

Reconnaître des langages avec un configurateur

Mathieu Estratat & Laurent Henocque

LSIS - UMR CNRS 6168,

Université d'Aix-Marseille III

Avenue Escadrille Normandie Niemen

13397 Marseille cedex 20

`{mathieu.estratat;laurent.henocque}@lsis.org`

Résumé

Les évolutions récentes des théories linguistiques font largement appel au concept de contrainte. De plus, les caractéristiques générales des grammaires de traits ont conduit de nombreux auteurs à pointer la ressemblance existant entre ces notions et les objets ou frames. Nous montrons qu'une évolution des programmes à base de contraintes, appelée programmes de configuration, peut être appliquée à la reconnaissance de langages et proposons une traduction systématique des concepts et contraintes présentés dans les grammaires de propriétés en terme de problèmes de configuration. Nous validons cette traduction par son application tout d'abord à un langage (hors contexte) récursif et à la sémantique associée, puis à un sous-ensemble du langage naturel contenant des ambiguïtés lexicales. Notre approche dépasse les implémentations existantes d'une part des grammaires de propriétés car la procédure de recherche est générique et d'autre part, celles par satisfactions de contraintes des grammaires de dépendance car la configuration est une extension des CSP. De plus, notre approche valide l'intégration de la sémantique associée aux langages. Nos expérimentations montrent l'efficacité pratique de cette approche qui ne requiert pas d'algorithmes ad hoc et permet une utilisation aussi bien analytique que générative ou encore hybride du programme.

1 Introduction

Les évolutions récentes des théories linguistiques font largement appel au concept de contrainte [PS94, Bla00, Bla01, Duc99]. En accord avec ces formalismes, une construction syntaxique est dite valide lorsque les contraintes portant sur les traits sont satisfaites. Au moins deux approches se déclarent explicitement être exclusivement à base de contraintes. L'implémentation des grammaires de dépendance dans [Duc99] utilise le cadre générique des CSP (suivant un paradigme de programmation par contraintes concurrentes) avec des ensembles de variables et des contraintes de sélection. Les grammaires de propriétés [Bla00, Bla01] sont proposées avec un algorithme de passage spécifique. Bien que très différents en nature, ces deux points de vue argumentent en

faveur du fait que la propagation de contraintes est un outil efficace pour désambigüer le langage naturel. De plus, les propriétés générales des grammaires de traits, ont mené de nombreux auteurs à pointer la ressemblance existante entre ces notions et les frames ou objets ainsi que le besoin de l'héritage multiple [PS94].

Dans le même temps, une évolution récente de la programmation par contrainte vers les problèmes de configuration a favorisé le développement de configurateurs efficaces avec de potentielles applications en IA [Mai98, SNTS01]. Un outil tel que JConfigurator [Mai98] repose entièrement sur des ensembles contraints de variables, tout comme l'implémentation des grammaires de dépendance dans [Duc99]. Dans cet article, le passage est vu comme une tâche de configuration. Il est important de noter que l'implémentation de Mozart en langage OZ repose sur des contraintes de traits qui possèdent une application immédiate à l'analyse du langage naturel [VBD⁺03], ce qui de plus, accentue l'idée de mélanger ces concepts.

Configurer consiste à simuler la réalisation d'un produit complexe à partir de composants choisis dans un catalogue de types. Ni le nombre ni les types des composants requis ne sont connus au départ. Les composants sont soumis à des relations (cette information est appelée "partonomique") et leurs types sont soumis à des relations d'héritage (information "taxonomique"). Les contraintes (appelées aussi règles de bonne formation) définissent l'ensemble des produits valides de manière générique. Un configurateur prend en entrée un fragment de la structure de l'objet à construire et l'étend en une solution du problème, si elle existe. Ce problème est indécidable dans le cas général.

Un tel programme est décrit par un modèle orienté objet (comme illustré par les figures 3 et 5), assorti de contraintes de bonne formation. Résoudre techniquement le problème d'énumération associé peut être fait en utilisant différents formalismes ou approches techniques : extensions des CSP [MF90, FFH⁺98], approches basées sur la connaissance [Stu97], logiques terminologiques [Neb90], programmation logique étendue (chaînage avant et arrière, sémantiques non standard) [SNTS01], approches orientées objet [Mai98, Stu97]. Nos expérimentations sont conduites avec le configurateur orienté objet Ilog JConfigurator [Mai98].

Les configurateurs ont montré leurs capacités à traiter des modèles objet complexes dans de nombreuses applications industrielles. Leur utilisation générale est de représenter la sémantique d'un domaine de connaissance fini. L'approche générale en analyse du langage naturel voit la syntaxe et la sémantique comme deux notions séparées, traitées par des formalismes différents (par exemple, HPSG¹ pour la syntaxe et λ -calcul plus logique pour la sémantique). Nous décrivons ici des expérimentations conduites sous un point de vue radicalement différent, comme une réponse à la question : un configurateur peut-il traiter à la fois la syntaxe et la sémantique d'un ensemble de phrases ayant trait à un certain domaine de connaissance ?

Notre intuition est que le procédé de construction d'un arbre de passage est similaire à une activité de configuration (en effet, de nouvelles constructions sont introduites pour grouper les mots au sein de catégories adéquates comme le syntagme verbal ou le syntagme nominal et ces constructions sont liées par des relations). Parmi les bénéfices potentiels se trouve le fait que nous pourrions prévoir la sémantique exacte pour des langages portant sur un domaine de connaissance défini et aussi qu'une étroite collaboration entre la syntaxe et la sémantique pourra être obtenue.

Nous présentons ici une application de la configuration au problème de l'analyse de langages, un travail qui a débuté avec [Est03]. A cette fin et parceque nous visons à analyser des langages naturels, nous ne proposons pas de formulation ad hoc du problème mais une traduction systématique des grammaires de propriétés [Bla01] en problème de configuration. Pour montrer

¹Head Driven Phrase Structure Grammars[PS94]

la validité de l'approche proposée, nous détaillons deux exemples : le premier est l'analyse de l'archétype des grammaires hors contexte $a^n b^n$ et le second est l'analyse d'un sous ensemble simple du langage naturel. La grammaire $a^n b^n$ est récursive, ainsi malgré son apparente simplicité, elle présente une difficulté inhérente au langage naturel (les syntagmes nominaux sont récursifs). Dans cet exemple, un unique modèle objet décrit à la fois la syntaxe et la sémantique fondamentalement simple. L'autre exemple est un sous ensemble du langage naturel contenant des ambiguïtés lexicales. Dans les deux cas, nous montrons que le configurateur utilisé pour l'analyse peut être exploité d'une manière analytique, générative ou mixte et que la propagation des contraintes pour résoudre ces problèmes ne nécessite que peu ou pas de recherche.

Il y a plusieurs motivations pour se placer dans le contexte des grammaires de propriétés (au lieu des grammaires de dépendance par exemple), qui accentuent la contribution originale de ce travail. Les grammaires de propriétés utilisent des catégories de même que la plupart des théories CL² depuis GPSG³ et HPSG[PS94]. Ceci procure un large corpus de recherche et de grammaires existantes. [Bla01] présente une grammaire pour le français suffisamment riche pour de nombreuses applications sérieuses. Les sept sortes de contraintes dans les grammaires de propriétés sont très facile à traduire en contraintes de configuration et sont également plus facile à lire ou à comprendre que certains de leurs équivalents en grammaires de dépendance. En outre, une partie de la puissance de l'implémentation à base de CSP des grammaires de dépendance est limitée par le fait qu'aucune catégorie intermédiaire n'est nécessaire, par conséquent que la taille totale du problème est connue au départ. Cette limitation peut rendre maladroite la formulation des contraintes. De plus, les CSP standards sont connus depuis longtemps pour être trop limités pour de réelles tâches de configuration [MF90, Mai98, GK99, SNTS01, AFM02].

En particulier, la nature dynamique des problèmes de configuration doit être justifiée, au moins par les variables d'activité comme dans les CSP dynamiques. Dans notre cas, lorsque la sémantique d'une phrase se rapporte à des objets nouvellement introduits dans le discours, aucune approche basée sur les CSP ne peut convenir pour la traiter. Ainsi, en utilisant le point de vue de la configuration à la fois sur la syntaxe et sur la sémantique, nous proposons une solution unique pour mélanger ces deux aspects de l'analyse d'un langage dans un seul cadre. D'un point de vue de programmation par contraintes, ceci mène à une propagation des contraintes potentiellement optimale entre ces deux sous problèmes, utilisables aussi bien de manière analytique que générative ou mixte.

1.1 Les grammaires de propriétés

Les grammaires de propriétés [Bla00, Bla01] sont un formalisme linguistique basé sur des contraintes. [Bla01] propose à la fois une classification des contraintes de traits appelées *propriétés* et un algorithme de parsage qui tente d'exploiter la propagation des contraintes (d'une manière "ad hoc" puisqu'aucun système de contraintes "standard" n'est utilisé) afin de résoudre les ambiguïtés lexicales le plus tôt possible. Les algorithmes mis à part, les grammaires de propriétés contiennent deux notions importantes : les *catégories* représentant toute unité syntaxique reconnue (les mots mais aussi les syntagmes) et les *propriétés* (un synonyme pour contrainte) qui s'appliquent aux catégories pour former les règles de bonne formation de la grammaire et les règles de cohésion de la phrase. Les catégories représentent aussi bien les mots que les syntagmes d'une phrase. Par exemple, la figure 8 montre que "la porte" est un *syntagme nominal(SN)* où "la" est un *déterminant* et "porte" un *nom*. A chacun de ces mots ou groupe de mots est associé

²Computational Linguistics

³Generalized Phrase Structure Grammars [GKPS85]

une catégorie, *SN*, *Det* et *N* respectivement.

1.2 Catégories et programmation par contraintes

Les catégories sont des structures de traits. Une structure de traits est un ensemble de couples (*attribut*, *valeur*) utilisés pour étiqueter une unité linguistique, comme sur la figure 1, où *livre* est un *nom masculin* à la *3^{ième} personne du singulier*. Cette définition est récursive : une valeur de trait peut être une autre structure de trait ou un autre ensemble de traits.

$$\left[\begin{array}{ll} \text{Cat :} & \text{N} \\ \text{Phon :} & \text{livre} \\ \text{Accord :} & \left[\begin{array}{ll} \text{Gen :} & \text{masc} \\ \text{Nb :} & \text{sing} \\ \text{Per :} & \text{3ième} \end{array} \right] \\ \text{Type :} & \{\text{Commun}\} \end{array} \right]$$

FIG. 1 – Un exemple de la catégorie N

Fonctionnellement, un trait peut être assimilé à une variable de CSP et une structure de trait peut être vue comme une assignation de valeurs à un agrégat de variables de trait. Une valeur de trait peut être une constante d'un domaine spécifique (par exemple, une énumération telle que $\{\text{Singulier}, \text{Pluriel}\}$ ou un entier tel que $\{1(\text{er}), 2(\text{nd}), 3(\text{ième})\}$). Une valeur de trait peut aussi être une (liste de, ensemble de) structure(s) de traits (comme *Accord* dans la figure 1). Ainsi, les domaines finis des variables CSP ne peuvent pas être utilisées pour modéliser les traits et une notion de relations, ou de variables ensemblistes doit être utilisé (comme dans [Mai98, Stu97, Duc99]). [BBF94] propose un traitement des logiques attribut-valeur (structure de traits à valeurs atomiques) sous la forme d'un système de résolution de CSP dynamiques. Ce système ne permet pas de prendre en compte les valeurs de traits complexes. De plus, les contraintes utilisées sont strictement binaires et alourdissent la formulation du problème. Il est utile de préciser que les structures de traits sont disponibles dans le langage construit en OZ[ST94] et supportant les contraintes de traits.

1.3 Les propriétés face aux contraintes

Les propriétés[Bla01] sont des contraintes portant sur les catégories et qui spécifient les règles de bonne formation syntaxique et les règles de cohésion de la phrase. Il y a sept sortes de propriétés : *constitution*, *noyau*, *unicité*, *exigence*, *exclusion*, *linéarité* et *dépendance* détaillées en section 2.2. Ces propriétés, et nous le verrons par la suite, se traduisent aisément dans un système à base de contraintes. Par exemple, la contrainte de linéarité (ou précédence) peut être implémentée en utilisant une relation d'ordre sur les entiers. Certaines contraintes, comme celles de constitution ou de noyaux, nécessitent une traduction plus orientée objet, impliquant au moins des variables ensemblistes.

1.4 Plan de l'article

La section 2 décrit une traduction des grammaires de propriétés en terme de problème de configuration. Les sections 3 et 4 présentent des exemples. Nous verrons enfin en section 5 quels sont les apports de notre approche ainsi que les pistes de recherche à explorer.

2 Des grammaires de propriétés à la configuration

2.1 Un modèle objet pour les catégories

Les éléments structurels des formalismes linguistiques se traduisent naturellement en terme de problèmes de configuration. Un *trait* correspond à une variable CSP. Une *structure de trait* est un agrégat de traits facilement représentable par des classes dans un modèle objet. Une *catégorie* se traduit naturellement en une classe d'un modèle objet, appartenant à une hiérarchie de classes et soumis à des relations d'héritage (éventuellement multiple [PS94]). De nombreux traits ont comme valeurs des (ensembles de) structures de traits. Cette situation peut être facilement représentée en utilisant des relations entre les classes dans un modèle objet. Lorsque le configurateur utilisé est lui-même orienté objet [Mai98], de telles relations sont implémentées par des variables ensemblistes. Par exemple, un syntagme nominal peut avoir comme noyau⁴ un nom. Cette relation entre catégories peut être adéquatement décrite en utilisant une relation dans le modèle objet correspondant. Par exemple, la catégorie de la figure 1 est traduite en une classe d'un modèle objet comme illustré par la figure 2. Sur cette figure, trois classes sont présentes :

- *Accord* qui représente le trait à valeur complexe *Accord* de la figure 1 et dont les attributs représentent les traits associés à la valeur complexe de ce dernier.
- *CatTerminale* permettant de factoriser l'attribut *phon*, présent dans toute catégorie terminale (*nom*, *déterminant*, *verbe*, *adjectif*, etc ...).
- *N* représentant la structure de trait de la figure 1. L'attribut *Type* lui est spécifique, c'est à dire que seule cette catégorie possède un attribut *Type*. Par héritage de la classe *CatTerminale* cette classe possède l'attribut *Phon* et une relation avec la classe *Accord*. Elle traduit ainsi parfaitement la structure de trait de la figure 1.

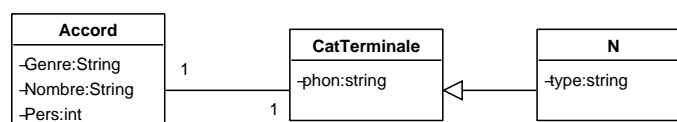


FIG. 2 – Un modèle objet pour la catégorie N

2.2 Contraintes sur le modèle objet pour les propriétés

Les propriétés définissent les contraintes ainsi que les relations sur le modèle objet construit à partir d'une grammaire de propriétés donnée. Nous utilisons des lettres majuscules pour dénoter les catégories (ex : $S, A, B, C \dots$). Nous utilisons aussi les notations suivantes : lorsqu'une relation existe entre deux catégories S et A , on note par $s.A$ l'ensemble des A liés à un s , instance donnée de S et par $|s.A|$ leur nombre. Par simplicité et dès lors que cela sera possible, nous utiliserons la notation $\forall SF(S)$ (où F est une formule appliquée au symbole S) plutôt que $\forall s \in SF(s)$. Les attributs d'une classe sont dénotés en utilisant la notation pointée standard. (par exemple, $a.debut$ représente l'attribut *debut* de l'objet a). I_A représente l'ensemble des indices possibles pour les instances de la catégorie A .

⁴Informellement, le noyau est l'élément central dans un syntagme, celui qui gouverne les propriétés linguistiques

- **Constituants** : $Const(S) = \{A_m\}_{m \in I_A}$ spécifie qu'un S ne peut être constitué que d'éléments de $\{A_m\}$. Cette propriété est décrite en utilisant des relations entre S et tous les $\{A_m\}$, comme dans les modèles objet sur les figures 3 et 5.
- **Noyaux** : La propriété $Noyaux(S) = \{A_m\}_{m \in I_A}$ liste l'ensemble des catégories noyaux possibles pour la catégorie S . Le noyau est unique et obligatoire pour tout syntagme. Par exemple, $Noyaux(SN) = \{N, Adj\}$. Le mot "porte" est le noyau dans le SN : "la porte". La relation de *Noyau* est un sous-ensemble de la relation *Const*. De telles propriétés sont implémentées en utilisant des relations comme pour la constitution et des contraintes de cardinalité adéquates.
- **Unicité** : La propriété $Unic(S) = \{A_m\}_{m \in I_A}$ limite une instance de la catégorie S à n'avoir qu'au plus une instance de chaque A_m , $m \in I_A$ en tant que constituant. La propriété d'*unicité* est traitée en utilisant les contraintes de cardinalité comme par ex : $\forall S |\{x : S.Const \mid x \in A_m\}| \leq 1$, que par simplicité et par la suite, nous noterons $|S.A_m| \leq 1$. Par exemple, dans un SN le déterminant *Det* est unique.
- **Exigence** : $\{A_m\}_{m \in I_A} \Rightarrow_S \{\{B_n\}_{n \in I_B}, \{C_o\}_{o \in I_C}\}$ spécifie que chaque occurrence de l'ensemble des A_m implique la présence dans sa totalité d'au moins un des ensembles de catégories $\{B_n\}$ ou $\{C_o\}$ parmi les constituants de S . Par exemple, dans un syntagme nominal, si un nom commun est présent, alors un déterminant doit aussi apparaître ("porte" ne forme pas un syntagme nominal valide, tandis que "la porte" oui). Cette propriété est traduite par la contrainte :

$$\forall S (\forall m \in I_A |S.A_m| \geq 1) \Rightarrow ((\forall n \in I_B |S.B_n| \geq 1) \vee (\forall o \in I_C |S.C_o| \geq 1)) \wedge (|S.B_n| + |S.C_o| \geq |S.A_m|)$$

- **Exclusion** : La propriété $\{A_m\}_{m \in I_A} \nRightarrow_S \{B_n\}_{n \in I_B}$ déclare les ensembles de catégories s'excluant mutuellement (ne pouvant être présentes ensemble dans un même syntagme). Elle peut être implémentée par la contrainte :

$$\forall S, \begin{cases} (\forall m \in I_A |S.A_m| \geq 1) \Rightarrow (\forall n \in I_B |S.B_n| = 0) \\ \wedge \\ (\forall n \in I_B |S.B_n| \geq 1) \Rightarrow (\forall m \in I_A |S.A_m| = 0) \end{cases}$$

Par exemple, un nom (N) et un pronom (Pro) ne peuvent pas apparaître ensemble dans un SN . Notons que dans la formulation de cette contrainte, \Rightarrow représente l'implication logique et non la propriété d'exigence.

- **Linearité** : La propriété $\{A_m\}_{m \in I_A} \prec_S \{B_n\}_{n \in I_B}$ signifie que toute occurrence d'un $\{A_m\}_{m \in I_A}$ précède toute occurrence d'un $\{B_n\}_{n \in I_B}$. Par exemple, dans un SN , un *Det* doit précéder un N (si ces deux catégories sont présentes). Pour traiter cette contrainte, il a été nécessaire en premier lieu, d'introduire dans la représentation des catégories dans le modèle objet, deux attributs entiers *debut* et *fin* qui représentent respectivement les positions du premier et du dernier mot d'une catégorie. Ensuite, la propriété de linéarité est traduite par la contrainte :

$$\forall S \forall m \in I_A \forall n \in I_B, \\ max(\{i \in S.A_m \bullet i.fin\}^5) \leq min(\{i \in S.B_n \bullet i.debut\})$$

⁵ $\{i \in S.A_m \bullet i.fin\}$ se lit : l'ensemble des valeurs des attributs *fin* de i , pour i appartenant à $S.A_m$. Voir [Hen03] pour plus de détails.

- **dépendance** : Cette propriété établit des relations spécifiques entre catégories distantes, en relation avec la sémantique du texte (comme pour dénoter, par exemple, le lien existant entre un pronom et son référent dans une phrase précédente). Par exemple, dans un syntagme verbal, il existe une relation de dépendance entre le syntagme nominal sujet et le verbe. Cette propriété est représentée de manière adéquate par une relation.

Les propriétés sont ainsi transcriptibles par des contraintes indépendantes. Mais il est également possible de factoriser certaines d'entre-elles dans le modèle, le plus souvent au moyen d'une relation et de sa cardinalité. Par exemple, constitution et unicité peuvent être groupées sous une relation vers chaque candidat constituant et de multiplicité [0,1] (nous avons fait ce choix dans les exemples suivant, voir figures 3 et 5).

3 Application au langage context free $a^n b^n$

Nous présentons maintenant une application de la traduction précédente à la description du langage $a^n b^n$, représentant archétypal des langages hors contexte. Dans [Bla01] le langage $a^n b^n$ est défini par les propriétés suivantes :

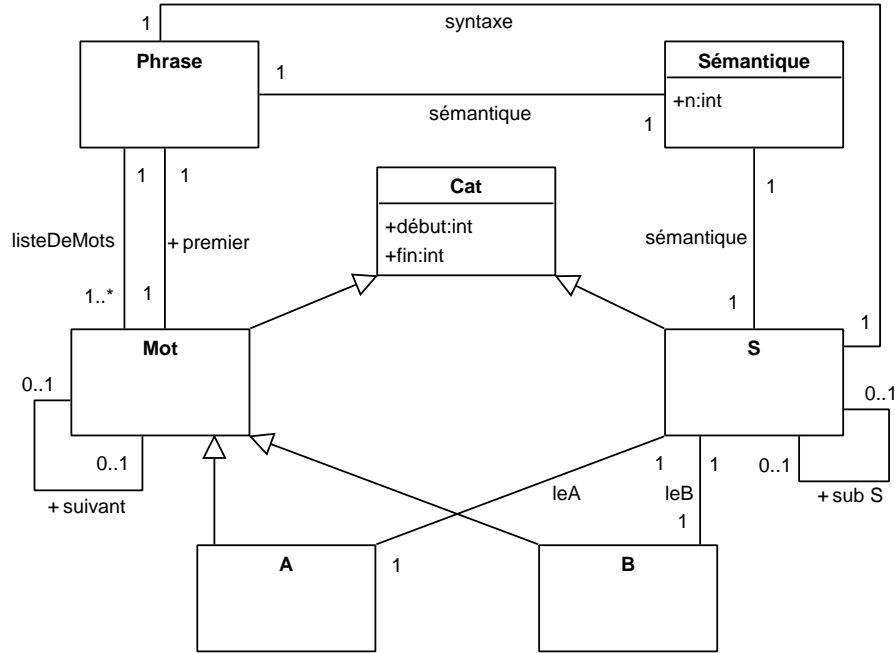
$$\left\{ \begin{array}{ll} \text{Constitution} : & \text{Const}(S) = \{S, a, b\}; \\ \text{Noyaux} : & \text{Noyaux}(S) = \{a\}; \\ \text{Unicité} : & \text{Unic}(S) = \{S, a, b\}; \\ \text{Exigence} : & a \Rightarrow b; \\ \text{Linéarité} : & a \prec b; a \prec S; S \prec b; \end{array} \right.$$

Nous implémentons cette grammaire par le modèle objet présenté sur la figure 3 et les contraintes associées. Dans ce modèle, les classes S , A et B correspondent aux catégories introduites dans [Bla01]. La classe Cat est une abstraction pour toutes les catégories. Elle fournit les attributs *debut* et *fin* nécessaires pour les contraintes de linéarité. La classe Mot est une abstraction pour les catégories terminales A et B . La classe $Phrase$ est en relation avec la liste de "mots" la composant : instances de la super classe Mot . Elle est aussi en relation avec le premier mot de cette liste. La liste de mots est implémentée en utilisant un attribut supplémentaire *suivant* réflexif sur la classe Mot . La classe $Phrase$ est liée avec la classe $Semantique$ et avec la classe S (chaque phrase est en relation avec à la fois sa syntaxe et sa représentation sémantique). La classe S est liée avec la classe $Semantique$: chaque catégorie non-terminale est liée à sa propre sémantique (nombre de A qu'elle contient, auquel elle est liée). Les propriétés de linéarité sont traduites via des contraintes additionnelles :

$$\left\{ \begin{array}{l} \forall S, S.A.debut < S.B.debut; \\ \forall S, (|S.S| == 1) \Rightarrow (S.A.fin \leq S.S.debut); \\ \forall S, (|S.S| == 1) \Rightarrow (S.S.fin \leq S.B.debut); \end{array} \right.$$

3.1 Sémantique

La sémantique associée à une phrase valide du langage $a^n b^n$ est évidemment le nombre n de a et b contenus dans la phrase. A cette fin, la classe "Semantique" du modèle possède un attribut entier n . Le nombre $S.Semantique.n$ représente le nombre de A dans un S .

FIG. 3 – Un modèle objet pour $a^n b^n$

Les contraintes liant la syntaxe et la sémantique sont :

$$\begin{cases} \forall S (|S.S| == 1) \Rightarrow S.Semantique.n = 1 + S.S.Semantique.n \\ \forall S (|S.S| == 0) \Rightarrow S.Semantique.n = 1 \\ \forall Phrase \, Phrase.Semantique = Phrase.S.Semantique; \end{cases}$$

Ces contraintes définissent récursivement la sémantique d'un S comme le nombre de A qu'elle contient et la sémantique d'une phrase comme la sémantique de sa catégorie non terminale de plus haut niveau.

3.2 L'analyse

Le configurateur est utilisé de la manière suivante : il prend en entrée un groupe d'objets partiellement connus (des instances de la classe *Mot* par exemple), partiellement interconnectés (via la relation *suivant*). Puis, il tente de compléter cette entrée en ajoutant de nouveaux objets et relations et en assignant un type à chaque objet et une valeur à chaque attribut dans le but de satisfaire toutes les contraintes du modèle.

La figure 4 présente différents fonctionnements de l'analyseur. Sur cette figure, les états du système sont décrits par des triplets $\langle mot, syntaxe, sémantique \rangle$ ("?" représente un objet inconnu, et "◇" un mot inconnu) et le comportement du système est décrit par les règles *état d'entrée* \mapsto *état de sortie*. Les deux premières lignes correspondent à des cas d'analyse pure. Dans

$$\left\{ \begin{array}{l} \langle aaabbb, ?, ? \rangle \mapsto \langle aaabbb, S(a, S(a, S(a, null, b), b), b), 3 \rangle \\ \langle abbb, ?, ? \rangle \mapsto false \\ \langle \diamond a \diamond b, ?, ? \rangle \mapsto \langle aabb, S(a, S(a, null, b), b), 2 \rangle \\ \langle ?, ?, 2 \rangle \mapsto \langle aabb, S(a, S(a, null, b), b), 2 \rangle \end{array} \right.$$

FIG. 4 – Des sessions de l'analyseur

la seconde ligne, la phrase n'appartient pas au langage (entrée inconsistante). Les deux dernières lignes de la figure 4 illustrent des utilisations génératives ou hybrides du programme.

3.3 Résultats expérimentaux

Le tableau 1 liste les résultats obtenus⁶ pour des entrées consistantes et inconsistantes de différentes tailles. La première colonne précise les entrées du configurateur et la sixième colonne le temps d'exécution en secondes.

p	$\#fails$	$\#cp$	$\#csts$	$\#vars$	$\#secs$
$aaabbb$	0	29	370	143	0,48 s
$\diamond a \diamond b$	0	22	469	119	0,39 s
$a(10) b(10)$	0	92	1672	430	0,77 s
$a(20) b(20)$	0	182	4892	840	1,33 s
$a(50) b(50)$	0	52	24152	2070	4,74 s
$a(51) b(49)$	1	0	1687	1417	3,92 s
$\diamond a(50) b(49)$	1	0	1684	1416	5,12 s

TAB. 1 – resultats expérimentaux

Ces résultats montrent que le programme fournit des temps d'exécution acceptables pour des phrases de plus de 100 mots. Ceci est dû à l'efficacité de la propagation des contraintes qui mène à la solution avec un très petit nombre de backtrack ("fails"), plus particulièrement en mode analyse pure. Le fait que le programme termine sans aucun point de choix lorsque l'entrée est inconsistante ($a(51) b(49)$ et $\diamond a(50) b(49)$) vient du fait que la seule propagation des contraintes détecte l'inconsistance. En effet, de nombreuses relations ont une multiplicité de 1, ce qui facilite une propagation efficace des contraintes. Les langages proches des langages naturels, comme illustré dans l'exemple suivant, ne possèdent pas cette particularité. Ce comportement est également souligné parceque la formulation du problème est sujette à une contrainte permettant de briser les symétries : nous nous assurons que la valeur des attributs *debut* de toutes les instances de S soient strictement croissantes.

⁶PC utilisé : P4 2,4GHz - 512 Mo DDR - Windows XP SP1 - Java 2 V.1.4.2 - Ilog Jconfigurator 2.1

4 Parser un langage naturel lexicalement ambigu

La figure 5 présente un fragment du modèle objet utilisé pour un sous-ensemble du français, où les propriétés de constitution et de linéarité sont explicitées. La figure 6 illustre quelques contraintes de bonne formation. Nous définissons dans la figure 7 un exemple de lexique simple. Dans le lexique les – correspondent à des valeurs non spécifiées ou non accessibles ou non connues.

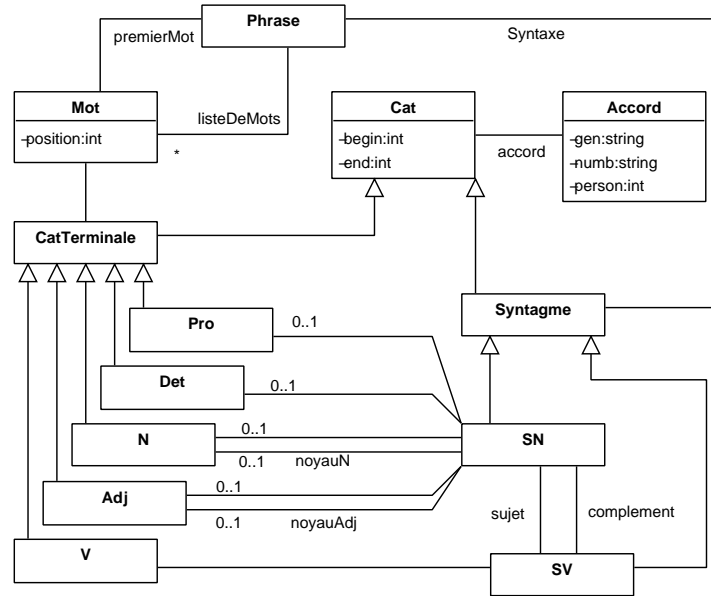


FIG. 5 – Modèle objet utilisé pour parser notre langage

Le langage reconnu par ce modèle objet est constitué de phrases contruites autour d'un sujet, un verbe et un complément, où le sujet et le verbe sont obligatoires et le sujet et le complément nécessairement des syntagmes nominaux.

Toutes ces contraintes sont exprimées simplement sur le modèle objet par l'intermédiaire des relations et de leurs cardinalités (comme présenté sur la figure 6). D'autres contraintes sont cependant nécessaires, comme la contrainte précisant qu'un noyau est unique ou la contrainte gérant les valeurs des attributs *debut* et *fin* dans les syntagmes.

Noyaux : $|SN.noyauN| + |SN.noyauAdj| = 1$;

Linéarité : $Det < N$; $Det < Adj$;

Exclusion : $(|SN.N| \geq 1) \Rightarrow (|SN.Pro| = 0)$ AND $(|SN.Pro| \geq 1) \Rightarrow (|SN.N| = 0)$;

Exigence : $(|SN.N| = 1) \Rightarrow (|SN.det| = 1)$

FIG. 6 – Quelques contraintes sur le *SN*

4.1 Résultats expérimentaux

Nous avons testé le modèle objet avec des phrases de niveau d'ambiguïté lexicale variable, suivant le lexique présenté sur la figure 7.

MOT	CAT	GEN	NBR	PERS
ferme	N	fem	sing	3
ferme	Adj	-	sing	-
ferme	V	-	sing	1,3
la	Det	fem	sing	3
la	Pro	fem	sing	3
mal	N	masc	sing	3
mal	Adj	-	-	-
porte	N	fem	sing	3
porte	V	-	sing	1,3

FIG. 7 – Une partie du lexique

La phrase (1), "la porte ferme mal" est totalement ambiguë. Dans cet exemple, "la" peut être un *pronom* ou un *déterminant*, "porte" peut être un *verbe* (conjugué au présent de l'indicatif à la troisième personne du singulier par exemple) ou un *nom*, "ferme" peut être un *verbe*, un *nom* ou un *adjectif* et "mal" peut être un *adjectif* ou un *nom*. Notre programme génère un étiquetage pour chaque mot et l'arbre syntaxique correspondant (figure 8).

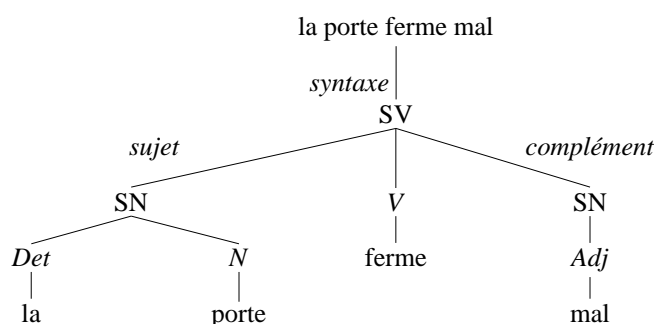


FIG. 8 – Arbre syntaxique pour la phrase en français "la porte ferme mal"

La phrase (2) est "la porte bleue possède trois vitres jaunes". Ici, "bleue" et "jaunes" sont des *adjectifs*, "vitres" est un *nom* et "trois" un *déterminant*. La dernière phrase, (3), "le moniteur est bleu" ne contient pas de mot ambigu. Le tableau 2 montre les résultats obtenus pour les phrases (1), (2) et (3).

Ces résultats montrent qu'un étiquetage syntaxique correct est obtenu après très peu de "backtracks" (*fails*). Le nombre de "fails" (ou mauvais choix) dépend du nombre de mots ambigus dans la phrase. Le temps d'exécution est dépendant, lui aussi, de la taille de la phrase. Le "backtrack" restant dans (3) provient du fait qu'une recherche minimale est obligatoirement exécutée.

p	$\#fails$	$\#cp$	$\#csts$	$\#vars$	$\#secs$
(1)	4	40	399	220	0,55 s
(2)	3	50	442	238	0,56 s
(3)	1	35	357	194	0,52 s

TAB. 2 – Résultats expérimentaux pour des phrases en français

5 Conclusion

Nous avons décrit une traduction des grammaires de propriétés en terme de problème de configuration. Nous avons également montré qu’une procédure de recherche générique, combinée à la propagation de contraintes, permet non seulement de résoudre le problème efficacement (le programme java peut parser en quelques secondes des phrases de plus de cent mots), mais offre également tous les modes d’interaction possibles. La capacité à compléter des phrases, ou à en générer, possède de nombreuses applications pratiques.

Notre approche dépasse les implémentations des grammaires de propriétés car la procédure de recherche est générique et celles des grammaires de dépendance, basées sur des contraintes car la configuration est une extension des CSP. Elle permet aussi l’intégration naturelle (même si ce n’est pas trivial) à l’analyseur syntaxique de la sémantique du langage naturel.

Nos perspectives de recherche prévoient l’implémentation d’un analyseur d’un sous-ensemble du français reposant sur la sémantique de la description de scènes en trois dimensions.

6 Remerciements

Ces travaux de recherches ont pu être réalisés grâce à un financement JemSTIC du CNRS.

Références

- [AFM02] Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic csp—application to configuration. *Artificial Intelligence*, 135(1-2) :199–234, 2002.
- [BBF94] Azeddine Belabbas, Hachemi Bennaceur, and Christophe Fouqueré. Résolution des csp par classification des contraintes : application aux logiques attribut-valeur. In *RFIA’94*, pages 409–420, Paris, 1994.
- [Bla00] Philippe Blache. Property grammars and the problem of constraint satisfaction. In *ESSLLI-2000 workshop on Linguistic Theory and Grammar Implementation*, 2000.
- [Bla01] Philippe Blache. *Les Grammaires de Propriétés : des contraintes pour le traitement automatique des langues naturelles*. Hermès Sciences, 2001.
- [Duc99] Denys Duchier. Axiomatizing dependency parsing using set constraints. In *Sixth Meeting on Mathematics of Language, Orlando, Florida*, pages 115–126, 1999.
- [Est03] Mathieu Estratat. Application de la configuration à l’analyse syntaxico sémantique de descriptions. Master’s thesis, Faculté des Sciences et Techniques de Saint Jérôme, LSIS équipe InCA, Marseille, France, submitted for the obtention of the DEA degree, 2003.

- [FFH⁺98] Gerhard Fleischanderl, Gerhard Friedrich, Alois Haselböck, Herwig Schreiner, and Markus Stumptner. Configuring large-scale systems with generative constraint satisfaction. *IEEE Intelligent Systems - Special issue on Configuration*, 13(7), 1998.
- [GK99] Andreas Günter and Christian Kühn. Knowledge-based configuration - survey and future directions. In *5th Biannual German Conference on Knowledge Based Systems, Würzburg, Germany, Lecture Notes in Artificial Intelligence LNAI 1570*, pages 47–66, March 1999.
- [GKPS85] Gerald Gazdar, Ewan Klein, Geoffrey Pullum, and Ivan Sag. *Generalized Phrase Structure Grammar*. Blackwell, Oxford, 1985.
- [Hen03] Laurent Henocque. Modeling Object Oriented Constraint Programs in Z. Technical report, LSIS Research Report, available at <http://arXiv.org/abs/cs/0312020>, 2003.
- [Mai98] Daniel Mailharro. A classification and constraint based framework for configuration. *AI-EDAM : Special issue on Configuration*, 12(4) :383 – 397, 1998.
- [MF90] Sanjay Mittal and Brian Falkenhainer. Dynamic constraint satisfaction problems. In *Proceedings of AAAI-90*, pages 25–32, Boston, MA, 1990.
- [Neb90] Bernhard Nebel. Reasoning and revision in hybrid representation systems. *Lecture Notes in Artificial Intelligence*, 422, 1990.
- [PS94] Carl Pollard and Ivan Sag. *Head-Driven Phrase Structure Grammar*. The University of Chicago Press, Chicago, 1994.
- [SNTS01] Timo Soinen, Ilkka Niemela, Juha Tiihonen, and Reijo Sulonen. Representing configuration knowledge with weight constraint rules. In *Proceedings of the AAAI Spring Symp. on Answer Set Programming : Towards Efficient and Scalable Knowledge*, pages 195–201, March 2001.
- [ST94] Gert Smolka and Ralf Treinen. Records for logic programming. *The Journal of Logic Programming*, 18(3) :229–258, April 1994.
- [Stu97] Markus Stumptner. An overview of knowledge-based configuration. *AI Communications*, 10(2) :111–125, June 1997.
- [VBD⁺03] Peter Van Roy, Per Brand, Denys Duchier, Seif Haridi, Martin Henz, and Christian Schulte. Logic programming in the context of multiparadigm programming : the Oz experience. *Theory and Practice of Logic Programming*, 2003. To appear.