Données Semistructurées et XML

Bernd Amann

amann@cnam.fr

Conservatoire National des Arts et Métiers

Paris

February 8, 2001

Contenu

INTRODUCTION

La révolution Internet

- Depuis 10 ans, Internet révolutionne l'informatique grand publique
- Au début, un serveur Web était essentiellement un **serveur de documents HTML**.
- Depuis, les serveurs Web sont devenus des vrais **serveur d'applications** :
  - qui accèdent aux bases de données
  - fournissent des notions de session, transaction, sécurité
  - avec des nouvelles technologies comme Java/Javascript, PHP, cookies,...

Nouvelles applications Web

- Commerce électronique:
  - Services: réservation de train, météo, ...
  - Bourse en-ligne
  - B2B, B2C
- Communautés Web (online communities): Napster, ...

Évolution des Systèmes d'Information

Années 70: Système de Fichiers $\rightarrow$ SGBD: séparation entre le stockage physique et la représentation logique des données par un modèle logique (architectures à deux niveaux/two-tier)

Années 90: Serveur de données monolithiques $\rightarrow$ Portails Web/Entrepôts de données/Portails d'entreprises: séparation entre les modèles logiques des données (SGBD relationnel ou objet, documents structurés, ...) et les applications par de modèles de médiation ou d'intégration (architectures à trois niveaux/three-tier)

Nouveau Besoin: Intégration de Données

Les nouvelles applications du Web ont besoin d'échanger, d'interroger et d'intégrer des données hétérogènes:

Bases de Données: relationnels, objets

Documents: XML, SGML, HTML

Données multimedia: figures, tableaux, photos, dessins, audio, video, spatiales

Modèles de Données Semistructurées

Modèles de données “universels” qui permet de représenter des structures

Irrégulières : on peut comparer des données dans formats différents (e.g. une chaîne de caractères avec un n-uplet)

Implicites : données et structures (grammaire, schéma) sont mélangées

Partielles : coexistence de données structurées et non-structurées

Exemples: OEM, XML, graphes/arbres étiquetés

Hétérogénéité des Données

Les données sont hétérogènes au niveau du structure et de la sémantique:

Structure :
- un même document peut exister sous format PostScript (vue plate), SGML/XML/HTML (semistructuré)
- le nom d'une personne peut être une chaîne de caractères ou un nuplet avec deux attributs (nom et prénom)

Sémantique : un nom d'attribut peut avoir différentes significations dans deux bases de données différentes (e.g. homonyme `adresse` = adresse professionnelle ou adresse personnelle)

Modèles Semi-structurées : Principes

Exemple

Trois sources de données:

- Base de données à objets avec une classe **Cinéma**:

```
Cinéma(nom: string,
       adresse: tuple(rue: string, numéro: integer),
       seance: set(tuple(heure: integer, film: string))
```

Une instance:

```
c1(nom = ‘‘St. André des Arts’’,
   adresse = tuple(rue = ‘‘St. André des Arts’’,
                   numéro: 13),
   seances = set(tuple(heure = 18 , film = ‘‘Brazil’’)
                 tuple(heure = 20 ,
                       film = ‘‘Apocalypse Now’’)))
```

- Base de données relationnelle avec une relation **Roles**:

```
create table Roles(acteur varchar(20), film varchar(20));
```

Une instance:

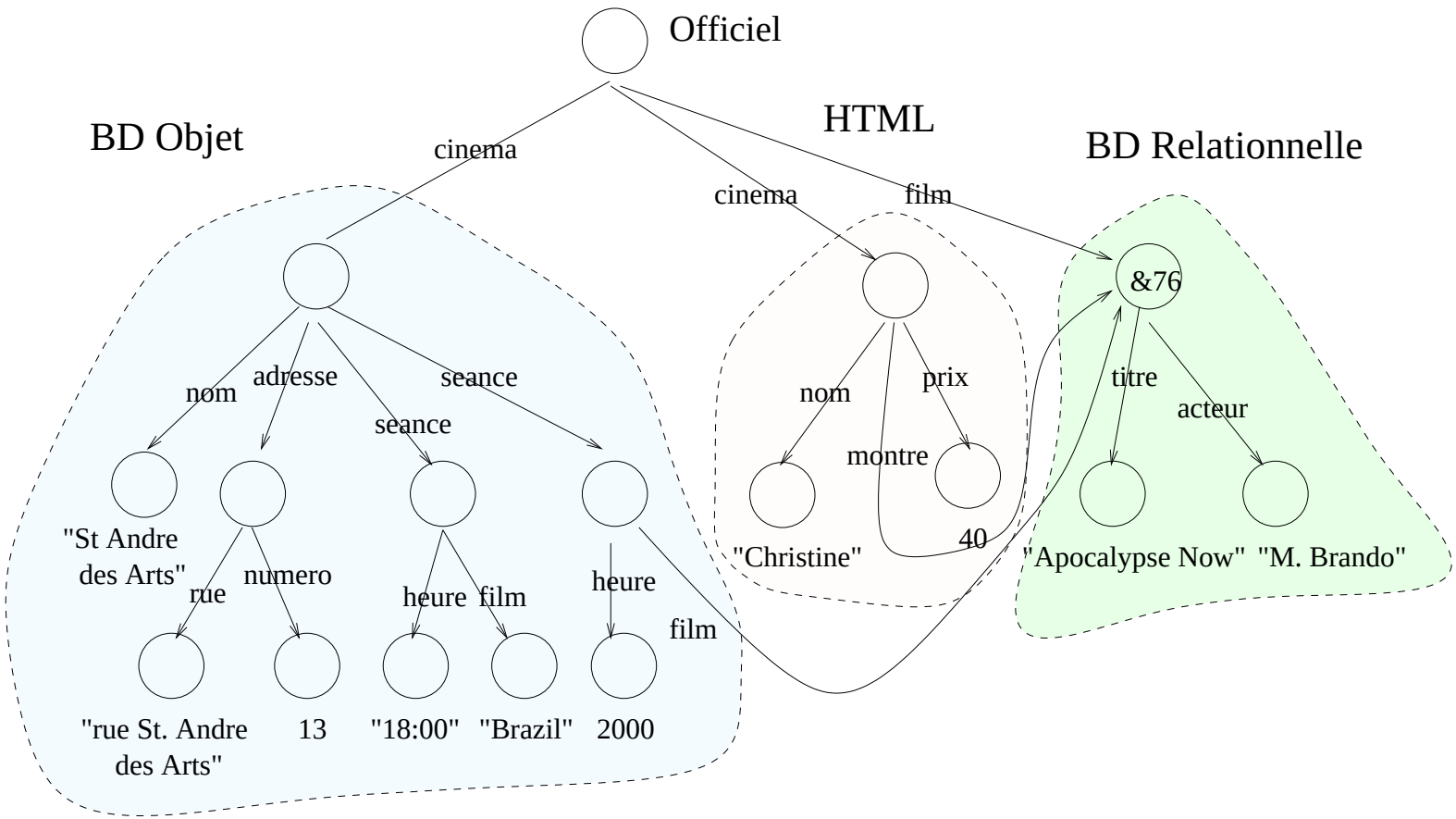
Roles	acteur	film
	Brando	Apocalypse Now
	Brando	Le Parrain
	...	...

- Un serveur Web qui contient des informations sur les films, les cinémas, les prix d'entrée:

```
<html>
 Le cinéma <bf>Action Christine</bf> (prix d'entrée 40 FF)
 montre actuellement le film <i>Apocalypse Now</i>.
</html>
```

Exemple: Graphe

Représentation uniforme des différentes types de données:



Exemple: Expression SSD

```
{ cinéma:
    { nom: ‘‘St. André des Arts’’,
      adresse:
        { rue: ‘‘rue St. André des Arts’’,
          numéro: 13 },
      séance:
        { heure: 2000,
          film: &76}},
 film: &76
    { titre: ‘‘Apocalypse Now’’,
      acteur: ‘‘M. Brando’’},
 ...
}
```

Exemple: Base de Données Relationnelle

Relation

R	A	B	C
	a1	b1	c1
	a2	b2	c2
	a3	b3	c3
	a4	b4	c4

Expression SSD

R: { tuple: {A: a1, B: b1, C: c1},
tuple: {A: a2, B: b2, C: c2},
tuple: {A: a3, B: b3, C: c3},
tuple: {A: a4, B: b4, C: c4}}

Donnée = Graphe étiqueté

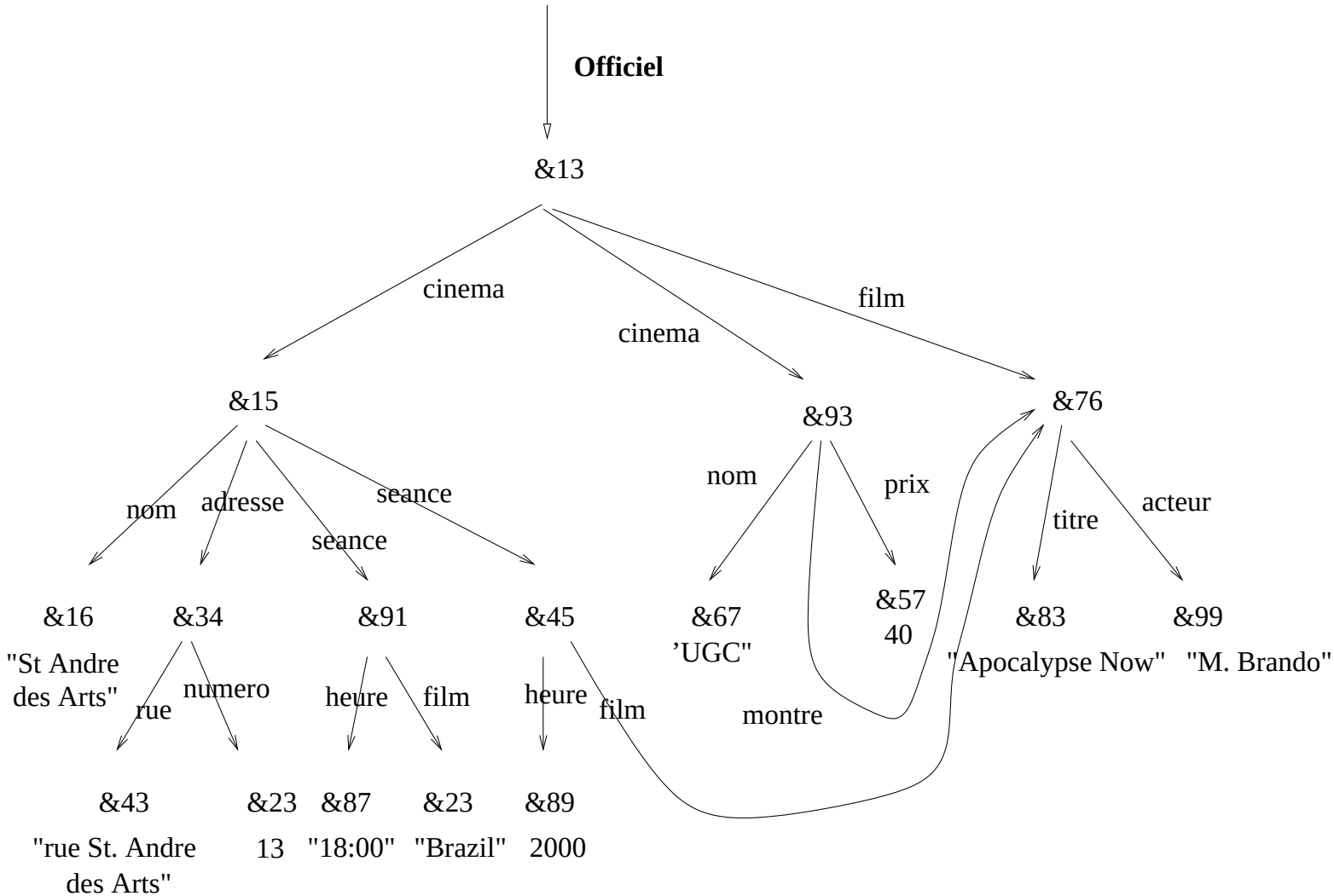
Plus formellement, on a

- ensemble fini de noms R (racines) et un ensemble de valeurs atomiques A
- un graphe étiqueté fini (V, E) où V est un ensemble d'oids pour les objets (atomiques ou complexes) et E est un ensemble d'arcs étiquetés par des chaînes de caractères.
- trois choix possibles: étiqueter les noeuds (XML, OEM), étiqueter les arcs ou étiqueter les deux (en principe il n'y a pas de grande différence)
- une fonction de nommage $name : R \rightarrow V$
- une fonction de déréférencage $valeur : V_a \rightarrow A$, qui associe à chaque objet atomique dans V une valeur atomique dans A.

Contraintes

- Les noeuds atomiques n'ont (par définition) pas d'arcs sortant.
- Chaque noeud doit être accessible à partir d'un objet $name(N)$ pour un N dans R.

OEM : Exemple Graphe d'Objets



XML: CONCEPTS DE BASE

HTML: Rappel

- HTML: HyperText Markup Language
- dernière version: XHTML 1.0
  - recommandation W3C
  - changement majeur depuis HTML 4.0: HTML devient une application de XML

Exemple

officiel.html:

```
<HTML>
  <HEAD><TITLE>Officiel du Spectacle</TITLE></HEAD>
  <BODY BGCOLOR="yellow">
    <H1>Officiel du Spectacle</H1>
    <H2>Cinémas</H2>
    <UL>
      <LI><B>St. André des Arts</B>,
        13, rue St. André des Arts, Paris
        <OL> <LI>18:00 - <A HREF="#brazil">Brazil</A></LI>
             <LI>20:30 - Matrix</LI> </OL> </LI>
    </UL>
```

```
<H2>Films:</H2>
      <UL>
        <LI><A NAME="brazil"><B>Brazil</B> avec:
          <OL>
            <LI>Jonathan Pryce</LI>
            <LI>Robert De Niro</LI>
          </OL>
        </LI>
      </UL>
    </BODY>
  </HTML>
```

Structure, Contenu et Présentation

- Dans un document HTML on “mélange” généralement le contenu et la présentation: le titre d’un film est entre les balises <B>...</B>, la couleur de la page est spécifié dans la balise <BODY BGCOLOR=...>, ...

  ==> Séparation du contenu et de la présentation: **HTML** avec **feuille de style CSS**

- Les informations dans une page HTML ne sont pas structurées: on ne peut pas distinguer le nom du cinéma, le titre du film, l’adresse du cinéma etc...

  ==> Utilisation de balises spécifiques à une application: **XML**.

Bibliographie

Bibliographie XML

Ces références biblio ont été trouvé avec Google. N'hesitez pas à en chercher d'autres!

- XML Générale:
  - W3C Web server : http://www.w3.org/
  - A. Michard, XML - Langage et Applications, Eyrolles
  - Transparents de ce cours (2 slides/page): http://sikkim.cnam.fr:/Cours/Cours-XML/poly.pdf
  - V. Aguiléra, XML et gestion de données semi-structurées, supports de cours
  - Tutorial XML et BD: http://www.cs.huji.ac.il/ atdb/Lectures/xml/index.htm
  - Serveur IBM Developerworks
- XSL:
  - Tutorial XSL:http://www.arbortext.com/xsl/
  - Projet XML Apache: http://xml.apache.org/

- Langages de Requêtes:
  - XML query languages : http://www.w3.org/TandS/QL/QL98/
  - Recommendation XQL: http://www.w3.org/TandS/QL/QL98/pp/xql.html, http://www.cuesoft.com/docs/cuexsl_activex/xql_users_guide.htm
  - XML-QL : http://www.w3.org/TR/NOTE-xml-ql/
  - Quilt : Quilt
- Xyleme: Serveur Web

Bibliographie Données Semistructurées

- Données-semistructurées:
  - S. Abiteboul, P. Buneman, D. Suciu: Data on the Web - from relations to semi-structured data and XML
- Langages de requêtes semi-structurés
  - S. Abiteboul et. al.: The Lorel query language for semistructured data, `ftp://db.stanford.edu/pub/papers/lorel96.ps`
  - Langages XML: Quilt, XML-QL (voir W3C)

Le langage de marquage XML

- XML: eXtensible Markup Language
- standard (recommandation W3C, www.w3.org) pour
  1. documents structurés : héritier de SGML
  2. documents Web: généralisation de HTML
- XML facilite(ra)
  1. l'échange de données sur le Web
  2. l'intégration d'applications Web
  3. l'interrogation du Web

Le World Wide Web Consortium (W3C)

- 400 partenaires industriels, parmi lesquels Oracle, IBM, Compaq, Xerox, Microsoft, etc..
- Laboratoires de recherche: MIT pour les États Unis, INRIA pour l'Europe, université Keio (Japon) pour l'Asie
- Objectif: définir un modèle pour faciliter l'échange de données sur le Web

Historique

- 1993: premiers travaux sur l'adaptation des techniques SGML au Web (Sperberg).
- Juin 1996: création d'un groupe de travail au sein du W3C
- 10 Février 1998: publication de la recommandation pour la version 1.0 du langage.

XML = Modèle Documents Structurés

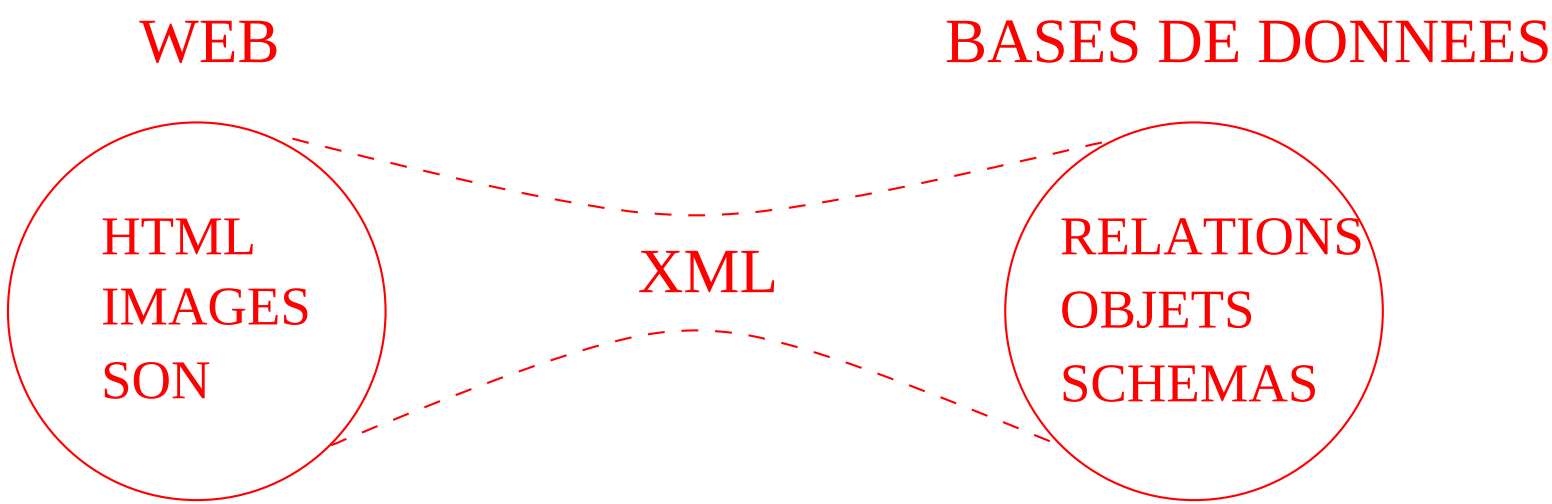
- XML est “compatible” avec SGML (standard pour documents structurés)
- l’édition de documents XML est simple (un éditeur texte standard suffit)
- la structure d’un document peut être prédéfinie par une grammaire (DTD) et analysée par un parseur
- le contenu d’un document est séparée de sa présentation: **feuille de style XSL**

XML = Syntaxe pour Transfert de Données

“ASCII du 21e siècle” (H.S. Thompson)

- ASCII (ISO 646) et UNICODE/ISO 10646: encodage de caractères
- XML: encodage/linéarisation de données
  - XML permet de représenter des données avec une structure irrégulière, implicite et partielle (semi-structurées).
  - les nouvelles techniques d’intégration et d’interrogation de données semi-structurées peuvent être appliquées.

Le Web et les Bases de Données



Notion de balisage structurel

- Principe clé de SGML
- Idée: séparer la structure logique de la présentation d'un document
- Avantages (par rapport à HTML):
  1. indépendance entre les outils de navigation (browser) et les outils de gestion de données (e.g. BD),
  2. différentes présentations pour le même document,
  3. indexation et l'interrogation "structurelle"

Un document XML

Fichier officiel.xml:

```
<Officiel> Ce document contient des informations sur des cinémas.
   <cinéma>
      <nom> St. André des Arts </nom>
      <adresse>
          <ville> Paris </ville>
          <rue> rue St. André des Arts </rue>
          <numéro> 13 </numéro>
      </adresse>
      <séance heure = ’18:00’ ref_film = ’&13’ />
      <séance heure = ’20:00’ ref_film = ’&14’ />
   </cinéma>  Voici l’information sur le film ’Brazil’ :
   <film film_id = ’&13’ actors = ’&156 &158’ />
      <titre> Brazil </titre><année> 1986 </année>
   </film></Officiel>
```

Explications

- l'élément `Officiel` est la racine du document de "contenu mélangé"
- un cinéma a un nom, une adresse et zéro ou plusieurs séances
- une séance contient un attribut `ref_film` qui permet de référencer d'autres éléments *dans le même document*. **Remarque** : les références ne sont pas typées.
- les films sont identifiés par la valeur de l'attribut `film_id`

DTD : Déclaration de la Structure d’un Document XML

Déclaration du Type de Document

Fichier officiel.dtd:

```
<!ELEMENT Officiel (#PCDATA | cinéma | film)*>
<!ELEMENT cinéma (nom, adresse, (séance)*)>
<!ELEMENT nom (#PCDATA) >
<!ELEMENT adresse (ville, rue, (numéro)?)>
<!ELEMENT séance EMPTY>
<!ATTLIST séance heure NMTOKEN #REQUIRED
          ref_film IDREF #REQUIRED>
<!ELEMENT film (titre, année>
<!ATTLIST film film_id ID #REQUIRED>
          actors IDREFS #IMPLIED>
<!ELEMENT titre (#PCDATA) >
...]>
```

DTD: Utilisation

On ajoute au début du document XML officiel.xml la clause DOCTYPE.

- Définition locale (Exemple):

```
<!DOCTYPE Officiel [
<!ELEMENT Officiel (#PCDATA | cinéma | film)*>
<!ELEMENT cinéma (nom, adresse, (séance)*)>
...]>
```

- Définition externe (Exemple):

```
<!DOCTYPE Officiel SYSTEM "officiel.dtd">
```

Documents XML valides et bien-formés

- document XML **bien-formé** :
  - pas de DTD
  - la structure est imbriquée (arborescence)
- document XML **valide** :
  - DTD existe
  - respecte la DTD (grammaire, élément racine, spécifications d'attributs)
  - respecte l'intégrité référentielle :
    - toutes les valeurs d'attributs de type ID sont distinctes
    - toutes les références sont valides

DTD: Pourquoi Validation?

Une DTD est une interface entre le producteurs et les consommateurs des données :

- le producteur peut contrôler la qualité des données produites
- le consommateur peut séparer la vérification syntaxique des données (parseur) de la logique de l’application

ELEMENT : Déclaration du Type de Élément

Un élément est défini par un nom et un modèle de contenu (MC):

- Chaque *expression régulière* (e.r.) e sur alphabet des noms d'éléments N est un MC
- `EMPTY` est un MC : élémént vide
- `ANY` est un MC : toute combinaison de tous les éléments
- `#PCDATA` est un MC : texte
- Contenu mixte : Si $n_1, n_2, ..., n_k$ sont des noms d'éléménts, alors $(\#PCDATA \mid n_1 \mid n_2 ... n_k)*$ est un MC.

Rappel: Expressions régulières

Expressions régulières sur alphabet des noms d'éléments N:

- chaque nom d'éléménd $n \in N$ est une e.r.
- si e est une e.r., alors $(e)*$ (cloture), $(e)+$ et $(e)?$ (option) sont des e.r.
- si e_1 et e_2 sont des e.r, alors (e_1, e_2) (séquence) et $(e_1|e_2)$ (alternative) sont des e.r.

ATTLIST : Déclaration des Attributs

- Syntaxe : `<!ATTLIST` élément nom type mode [default] `>`
- Types d'attributs (type):
  1. String : `CDATA`
  2. Enumerated : séquence de valeurs alternatives séparées par |
  3. ID, IDREF, IDREFS : identification et références
  4. ENTITY/ENTITIES : nom d'une entité non analysée déclarée ailleurs
  5. NMTOKEN/NMTOKENS : chaîne de caractères sans blancs
  6. NOTATION : une ou plusieurs notations (séparées par |)

ATTLIST : Mode

Modes d’attributs:

- `#REQUIRED` : la valeur doit être définie
- `#IMPLIED` : la valeur est optionnelle
- `#FIXED` : la valeur est constante

ATTLIST: Exemples

```
<!ATTLIST séance  heure NMTOKEN #REQUIRED
                  ref_film IDREF #REQUIRED>
<!ATTLIST film    film_id ID #REQUIRED
                  actors IDREFS #IMPLIED
                  langue (AN|FR|AL|ES|IT) #IMPLIED>
<!ATTLIST adresse ville CDATA #IMPLIED ’Paris’>
```

Entités générales et Entités paramètres

```
<!DOCTYPE Officiel [
<!ENTITY copyright ’Copyright B. Amann’>
<!ELEMENT Officiel (p, annéé) >
<!ELEMENT p (#PCDATA) >
<!ENTITY % text ’#PCDATA’>
<!ELEMENT année (%text;) >
]>
<Officiel>
<p> %copyright; </p><année>2000</année>
</Officiel>
```

- Entités paramètres : déclaration et utilisation dans DTD.
- Entités générales : déclaration dans DTD et utilisation dans DTD et document.

Entités externes

- Segmentation du document en plusieurs sous-documents
- Réutilisation de DTDs et de déclarations
- Références vers données non-XML (NOTATION)

Adressage:

- URL :

```
<!ENTITY % autre SYSTEM 'http://pariscope.fr/ext.xml' >
```

- FPI : formal public identifier

```
<!ENTITY % autre PUBLIC '-//CNAM//Texte libre//FR'>
```

NOTATION : entités non-XML

Utilisation:

- déclaration du format (type = application) pour entités non-XML
- référence à une entité de type notation seulement possible comme valeur d'attribut

```
<!DOCTYPE exemple [
<!NOTATION gif SYSTEM '/usr/local/bin/xv' >
<!ENTITY myphoto SYSTEM './moi.gif' NDATA gif >
<!ELEMENT person EMPTY >
<!ATTLIST person photo NOTATION (gif) #IMPLIED>
]>
<person photo='myphoto' >
```

Domaines nominaux (namespace)

- Un domaine nominal XML (namespace) est une collection de noms d’éléments ou noms d’attributs (identifiée par un URI).
- Utilisation: éviter les conflits de noms (par exemple, quand on utilise plusieurs DTD externes)

Domaines nominaux: Exemple

```
<?xml version='1.0'?>
<film xmlns:fi='http://www.pariscope.fr/films.dtd'>
  <acteur>
    <nom> George Clooney </nom>
  </acteur>
</film>
<theatre xmlns:th='http://www.comedie.fr/pieces.dtd'>
  <acteur nom='Juliette Binoche' />
</theatre>
```

- Noms qualifiés: `fi:film`, `fi:acteur`, `fi:nom`, `th:theatre`, `th:acteur` et `th:nom`

SELECTION DE FRAGMENTS XML

XPath: Sélectionner des Fragments XML

- XPath est fondé sur une représentation arborescente (DOM) du document XML
- Objectif : référencer noeuds (éléments, attributs, commentaires, ...) dans un document XML

XPath: Utilisation

XPath est utilisé par

- XML Schéma pour créer des clés et références
- XLink pour créer des liens entre documents/fragments XML
- XSL pour sélectionner des règles de transformation

XPath: Exemples

- La racine du document `Officiel.xml`:

  ```
  Officiel.xml/
  ```

- Tous les fils de type `film` de la racine du document XML :

  ```
  Officiel.xml/child::film
  ```

  ou (syntaxe simplifié)

  ```
  Officiel/film
  ```

- Tous les éléments de type `film` :

  ```
  Officiel.xml/descendant::film
  ```

  ou (syntaxe simplifié)

  ```
  Officiel.xml//film
  ```

Étape de positionnement

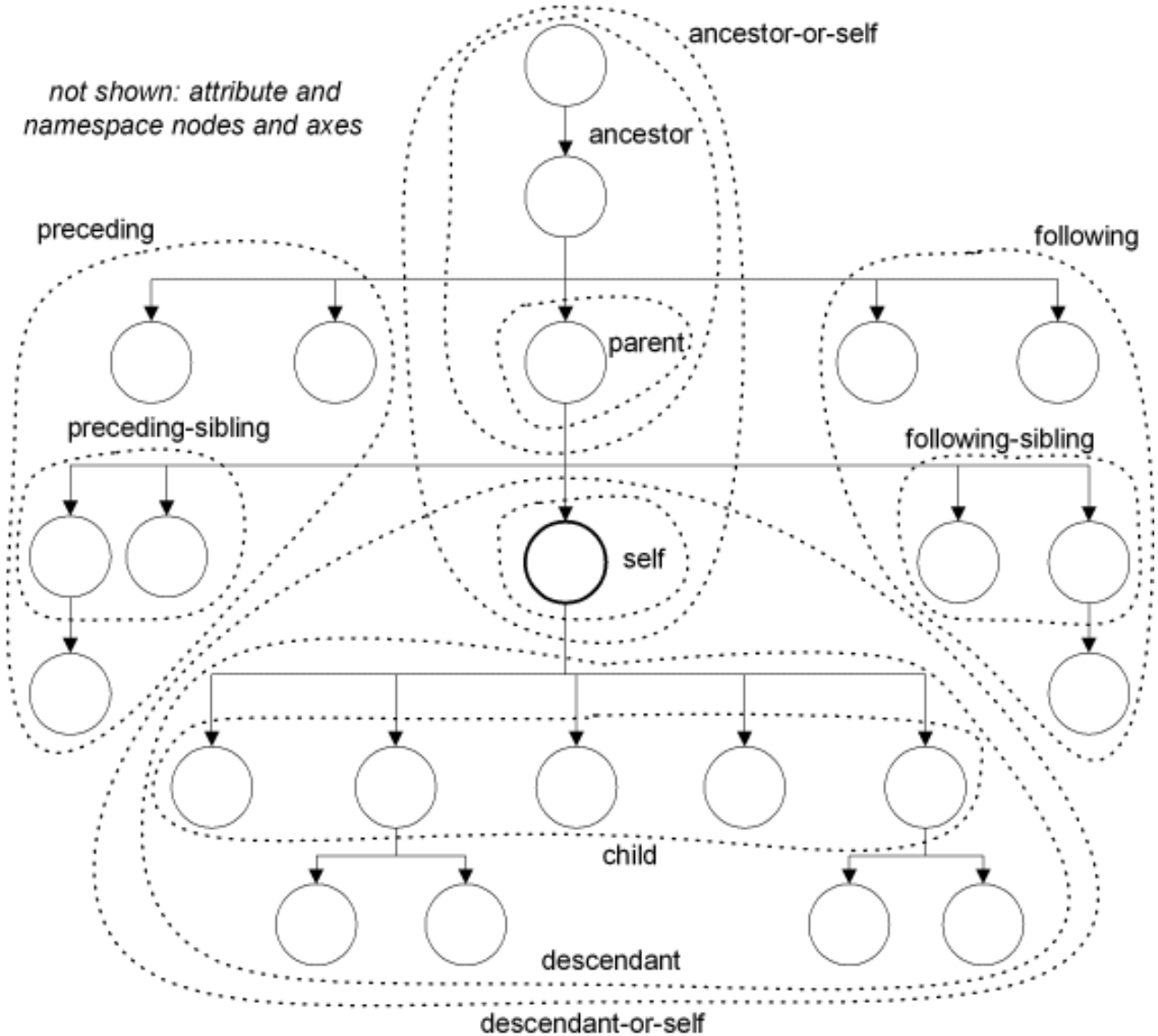
Une étape de positionnement est défini par un axe et un test:

1. l’axe sélectionne un ensemble de noeuds par rapport à leur position absolue ou relative à un autre noeud.
2. le test est évalué pour chaque noeud dans la sélection.

XPath: Axes

On distingue entre les

- *axes absolues* qui identifient la racine du document (/) et tout les éléments par leurs identificateurs (id(x)) et les
- *axes qui sélectionnent* les noeuds par leur position relatifs à un noeud:
  - noeuds enfants et descendants: `child, descendant`
  - noeud parent et ancêtres: `parent, ancestor, parent`
  - frères: `preceding-sibling, following-sibling`
  - noeuds précédents et suivants: `preceding, following`
  - noeud même: `self`


ancestor-or-self
not shown: attribute and namespace nodes and axes
ancestor
preceding
following
parent
preceding-sibling
following-sibling
self
child
descendant
descendant-or-self

Test: Nom/type des Noeud

- nom d'élément: `film`
- nom d'attribut: `@titre`
- type de noeud:
  - noeuds: `node()` (pas de sélection).
  - éléments: `*`,
  - attributs: `@*`,
  - noeuds de type texte (PCDATA): `texte()`:
  - commentaires: `comment()`
  - instructions d'exécution: `processing-instruction()`

Test : Prédicats Optionnels

- la position par rapport à l'axe choisi :
  - `child::*[3]`: le 3e enfant
  - `child::*[position()=2]`: le 2e enfant
  - `descendant::[position()=last()]`: le dernier descendant (parcours en pré-ordre)
- le nombre d'occurrences :
  - `cinema[count(child::seance) > 1]`: cinémas avec au moins 2 séances
  - `film[count(child::acteur) = 0]`: films sans acteurs
- l'existence d'attributs et d'éléments:
  - `film[not(child::acteur)]`: films sans acteur

- sélection par valeur:
  - `film[child::@titre='Brazil']`: le film Brazil
  - `acteur[normalize-space(child::prénom)='Bruce']`: les acteurs avec Bruce comme prénom
- la structure locale : chemins imbriqués avec connecteurs logiques (qualifiers)
  - `acteur[child::nom and child::datenaissance]`: les acteurs avec un nom et une date de naissance
  - `film[child::@titre='Brazil' and child::acteur/child::nom='De Niro']`: le film Brazil avec l'acteur De Niro

Chemins de Positionnement

Un *chemin* (de positionnement) est une séquence d'étapes de positionnement (location step) par rapport à un ensemble de noeuds données (contexte) :

`axe::test/axe::test/.../axe::test`

Exemples:

- /child::film/descendant::acteurs/child::@nom
- /descendant::cinema[@nom='Odeon']/descendant::film/child::@titre

Évaluation d’un Chemin de Positionnement

- Chaque étape de positionnement est évalué par rapport à un *contexte* créé par l’étape précédente:
- La première étape est un étape absolue (généralement la racine du document) et crée le contexte pour la deuxième étape.

XPath : Syntaxe simplifiée

XPath	Syntaxe simplifiée
`/child::film/child::acteur`	`/film/acteur`
`/child::cinéma/descendant::acteur`	`/cinéma//acteur`
`/descendant::*`	`//*`
`/descendant::film[@année='2000']`	`//film[@année='2000']`

Exemples

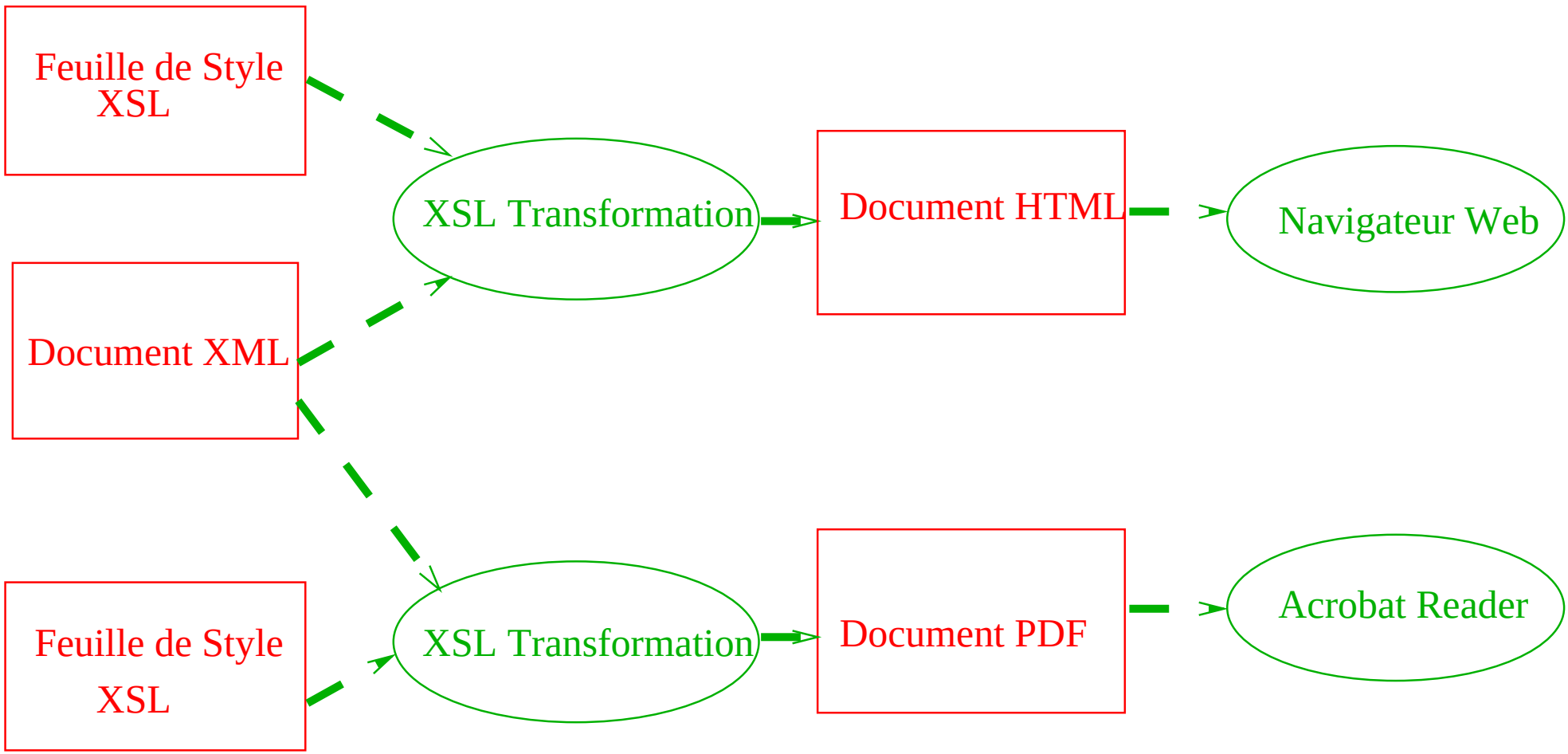
- `/Officiel/text()` ou `(/Officiel/child::text())`
- `//*` ou `descendant-or-self::*`
- `/..` ou `/ancestor::*`
- `//film|cinema`
- `*[name(.)='cinema' or name(.)='film']`
- `//text()` ou `descendant-or-self::text()`
- `//titre` `descendant-or-self::titre`
- `titre/..` ou `(titre/parent::*)`
- `//cinema/ancestor::*`
- `//annee/preceding-sibling::*`
- `//annee/preceding-sibling::*/text()`

- `//cinema[1]/nom`
- `//cinema[2]/nom`
- `//cinema[position()=1]/nom`
- `/*/*[position()=1]`
- `/*/*[position()=last()]`
- `/*/*[position()=1 and self::cinema]/nom`
- `/*/*[position()=2 and self::cinema]/nom`
- `//@film_id` or `(//descendant-or-self::node()/attribute::film_id)`
- `//cinema[contains(., 'Andr')]`
- `*[@*]`
- `//*[count(*)<2]`

TRANSFORMATION DE DOCUMENTS XML

XSL

- Working draft W3C (avril 1999)
- Efforts antérieurs : XSL s'est fortement inspiré de CSS (extension: XSL permet la transformation de la structure) et de DSSSL (SGML)
- Une feuille de style XSL utilise deux langages (la syntaxe est XML) :
  - XSLT: pour la transformation de la structure du document
  - Vocabulaire pour la spécification de la présentation (papier, écran, ...) $\rightarrow$ CSS
- XSLT domaine nominal (namespace) = http://www.w3.org/1999/XSL/Transform


Feuille de Style XSL
Document XML
Feuille de Style XSL
XSL Transformation
XSL Transformation
Document HTML
Document PDF
Navigateur Web
Acrobat Reader

Fonctions d’une Feuille de Style

Fonction de Base: Langage Transformation d’Arbres (documents XML):

- Génération de texte
- Suppression de contenu (noeuds)
- Déplacer texte (noeuds)
- Dupliquer texte (noeuds)
- Trier

Qu’est-ce qu’on peut faire?

- Transformer un document XML en un document HTML
- Transformer un document XML en un autre document XML
- Interroger un document XML

Example: Transformation en HTML

- XML
- Feuille de Style XSL
- Résultat Dynamique
- Résultat Stocké

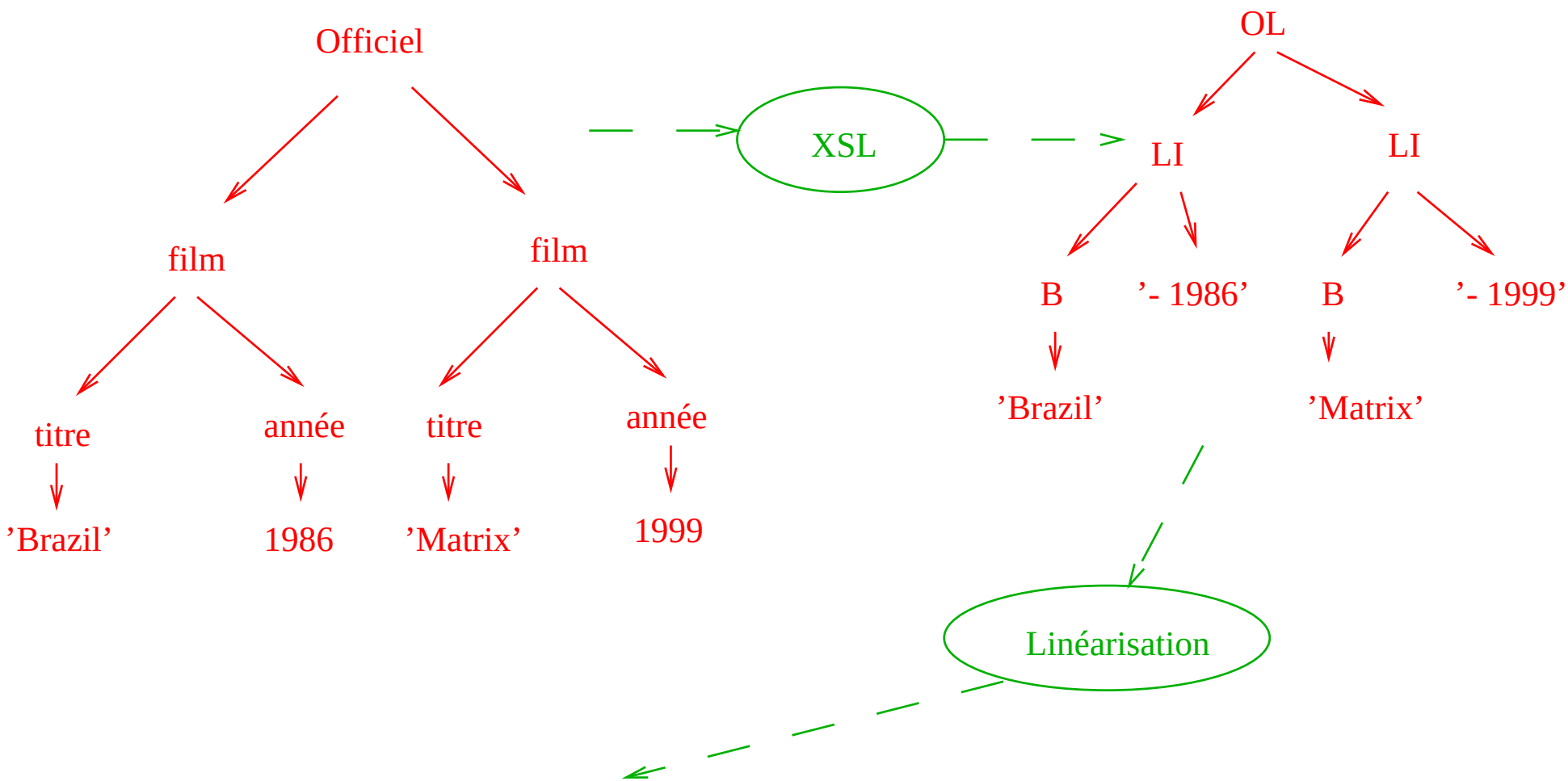
Structure d’une feuille XSL

- Une feuille XSL est un ensemble de règles
- Une règle associe un motif (expression XPath) à un constructeur ou modèle
- Le motif définit la structure à laquelle peut s’appliquer cette règle
- Le constructeur définit la structure du sous-arbre généré lors de l’activation de la règle
- Si plusieurs règles s’appliquent à un élément, la plus spécifique est retenue

Règles XSL : Exemple

Générer une liste HTML avec les titres et les années de production de films :

```
<xsl:template match='Officiel'>
   <ol>
     <xsl:apply-template match='film'/>
   </ol>
</xsl:template>
<xsl:template match='film'>
  <li><A name='film{@film_id}'/>
  <B><xsl:value-of select='titre'/></B> -
    <xsl:value-of select='annee'/>
  </li>
</xsl:template>
```



<OL><LI><B>Brazil</B>- 1986</LI><LI><B>Matrix</B>- 1999</LI></OL>

Règles XSL : Modèle de Traitement

- Une liste de noeuds (contexte) est traitée pour créer un fragment résultat
- Pour chaque noeud, on :
  - sélectionne la règle avec le motif qui correspond au noeud,
  - évalue le constructeur,
  - ajoute le fragment généré pour chaque noeud dans le résultat

Récursion: dans le constructeur d'une règle il est possible d'activer d'autres règles.

Règles XSL : Exemple

```
<xsl:template match='Officiel'>
   <ol>
     <xsl:apply-template match='film'/>
   </ol>
</xsl:template>
<xsl:template match='film'>
  <li><A name='film{@film_id}'/>
  <B><xsl:value-of select='titre'/></B> -
    <xsl:value-of select='annee'/>
  </li>
</xsl:template>
```

Règles XSL : Récursion

Problème : bien que XSL travaille sur des arbres (sans cycles) il est possible de créer des boucles infinies (“goto”) :

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

Résolution de conflits: Exemple

Cinémas parisien en gras avec la rue et autres cinémas avec le nom de la ville (XML, XSL, résultat dynamique, résultat stocké)

```
<xsl:template
  match='cinema[normalize-space(adresse/ville)="Paris"]'>
  <li><B><xsl:value-of select='nom'/></B>
  (<xsl:value-of select='adresse/rue'/>)</li>
</xsl:template>

<xsl:template match='cinema'>
  <li><I><xsl:value-of select='nom'/></I>
  (<xsl:value-of select='adresse/ville'/>)</li>
</xsl:template>
```

Résolution de conflits

Motif	Priorité par defaut
`film, @titre`	0
officiel:*	-0.25
*	-0.5
sinon	0.5

- Attribut `priority` pour les règles
- Les règles importées sont ignorées en faveur de règles locaux
- Une règle avec plusieurs alternatives (| dans le motif) est traduit en un ensemble de règles
- S'il restent plusieurs règles possibles : message d'erreur ou choix dans l'ordre inverse des déclarations des règles (?)

Règles par défaut

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

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

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

Les règles par défaut sont importées implicitement (priorité plus basse).

Selection de Règles: Modes

L'attribut `mode` permet de choisir une règle explicitement (XML, XSL, résultat dynamique, résultat stocké):

```
<xsl:template match="film/titre">
  <B><xsl:value-of select='.'/></B>
</xsl:template>
<xsl:template match="film/titre" mode='crossref'>
   <I><xsl:value-of select='.'/></I>
</xsl:template>
```

Appel:

```
<xsl:apply-templates select='//film[@film_id=$ref]/titre'
     mode='crossref'/>
```

Règle = Procédure avec Paramètres

La liste des cinémas (XML, XSL, résultat dynamique, résultat stocké):

```
<xsl:template match='cinema'>
  <xsl:call-template name='displaycinema'>
    <xsl:with-param name='nom'>
      <xsl:value-of select='nom'/>
    </xsl:with-param>
  </xsl:call-template>
</xsl:template>

<xsl:template name='displaycinema'>
  <xsl:param name='nom'>Nom inconnu</xsl:param>
  <li><I><xsl:value-of select='$nom'/></I></li>
</xsl:template>
```

Structures de controle: for-each, if, choose

La liste des cinémas (XML, XSL, résultat dynamique, résultat stocké):

```
<xsl:template match="//Officiel">
 <HTML><BODY><H1> La liste des cinémas</H1><ol>
  <xsl:for-each select='cinema'>
   <li><xsl:choose>
    <xsl:when test='adresse/ville="Paris"'>
     <b><xsl:value-of select='nom'/></b>
      (<xsl:value-of select='adresse/rue'/>) </xsl:when>
    <xsl:otherwise>
     <I><xsl:value-of select='nom'/></I>
     (<xsl:value-of select='adresse/ville'/>) </xsl:otherwise>
 </xsl:choose></li> </xsl:for-each></ol>
</BODY></HTML> </xsl:template>
```

Variables

- `<xsl:variable>` permet d'associer une variable avec une chaine de caractères, une liste de noeuds ou un fragment (arbre) XML.
  - `<xsl:variable name="foo" value="."/>`
  - `<xsl:variable name="foo"><A>...</A></xsl:variable>`
- Visibilité "léxicale":

```
<xsl:if test="...">
   <xsl:variable name="foo">...</xsl:variable>
</xsl:if>
```

Variables: Exemple

La liste des cinémas (XML, XSL, résultat dynamique, résultat stocké):

```
<xsl:variable name='counter'>0</xsl:variable>
<xsl:for-each select='cinema'>
  <xsl:variable name='counter'>
    <xsl:value-of select='$counter+1'/>
  </xsl:variable>
  <xsl:value-of select='$counter'/>.
  <I><xsl:value-of select='nom'/></I>
</xsl:for-each>
```

Attention: on n'obtient pas le résultat attendu.

Trier

Cinémas par ordre alphabétique (XML, XSL, résultat dynamique, résultat stocké):

```
<xsl:apply-templates select="cinema">
  <xsl:sort data-type="text" select=’nom’
            order =’ascending’/>
</xsl:apply-templates>
```

Clés

Les clés sont une généralisation du mechanisme ID/IDREF, et permettent un accès associatif rapide. XML, Feuille de Style XSL, résultat dynamique, résultat stocké :

```
<xsl:key name='filmkey' match='//film' use='@film_id'/>
...
<xsl:variable name='ref'
   select='key("filmkey",string(@ref_film))'/>
<xsl:choose>
  <xsl:when test="$ref">
    <A href='#film{@ref_film}'>
      <xsl:apply-templates select='$ref' mode='crossref'/>
    </A>
  </xsl:when>
  <xsl:otherwise><b>Film inconnu</b></xsl:otherwise>
</xsl:choose>
```

Génération d'Identificateurs d'Objets

La fonction `generate-id(.)` permet de créer et d'utiliser à des identificateurs de noeuds.

Exemple: Création de liens HTML vers des cinémas: (XML, XSL, résultat dynamique, résultat stocké):

```
<xsl:for-each select="//cinema">
  <a href="#{generate-id(.)}"><xsl:value-of select="nom"/></a>

</xsl:for-each>
...
<xsl:template match='cinema'>
  <a name="{generate-id(.)}"/>
....
```

Objets de Formatage (FO)

- deuxième partie du standard XSL
- vocabulaire XML pour la spécification de la forme (layout) d'un document
- résultat de la transformation: document XML avec FO
- résultat du formattage: PDF (apache FOP), Word, ...

DTD FO

- 56 types d’elements: fo:title, fo:page-number, fo:block
- > 250 attributs
- basé sur un découpage d’une page en rectangles (areas)
  1. regions: ex: page = 3 régions (entête, corps, bas de page)
  2. block areas: paragraphs, entré dans une liste
  3. line areas: ligne de texte
  4. inline areas: partie d’une ligne et “objets externes”

FO: Exemple Simple

```
<?xml version="1.0"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <fo:layout-master-set>
    <fo:simple-page-master master-name="only">
      <fo:region-body/>
    </fo:simple-page-master>
  </fo:layout-master-set>
 <fo:page-sequence master-name="only">
    <fo:flow>
      <fo:block font-size="20pt" font-family="serif" line-heigh
        Alien 1
      </fo:block>
      <fo:block font-size="20pt" font-family="serif" line-heigh
        Alien 2
```

```
      </fo:block>
    </fo:flow>
  </fo:page-sequence>
</fo:root>
```

Objets de Formatage (FO): Exemple PDF

- XML
- Feuille de Style XSL
- Résultat Dynamique
- Résultat Stocké PDF

INTERROGATION DE DOCUMENTS XML

XML-QL

- Proposition Workshop W3C, WWW99, Logiciel
- Syntaxe XML
- Expressions de chemins et motifs
- “Templates” pour la construction du résultat
- Complet dans le sens relationnel (algèbre/calcul)
- Modèle de données : graphe avec étiquettes sur les arcs (XML : noeuds étiquetés)
- Autres langages: XQL, Quilt

Exemple

Les films qu'on peut voir à l'Odéon :

```
where <cinema>
        <nom> Odéon </nom>
        <seance><film> $T </film> </seance>
      </cinema>
   in ‘‘www.officiel.com/cinemas.xml’’
construct $T
```

Résultat n'est pas bien-formé.

Construction de résultats

Les cinémas, séances et les films (relation) :

```
<answer>
where <cinema> <nom> $N </>
              <seance heure=$H>
                   <film> $T </>
              </seance></cinema>
       in ‘‘www.officiel.com/cinemas.xml’’
construct <result>
            <cinema> $N </> <heure> $H </> <film> $T </>
          </result>
</answer>
```

Transformation de l’attribut `heure` en élément.

Requêtes imbriquées : Éléments optionnels

Les films avec l'heure de présentation (si connue):

```
<answer>
where <cinema> <nom> Odéon </>
              <seance> $S </>
      </cinema>
   in ‘‘www.officiel.com/cinemas.xml’’,
      <film> $T </> in $S
construct <result>
           <film> $T </>
              where <heure> $H$ in $S
              construct <date> $H </>
          </result>
</answer>
```

Requêtes imbriquées : Regroupement

Chaque film avec les cinémas où on peut le voir :

```
<answer>
where <film> $F </>
      </film> in ‘‘www.officiel.com/cinemas.xml’’,
construct
   <film> $F
      where <cinema> <nom> $N </>
                <seance> <film> $F </> </>
            </cinema> in ‘‘www.officiel.com/cinemas.xml’’
      construct <cinema> $N </cinema>
   </film>
</answer>
```

“Syntactic Sugaring”

```
where <tag>...</tag>
      element_as $B in URL
construct ...
```

$B est affecté par le noeud <tag> ... <tag>

```
where <tag> ... </tag>
      content_as $B in URL
construct ...
```

$B est affecté par le contenu du noeud <tag> ... <tag>

Jointures

Les cinémas qui montrent le même film que l'Odéon :

```
where <cinema> <nom> Odeon </>
                <seance> <film> $T </> </>
      </cinema>
      content_as C1,
      <cinema> <nom> $N </>
                <seance> <film> $T </> </>
      </cinema>
      content_as C2 in ‘‘www.officiel.com/cinemas.xml’’,
      C1 <> C2
construct <cinema> $N </cinema>
```

Interrogation du schéma

Les éléments avec un attribut “id = &123” :

```
<answer>
where <$E id = ‘‘&123’’> $C </>
construct <$E> $C </>
</answer>
```

Expressions régulières

Les descendants de Charlemagne :

```
<answer>
where <personne><nom|name> $N </>
                <(enfant|child)*> $E </>
      </personne>
construct
      <descendant> $E </>
</answer>
```

Ne pas confondre avec la sémantique “horizontale” de `(enfant|child)*` dans une DTD.

Integration

Les films avec les cinémas et les critiques :

```
    where <film><titre> $T </> <critique> $C </></>
       in ‘‘www.cahier.com/films.xml’’
construct <film><titre> $T </>
             <critique> $C </>
             (    where <cinema> <nom> $N </>
                          <seance> <film> $T </> </> </>
                    in ‘‘www.officiel.com/cinemas.xml’’
              construct <cinema> $N </>)
       </film>
```

Integration: Fonctions de Skolem

Les films avec les cinémas et les critiques :

```
{where <film><titre>$T</> <critique> $C </> </>
    in ‘‘www.cahier.com/films.xml’’
 construct <film ID=filmid($T)>
             <titre> $T </>
             <critique> $C </>
           </film>
}
{where <cinema> <nom> $N </>
          <seance> <film> $T </> </> </>
    in ‘‘www.officiel.com/cinemas.xml’’
 construct <film ID=filmid($T)><cinema> $N </></>
}
```

Ordre

Deux sémantiques :

- modèle sans ordre
- modèle avec ordre :
  - le filtrage (pattern matching) dans la clause `where` est fait sans ordre
  - le résultat (clause `construct`) est trié par rapport à l'ordre dans le motif de la clause `where`

Ordre : Exemple

```
where <a><b> $B </b><c> $C </c></a>
construct <b> $B </b><c> $C </c>
```

- Document : `<a><b>b1<b><c>c1</c><b>b2</b><b>b3</b><c>c2</c>`
- Résultat :

```
<b>b1</b><c>c1</c>
<b>b1</b><c>c2</c>
<b>b2</b><c>c1</c>
<b>b2</b><c>c2</c>
<b>b3</b><c>c1</c>
<b>b3</b><c>c2</c>
```