Modélisation des ressources linguistiques d'une application industrielle

Frédéric Meunier

Laboratoire central de Recherches
Thomson-CSF
Domaine de Corbeville
91404 Orsay cedex
frederic.meunier@lcr.thomson-csf.com
http://www.lcr.thomson-csf.com

Résumé

Cet article présente les avantages qu'apporte la modélisation des ressources linguistiques utilisées dans une application. Le lecteur trouvera également dans cet article une présentation rapide de deux méthodes répandues dans le monde de l'informatique (Merise et UML) et leur modèle associé (entité relation et objet). Enfin, nous donnerons un exemple de modélisation des ressources linguistiques d'une application en cours de développement.

1. Problématiques

1.1. Historique

Nous disposons depuis quelques années d'un formalisme de génération de textes multilingues. Ce formalisme, GTAG (Danlos 94, Danlos 98), a été validé par une première implémentation : Flaubert (Danlos & Meunier 96, Meunier 97). L'objectif de notre laboratoire est maintenant de déployer le formalisme, *i.e.* le rendre accessible à la communauté scientifique désireuse de développer des prototypes de génération de textes.

1.2. Déployer le formalisme GTAG

Notre objectif est donc de déployer un formalisme. Pour faire un parallèle, prenons l'exemple des grammaires TAG (Joshi et al. 1975). La communauté dispose d'un formalisme permettant de décrire certaines grammaires : le formalisme TAG. Afin de déployer ce formalisme, (Paroubek et al. 1992) ont développé un outil : XTAG. Cette outil est utilisé par la communauté scientifique pour développer différentes grammaires et les tester. Grâce à cet outil, de nombreux chercheurs ont pu implémenter des grammaires et montrer puis repousser les limites du formalisme. Notre objectif est similaire : diffuser un outil permettant à la communauté scientifique de développer et de tester des générateurs de texte, et attendre d'elle un retour pour améliorer le formalisme.

Pour atteindre de tels objectifs, il nous faut donc diffuser un outil de développement de générateurs. Ceci n'est possible que si le moteur de génération livré avec cet outil est entièrement générique. La figure 1 donne l'architecture général d'un tel générateur. On y voit que seules les ressources linguistiques dépendent du domaine d'application (noté D), de la langue cible (L) et éventuellement du style (S).

Dans Flaubert, cette architecture est en partie respectée. Il y subsiste quelques algorithmes bien identifiés (parallélisme, réduction des subordonnées et attributs non spécifiés, tels que définis par Danlos 97), qui dépendent de certaines ressources (en l'occurrence la topologie de certains schémas d'arbres élémentaires). Ainsi, Flaubert ne peut pas déployer le formalisme

GTAG car il y est fait des hypothèses sur le contenu de certaines bases linguistiques. Pour pallier ce problème il nous faut :

- revoir la structure des ressources linguistiques ;
- ré-implémenter Flaubert.

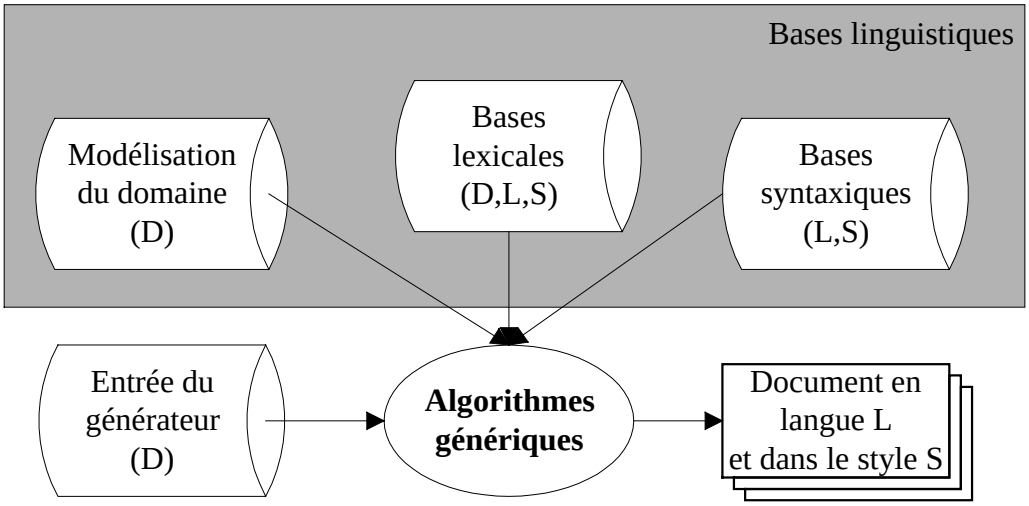

Figure 1 : Architecture du générateur

Par ailleurs, ces ressources, pour un déploiement optimal, doivent être accessibles facilement par le linguiste, pour deux raisons :

- prototypage rapide de générateurs ;
- maintenance facilitée des générateurs déjà développés.

La figure 2 montre l'architecture générale de l'outil qui nous permettra de déployer le formalisme.

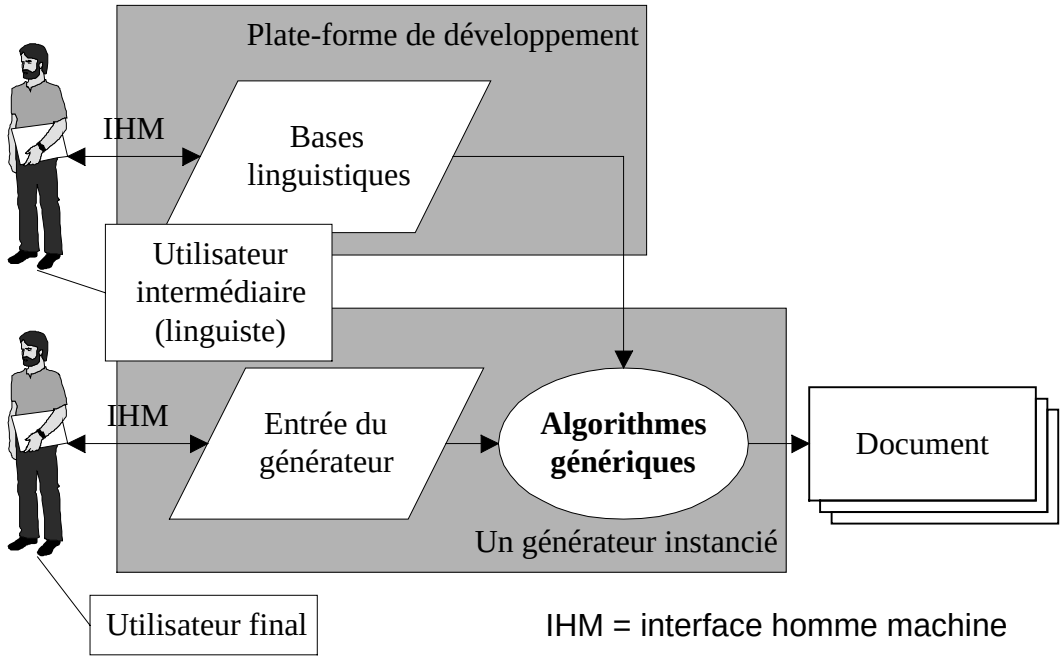

Figure 2 : Architecture de la plate-forme

Dans Flaubert, nous disposons d'une seule interface homme machine (IHM), celle permettant à l'utilisateur final de construire facilement une entrée du générateur. Il nous faut donc développer l'autre IHM qui permet à l'utilisateur intermédiaire de développer et maintenir facilement les bases linguistiques.

Ainsi, il nous faut :

- ré-implémenter Flaubert et son IHM pour l'utilisateur final ;
- implémenter une autre IHM pour l'utilisateur intermédiaire ;
- poursuivre les travaux sur le formalisme lui-même...

Pour cela nous avons mis en commun plusieurs ressources (humaines). Or, pour s'assurer que l'équipe de développement puisse communiquer sans ambiguïté, il nous faut utiliser un langage commun.

Pour ces deux raisons (revoir les ressources linguistiques en détail et avoir un langage commun), nous sommes arrivés à la conclusion suivante : formaliser ne suffit plus, **il faut modéliser les ressources linguistiques**.

2. Intérêts de la modélisation

En informatique, lorsqu'on parle de modélisation, on sous-entend l'utilisation d'une méthode de développement. Une méthode de développement présente plusieurs avantages. Tout d'abord, elle est en général la synthèse de différentes expériences (heureuses ou douloureuses), et évite au développeur de tomber dans certains pièges. Ensuite, certaines méthodes, du fait de leur large utilisation, sont outillées. Grâce à ces outils, certaines étapes peuvent être automatisées, ce qui interdit les erreurs humaines (qui sont à l'origine de la quasi totalité des *bugs*). Il existe même des outils incluant le *reverse engineering*, qui répercutent les modifications faites par le développeur à l'étape de développement précédente.

La figure 3 montre le cycle de développement d'une application du type de celles que développe un laboratoire industriel. On y voit les différents niveaux d'abstraction d'une application ainsi que les interactions entre chercheur, développeur et utilisateur. Une méthode bien outillée permet au développeur de passer sans trop d'encombres du niveau d'abstraction le plus élevé (formalisme défini par le chercheur), jusqu'au niveau physique (qui ne concerne que le développeur et la machine), puis de remonter jusqu'à l'interface utilisateur. Ainsi, la modélisation peut être vue comme une interface entre chercheur et utilisateur.

Il est communément admis (Gaudel et al. 1996, Rosen 95) que les erreurs les plus chères sont les erreurs conceptuelles, *i.e.* celles faites aux niveau d'abstraction le plus élevée. Ce type « d'erreurs » est fréquent dans le cadre d'un laboratoire industriel, car la recherche est en perpétuel mouvement, et nous ne pouvons interdire aux chercheurs de remettre en question ses spécifications. Cependant, il est aussi vrai que si nous disposons d'une modélisation, il nous est plus facile d'évaluer les répercussions (financières ou autres) d'une modification au niveau conceptuel.

Enfin, une modélisation bien menée, nous oblige à distinguer les différents niveaux d'abstraction, ainsi que les personnes qui y ont accès. Or l'utilisateur et le chercheur communiquent avec l'application au même niveau d'abstraction. Ceci facilite grandement le retour sur expérience, qui est un de nos objectifs (améliorer le formalisme initial grâce au déploiement).

3. Choisir une méthode

Nous avons montré que modélisation et méthode étaient liées. Ainsi choisir un modèle revient à choisir une méthode. Nous allons présenter deux d'entre elles dans les sections suivantes[1].

[1] Nous invitons le lecteur à se reporter à (Moréjon 92, Gabay 98) pour une présentation complète de ces deux méthodes.

3.1. Merise

Cette méthode (Tardieu et al. 83) s'appuie sur le modèle entité relation (Centre technique informatique 1979). Elle est largement utilisée spécialement dans le domaine transactionnel (bases de données relationnelles). De ce fait, le marché foisonne d'outils d'automatisation et de *reverse engineering* pour des coûts accessibles. La méthode Merise est par ailleurs complète, dans le sens où elle accompagne le développeur tout au long du cycle de développement.

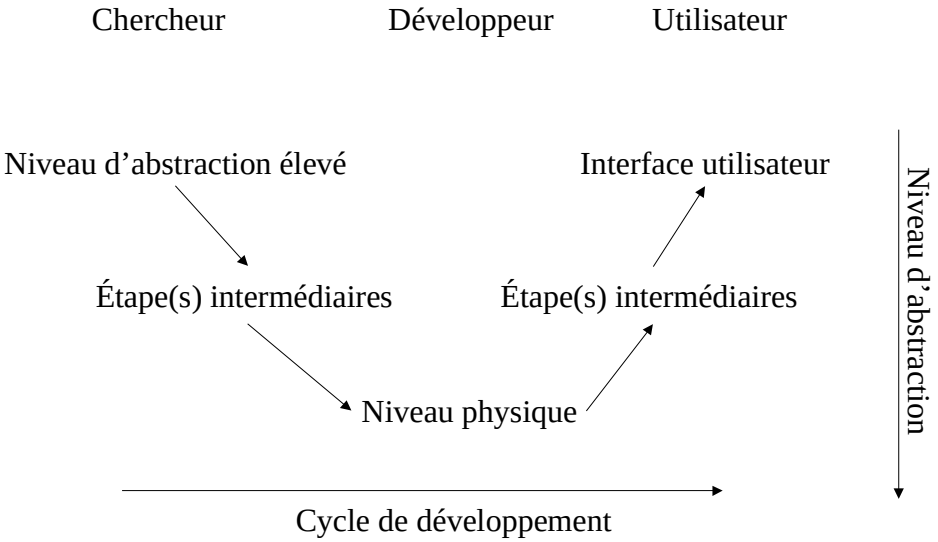

Figure 3 : Méthode et niveaux d'abstraction

3.2. UML (Unified Modeling Language)

Cette méthode, émergente, est la synthèse de trois méthodes (OMT, BOOCH, et OOSE) et s'appuie sur un modèle plus puissant que le modèle entité relation : le modèle objet. Elle tend à remplacer la méthode Merise, même dans le monde des bases de données (bases de données objet). Cependant, elle n'est pas encore stabilisée, comme le montre (Gabay 98), et n'est pas aussi outillée que la méthode Merise.

3.3. Notre choix

Nous avons donc opté pour la méthode Merise et le modèle entité relation. Il est vrai que ce choix paraît incompatible avec une autre contrainte imposée par le laboratoire : l'utilisation d'un langage objet (java) pour les IHM. Cependant, le marché dispose d'outil pouvant mixer les deux approches.

La figure 4 montre les outils que nous utilisons pour chaque étape du cycle de développement. Le passage de GTAG au modèle conceptuel de données[2] est l'étape qui est la plus difficile, et que le développeur doit répéter pour chaque application. Le passage du MCD au modèle physique de données[3] est automatique. À l'intérieur du SGBDR il existe un niveau intermédiaire, que nous appelons abusivement SQL[4]. Ce niveau s'interface en standard sous

[2] MCD, niveau d'abstraction le plus élevée du modèle entité relation.

[3] MPD, niveau d'abstraction le plus bas du modèle entité relation ; il se situe au niveau d'abstraction le plus élevé des systèmes de gestion de bases de données relationnelles — SGBDR.

[4] *Standard Querying Language*.

Microsoft Windows avec le pont ODBC[5] quel que soit le SGBDR utilisé. Ensuite, le pont ODBC s'interface en standard avec l'API java via le pont JDBC[6]. Puis, l'accès au JDBC est facilité grâce à une API java commercialisée (DataSet). Enfin, nous avons encapsulé les DataSets en utilisant une API java que nous avons développée (DBxxx). Cette API donne une vue objet de la base de données relationnelle[7], ce qui permet de développer une IHM en java en respectant la « philosophie » java (codes plus lisibles, donc maintenables, etc.).

L'atout majeur des outils choisis est que seules deux étapes sont prises en charge par le développeur, les autres étant automatisées. Par ailleurs, ce sont des outils génériques, ce qui autorise un déploiement plus facile. Tout SGBDR convient dès l'instant où il interprète le langage SQL, et qu'il s'interface directement ou indirectement (via ODBC) au JDBC.

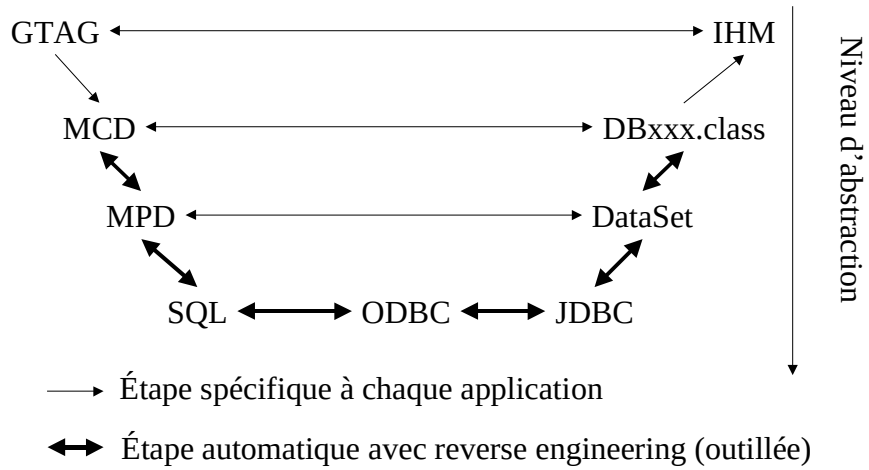

Figure 4 : Méthode et outils utilisés par le laboratoire

4. Exemple d'une application : CLEF

Dans cette section nous allons montrer un extrait des travaux de modélisation que nous avons fait dans le cadre de la ré-implémentation de Flaubert (Reyes 99, Meunier & Reyes 99) : le système CLEF (**c**hoix **l**exicaux **é**tendus et **f**ormalisés).

4.1. Rappel des spécifications

L'objectif de déploiement de GTAG impose de nouvelles spécifications :

- système multi-utilisateur ;
- système multi-domaine ;
- système multilingue.

En effet, le système doit prendre en compte qu'un générateur puisse être développé par plusieurs utilisateurs, et inversement qu'un utilisateur développe plusieurs générateurs. Par ailleurs, le système doit gérer pour chaque domaine plusieurs langues cibles, voire plusieurs styles (*i.e.* niveau de langue, comme le français contrôlé) pour chaque langue.

Pour faciliter encore le déploiement, nous avons introduit des spécifications originales : la réutilisation de ressources. En effet, un des objectifs est de faire un parallèle avec les

[5] *Open Database Connectivity.*

[6] *Java Database Connectivity.*

[7] Ce type d'API est apparu depuis sur le marché.

composants sur étagères (COTS — *Component On The Shelf*) : développer de nouveaux générateurs à partir de générateurs existant. Pour cela, il faut pouvoir :

- réutiliser des ressources d'un domaine à un autre ;
- réutiliser des ressources d'une langue à l'autre ;
- partager des ressources entre utilisateurs.

Moyennant ces spécifications, nous allons donner un extrait du modèle conceptuel de données. Nous avons choisi le niveau syntaxique, car il fait appel à des notions connues.

4.2. Exemple : modélisation du niveau syntaxique

Nous invitons le lecteur à se reporter à la littérature abondante sur le modèle entité relation. Nous supposons connues les notions suivantes :

- entité et relation ;
- identifiant, clé primaire ;
- propriété ;
- cardinalité ;
- dépendance fonctionnelle et contrainte d'intégrité fonctionnelle (notée CIF) ;
- contrainte d'inclusion.

4.2.1. MCD

La figure 5 donne un extrait du MCD du système CLEF. Cet extrait modélise une partie du niveau syntaxique des ressources linguistiques. Les entités mises en présence correspondent aux structures formelles décrites dans (Meunier 97). Nous avons donné des noms explicites aux entités modélisant ces structures formelles (par exemple l'entité ENTREE LEXICALE modélise les entrées du lexique).

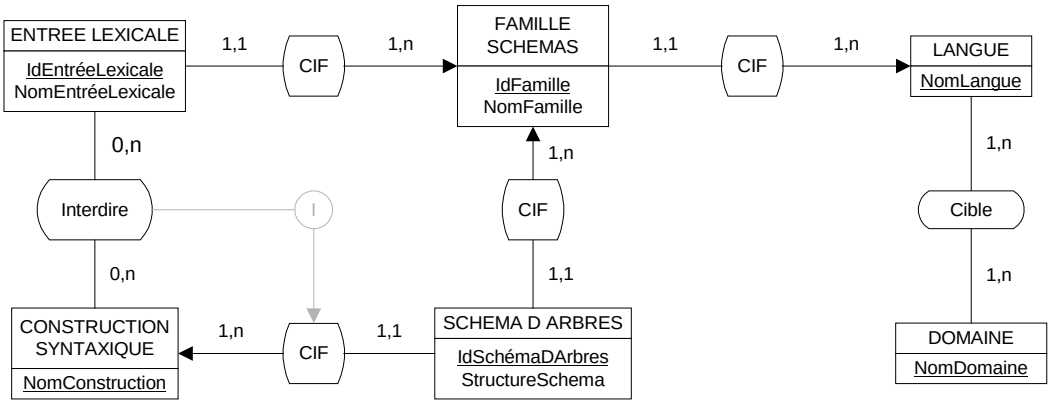

Figure 5 : Extrait du MCD du niveau syntaxique

Nous faisons volontairement pointer toute entrée lexicale vers une famille de schémas d'arbres, bien que cela ne soit nécessaire que pour les entrées prédicatives (les déterminants par exemple n'ont pas dans les grammaires TAG classiques de famille associée, mais un unique arbre élémentaire). Ce choix a été fait pour avoir un MCD allégé et des structures homogènes.

Par ailleurs, nous supposons qu'une entrée lexicale a une et une seule famille de schémas d'arbres associée comme le montrent les cardinalités données dans la figure 5 (à gauche de la contrainte d'intégrité fonctionnelle reliant ENTREE LEXICALE et FAMILLE SCHEMAS). Cette contrainte est imposée par la génération, en analyse elle n'est pas nécessaire. Ainsi, il n'est pas possible d'identifier une entrée lexicale par le lemme qu'elle utilise. En effet, prenons l'exemple du verbe *voler*, il existe au moins deux entrées lexicales utilisant ce verbe : celle de

l'intransitif (*l'avion vole*) et du transitif (*Paul vole une pomme*). Nous ne pouvons donc identifier ces deux entrées lexicales par le verbe *voler*. Une solution est de numéroter les entrées lexicales : **voler1** pour l'intransitif et **voler2** pour le transitif, et faire du verbe et du numéro l'identifiant de l'entrée lexicale. Cette solution pose un problème car nous rappelons que notre lexique est multilingue. Ainsi si un verbe anglais est homographe avec un substantif français (*to figure, une figure*), il faudrait donner des numéros différents (**figure1** pour *to figure*, **figure2** pour *une figure*), ce qui n'est pas souhaitable, ou ajouter la langue au nom de l'entrée lexicale (**figureEn** pour *to figure*, et **figureFr** pour *une figure*). Mais, nous serions redondant, car nous connaissons par l'intermédiaire de l'entité FAMILLE SCHEMAS la langue de l'entrée lexicale. Or la redondance est rigoureusement interdite dans toute modélisation (une redondance entraîne à un moment ou un autre des erreurs de synchronisation). Nous avons donc décidé d'identifier une entrée lexicale par un numéro (que l'utilisateur ne connaît pas), et de lui adjoindre comme propriété un nom d'entrée lexicale (qui peut être le lemme qu'utilise l'entrée lexicale).

Nous voyons également dans la figure 5, qu'une famille de schémas d'arbres est identifiée par un numéro, et non par son nom. Ce choix est encore motivé par le multilinguisme, car rien n'interdit d'avoir une famille N1VN2 en français et en anglais. Le même raisonnement s'applique aux schémas d'arbres, pour lesquels la propriété StructureSchema est une chaîne de caractères caractérisant la structure du schéma d'arbres.

Ensuite, nous voyons qu'un schéma d'arbres pointe vers une et une seule construction syntaxique que nous identifions par son nom (actif, passif sans agent par exemple). Or nous savons que certaines entrées lexicales ont des constructions interdites (par exemple le verbe *coûter* n'a pas de passif avec ou sans agent). Ainsi, l'entité CONSTRUCTION SYNTAXIQUE est en relation, via Interdire, avec l'entité ENTREE LEXICALE. Les cardinalités indique qu'une entrée lexicale peut avoir zéro, une ou plusieurs constructions interdites, et qu'une construction syntaxique peut ne jamais être interdite, ou être interdite pour une ou plusieurs entrées lexicales.

Cependant la relation Interdire ne peut pas relier n'importe quelle construction avec n'importe quelle entrée lexicale. En effet, il faut s'assurer que la construction syntaxique soit celle d'un schéma d'arbres de la famille vers laquelle pointe l'entrée lexicale. Par exemple, l'entrée lexicale **coûter** pointe vers la famille N1VN2 (*le livre coûte dix francs*), dans cette famille nous avons des schémas d'arbres décrivant le passif avec agent et le passif sans agent. Il est alors possible de mettre en relation, via Interdire, l'entrée **coûter** est les deux constructions syntaxiques du passif, cependant, il est impossible de la mettre en relation avec la construction antéposée (spécifique des connecteurs). Cette contrainte est appelée contrainte d'inclusion, et est décrite par un I dans le MCD en orientant le sens de l'inclusion.

4.2.2. MPD

Le passage au MPD est quasi immédiat, et ne nécessite pas forcément d'outil. Les règles sont simples : chaque entité et relation donne lieu a une table, sauf dans le cas où les cardinalités sont 0,1 ou 1,1. Pour les entités les identifiants deviennent la clé primaire de la table, et pour les relations, la clé primaire est donnée par les identifiants des entités mis en relation. Dans le cas de cardinalité 0,1 ou 1,1 on parle de clé externe ou étrangère qui est ajoutée à la table de l'entité portant cette cardinalité. Cette clé externe est l'identifiant de(s) l'autre(s) entité(s) mis en relation. Ainsi, en prenant la convention suivante (le + et le * désignent les opérateurs classiques des expressions rationnelles) :

Nom de la table((clé primaire de la table)+, (*clé externe*)*, (propriété)*)

nous avons les tables suivantes :

- **EntréeLexicale**(IdEntréeLexicale, *IdFamille*, NomEntréeLexicale) ;
- **FamilleSchémas**(IdFamille, *NomLangue*, NomFamille) ;
- **SchémaDArbres**(IdSchéma, *IdFamille*, *NomConstruction*, StructureSchéma) ;
- **ConstructionSyntaxique**(NomConstruction) ;
- **Langue**(NomLangue) ;
- **Domaine**(NomDomaine) ;
- **Cible**(NomLangue, NomDomaine) ;
- **Interdire**(IdEntréeLexicale, NomConstruction).

4.2.3. SQL

Les instructions SQL de création des première et dernière tables sont données ci-dessous :

```
CREATE TABLE EntreeLexicale(
   IdEntreeLexicale INTEGER CONSTRAINT LaCle PRIMARY KEY,
   IdFamille INTEGER CONSTRAINT LaCleExterne
      REFERENCES FamilleSchemas(IdFamille),
   NomEntreeLexicale TEXT);
```

```
CREATE TABLE Interdire(
   IdEntreeLexicale INTEGER CONSTRAINT LaCleExterne1
      REFERENCES EntreeLexicale(IdEntreeLexicale),
   NomConstruction TEXT CONSTRAINT LaCleExterne2
      REFERENCES ConstructionSyntaxique(NomConstruction),
   CONSTRAINT LaCle PRIMARY KEY (IdEntreeLexicale,NomConstruction))
```

Ces requêtes sont standards et ne dépendent pas du SGBDR utilisé. Quant à la contrainte d'inclusion de notre exemple, elle ne peut être traduite simplement dans certains SGBDR ; elle est donc prise en compte par l'implémentation (*i.e.* l'IHM de développement de générateurs).

4.2.4. DBxxx

Nous avons implémenté une API permettant d'encapsuler une base de données relationnelle dans un modèle objet. Grâce à cette API, nous pouvons donner une vue objet de nos ressources au travers de classes d'objets, que nous notons DBxxx.

4.2.5. IHM et moteur de génération

(Meunier & Reyes 99) se sont appuyés sur les objets DBxxx pour implémenter l'IHM permettant à un utilisateur intermédiaire de développer des générateurs[8]. Les figures 6 et 7 donnent deux copies d'écran de cette IHM, donnant deux vues différentes des ressources linguistiques : celle du lexique et celle de la grammaire. On retrouve les relations entre entrées lexicales et constructions syntaxiques (forbidden constructions, dans la figure 6), entre entrées lexicales et familles de schémas d'arbres (une entrée lexicale est associée à une et une seule famille, dans la figure 6), et, entre schémas d'arbres, familles et constructions syntaxiques (les deux contraintes d'intégrité fonctionnelle qui a un schéma associe une et une seule famille et une et une seule construction syntaxique, dans la figure 7).

[8] Cette IHM peut également être utilisée pour développer des grammaires d'analyse. Pour cela, il suffit de masquer les ressources spécifiques à la génération de texte.

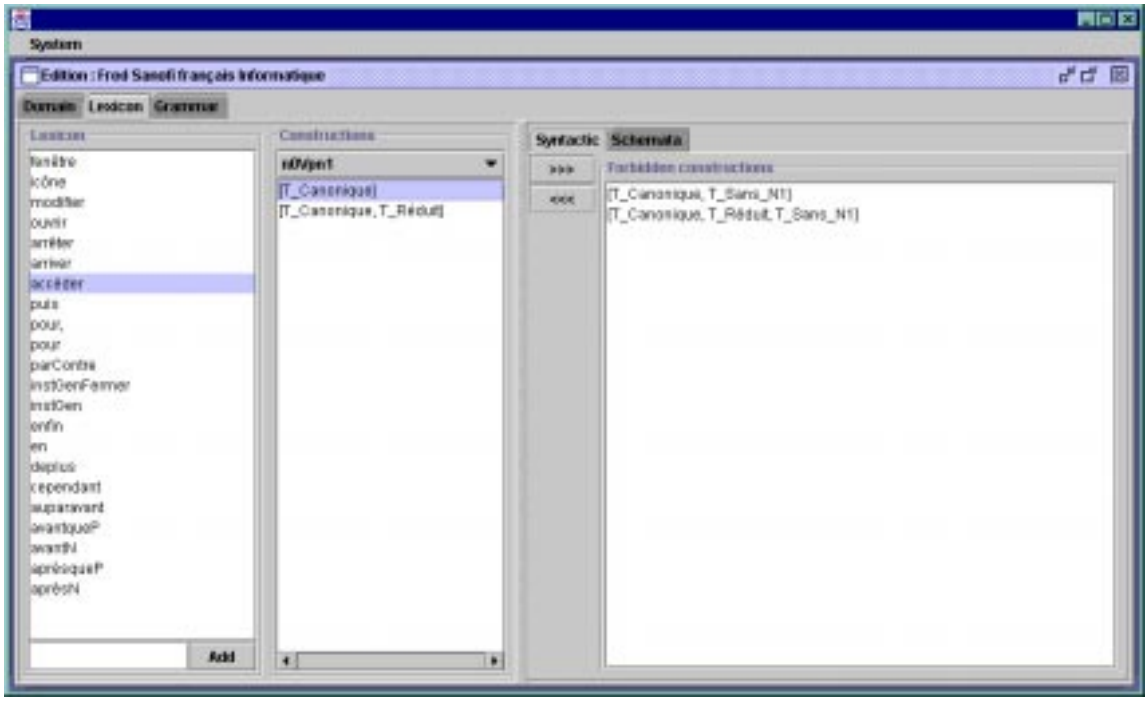

Figure 6 : L'IHM, vue du lexique

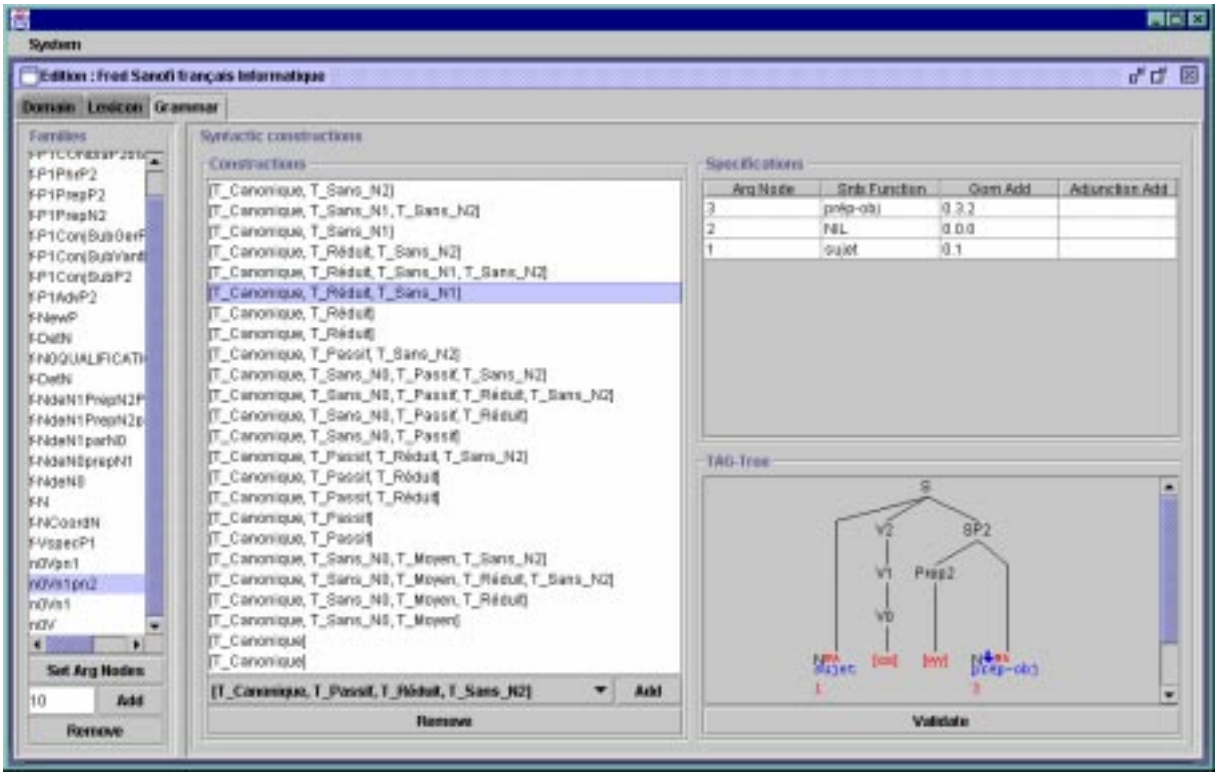

Figure 7 : L'IHM, vue de la grammaire

Conclusion

Il est vrai que l'investissement que nous avons fait dans la première étape de modélisation (passage de GTAG au MCD) est assez important (trois hommes mois), mais le retour sur investissement se fait déjà sentir. En effet, la modélisation que nous avons faite, et le choix des outils utilisés, nous a permis, en quelques minutes, d'enrichir certaines ressources de propriétés supplémentaires (le trait form tel que défini par Danlos 98). Les modifications de

ce type ne représentent donc pas un surcoût prohibitif, et permettent donc au formalisme de vivre.

Par ailleurs, même si nos objectifs concernant le déploiement d'un formalisme ne sont pas novateurs (le lecteur pourra, par exemple, se reporter au travaux de John Bateman disponibles sur ftp.darmstadt.gmd.de/pub/komet/KPML-2.0/ dont les objectifs sont similaires), nous avons montré ici que l'application d'une méthode de génie logiciel peut constituer une avancée bénéfique au traitement automatique du langage naturel.

Remerciements

Nous tenons à remercier Laurence Danlos pour ses conseils judicieux qui nous ont aidé à rédiger cet article. Nous tenons également à remercier Rodrigo Reyes qui a su mettre en valeur nos travaux au travers de ses développements. Enfin, nous remercions Thierry Poibeau pour sa contribution aux relectures de cet article.

Références

Centre technique informatique (1979), *Méthode de définition d'un système d'information*, Vol. 0 à 4

Danlos L. (1994), « Génération de textes dans le formalisme GTAG inspiré des grammaires d'arbres adjoints », *Rapport technique TALANA*, Vol. 10, projet Flaubert

Danlos L. (1997), « Les fonctions syntaxiques dans G-TAG », *Rapport technique TALANA*, Vol. 13

Danlos L. (1998), « G-TAG : un formalisme lexicalisé pour la génération de textes inspiré de TAG », *TAL*, Vol. 39.2

Danlos L., Meunier F. (1996), « La génération multilingue : applications industrielles et réalisation scientifique », *Langues situées, technologie communications*, 1996

Gabay J. (1998), *Merise. Vers OMT et UML. Un guide complet avec études de cas, 3ème édition*, InterEditions

Gaudel M.-C., Marre B., Schlienger F., Bernot G. (1996), *Précis de génie logiciel*, Masson

Joshi A., Levy L., Takahashi M. (1975), « Tree Adjunct Grammars », *Journal of the Computer and System Science*, Vol. 10:1

Meunier F. (1997), *Implantation du formalisme de génération GTAG*, Thèse de doctorat, Paris 7

Meunier F., Reyes R. (1999), *La plate forme de développement de générateurs de textes CLEF*, à paraître

Moréjon J. (1992), *Principes et conception d'une base de données relationnelle*, Éditions d'organisation

Paroubeck P., Schabes Y., Joshi A. (1992), « X-TAG: A graphical Workbench for developing TAGs », *4th Conference on Applied NL Processing*, 1992

Rosen J.-P. (1995), *Méthodes de génie logiciel avec Ada 95*, InterEditions.

Reyes R. (1999), *Le moteur de génération de textes de CLEF*, à paraître

Tardieu H., Rochfeld A., Coletti R. (1983), *La méthode Merise. Tome 1 : principes et outils*, Éditions d'organisation

Tardieu H., Rochfeld A., Coletti R., Panet G., Vahee G. (1985), *La méthode Merise. Toms 2 : démarche et pratiques*, Éditions d'organisation