendants* de type **Element** du nœud donné et utilise la fonction *testNode()* définie plus haut.

```
NodeList fonction descendant(Node node, DOMString elType) {
     Node fils;
     DocumentFragment résultat = new DocumentFragment;
     NodeList descList;
     integer i=0, j;
     tant que i < node→childNodes()→length() {
          fils = node→childNodes()→item(i);
          si testNode(fils, elType)
               résultat = résultat→appendChild(fils)
          descList = descendant(fils, elType);
          nombreDesc = descList→length();
          j = 0;
          tant que j < descList→length() {
               résultat = résultat→appendChild(descListitem(j));
               j = j+1;
          }
          i = i+1;
     }
     retourner résultat→childNodes();
}
```

Comme la fonction `child()`, cette fonction parcourt les fils du nœud donné et sélectionne ceux qui satisfont le critère de sélection défini par le paramètre `elType`. Mais elle ne s'arrête pas là, et demande, par un appel récursif avec le même critère de sélection, les descendants de tous ses fils. L'évaluation de cette fonction est illustrée dans le tableau 19.

Appel de fonction	Résultat
`descendant(1,"*")`	[2,4,7,9]
`descendant(1,"A")`	[2]
`descendant(1,"B")`	[4]
`descendant(1,"C")`	[7]
`descendant(1,"D")`	[9]
`descendant(2,"*")`	[4,7,9]
`descendant(2,"B")`	[4]
`descendant(2,"C")`	[7]
`descendant(2,"D")`	[]

TAB. 19 – *Application de la fonction* `descendant()` *à ExXML2DOMa.xml*

L'appel `descendant(1,"*")`, ou 1 est l'identificateur de la racine du document, retourne tous les éléments du document XML.

8.4 Exemples en Java avec le parseur *Xerces*

Voici le fichier *Preordre.java*. Il contient les principales méthodes DOM.

Exemple 8.5 *Preordre.java* : *Parcours avec numérotation des nœuds*

```
// Programme analysant avec DOM un document XML
// et placant dans Sortie.xml le même document
// dans lequel on a ajouté le numéro
// de noeuds texte

// Import des classes Java

import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.apache.xalan.serialize.*;

class Preordre
{
  public static void main (String args [])
  {
    try
    {
       // Instanciation du parseur
       File fdom = new File (args[0]);
       DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
       DocumentBuilder builder = factory.newDocumentBuilder();
       // Analyse du document
       Document dom = builder.parse(fdom);
       Node elementRacine = dom.getDocumentElement();

       // On initialise le parcours avec le numéro 1
       Parcours (elementRacine, 1);

      // Sérialisation du résultat
```

```
      try
      {
        Serialiseur ser = new Serialiseur (dom);
        ser.sortie ("Sortie.xml");
      }
      catch (IOException e)
      {
        System.out.println ("Erreur ! " + e.getMessage());
      }
    }
    catch (Exception e)
    {
      System.out.println ("Erreur ! " + e.getMessage());
    }
 }

  private static int Parcours (Node noeud, int numero)
  {
      String str = new String();

      numero++;

      // Numérotation du noeud s'il est de type texte
      if (noeud.getNodeType() == Node.TEXT_NODE)
      {
         str = "(" + numero + ") " + noeud.getNodeValue();
         noeud.setNodeValue (str);
      }

      // Parcours récursif si le noeud a des fils
     if (noeud.hasChildNodes())
     {
       // Récupère la liste des fils du
       // noeud courant (liste de type NodeList)
       NodeList fils = noeud.getChildNodes();

       // Parcours de la liste et appel récursif
       for (int i=0; i < fils.getLength(); i++)
         numero = Parcours (fils.item(i), numero);
     }
     return numero;
  }
}
```

Les tableaux 40 et 41 donnent respectivement les principales méthodes des classes **Node** et **Document**. La méthode *getNodeType* de **Node** renvoie le type du nœud qui peut être `Node.ATTRIBUTE_NODE`, `Node.CDATA_SECTION_NODE`,`Node.ELEMENT_NODE`,`Node.DOCUMENT_NODE`,`Node.COMMENT_NODE` ou `Node.PROCESSING_INSTRUCTION_NODE`.

En ce qui concerne la classe **NodeList**, il n'existe que deux méthodes :

1. `int getLength()` retourne le nombre de nœuds dans la liste ;
2. `Node item(int i)` retourne le *i*ème nœud de la liste.

La classe **NamedNodeMap** a également une méthode *getLength()*, ainsi qu'une méthode *getNamedItem(String nom)* pour récupérer un attribut par son nom.

`Node appendChild(Node newChild)`	Adds the node newChild to the end of the list of children of this node.
`NamedNodeMap getAttributes()`	A NamedNodeMap containing the attributes of this node (if it is an Element) or null otherwise.
`NodeList getChildNodes()`	A NodeList that contains all children of this node.
`Node getFirstChild()`	The first child of this node.
`Node getLastChild()`	The last child of this node.
`Node getNextSibling()`	The node immediately following this node.
`java.lang.String getNodeName()`	The name of this node, depending on its type
`short getNodeType()`	A code representing the type of the underlying object
`java.lang.String getNodeValue()`	The value of this node, depending on its type
`Document getOwnerDocument()`	The Document object associated with this node.
`Node getParentNode()`	The parent of this node.
`Node getPreviousSibling()`	The node immediately preceding this node.
`boolean hasAttributes()`	Returns whether this node (if it is an element) has any attributes.
`boolean hasChildNodes()`	Returns whether this node has any children.
`void setNodeValue(String nodeValue)`	

FIG. 40 – *Méthodes de la classe* `Node`

`Attr createAttribute(String name)`	Creates an Attr of the given name.
`CDATASection createCDATASection(String data)`	Creates a CDATASection node whose value is the specified string.
`Comment createComment(String data)`	Creates a Comment node given the specified string.
`Element createElement(String tagName)`	Creates an element of the type specified.
`Text createTextNode(String data)`	Creates a Text node given the specified string.
`Element getDocumentElement()`	This is a convenience attribute that allows direct access to the child node that is the root element of the document.

FIG. 41 – *Méthodes de la classe* `Document`