# Rapport du projet XMLLiteParser

**Modélisation**

Mathis Deloge, Antoine Petot, Ange Picard

## 1 Descriptif du sujet

Un parseur / validateur XML-Lite est un programme capable de lire un fichier, d'indiquer s'il vérifie la norme XML-Lite et si oui, de l'analyser et de retenir sa structure ainsi que son contenu. Pour nous permettre de concevoir un programme réalisable, notre parseur / validateur opère sur un langage simplifié de XML, le XML-Lite conçu pour faciliter l'utilisation, les performances ainsi que les normes de conformité (XML 1.0).

### 1.1 Le XML-Lite

Pour être considéré comme du XML-Lite, les fichiers parsés / validés par notre programme doivent respecter certaines règles :

- Une balise possède un nom.
- Une balise doit être ouverte puis fermée.
- Une balise peut contenir du texte.
- Une balise peut contenir d'autre balises.
- L'ordre des balises filles n'a pas d'importance et tout le texte contenu dans une balise est regroupé en un seul bloc.
- Une balise fille doit être fermée avant la fermeture de la balise parent.
- Une balise peut contenir une balise du même nom.
- Un document doit commencer par l'ouverture d'une balise se fermant à la fin du document.

### 1.2 Exemple de fichiers XML-Lite

#### 1.2.1 Fichier XML-Lite correct

```
<FirstTag>
    <ChildTag>
        <AnotherChildTag>
        </AnotherChildTag>
    </ChildTag>
    <tag>
    </tag>
</FirstTag>
```

#### 1.2.2 Fichier XML-Lite invalide

```
  <FirstTag>
      <SecondTag>
        <EndTag>
          <AloneTag>
        </>
  </FirstTag>
  </SecondTag>
Un peu de texte
```

### 1.3 Structure du document

Le parseur / validateur doit être capable de lire n'importe quel fichier XML-Lite mais doit aussi être en mesure d'attendre une certaine structure de document grâce à l'ajout d'un fichier .dtd appelé schéma. Grâce aux fichiers schéma, le parseur / validateur connais avec plus de finesse les balises filles autorisées ou non pour chaque balises. C'est une sorte de modèle qui permettra la validation du fichier XML-Lite.

# 2 Journal de bord

## 2.1 Séance 1

Lors de la première séance, nous avons tout d'abord effectué le choix de sujet. Le parseur / validateur XML-Lite nous a intéressé étant donné le grand nombre de programmes fonctionnant avec XML pour la persistance et la souplesse de ce format de base de données, nous étions intéressé de découvrir les notions de bases du XML.

Par ailleurs, durant cette séance, nous avons trouvé des informations sur les validateurs de documents et avons pensé à implémenter un automate fini pour modéliser notre validateur. Le design objet "state pattern" semblait particulièrement adapté.

## 2.2 Séance 2

Lors de la deuxième séance, nous avons modélisé l'automate fini schématiquement, puis, nous l'avons implémenté. Il est utilisé pour valider le document. Nous avons également codé les différents états.

**Exemple d'un état du validateur**

```
1  public class NewTag implements State {
2      @Override
3      public State transition(char c) {
4          if (c == '/')
5              return new NewClosingTag();
6          else if ((c != '<') && (c != '>')) {
7              XMLLiteParser.getInstance().fillBuffer(c);
8              return new NewTagName();
9          } else
10             return new Error();
11     }
12
13     @Override
14     public boolean isFinal() {
15         return false;
16     }
17 }
```

Puis, nous avons réfléchis à la structure mathématique du pasreur, nous sommes vite arrivé à celle d'un arbre. Cette structure à l'avantage d'être facile à designer en objet. Nous avons donc implémenté deux classes :

- XMLLiteNode : Pour représenter une feuille ou un nœud de l'arbre.
- XMLLiteParser : Pour construire l'arbre.

Il a également fallu implémenter un buffer afin de stocker caractère par caractère les informations provenant des états du validateur.

## 2.3 Séance 3

Après avoir implémenté le parseur en structure d'arbre lors de la séance 2, nous avons pu interfacer une IHM basée sur les noeuds et feuilles de l'arbre nous permettant de faire une représentation claire et précise du fonctionnement du parseur / validateur qui nous serivra principalement lors de la présentation de projet.

L'implémentation de la classe JTree nous permet alors de représenter visuellement la structure du fichier XML-Lite analysé formaté en structure d'arbre.

Lors de cette séance, nous avons également mis en place un débogage à la volée des fichiers XML grâce à l'automate fini. Ainsi, on lève des exceptions quand il y a un problème avec l'arbre généré par nos états. Cela nous assure d'avoir en permanence un arbre cohérent ou une erreur le cas échéant.

## 2.4 Séance 4

Lors de la séance 4, nous avons dû faire face à une erreur n'arrivant uniquement lors de la lecture de gros fichiers. En effet, la validation de ces fichiers posait problème pour les dernières balises, le Parser rajoutait des balises filles au root node, à la fin, alors qu'elles n'existaient pas...

Nous avons pensé que ce problème provenait sans doute de notre classe qui lit les fichiers. Pour vérifier son fonctionnement, nous avons donc essayé de lire le fichier et de réécrire son contenu directement dans une copie. Et il est apparu qu'en effet, notre lecteur rajoutait des mauvais caractères en fin de fichier. Ceci a cause d'un buffer mal géré. Nous avons donc réécrit la classe de lecture des fichiers caractère par caractère avec des objets mieux adaptés.

## 2.5 Hors séance

Nous avons réalisé les prolongements concernant la vérification de structure d'un document, la définition de cette structure dans un fichier (DTD) et l'interprétation d'un fichier XML.

Nous avons d'abord réalisé les objets représentants les contraintes. La première « contrainte » sert à représenter une contrainte (enfant autorisé, obligatoire) sur un nom de balise donnée.

Puis en implémentant un deuxième automate fini, mais cette fois-ci avec la syntaxe d'un DTD-Lite. Nous avons construit les contraintes à partir d'un fichier externe de la même manière que nous le faisions dans l'automate pour valider le XML. Mais cette fois-ci au lieu de construire un arbre, on construit un ensemble de contraintes qui définissent un schéma.

Nous avons ensuite construit un algorithme (récursif) qui s'occupe de parcourir l'arbre, et s'il trouve une balise qui porte un nom ayant une contrainte dans le schéma, de vérifier si ses balises filles respectent la dite contrainte.

# 3 Choix du modèle mathématique

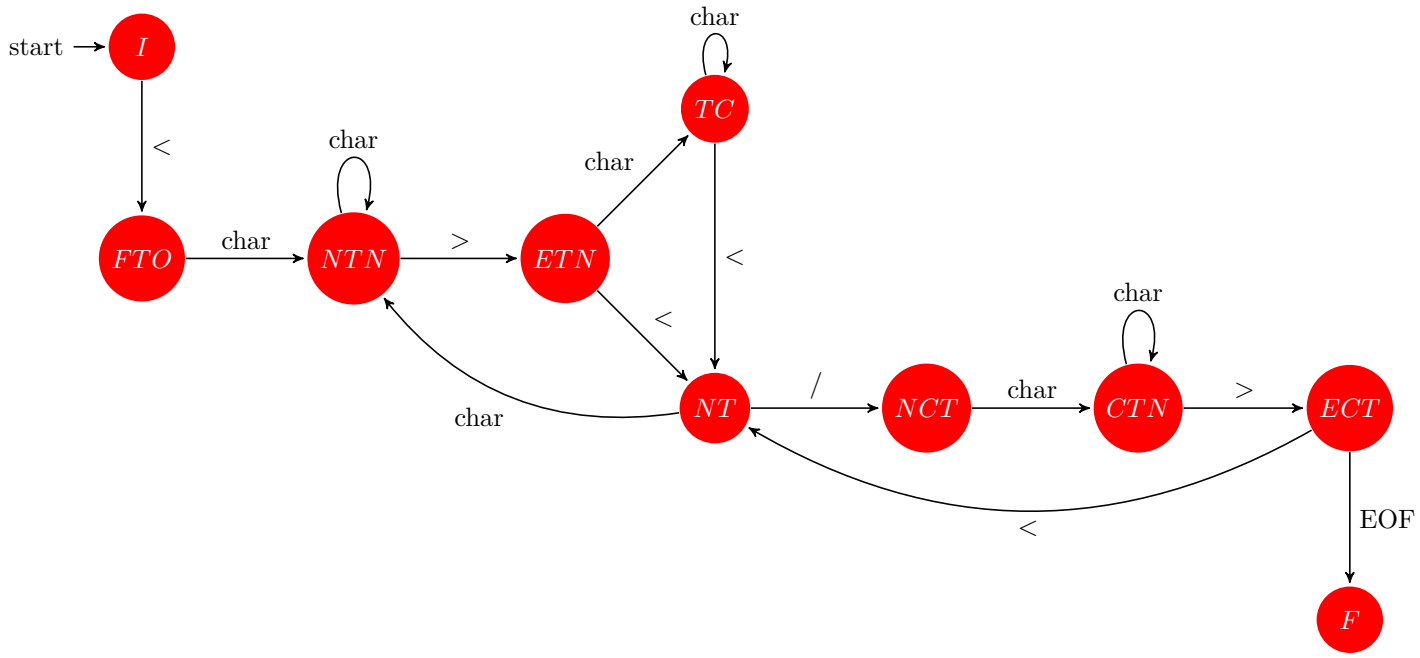
## 3.1 Le modèle mathématique

La principale contrainte du validateur était de permettre la correspondance au modèle syntaxique du fichier XML-Lite. // Pour nous permettre de vérifier la structure du document caractère par caractère (pour simplifier par la suite le débuggage du document), nous avons choisi de modéliser le parser par un automate fini

Enfin, ce modèle mathématique avec une implémantation et un débuggage relativement simple permet également une amélioration facile du programme grâce à la différenciation de tous les états dans différentes classes.

## 3.2 Représentation de l'automate fini

### 3.2.1 Schéma




### 3.2.2 Description des états

**I** Initial

**FTO** First Tag Opening

**NTN** New Tag Name

**ETN** End Tag Name

**TC** Text Content

**NT** New Tag

**NCT** New Closing Tag

**CTN** Colsing Tag Name

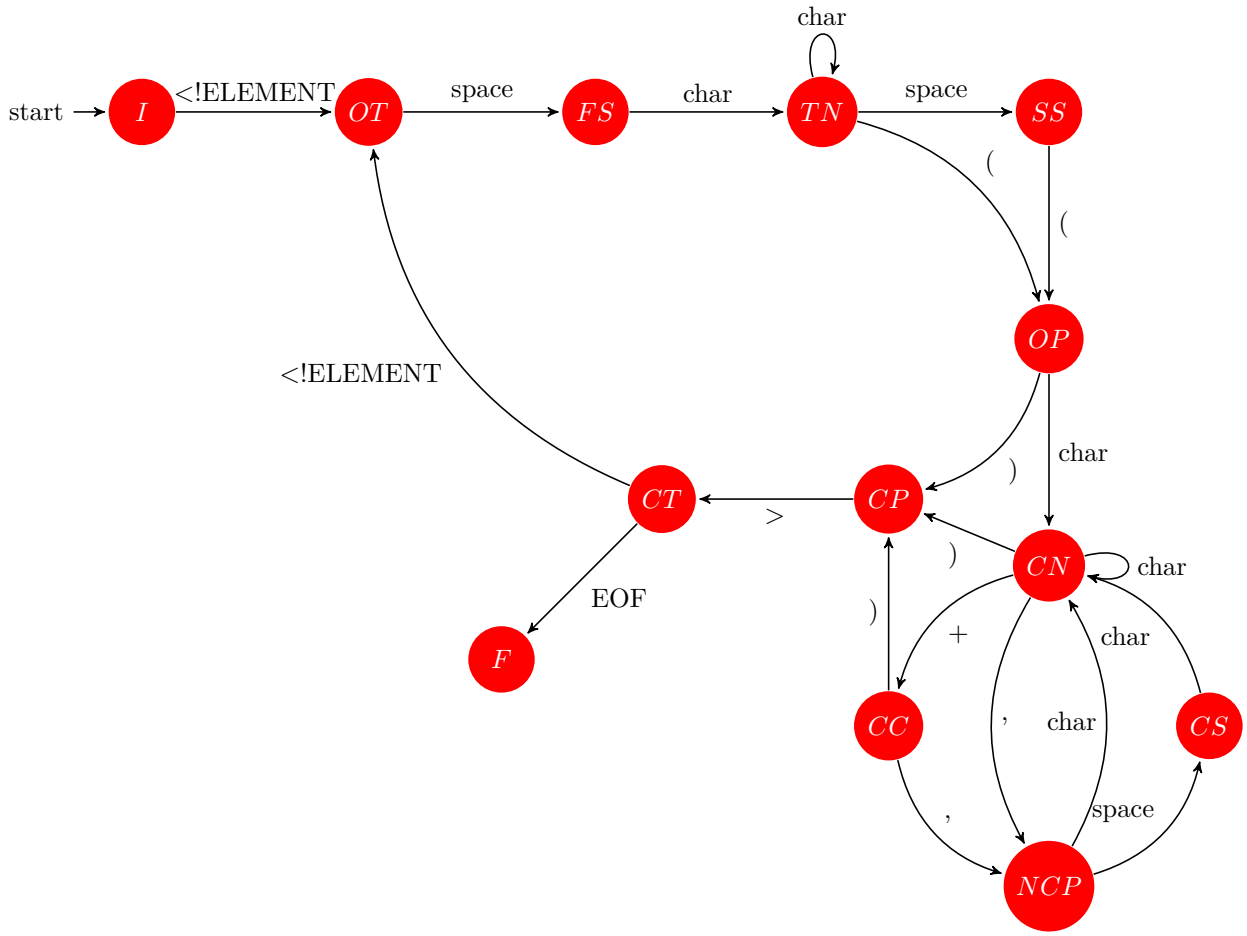
**ECT** End Closing Tag

**F** Final

## 3.3 Automate de validation du fichier DTD

Afin d'utiliser un fichier DTD, nous devons vérifier ce fichier DTD. Nous avons ainsi implémenter un automate fini pour vérifier la structure du schéma.

### 3.3.1 Schéma




### 3.3.2 Description des états

**I** Initial

**OT** Opening Tag

**FS** First Space

**TN** Tag Name

**SS** Second Space

**OP** Opening Parenthesis

**CN** Child Name

**NCP** Next Child Point

**CS** Child Space

**CP** Closing Parenthesis

**CT** Closing Tag

**F** Final

# 4 Conclusion

## 4.1 Les difficultés rencontrées

Comment débugger un document invalide ?
Arréter de parser à la première erreur et faire remonter l'erreur. (Exception)

Comment connaitre la position actuelle dans le document ?
Incrémenter un index à la lecture de chaque nouveau caractères.

## 4.2 Les outils acquis

En développant le débugger nous avons acquis une meilleur compréhension du systèmes d'exception. Déléguer le traitement d'un cas exceptionelle à l'utilisateur si il est plus approprié que celui le fasse. (Exemple : le parser renvoie les exceptions au TransitionSystem celui ci ayant accés à la position courante dans le document).

# Sommaire