CorpusReader : construction et interrogation de corpus multiannotés

Sylvain Loiseau*

* *LIMSI (CNRS)*
B.P. 133
91403 ORSAY CEDEX
sloiseau@limsi.fr

RÉSUMÉ. *CorpusReader est une plate-forme pour construire et interroger des corpus multiannotés. Ces corpus articulent plusieurs niveaux de description linguistique (morphologie, syntaxe, sémantique, etc.), de façon à permettre la construction d'observables associant plusieurs niveaux, ainsi que la description de corrélations entre niveaux de description. Les corpus multiannotés sont complexes à construire, à représenter et à interroger. Les spécificités de CorpusReader tiennent principalement au choix d'opérer une fusion* a posteriori *des annotations que produisent les outils d'analyse existants, plutôt qu'à organiser leur interopérabilité.*

ABSTRACT. *CorpusReader is a framework for creating and querying multi-layer corpora, which contain several levels of analysis (morphology, syntax, semantics, etc.) and which are aimed at observing correlations between these levels. Building, representing and querying multi-layer corpora is complex. CorpusReader's specificity essentially lies in merging the outputs of existing corpus analysis tools, avoiding the problem of integrating them at the software level.*

MOTS-CLÉS : *corpus multiannotés, linguistique quantitative, linguistique de corpus, XML, graphes d'annotation.*

KEYWORDS: *multi-layer corpora, quantitative linguistics, corpus linguistics, XML, annotation graphs.*

1. Introduction

CorpusReader est une plate-forme logicielle qui permet la construction de larges corpus articulant plusieurs annotations, que j'appellerai « multiannotés »[1]. Elle permet également de manipuler ces corpus multiannotés, au formalisme complexe, pour en extraire des sous-corpus, des représentations ou des quantifications.

Elle vise ainsi à rendre accessibles à la description quantitative de nouveaux objets empiriques. En effet, alors que de nombreux niveaux de description peuvent être annotés automatiquement, et que les outils dans ce domaine se sont stabilisés (au moins les niveaux morphologiques, morphosyntaxiques, syntaxiques et lexicaux), il est encore difficile de constituer des corpus articulant les annotations de plusieurs de ces outils. De plus, parallèlement aux instruments d'annotations linguistiques, c'est l'ensemble des briques méthodologiques des linguistiques de corpus qui sont souvent peu adaptées à des données plus « réalistes » où des annotations diverses sont coprésentes. Ainsi, les formats d'annotations les plus répandus (notamment XML) se prêtent mal aux représentations d'une pluralité de jeux d'annotations enchevêtrés.

Or, des corpus articulant plusieurs annotations rendraient accessibles à la description des objets d'un nouvel ordre de complexité empirique. La mise en regard de plusieurs niveaux de description linguistique augmente considérablement la diversité des régularités qui peuvent être observées. L'enjeu de tels corpus est notamment de pouvoir décrire les corrélations et les interactions entre niveaux de description et de construire des observables associant plusieurs niveaux de description. Cet objectif s'inscrit donc dans un mouvement général de développement des méthodes quantitatives en linguistique (Glessgen, 2007, p. 424) :

> « Les méthodes de quantification sont sur le point de transformer la recherche et l'histoire linguistiques comme elles ont transformé, il y a quelques décennies, la recherche sociologique ou psychologique. »

Les observables nouveaux que l'on peut construire dans des corpus multiannotés intéressent notamment la description des types de textes. On sait en effet que la prise en compte des types de textes est un préalable à l'amélioration qualitative des instruments d'annotations[2]. Ces types de textes sont le plus souvent caractérisés au moyen de descripteurs issus d'un seul niveau de description, notamment morphosyntaxique (morphologie flexionnelle). Or, il n'y a aucune raison pour que les régularités caractéristiques de types de textes ou les phénomènes permettant de les identifier le

1. J'ai plaisir à remercier pour leur relecture et leurs conseils Cyril Grouin, Sarah Leroy, Patrick Paroubek et Pierre Zweigenbaum. Je remercie également les trois relecteurs anonymes de la revue TAL, dont les commentaires et les critiques ont été extrêmement utiles. Ce travail a bénéficié d'un financement de l'Agence National de la Recherche (ANR), pour le projet Passage (ANR-06-MDCA-013-02).

2. Ainsi Habert écrivait-il en 2000 : « Plusieurs études convergent en effet pour rendre plausible l'hypothèse selon laquelle la fiabilité des traitements automatiques dépendrait de l'homogénéité des données en cause » (Habert, 2000).

plus efficacement soient limités à l'intérieur d'un seul niveau de description et n'établissent pas des solidarités entre niveaux de description. Dès lors que l'objectif n'est plus seulement la modélisation d'un niveau de description, mais la prise en compte de leur fonctionnement dans la variation des usages pour caractériser les objets empiriques que sont les textes, la prise en compte des interactions entre niveaux est une nécessité méthodologique.

C'est ce que soulignait, dès 1994, (Baayen, 1994, p. 32), du point de vue d'un objectif de classification de textes : « *This suggests that the combined quantitative analysis of morphology on the one hand [. . .] and syntax and pragmatics on the other [. . .] constitutes a robust and fruitful line of inquiry into the sociolinguistic and stylistic aspects of language use.* » ; ou, plus récemment, en apprentissage, (Yvon, 2006, p. 34) : « [. . .] la description linguistique fournit des descripteurs de plusieurs niveaux, qu'il peut être souhaitable d'utiliser conjointement »[3].

La description des solidarités entre niveaux peut fortement contribuer, en retour, à l'amélioration des instruments d'annotations si elle permet de prendre en compte les régularités des types de textes.

CorpusReader tente donc d'aider à la description quantitative des interactions entre niveaux de description. Il n'apporte aucune compétence en annotation linguistique ni en statistique, mais tente d'être un « pont » permettant de soumettre à l'analyse quantitative, et à la description, des corpus articulant plusieurs niveaux d'annotations.

Il s'inscrit dans une génération d'outils et de formalismes qui tentent de répondre à la même problématique, notamment Gate (Bontcheva *et al.*, 2004), NITE (Carletta *et al.*, 2003), Atlas (Bird *et al.*, 2000) ou LinguaStream (Bilhaut et Widlöcher, 2006). Les spécificités de CorpusReader tiennent particulièrement au fait que le corpus produit est aussi peu dépendant que possible du dispositif logiciel.

Dans une première partie je présenterai et justifierai les choix d'architecture de CorpusReader en insistant sur l'importance particulière prise par le format de données dans ce dispositif. Dans une deuxième partie je présenterai deux aspects de la question de la réalisation de corpus multiannotés : le choix d'un formalisme de représentation et le choix d'une méthodologie de fusion des instruments d'analyseurs tiers[4]. Dans une dernière partie j'essayerai de tirer les conséquences des choix de CorpusReader du point de vue des implications méthodologiques et de la nature de l'artefact linguistique produit.

3. On peut également remarquer que Biber, dès ses premières expériences, utilisait un jeu de variables relativement hétérogène du point de vue des niveaux de description. Les variables sont cependant présentées comme relevant de « six catégories grammaticales majeures » (Biber, 1988, p. 72), sans que ne soit thématisée cette hétérogénéité. Ces variables sont sélectionnées pour permettre l'observation de l'opposition entre les modalités orale et écrite. *Cf.* également (Loiseau, sous presse)

4. Je passerai plus rapidement sur ces deux aspects qui ont été exposés plus en détail dans (Loiseau, 2007).

2. Objectifs et choix

2.1. *Objectifs : explorations quantitatives de corpus complexes*

Les objectifs de CorpusReader sont d'une part de permettre de construire des corpus articulant les sorties de plusieurs instruments d'annotation, et d'autre part de pouvoir extraire de ces corpus au formalisme complexe des structures de données plus simples résultant d'une composition de traits à façon dans le corpus d'origine : liste de fréquences, matrice de cooccurrences, graphes de cooccurrences – ou simplement concordancier.

Une analyse produite avec ce dispositif (Loiseau, 2006) et illustrant cet objectif est présentée ci-dessous, en insistant sur les difficultés de la construction de cet objet.

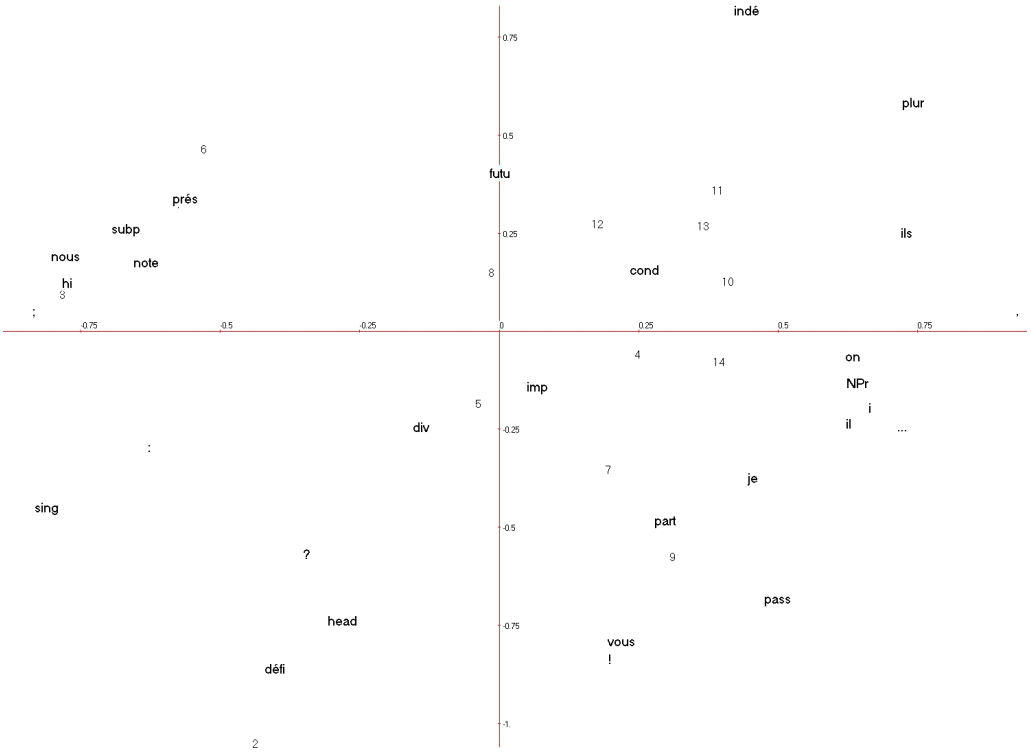

Figure 1. *Analyse factorielle (ACP) de quatorze textes de Gilles Deleuze*

Dans cet exemple, il s'agit de caractériser, par une analyse factorielle, quatorze textes de Gilles Deleuze. Des descripteurs intéressant principalement le système énonciatif ont été utilisés. Chaque texte a été caractérisé par plusieurs variables, relevant de plusieurs niveaux de descriptions ou de plusieurs sémiotiques : fréquences relatives des temps et des modes, des pronoms personnels, des signes de ponctuation, des noms

propres parmi l'ensemble des noms, et enfin de différentes marques de segmentation textuelle (italiques, citations, divisions, titres, paragraphes). Certaines variables représentent des informations encore plus « fines », comme les proportions relatives de traits « indéfinis » et « définis » dans les déterminants.

Une analyse en composantes principales (ACP) a permis de caractériser les textes par des ensembles de variables. Sur le plan issu des deux premiers facteurs (figure 1), les textes sont numérotés chronologiquement. Sur l'axe du premier facteur (horizontal), on retrouve l'ordre chronologique (les « retours en arrière » sont dus aux commentaires, genre plus conservateur que l'essai).

Du point de vue des variables, tous les niveaux de descriptions représentés varient sur les deux axes représentés. Par exemple, du point de vue des personnes, les textes sont caractérisés successivement par la première personne du pluriel (le *nous* académique), puis les deuxièmes personnes, enfin par le *je* et le *on* dans les derniers textes. Sur le plan des signes de ponctuation, les premiers textes sont caractérisés par le point-virgule, les textes centraux par le point d'exclamation, enfin les derniers textes par les points de suspension. Les marques de structuration du texte comme les italiques (*hi*), titres (*head*), ou notes (*note*) sont plus fréquentes dans les premiers textes. Les temps et modes les plus caractéristiques sont d'abord le subjonctif présent (*subp*), puis l'impératif (*imp*), et enfin le conditionnel (*cond*).

On observe donc des corrélations dans la variation des différents niveaux : l'ensemble des niveaux contribue à opposer le discours académique des premiers textes, le discours politisé des textes centraux, et le discours littéraire des derniers textes.

Cette analyse nécessite un corpus où sont présentes des informations issues de plusieurs « sources » : les traits morphosyntaxiques sont issus d'un instrument d'annotations, tandis que les occurrences des notes, citations ou italiques sont des propriétés de mise en forme ou de structuration. Le corpus doit donc contenir et permettre de représenter ensemble ces différents traits. Certaines étiquettes ont été décomposées en éléments plus fins : le trait indéfini, par exemple, n'était pas donné comme unité par l'instrument utilisé.

Pour une analyse de cette sorte, il est nécessaire de pouvoir désigner facilement les phénomènes à quantifier dans le corpus, de les regrouper en observation sur le palier voulu dans le corpus (ici, les textes), et d'obtenir la matrice de cooccurrences produite dans un format facilement manipulable du point de vue des outils statistiques. L'objectif de CorpusReader est de faciliter la définition de l'ensemble de ces étapes pour permettre de composer des observables à façon.

D'autres applications de CorpusReader ont permis d'utiliser conjointement deux analyseurs syntaxiques – c'est-à-dire deux outils intervenant sur le même niveau de description, et non pas seulement des outils intervenant sur des niveaux différents. Ainsi, le meilleur de chacun des deux analyseurs syntaxiques a pu être utilisé en même temps : notamment une analyse robuste en groupes syntaxiques chez l'un, et un jeu d'étiquettes morphosyntaxiques plus fin chez l'autre. CorpusReader a été éprouvé sur

des corpus volumineux, notamment Wikipédia (dans le cadre du projet ANR Autograph).

2.2. *Choix de conception*

L'objectif de CorpusReader est donc de permettre la fusion d'annotations puis la composition à façon, à partir de corpus complexes, de différentes structures de données – telles que des matrices mais également des graphes ou des structures plus classiques comme des concordanciers. Pour cela, il faut un accès fin à la structure de données représentant le corpus ainsi qu'une collection de fonctions de sélection, de quantification, ou de transformation.

Les choix qui ont déterminé CorpusReader ont d'abord été de se limiter à une seule structure de données et même un seul format de données, de façon à pouvoir offrir un contrôle fin sur les données. Le format XML a été retenu pour le compromis qu'il propose entre expressivité et facilité d'utilisation, pour le fait qu'il est standardisé et largement supporté, et enfin parce qu'une riche bibliothèque logicielle l'accompagne[5].

Cette spécialisation dans la manipulation d'un format découle de l'hypothèse qu'une plate-forme comme CorpusReader est un type d'outil distinct des instruments d'annotations. Peut-être deux types d'outils doivent-ils être distingués et n'ont-ils pas nécessairement intérêt à être fusionnés ? Il s'agit d'une part des outils spécialisés dans des tâches de modélisation (analyse linguistique), qui n'ont pas vocation à prendre en charge des formats complexes et qui fournissent nécessairement des données peu « conciliantes » dans des formats souvent idiomatiques et d'autre part des outils qui ont vocation uniquement à mettre en relation ces instruments existants. En d'autres termes CorpusReader résulte d'une approche non intégrative, plus modulaire : plutôt que de chercher à intégrer dans la plate-forme des compétences d'annotations, elle s'appuie entièrement sur des outils d'annotations existants et extérieurs qu'il faut pouvoir intégrer.

Le développement de plate-formes intégrant réellement les outils eux-mêmes est sans doute un objectif en soi, notamment pour la réalisation d'outils plus complexes et capables de bénéficier d'annotations préalables. Mais, du point de vue de la description elle-même, la disponibilité de plates-formes consacrées à la fusion *a posteriori* est un enjeu, puisque des annotations utiles à la description existeront toujours à côté de plates-formes intégrées. Le choix d'une fusion *a posteriori* permet également une réutilisabilité immédiate des instruments existants.

Dans cette perspective, le format de données n'est pas quelque chose d'entièrement interne à l'outil : l'essentiel est que les outils de type « plates-formes » puissent eux-mêmes communiquer, et qu'ils ne reproduisent pas le babélisme des formats, qui

5. Pour la même raison le langage Java a été privilégié : l'essentiel des bibliothèques associées à XML sont conçues dans ce langage.

prévaut au niveau des instruments d'annotations. CorpusReader n'impose aucun idiomatisme dans le format de données et s'appuie sur les éléments les plus standard[6].

Ces différents points distinguent CorpusReader d'une plate-forme comme Gate. Dans ce dernier dispositif, différents formats de stockage sont interchangeables, notamment les bases de données ou XML. Ceci implique une abstraction du format dans la représentation donnée du corpus à l'utilisateur ((Bontcheva *et al.*, 2004, p. 353 et 357) : « *GATE users are isolated from the ways in which LRs [language resources] are stored* »). Le choix de CorpusReader est au contraire de considérer que l'ensemble des propriétés du format intéresse le corpus comme artefact linguistique et qu'il ne peut pas être abstrait sans perte. De même, Gate propose une intégration des outils d'annotations dans la plate-forme et définit des interfaces pour la communication entre les composants logiciels. Plutôt que la communication entre composants logiciels, CorpusReader fournit des moyens pour insérer dans un corpus existant les annotations produites par les instruments tiers : il s'agit d'une fusion « par les formats » et non « par les instruments » (voir section 4.2).

3. Présentation de CorpusReader

CorpusReader est présenté ici à travers son utilisation concrète. Du fait de son objectif de donner un accès fin à la manipulation d'un format déterminé, il s'appuie sur un composant logiciel pour le traitement de ce format. Il est nécessaire de présenter rapidement ce composant puisqu'il détermine la représentation des données proposée par l'outil lui-même. La construction d'une requête est ensuite présentée en détail, puis différents mécanismes essentiels à la mise en œuvre de CorpusReader sont abordés. Enfin, un inventaire rapide des fonctions disponibles est proposé.

3.1. *Une architecture en pipeline de filtres*

CorpusReader se fixe donc l'objectif de rendre possible le modelage/l'extraction d'informations, notamment quantitatives. Cela implique en premier lieu de donner accès à une représentation du format XML. CorpusReader est très fortement associé à l'API[7] SAX (Simple API pour XML) : non seulement parce qu'il en dépend techniquement, mais aussi parce qu'il choisit de rendre visible à l'utilisateur le fonctionnement de l'API, plutôt que de l'abstraire dans des objets de plus haut niveau. Un détour par les propriétés de cette API est donc nécessaire.

6. La TEI (*Text Encoding Initiative*) est particulièrement supportée, mais CorpusReader peut être utilisé avec n'importe quel vocabulaire.

7. Une API (*Application programming interface*) est une convention pour faciliter la communication entre deux briques logicielles. Ici, il s'agit de définir une communication entre un parseur de documents XML, qui assure la conversion du document en une structure de données, et une application qui utilise un parseur pour accéder à cette structure de données. Le respect d'une API permet à l'application d'être indépendante d'un parseur particulier.

L'API SAX se caractérise par le fait qu'elle transmet le contenu du document XML à l'application comme une succession de « briques ». C'est donc une API séquentielle : l'ordre dans lequel les briques sont transmises compte. Elle s'oppose aux méthodologies « arborescentes » qui privilégient la logique inverse : construire en mémoire une structure de données représentant l'arbre XML, et la transmettre d'un seul coup à l'application. Dans une représentation arborescente, l'application peut accéder à tout moment à n'importe quelle partie de l'arbre XML et utiliser des langages de haut niveau, comme XQuery ou XSLT. L'API SAX est de plus bas niveau puisqu'on ne représente pas la nature hiérarchique des données du document à travers une structure de données elle-même arborescente, mais à travers un flux d'objets représentant les nœuds de l'arbre, transmis dans un ordre reproduisant la traversée profondeur droite de l'arbre[8].

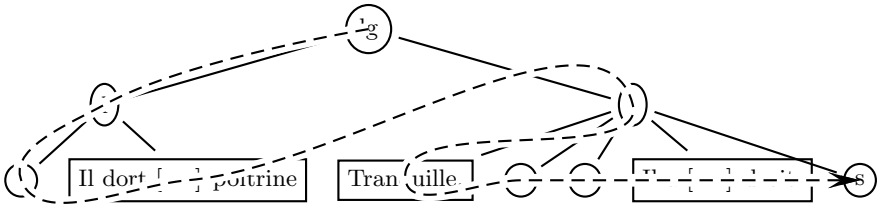

Figure 2. *Parcours d'un document XML avec l'API SAX.*

L'API SAX offrait plusieurs avantages déterminants :

– en premier lieu, elle était nécessaire pour permettre l'analyse de corpus sans limitation de taille. En effet, les méthodologies arborescentes nécessitent de construire l'arbre XML en mémoire, ce qui fait peser une contrainte forte sur la taille des données manipulables ;

– l'accès séquentiel permet de définir des portions du document XML en privilégiant l'axe de la précédence indépendamment des relations de dominance de l'arbre, à la différence des méthodologies arborescentes où la dominance est première. Cela facilite le traitement de corpus contenant des annotations enchevêtrées (voir *infra*) ;

– enfin, l'API SAX se prête à une décomposition des tâches complexes en une succession de tâches simples. En effet, dans un cadre séquentiel on peut décomposer un traitement en filtres. Chaque filtre reçoit le contenu, le transmet au suivant en réalisant une modification précisément définie du flux reçu[9]. Les filtres peuvent être

8. Dans ce flux, un nœud non terminal comme un élément doit donc être représenté par deux briques : l'une représente l'élément ouvrant et l'autre l'élément fermant, la descendance de l'élément étant ce qui est inclus entre ces deux briques.

9. On peut se représenter ces filtres comme des filtres UNIX : « *[. . .] programs that read some input, perform a simple transformation on it, and write some output* » (Kernighan et Pike, 1984, p. 101). Le document n'a pas besoin d'être entièrement traité par un filtre avant d'être transmis au suivant (certains filtres peuvent néanmoins, au besoin, mémoriser une partie du contenu avant de la transmettre au filtre suivant).

combinés en « pipeline ». La décomposition de l'application en modules réutilisables peut donc être faite au niveau de l'API elle-même, et non dans une logique propre à la plate-forme, ce qui assure une réutilisabilité maximale de ces composants, y compris en dehors de la plate-forme proposée.

Toutes les fonctions du programme sont implémentées comme des filtres qui peuvent se combiner dans un flux de filtres, ou « pipeline », définissant un traitement. Ainsi, chaque filtre peut se concentrer sur une tâche précise et prendre un nombre limité d'arguments. Les filtres sont fortement réutilisables et combinables ; ils peuvent être enchaînés de façon modulaire et cumulative. Tandis que chaque filtre reste simple, le pipeline de filtres peut exprimer des tâches complexes.

3.2. *Exemple de requête*

Un exemple simple permettra de présenter la mise en œuvre de ces principes. Supposons une tâche consistant à extraire d'un corpus une liste de fréquences des lemmes (notés par hypothèse dans les attributs `@lemma`) des éléments (`w`) dans les paragraphes contenant le lemme *x*. Cette requête pourra être exprimée dans CorpusReader au moyen d'un pipeline de trois filtres :

– un premier filtre réalise la sélection du sous-corpus. Les seuls arguments nécessaires à cette sélection sont les racines des sous-arbres à considérer (ici les sous-arbres dont les éléments racines sont `p`), et une condition sur leur contenu. Ce filtre devra donc « stocker » le flux entre une brique `p` ouvrante et une brique `p` fermante avant d'y rechercher le lemme *x* et donc, de décider si l'ensemble du sous-arbre doit être transmis ou supprimé. Il transmet au filtre suivant le document inchangé, moins les paragraphes ne satisfaisant pas la condition fixée ;

– un filtre chargé de la construction de la liste de fréquences à mesure qu'il reçoit le flux. Ce filtre ignorant tout de la tâche de définition de sous-corpus, il ne prend comme argument qu'une définition des endroits du corpus correspondant au phénomène à quantifier. Ce filtre ne modifie pas le document reçu : il le transmet inchangé et se contente de mémoriser la liste de fréquences ;

– un troisième filtre est spécialisé dans l'écriture de listes de fréquences. Il prend comme argument le nom du filtre auquel demander cette liste. Lorsqu'il reçoit l'événement signalant la fin du document, il demande donc à ce filtre la liste de fréquences, collectée jusqu'à présent, et il la représente dans le vocabulaire de la TEI (Burnard et Bauman, 2007). Dans ce dernier cas de figure, le document est donc entièrement changé. Ce filtre peut optionnellement écrire dans un format « tabulé », plus simple, la liste de fréquences produite ;

La répartition sur deux filtres des fonctions de lecture et d'écriture des listes de fréquences (comme de toutes les autres structures de données) permet une plus grande réutilisabilité des composants (par exemple, le filtre réalisant les quantifications est également utilisé pour réaliser des matrices de cooccurrences).

```
<query>
  <header>
    <name />
    <date />
    <desc />
  </header>
  <corpus inURI"corpus.xml" outURI="output.xml" />
  <filterList>
    <filter name="sous-corpus" javaClass="tei.cr.filters.SelectSubTrees">
      <args>
        <treeRoot elxpath="p" />
        <query elxpath="//w[@lemma='x']" />
      </args>
    </filter>
    <filter name="fl" javaClass="tei.cr.filters.ExtractFrequencyList">
      <args>
        <lexiconEntry>
          <use elxpath="w/@lemma" />
        </lexiconEntry>
      </args>
    </filter>
    <filter name="Lexicon" javaClass="tei.cr.filters.WriteFrequencyList">
      <args>
        <lexicon filterName="fl" />
      </args>
    </filter>
  </filterList>
</query>
```

Figure 3. *Un document requête pour construire une liste de fréquences*

Un tel pipeline de filtres est spécifié dans un document XML dit « document requête » qui est fourni en argument à CorpusReader (voir figure 3). Ce document possède trois éléments, descendant d'une racine `query` :

– `header` : un en-tête, permettant de documenter le traitement ;

– `corpus` : les URLs de l'entrée du corpus (un document XML valide) et de la sortie du pipeline ;

– `filterList` : la spécification du pipeline.

Dans l'élément `filterList` on retrouve les trois filtres, définis dans trois éléments `filter` successifs. Le corpus traverse les filtres dans l'ordre de leur déclaration. Chaque élément `filter` contient le nom de la classe implémentant le filtre dans un attribut `@javaClass`, un nom l'identifiant dans le document requête dans un attribut `@name`, et des arguments dans un nœud `args`. On retrouve dans ces nœuds `args` les arguments que nous avons mentionnés ci-dessus pour chacun des filtres : définition

d'une racine des sous-arbres à tester et d'une expression testée dans le premier, définition des phénomènes à quantifier dans le deuxième, et renvoi au filtre fournissant la liste de fréquences dans le troisième, au moyen d'une référence au nom du filtre précédent. Le dernier filtre du pipeline, non représenté dans un document requête, est toujours un filtre qui réécrit sous forme de document XML, dans un fichier[10], le résultat reçu du dernier filtre spécifié dans le document requête.

Ce document requête est défini par un schéma validé par le programme avant exécution, sauf pour les descendances des nœuds `args` qui sont validées séparément par chaque filtre concerné.

Cet exemple permet d'attirer l'attention sur des caractéristiques de CorpusReader et de souligner les conséquences des principaux choix d'architecture.

Tout d'abord, chaque filtre implémente donc une fonctionnalité aussi précise et simple que possible. Les filtres prennent ainsi peu d'arguments, comme l'illustre l'exemple ci-dessus. C'est l'accumulation des filtres qui permet de réaliser un traitement complexe. La tâche de sélection de sous-corpus est distinguée de toute tâche de quantification ou de transformation. L'ensemble des tâches de sélection de sous-corpus pour des tâches ultérieures peut être réalisé à partir d'un petit nombre de filtres. De même, le filtre `ExtractFrequencyList` assure la fonction de quantification pour l'ensemble du programme.

Dans ces filtres, tous les phénomènes exprimés dans l'infoset XML[11] peuvent être désignés et quantifiés, notamment grâce à l'utilisation d'expressions XPath dans les arguments. Le programme ne propose pas de manipuler d'autres objets que les phénomènes de l'infoset XML.

Le rôle particulier du corpus dans ce dispositif doit être souligné. C'est sur lui que repose la communication entre les éléments du programme eux-mêmes (les filtres). Ainsi la combinaison d'une tâche de sélection de sous-corpus et de quantification est réalisée grâce à la transmission du corpus entre les deux filtres. Le corpus est le lien entre les différentes parties du programme. Le principe peut être poussé plus loin. Imaginons, par exemple, que l'on souhaite quantifier un phénomène plus difficile à désigner dans un flux que les occurrences des valeurs d'un attribut XML. Par exemple, la cooccurrence entre deux éléments donnés au sein d'un troisième. Dans ce cas, plutôt que de complexifier les arguments du filtre `ExtractFrequencyList`, une solution est de faire précéder ce filtre d'une étape de transformation avec XSLT, qui ajoute un élément reconnaissable quelconque à chaque occurrence de ce phénomène, puis de faire compter les occurrences de cet élément par `ExtractFrequencyList`. Le corpus transmis de filtre à filtre est à la fois la donnée du programme et un format de sérialisation de filtre à filtre. En somme, on tire bénéfice de la capacité d'XML à exprimer des données de différents degrés de formalisation, entre les deux pôles du

10. Ou sur la sortie standard, ce qui permet d'utiliser CorpusReader comme un filtre UNIX.
11. L'infoset XML définit XML en tant que structure de données.

```
<filterList>
  <!-- [...] -->
  <split elxpath="TEI">
    <filterList>
      <filter name="transformation" javaClass="tei.cr.filters.XSLT">
        <args>
          <stylesheet URI="stylesheet.xsl" />
        </args>
      </filter>
    </filterList>
  </split>
  <!-- [...] -->
</filterList>
```

Figure 4. *Utilisation de l'élément* `split`

document « orienté données » (*data-oriented*) et du document « narratif » (*narrative-oriented*).

3.3. *Diviser un document en sous-documents*

Certains types de filtres (par exemple, ceux qui réalisent une transformation XSLT ou une requête XQuery) nécessitent de charger le document entier en mémoire, ce qui est impossible sur les corpus volumineux qui dépassent très rapidement les capacités en mémoire vive.

Une solution est de « découper » le corpus : la transformation est appliquée à des sous-ensembles successifs du corpus entier. Les sous-ensembles sont chargés en mémoire l'un après l'autre. Une fois transformés, ces sous-ensembles sont envoyés au filtre suivant et sont recomposés en un seul document.

Dans une chaîne de filtres SAX ce « découpage » consiste à ajouter, autour des éléments choisis comme élément racine des sous-documents, les briques qui indiquent le début et la fin des documents. Ainsi, les filtres suivants du pipeline reçoivent plusieurs documents successifs. Ces événements ajoutés peuvent être retirés, en sortie du ou des filtres nécessitant un découpage du corpus, pour les faire recomposer à nouveau un seul document.

Cette solution est intéressante comme solution générale pour rendre disponible l'ensemble des langages de haut niveau (XQuery, XSLT), ainsi que la validation avec des schémas, sur des corpus très volumineux.

Dans le document requête, l'élément `split`, qui peut apparaître aux mêmes endroits qu'un élément `filter` permet d'insérer les filtres réalisant cet ajout/suppression d'événements autour d'une section du pipeline (figure 4).

L'élément `split` insère un filtre chargé de « découper » le document en sous-ensembles de façon à ce que les filtres suivants le traitent comme une succession de documents plus petits. L'élément racine des documents à constituer est donné par l'attribut `@elxpath`. Si des éléments satisfaisant cette condition sont enchâssés, seul le premier rencontré (l'ancêtre des autres) est utilisé comme racine d'un sous-document. Les portions du corpus qui sont extérieures à ces sous-documents passent « par-dessus » cette section du pipeline et atteignent directement les filtres suivant l'élément `split`.

Ce dispositif joue un rôle important dans la plate-forme en permettant au pipeline de modéliser des traitements plus complexes que la simple séquentialité. Ce rôle dépasse le simple découpage d'un corpus volumineux en éléments plus petits. Il permet, par exemple, d'exprimer une itérativité dans le traitement du corpus.

Ainsi, plusieurs filtres qui reposent sur une itérativité peuvent être utilisés avec l'élément `split`. Par exemple, la construction d'une matrice de cooccurrences repose sur le filtre `ExtractFrequencyList` pour quantifier les phénomènes voulus à l'intérieur d'une unité définie par `split/@elxpath` (figure 5). À chaque changement de document, cette liste de fréquences est ajoutée par `ExtractCooccurrencyMatrix` à une matrice, puis est remise à zéro. À la véritable fin du document, la matrice formée peut être écrite, par exemple avec `WriteCooccurrencyMatrix`.

L'élément `split` peut être utilisé plus généralement pour affiner l'application d'un filtre. Par exemple, un filtre comme `SelectSubTrees` (*cf. supra*) peut être utilisé à l'intérieur d'un élément `split` pour n'être appliqué qu'à certains sous-ensembles du document.

Plus généralement encore, `split` peut être utilisé pour désigner un chemin à l'intérieur d'un document par un simple jeu d'aiguillage. Dans l'exemple de la figure 6, où la logique est poussée à l'extrême, un enchâssement d'éléments `split` permet de quantifier un phénomène dans les seuls premiers vers des seuls tercets des seuls sonnets d'un recueil de poèmes.

3.4. *Désigner des éléments du flux avec XPath*

Certains noms d'arguments sont relativement fréquents dans les arguments des filtres. C'est notamment le cas de l'attibut `@elxpath`, rencontré, par exemple, sur l'élément `split`, le filtre `ExtractFrequencyList` ou le filtre `SelectSubTrees`. Cet attribut est systématiquement utilisé pour désigner des éléments dans le flux reçu.

L'attribut `@elxpath` attend comme argument une expression XPath. Cette expression est évaluée[12] sur chaque élément du document isolément, comme s'il formait un document autonome avec ses seuls attributs. Par exemple, dans le fragment suivant :

12. Au moyen du moteur XPath Jaxen, `http://jaxen.codehaus.org`.

```
<filterList>
  <split elxpath="p">
    <filterList>
      <filter name="quantifier"
              javaClass="tei.cr.filters.ExtractFrequencyList">
        <args>
          <lexiconEntry>
            <use elxpath="w/@lemma" />
          </lexiconEntry>
        </args>
      </filter>
      <filter name="accumuler"
              javaClass="tei.cr.filters.ExtractCooccurrencyMatrix">
        <args>
          <lexicon filterName="quantification" />
        </args>
      </filter>
    </filterList>
  </split>
  <filter name="ecrire"
          javaClass="tei.cr.filters.WriteCooccurrencyMatrix">
    <args>
      <matrix filterName="cumule" />
    </args>
  </filter>
</filterList>
```

Figure 5. *Utilisation de l'élément* `split` *pour réaliser une itération*

```
<p id="p4"><w lemma="un" n="1">Un</w><p>
```

le flux est considéré comme une succession de minidocuments autonomes comme

```
<?xml version="1.0"?> <p id="p4" />
```

```
<?xml version="1.0"?> <w lemma="un" n="1" />
```

sur lesquels l'expression XPath est successivement appliquée. Ce n'est pas la syntaxe XPath qui est limitée, mais les données sur lesquelles l'expression est appliquée. Ceci permet d'utiliser toute l'expressivité de la syntaxe ; par exemple de désigner un élément ouvrant du flux SAX avec l'expression `//*[contains(@ana,'pos.verb')]`.

```
<split elxpath="TEI[@type='sonnet']">
  <filterList>
    <split elxpath="lg[@type='tercet']">
      <filterList>
        <split elxpath="l[@n='1']">
          <filterList>
            <filter name="fList"
                    javaClass="tei.cr.filters.ExtractFrequencyList">
              <args>
                <lexiconEntry>
                  <use elxpath="w/@lemma" />
                </lexiconEntry>
              </args>
            </filter>
          </filterList>
        </split>
      </filterList>
    </split>
  </filterList>
</split>
```

Figure 6. *Utilisation de l'élément* `split` *pour désigner un chemin*

3.5. *Communications entre filtres*

Les premier et deuxième filtres de l'exemple de la figure 3 ne « communiquent » pas directement, sinon par l'intermédiaire du flux d'événements transmis. Au contraire, les deuxième et troisième filtres communiquent directement, par un mécanisme spécifique, de façon à se transmettre la liste de fréquences indépendamment du flux XML. Ce mécanisme est utilisé entre les filtres qui échangent un objet connu d'avance : une liste de fréquences, une matrice de cooccurrences, etc. Il permet de renforcer la modularité et la réutilisabilité des filtres dans le programme. Ainsi, par exemple, un seul filtre, `ExtractFrequencyList`, est chargé d'opérer des quantifications. Il peut-être utilisé dans un pipeline pour produire une liste de fréquences (figure 3) comme pour produire chaque observation d'une matrice de cooccurrences (figure 5).

On peut illustrer cette réutilisabilité des composants avec le filtre `ExtractLocationImpl` (figure 7). Ce filtre est spécialisé dans la construction de « références » dans le flux : en fonction de ses arguments, qui indiquent des nœuds de l'arbre à observer, il peut mémoriser par exemple, au fur et à mesure de la réception du flux, le numéro de la page actuellement traitée, le titre de la section en cours, le nombre d'occurrences écoulées d'un phénomène (par exemple le numéro du paragraphe en cours, depuis le début du document ou depuis la dernière occurrence de tel autre phénomène), ou l'identifiant du paragraphe en cours. Ce filtre ne change en rien le document reçu. Il peut en revanche être utilisé par tout autre filtre qui

```
<filterList>
  <filter name="f1" javaClass="tei.cr.filters.ExtractLocationImpl">
    <args>
      <text value=" (p. "></text>
      <use elxpath="tei:pb/@n"></use>
      <text value=")"></text>
    </args>
  </filter>
  <filter name="f2" javaClass="tei.cr.filters.Concordance">
    <args>
      <node elxpath="tei:w[@lemma='y']" />
      <locator filterName="f1" />
    </args>
  </filter>
<filterList>
```

Figure 7. *Utilisation de l'élément `ExtactLocationImpl`*

peut nécessiter de donner des « références » à des objets produits en termes de positions dans le document original. C'est par exemple le cas d'un filtre produisant des concordances : on peut souhaiter ajouter, à chaque ligne de la concordance, un numéro de la page dont est issue l'attestation. Le document requête de la figure 7 utilise `ExtractLocationImpl` pour ajouter des numéros de pages aux lignes d'une concordance des occurrences du lemme *y*. Le même filtre permet de nommer les observations d'une matrice de cooccurrences.

3.6. *Formats de données de sorties*

Une attention particulière a été apportée à la production de formats de sorties. Les différents filtres produisant des structures de données (matrice de cooccurrences ou graphe notamment) sont associés à des filtres capables de sérialiser ces structures dans les formats les plus courants des logiciels d'exploration statistique. Ainsi, les matrices de cooccurrences peuvent être sauvées directement dans les format de R[13], DTM[14], ou Matlab (dans un format de matrices creuses). Les graphes peuvent être sauvés dans des formats utilisés par R ou Pajek[15].

L'interaction avec le « monde extérieur » passe également par la prise en charge d'autres sources de données. Des filtres permettent ainsi de se connecter à une base de données relationnelles (et de transmettre la table retournée par une requête SQL à la suite du pipeline, exprimée en XML dans le vocabulaire de la TEI), ou XML

13. `http://www.r-project.org`
14. `http://www.lebart.org`
15. `http://vlado.fmf.uni-lj.si/pub/networks/pajek/`

natives. Un autre filtre permet à rebours d'insérer dans une base de données des enregistrements à partir de champs désignés dans le document et d'itérations sur des enregistrements.

3.7. *Filtres disponibles*

Les filtres disponibles dans CorpusReader respectent certaines conventions. En particulier, les filtres ne peuvent pas retourner un document mal formé. Cela signifie, par exemple, qu'un filtre `SelectSubTrees` ne pourra pas supprimer l'élément racine d'un document.

En conséquence, les filtres peuvent être chaînés dans n'importe quel ordre : il n'y a pas de filtre obligatoirement terminal. Rien n'interdit *a priori* de faire suivre un filtre retournant une matrice de cooccurrences exprimée en XML, par d'autres filtres travaillant sur la représentation en XML de cette matrice.

Plusieurs filtres permettent d'utiliser des syntaxes de haut niveau : c'est le cas des filtres `XSLT` et `XQuery`[16], ou encore de filtres permettant la validation de schémas RelaxNG ou W3C[17].

De même, plusieurs filtres peuvent aider à la confection de pipelines. Ainsi, `DefaultWriter` sérialise, dans un fichier donné en argument, le flux reçu, qu'il transmet inchangé. On peut ainsi mettre plusieurs de ces filtres à différents points d'un pipeline où l'on veut observer ce qu'est devenu le corpus à cette étape intermédiaire du traitement. `Silent` permet de réduire tout le flux reçu au seul élément racine, ce qui est utile, par exemple, pour se débarrasser d'une sortie volumineuse du pipeline quand le résultat voulu est déjà fourni par un filtre[18].

Parmi les filtres utiles à la sélection de sous-corpus, comme `SelectSubTrees`, on peut mentionner `Sample` qui permet de faire des tirages aléatoires dans un sous-ensemble défini en termes de phénomènes de l'infoset XML.

Des fonctions documentaires, comme la production de concordances, ont été réimplémentées. Là encore il s'agit d'adapter les outils à XML : les nœuds des concordances peuvent être n'importe quel phénomène de l'infoset XML, désigné par une expression XPath.

Des filtres permettent de faire des éditions localisées, comme appliquer une expression régulière, ajouter des éléments en fonction des occurrences d'une expression régulière, etc.

---

16. Ces deux filtres utilisent les logiciels Xerces (`http://wwww.apache.org`) et Saxon (`http://www.saxonica.org`).
17. Ces filtres utilisent le validateur Jing.
18. `Silent` est un peu analogue au fait de diriger un flux vers `/dev/null` sur un système UNIX.

4. Corpus complexes et nœuds-bornes

Deux filtres essentiels au programme sont présentés plus en détail : le premier permet de manipuler des enchevêtrements de hiérarchies, et le second de fusionner une annotation produite par un instrument tiers au corpus.

4.1. *Problèmes de représentations*

Les contraintes sur les formalismes de représentations imposées par les corpus multiannotés sont fortes. Or, les modèles de données arborescents comme XML sont trop faiblement expressifs pour représenter l'imbrication des annotations. La simple représentation conjointe des segmentations en phrases et en vers est impossible avec XML : les deux unités créent des enchevêtrements de hiérarchies (*intersecting hierarchies*). Dans les cas de corpus multiannotés, ce type de chevauchements est omniprésent et une solution particulièrement robuste est nécessaire.

L'alternative devant cette difficulté est soit de recourir à l'annotation « débarquée » (*stand-off*), soit de recourir à des « nœuds-bornes » (ou « balises autofermantes », « jalons » – *milestone*). Dans le premier cas, différents documents, reliés par un système de pointeurs, contiennent chacun un ensemble « compatible » d'annotations tel que, dans chaque document, il n'y ait pas d'enchevêtrements. Dans le second cas, on remplace certains éléments dont les frontières entrent en conflit avec d'autres éléments par des nœuds-bornes, c'est-à-dire des nœuds sans contenu qui notent le début et la fin de l'unité (DeRose, 2004). Cette fois, c'est l'association entre le nœud-borne de début et le nœud-borne de fin qui doit être exprimée au moyen d'un mécanisme ou d'une convention supplémentaire.

On peut interpréter cette alternative comme une opposition entre le fait de privilégier la dominance ou la précédence (Loiseau, 2007). Dans le cas de l'annotation débarquée, en effet, la dominance est conservée au prix d'une perte d'intégration des données, tandis que l'axe de la précédence doit être reconstitué. Cette solution suppose que l'annotation puisse être distribuée en différents sous-ensembles non contradictoires ou, en tout cas, peu nombreux. Dans le second cas, on perd, avec les nœuds-bornes, l'expression directe de la dominance, mais toutes les informations sont intégrées sur un axe linéaire commun – celui de la précédence. Les conséquences de ce choix sur les propriétés empiriques de l'artefact produit sont donc importantes : selon l'une ou l'autre solution, on privilégie différents types de contextualité.

Trois arguments peuvent être retenus en faveur de l'annotation en nœuds-bornes. Le premier est que l'intégration entre les données (et donc la contextualité) est plus forte : la contextualisation des informations est préservée dans la structure de données, sans nécessité de reconstruction. Le deuxième est que cette solution est plus robuste pour un grand nombre d'intersections possibles : l'ajout d'une nouvelle annotation ne change rien à la manipulation du corpus, tandis qu'elle nécessite l'intégration et l'alignement d'un nouveau document, même pour une annotation très partielle du

document, dans le cas de l'annotation débarquée. Elle permet d'éviter la gestion de pointeurs qui deviennent d'autant plus complexes à maintenir à jour que le nombre de documents augmente.

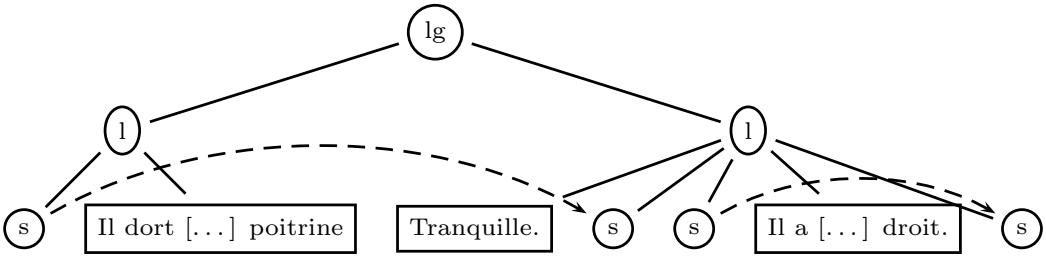

Figure 8. *Représentation d'une segmentation en phrases au moyen de nœuds-bornes*

Enfin, un troisième argument, propre aux choix d'architecture de CorpusReader, doit être souligné : tandis que la gestion des nœuds-bornes est délicate avec les langages de haut niveau qui privilégient la dominance, elle est extrêmement simple avec un accès au document par l'API SAX, puisque cette API donne précisément accès à la précédence. Depuis un flux SAX, il est aisé d'identifier un ensemble de nœuds-bornes ouvrants et fermants et de les convertir, en fonction du besoin, en éléments « normaux » pour faciliter les traitements futurs.

On peut rapprocher la structure de données produite par l'utilisation de nœuds-bornes des graphes d'annotations proposés par (Bird et Liberman, 2001) pour la représentation de corpus oraux. En effet, l'ajout de nœuds-bornes équivaut à introduire des arêtes dans le document. Dès lors, les deux types d'annotations (nœuds-bornes et éléments hiérarchiques) peuvent être représentés par un formalisme commun dans le cadre des graphes d'annotations (figure 9) : les deux types d'annotations représentent des arêtes orientées et indexées sur un axe commun. À partir de ce graphe, on peut définir autant de documents XML qu'il y a d'ensembles d'arêtes sans intersection entre elles (incluant dans tous les cas l'élément racine, c'est-à-dire l'arête la plus extérieure, nécessaire à la bonne formation du document). Les documents de cet ensemble peuvent être dits *homologues* en ce sens : ils ne diffèrent que par le choix d'utiliser l'un ou l'autre mécanisme de notations pour chaque arête.

L'hypothèse qui motive le choix des nœuds-bornes est que seul un sous-ensemble est effectivement nécessaire, à chaque étape d'un traitement, pour utiliser des syntaxes reposant sur l'expression de la dominance. Par exemple, si une itération est exprimée en XSLT, l'élément sur lequel opère cette itération doit être représenté au moyen d'un élément « normal », mais si on opère une quantification de phénomènes dans la descendance de ces éléments, alors ces phénomènes peuvent être indifféremment exprimés en éléments normaux ou en nœuds-bornes. L'annotation « normale », c'est-à-dire l'expression de l'axe de la dominance, n'est pas considérée comme une fin en soi ou un modèle (capturant une propriété intrinsèque des données linguistiques),

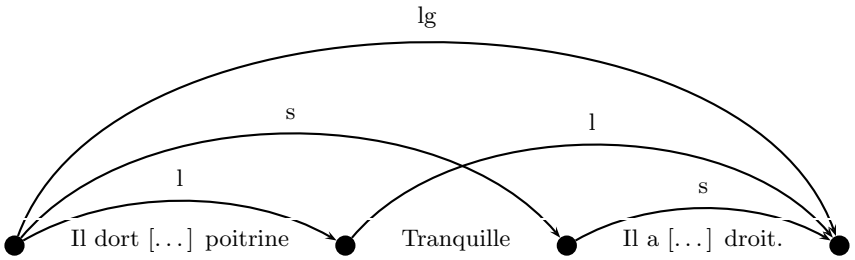

Figure 9. *Représentation unifiée d'éléments normaux et de nœuds-bornes dans un graphe d'annotations.*

mais seulement comme une représentation pratique pour exprimer des itérations ou des sélections dans certains langages.

Pour justifier les graphes d'annotations, (Bird et Liberman, 2001) mentionnent des besoins extrêmement proches de ceux qui ont conduit à l'utilisation des nœuds-bornes dans CorpusReader :

> « Les modèles arborescents traitent tous la relation de dominance comme fondamentale. Nous pensons que cela conduit à des difficultés non triviales. »

En conséquence, une fonction essentielle de CorpusReader est de permettre le « passage » entre plusieurs documents « homologues » en fonction des besoins de traitements. Ces transformations peuvent même être réalisées plusieurs fois dans un même pipeline.

Dans CorpusReader, un filtre (`ExtractMilestone`) est consacré à cette conversion d'un sous-ensemble des nœuds-bornes en éléments « normaux ». Ce filtre accepte comme argument une expression XPath désignant n'importe quel sous-ensemble de nœuds du document. Cette expression peut désigner un nœud-borne – mais également au besoin n'importe quel autre élément – pour segmenter le document. Un deuxième argument détermine le nom d'élément utilisé pour les éléments créés. Sur cette base, le flux est « découpé » en fonction des occurrences de l'élément désigné, et les éléments incompatibles avec cette nouvelle segmentation sont convertis eux-mêmes en nœuds-bornes. Là encore, l'élément `split` permet de contrôler, de l'extérieur, le comportement de ce filtre. En effet, si le filtre est placé à l'intérieur d'un élément `split` qui ne lui transmet, par exemple, que les paragraphes, cette segmentation se fera à l'intérieur des paragraphes : sans rompre les éléments paragraphes d'une part, et sans s'étendre à l'ensemble du document d'autre part. Enfin, `ExtractMilestone` peut prendre un troisième argument qui désigne les nœuds-bornes fermants. Dès lors, le filtre ne découpe plus le flux reçu en éléments consécutifs adjacents mais uniquement en fonction des paires de nœuds-bornes ouvrants et fermants trouvées.

L'originalité de ce choix est de représenter un graphe d'annotations dans un document XML « narratif ». Il implique que l'annotation est, pour une part, transformable et ajustable en fonction des besoins de traitement, et non pas immuable dans le cadre de l'outil qui l'utilise. Enfin, il permet de caractériser CorpusReader comme un dispositif définissant des transitions entre un *corpus cumulatif*, qui contient l'ensemble des informations disponibles dans un format complexe, et des *corpus de travail*, où un sous-ensemble de cette annotation est sélectionné, en fonction des besoins, pour être exprimé dans un formalisme « traitable » par des outils plus généralistes. Pour des corpus multiannotés, cette distinction de deux états du corpus est cruciale : elle offre la possibilité de dissocier les exigences d'expressivité et les exigences de maniabilité.

4.2. *Fusion d'annotations : une stratégie*

Une seconde opération cruciale est le passage de multiples annotations, issues de plusieurs instruments et chacune dans un format différent, en un corpus « cumulatif » articulant toutes ces annotations. Cette opération est essentielle puisque le dispositif propose de privilégier l'alignement des corpus à l'intégration des logiciels (voir ci-dessus, section 2.2).

L'intégration d'une annotation produite par un outil tiers dans un corpus déjà annoté peut être résumée par une séquence typique de trois étapes.

Dans un premier temps, le texte à annoter est extrait du corpus. Cette extraction peut être réalisée à faible coût (par exemple, avec XSLT, ou avec CorpusReader lui-même). Elle permet de prendre en compte les exigences de chaque analyseur (par exemple, la suppression des retours à la ligne en début de vers, pour l'analyse de sonnets, que la plupart des analyseurs interprètent comme un changement de phrase). Elle permet de soustraire à l'analyse certaines portions du document, comme l'entête. La seule limite à cette composition à façon du texte soumis à l'analyseur est que les critères de sélection du texte soient exprimés en termes d'éléments XML, de façon à permettre, aux étapes ultérieures, l'intégration des annotations produites dans le corpus.

Dans un second temps, l'annotation produite par l'instrument est convertie en XML, en conservant le texte d'origine restitué par l'analyseur (les formes fléchies). Cette conversion est là encore réalisable à faible coût, parfois au moyen de simples expressions régulières. Pour certains outils (Cordial, Syntex par exemple) CorpusReader propose d'intégrer cette étape.

À l'issue de ces deux étapes, on dispose de deux documents XML, contenant des annotations différentes, et certains fragments de texte communs. La fusion entre les deux annotations implique deux étapes : d'abord un alignement du texte commun, puis une insertion des annotations nouvelles dans le corpus existant. Pour réaliser l'alignement, les lieux, dans les deux documents, du texte commun (les formes fléchies originales) sont spécifiés au moyen d'expressions XPath. Le texte commun n'est pas nécessairement les nœuds textes des deux documents. Un algorithme de programma-

tion dynamique est utilisé pour être robuste aux éventuelles différences entre les deux chaînes de caractères : aucun outil ne restitue tout à fait fidèlement en sortie les formes fléchies qu'il a réellement reçues en entrée. Une fois les deux flux alignés, les annotations fournies par l'instrument sont importées dans le corpus de destination. Un jeu de règles peut être spécifié pour contrôler cette importation : par exemple, pour forcer certains éléments importés à être parents ou descendants d'éléments existants dans le corpus de destination, ou pour résoudre au moyen de nœuds-bornes les enchevêtrements de hiérarchies produits.

L'intégration réalisée ici entre annotations est une intégration « faible » au sens où les divergences de segmentation en paliers et les différences entre jeux d'étiquettes sur un même niveau, ont été acceptées, et que l'on n'a pas cherché à unifier les jeux d'étiquettes d'instruments concurrents sur un même niveau. C'est d'une certaine façon la diversité des analyses qui est supposée intéressante, et la possibilité d'utiliser chacune pour ce en quoi elle est performante ou adaptée à une hypothèse. Le dispositif n'a aucune connaissance des jeux d'étiquettes ou des contenus des annotations : il n'intervient qu'au niveau de la manipulation du format.

L'un des avantages de cette méthode est que seule la seconde étape est spécifique à un format idiomatique ; de nouveaux instruments peuvent être associés à très faible coût, pour peu qu'on puisse convertir leur format de sortie en XML. Cette stratégie d'intégration d'annotations repose sur l'utilisation d'XML comme format commun. On peut la distinguer d'une autre méthode consistant à redéfinir les instruments d'annotations, ou du moins à les intégrer dans un dispositif logiciel, pour permettre la multiannotation. L'*intégration par le format* adoptée ici est également proposée dans la plate-forme Atlas (Bird *et al.*, 2000), dans le cadre des graphes d'annotations. À l'inverse, l'*intégration par les instruments* est le choix que fait GATE : cette plate-forme définit des interfaces, c'est-à-dire des comportements types des logiciels, que des analyseurs doivent respecter de façon à interagir. La multiannotation ne peut être réalisée qu'avec des analyseurs partiellement (ré)écrits pour ce cadre. Le format d'annotations, quant à lui, peut alors être interne et spécifique à la plate-forme. CorpusReader fait le choix inverse : c'est le format (XML) qui est fédérateur, pas le comportement des logiciels d'annotations.

L'enjeu de cette solution est également la réutilisabilité, puisque les plates-formes aux formats d'annotations idiomatiques ne peuvent pas interopérer et reproduisent, au niveau de la multiannotation, le babélisme des formats. CorpusReader peut aussi bien intégrer une annotation produite par un instrument isolé qu'une annotation produite par une plate-forme de multiannotation. De plus, l'intégration par les formats rend plus immédiatement disponible, à faible coût, l'ensemble des instruments existants (voir section 2.2).

5. Plate-forme et concepts méthodologiques

Ce dispositif procède de plusieurs hypothèses sur l'enjeu descriptif de l'accès à la multiannotation et sur les propriétés du corpus comme artefact linguistique. En tant que dispositif technique, il porte avec lui des propositions et des implications qui vont être maintenant explicitées.

5.1. *Faire d'une API un outil*

CorpusReader peut être utilisé à différents niveaux de technicité :

– il fournit des fonctions prêtes à l'emploi pour différentes tâches documentaires, de quantification, de sélection ou d'enrichissement de corpus ;

– il peut être utilisé comme une façon de mettre en œuvre les langage XSLT ou XQuery, ou de définir une succession d'étapes de validation/transformation où la structure en pipeline est utile (notamment pour découper un document volumineux en documents de petites tailles) même si l'on n'utilise que des langages de haut niveau ;

– les fonctionnalités peuvent être étendues en important dans le pipeline des filtres extérieurs au programme. En effet, l'attribut `@javaClass` des éléments `filter` du document requête peut recevoir n'importe quel nom qualifié de classe java ; ces classes peuvent être fournies aussi bien par CorpusReader que par des bibliothèques externes[19]. À ce titre CorpusReader est un support de greffons[20]. Le choix de lier l'architecture du programme à l'API permet une interaction aussi forte que possible avec les autres outils prévus pour le même format. Par sa capacité à inclure d'autres filtres, CorpusReader peut être utilisé comme une façon de simplifier la mise en œuvre de SAX : seul le filtre proprement dit nécessite d'être défini, tandis que la gestion du parseur XML, le chaînage des filtres et la sérialisation de la sortie du pipeline sont assurés par le programme ;

– enfin, un filtre, `Script`, permet de définir entièrement un filtre dans le document requête : il prend en argument le code java définissant la gestion des briques reçues et ce code, exécuté à la volée[21], défini le comportement du filtre. La mise en œuvre du langage de programmation est ici la plus simple possible.

Cette variation dans le niveau d'abstraction de la représentation des traitements souligne l'objectif de CorpusReader d'être à mi-chemin de l'outil et de l'API. L'objectif en effet est de rendre simple la manipulation de l'API, de ne pas en dissimuler le fonctionnement derrière des abstractions propres à la plate-forme. Mon hypothèse à cet égard est que l'ensemble des propriétés du format de données peuvent intéresser

19. La localisation de ces bibliothèques peut être précisée dans un élément `libraries` du document requête.
20. À l'inverse, les filtres de CorpusReader peuvent être réutilisés par un autre programme.
21. L'interprétation du code java à la volée est permise par la bibliothèque Beanshell (`http://www.beanshell.org`)

la description de corpus. CorpusReader se présente donc, en un certain sens, comme une simple interface pour composer des pipeline de filtres SAX ; comme une interface pour faciliter l'utilisation de l'API SAX, ou pour faire de l'API un outil.

5.2. *Notions méthodologiques impliquées*

En tant que dispositif expérimental, CorpusReader propose une typologie d'objets représentant la manipulation d'un corpus. Il est intéressant de noter que les architectures permettant une multiannotation proposent toutes un nouveau type d'objets, qui ne sont ni des objets modélisant des concepts linguistiques, ni des objets modélisant des concepts informatiques, mais qui correspondent davantage à une modélisation du processus d'interprétation. Par exemple, GATE est construit autour d'une distinction entre les ressources (LR, *Language Resources*) et les composants logiciels (LE, *Language Engineering*). LinguaStream propose les concepts de vue et de modèle d'analyse. Ce type de concepts semble caractéristique de cette génération de dispositifs[22]. Ces distinctions sont souvent peu explicitées théoriquement, alors qu'il s'agit, indépendamment de leur implémentation, de réelles propositions théoriques ou méthodologiques. La distinction entre LE et LR peut sembler de bon sens mais c'est un choix théorique sur les éléments manipulés par le linguiste : elle reconduit une distinction entre données et programmes issue de l'intelligence artificielle. Des typologies de ces concepts seraient sans doute possibles ; par exemple, les concepts de modèle d'analyse ou de vue proposés par LinguaStream (Bilhaut et Widlöcher, 2006) sont des concepts qui portent sur l'activité de l'analyste : il s'agit de modéliser non pas l'objet mais l'activité de recherche ou d'exploration de cet objet. LinguaStream propose également une représentation d'un traitement par une succession de filtres, semblables à ceux de CorpusReader, cependant il s'agit d'un objet différent des filtres de l'API SAX.

De ce point de vue, une caractéristique de CorpusReader est de proposer aussi peu d'abstraction que possible. Plusieurs objets introduits par ce dispositif portent sur des états différents du corpus, comme l'opposition entre corpus d'accumulation et corpus d'exploration (voir section 4.1 ci-dessus). Les objets proposés reflètent des concepts de bas niveau issus de l'API, comme la notion de filtre SAX, définit dans l'API elle-même.

Plusieurs mécanismes relèvent du cahier d'expériences. Ainsi, le document requête propose d'enregistrer les paramètres d'une requête dans un élément `header`. L'en-tête permet de documenter la tâche et ainsi garantit la traçabilité et aide à la réexécutabilité des requêtes[23]. Pour garantir cette traçabilité, le programme génère éga-

22. On peut trouver des concepts proches dans des dispositifs plus anciens, par exemple dans Lexico3 et le journal de la session de travail qu'il propose de créer. Cependant, il s'agit là d'un équivalent d'un concept interprétatif existant en dehors d'un dispositif, celui du cahier d'expériences ; les logiciels de multiannotations proposent, quant à eux, un type d'objets nouveaux, spécifiques au dispositif expérimental.

23. La capacité à réexécuter facilement des expériences publiées, à réutiliser les outils, et à pouvoir les appliquer facilement à des données variées est certes importante, mais il est peut-être in-

lement lors de toute exécution un journal de l'exécution, qui peut être réglé à différents degrés de détail.

5.3. *Corpus et outil*

Une notion récurrente de cette présentation est l'importance accordée à l'autonomie du corpus par rapport à la plate-forme, notamment à travers le choix d'une annotation embarquée et d'une intégration par le format. Vis-à-vis du programme, le corpus cumule de nombreux rôles : le corpus est un format de communication interne aux composantes du programme (les filtres). La plate-forme n'impose aucun idiomatisme, et se rend par là interopérable avec d'autres plates-formes. CorpusReader se présente en définitive comme un simple outil pour manipuler un format de corpus.

D'un point de vue descriptif, il importe d'avoir accès à toutes les propriétés du corpus en tant que structure de données qui est elle-même une interprétation. Analysant les transcriptions de l'oral, (Mondada, 2000) montre que « La transcription n'est pas simplement une activité sélective, mais plus radicalement une entreprise interprétative [...] ». Ici, la structure de données n'est pas extérieure à l'objet observable.

D'autre part, ce qui caractérise, dans une certaine mesure, l'usage des données annotées est un étagement des interprétations : des données complexes, résultats d'une phase antérieure de description, sont utilisées comme matière première pour de nouvelles descriptions. C'est le cas des dictionnaires, par exemple, qui sont le résultat d'une pratique descriptive autonome, et qui servent comme ressources dans de nombreuses expériences. Ce besoin de réutilisation et de mise en interaction des interprétations que sont par exemple les dictionnaires ou les sorties d'analyseurs pour construire de nouveaux observables détermine un certain foisonnement que les plates-formes monolithiques ne sont pas à même d'assister au mieux.

6. Conclusion

CorpusReader a pour objectif de faciliter la création et l'interrogation de corpus empiriquement complexes. Il permet de modeler à façon des corpus multiannotés pour en extraire des données quantitatives ou documentaires.

---

exact d'employer le terme « reproductibilité » pour cela. Ce terme est propre aux sciences expérimentales, et il désigne une propriété d'une expérience qui lui permet de valider une hypothèse. Dans les sciences expérimentales, ce sont les résultats de l'expérience qui doivent être reproduits, pas seulement les conditions de l'expérience elle-même. L'exécution d'un programme est toujours déterministe et n'est pas une « expérience » dans le sens des sciences expérimentales. La capacité à le réexécuter ne prouve rien sur le plan de la théorie. Il est peut-être dangereux de sous-entendre, avec cette métaphore, que la réexécutabilité entraîne une validation théorique de quelque chose.

CorpusReader peut être considéré comme une plate-forme dédiée à la simple manipulation d'un format, de façon à rendre accessible toutes les propriétés empiriques de l'objet et à donner un contrôle très fin à l'utilisateur sur les données extraites du corpus. Sur le plan descriptif, l'enjeu de cet outil est de pouvoir décrire des phénomènes qui associent plusieurs analyseurs voire plusieurs niveaux de descriptions.

CorpusReader se distingue des plates-formes intégratives comme Gate qui repose sur l'intégration des logiciels : il essaye au contraire de rendre immédiatement disponibles l'ensemble des outils d'analyse de corpus existants, et de se rendre aussi réutilisable que possible, en privilégiant une intégration par les formats.

Ce dispositif repose sur la distinction de deux états du corpus : le corpus cumulatif et le corpus de travail. Il est dédié à des tâches de transformation entre ces deux états. Cette distinction est peut être nécessaire pour rendre accessible de nouveaux corpus à la fois volumineux et lourdement annotés.

7. Bibliographie

Baayen H., « Derivational productivity and text typology », *Journal of Quantitative Linguistics*, vol. 1, n° 1, p. 16-34, 1994.

Biber D., *Variation across speech and writing*, Cambridge University Press, Cambridge, 1988.

Bilhaut F., Widlöcher A., « LinguaStream : An Integrated Environment for Computational Linguistics Experimentation », *Proceedings of the 11th Conference of the European Chapter of the Association for Computational Linguistics, (EACL'06), Trento, Italy, April 3-7, 2006*, p. 95-98, 2006.

Bird S., Day D., Garofolo J., Henderson J., Laprun C., Liberman M., « ATLAS : A Flexible and Extensible Architecture for Linguistic Annotation », *Proceedings of the Second International Conference on Language Resources and Evaluation*, p. 1699-1706, 2000.

Bird S., Liberman M., « A formal framework for linguistic annotation », *Speech Communication*, vol. 33, n° 1/2, p. 23-60, 2001.

Bontcheva K., Tablan V., Maynard D., Cunningham H., « Evolving GATE to meet new challenges in language engineering », *Natural language Enginering*, vol. 3/4, n° 10, p. 349-373, 2004.

Burnard L., Bauman S., *TEI P5, Guidelines for Electronic Text Encoding and Interchange*, TEI Consortium, 2007.

Carletta J., Kilgour J., O'Donnel T., Evert S., Voormann H., « The NITE Object Model Library for Handling Structured Linguistic Annotation on Multimodal Data Sets », *Proceedings of the EACL Workshop on Language Technology and the Semantic Web (3rd Workshop on NLP and XML, NLPXML-2003)*, 2003.

DeRose S., « Markup Overlap : A Review and a Horse », *Proceedings of Extreme Markup Language, Montréal, Québec, August 2-6, 2004*, 2004.

Glessgen M.-D., *Domaines et méthodes en linguistique française et romane*, Armand Colin, Paris, 2007.

Habert B., « Création de dictionnaires sémantiques et typologie des textes », *L'Imparfait, Philologie électronique et assistance à l'interprétation des textes, Actes des Journées scientifiques 1999 du CIRLEP*, p. 171-188, 2000.

Habert B., *Instruments et ressources électroniques pour le français*, Ophrys, Paris, 2005.

Habert B., Zweigenbaum P., « Régler les règles », *TAL*, vol. 43, n° 3, p. 83-105, 2002.

Heiden S., « Un modèle de données pour la textométrie : contribution à une interopérabilité entre outil », *Actes des 8es Journées d'analyse des données textuelles (JADT 2006), Besançon, 19–21 avril 2006*, 2006.

Kernighan B. W., Pike R., *The UNIX Programming Environment*, Prentice-Hall, Englewood Cliffs, 1984.

Loiseau S., Sémantique du discours philosophique : du corpus aux normes, Thèse de doctorat, Université Paris X-Nanterre, 2006.

Loiseau S., « CorpusReader : un dispositif de codage pour articuler une pluralité d'interprétations », *Corpus*, vol. 1, n° 6, p. 153-186, 2007.

Loiseau S., « Corpus, observables et typologies textuelles », *Syntaxe et sémantique*, sous presse.

Mondada L., « Les effets théoriques des pratiques de transcription », *Linx*, n° 42, p. 131-150, 2000.

Yvon F., Des apprentis pour le traitement automatique des langues, HDR, Université Pierre et Marie Curie, 2006.