XTiger

Templates, Microformats et d'Autres (Petites) Choses

Auteur : **Francesc Campoy Flores**

Tuteurs : **Vincent Quint, Irène Vatton**

INRIA — Web Adaptation Multimédia

Université Joseph Fourier — ENSIMAG INPG — Universitat Politècnica de Catalunya

Remerciements

Je remercie Vincent Quint et Irène Vatton pour toute l'aide, les corrections et l'attention qu'il m'ont offert pendant mon stage au sein de leur équipe. De la même façon, je remercie aussi tous les membres du projet WAM de l'INRIA Rhône-Alpes pour m'avoir accueilli, et plus concrètement je remercie Laurent Carcone qui a gentiment partagé son bureau avec moi. Finalement je remercie les membres de l'administration de relations internationales de l'Université Joseph Fourier, de l'ENSIMAG et de l'Université Polytechnique de Catalogne.

Table des matières

Chapitre 1

Introduction

Depuis quelques années personne doute qu'internet et, plus concrètement, les sites web sont de plus en plus répandus. Il y a une dizaine d'années, les sites web n'étaient conçus que par des entreprises et d'importantes organisations, et réalisés par des experts du domaine. Mais maintenant, peu d'entreprises, pour petites qu'elles soient, n'ont pas de site web. Et les blogs (ou weblogs) sont de plus en plus nombreux, il en existe à présent plus de cinquante millions [23].

En accord avec cette vulgarisation de l'édition du web, le nombre de navigateurs du web croît de façon très impressionnante, avec plus d'un billion d'utilisateurs en 2005 [3].

Avec cette croissance du web et de ses utilisateurs plusieurs problèmes se sont trouvés en face des utilisateurs de tout niveau.

Structuration des documents La difficulté de créer des documents structurés est un des problèmes les plus importants pour la plupart d'utilisateurs. Ce problème est dû principalement à la complexité des documents et au fait que les utilisateurs, en général, ne sont pas conscients des règles à prendre en compte. Ces règles sont normalement exprimées sous la forme d'une DTD. A ces règles très strictes s'ajoutent souvent des règles plus fines, spécifiques au type de document. Par exemple, pour un article de recherche écrit en `XHTML`, en plus de la `DTD HTML` nous pouvons ajouter des contraintes spécifiques aux articles comme l'obligation d'écrire un *abstract*, une série de *keywords*, etc.

Conformité Pour les règles représentées par une DTD, il existe des outils : des validateurs qui vérifient à posteriori que le document est conforme à la spécification de la DTD, ou des éditeurs comme *Amaya* [22] qui guident

l'utilisateur et garantissent que les documents produits sont toujours valides. Si on veut ajouter des contraintes plus fines, on pourrait écrire une DTD ou un schéma XML, mais cela implique la mise en oeuvre de techniques assez lourdes et peu conviviales pour un utilisateur non expert. Une alternative intéressante est d'exploiter le mécanisme des templates. C'est la solution que nous allons explorer.

Sémantique Le manque de sémantique inhérente à des langages comme XHTML est aussi un problème. Une des propositions les plus répandues pour ce problème est le web sémantique. T.Berners [16] dit que l'approche du web sémantique est de développer des langages pour exprimer l'information d'une façon processable par une machine. Les microformats ont aussi pour objectif d'ajouter de la sémantique au web [13], mais plutôt que d'utiliser de nouveaux langages comme T.Berners le propose, les microformats se basent sur les éléments des langages déjà existants et universellement adoptés (XHTML, RSS).

Le mécanisme des templates peut servir de support à la gestion des microformats et donc combler le manque de sémantique inhérent à des langages comme XHTML.

Les templates Quand on demande à un utilisateur standard qu'est-ce qu'un template pour lui, la réponse la plus commune est qu'un template est une ébauche d'un document à compléter par l'utilisateur final pour rendre le travail plus rapide et simple. Par exemple, nous pouvons trouver des templates pour des Curriculum Vitae, pour des Rapports d'activité, pour des factures, etc.

Dans les templates, nous trouvons des parties modifiables à remplir par l'utilisateur et des parties figées. Par exemple, un formulaire peut être rempli mais il y aura des parties qui ne doivent pas être modifiées. Cette utilité est présente dans tous les formulaires comme les fiches d'inscription, les factures, etc.

Pour quelqu'un qui gère des milliers de documents d'un même type, ils est très important que les documents suivent une convention plus ou moins stricte, qui va lui permettre de travailler d'une façon plus efficace. Ces conventions peuvent avoir de l'influence sur le contenu mais aussi sur la forme et la mise en page. Par exemple, nous pouvons vouloir que toute une communauté crée des documents uniformes en style, ou que tous les documents utilisent une façon commune, établie par convention, de décrire des événements, des adresses, etc. Cela nous amène encore aux microformats, qui ne sont que des conventions sur la forme de décrire des concepts.

Nous voyons donc trois utilités principales des templates, à savoir :

- Rendre l'édition de documents plus simple en respectant un ensemble de règles d'organisation.
- Ajouter de la sémantique au langage cible, en utilisant par exemple des templates comme des microformats.
- Uniformiser la forme et le contenu d'un ensemble de documents.

A présent, n'existe aucun outil fournissant les trois fonctionnalités citées ci-dessus, cette affirmation est justifiée avec plus de détail dans la section consacrée à l'étude de l'état de l'art.

Le document est composé des chapitres suivants :

État de l'art Mentionne des solutions déjà proposées pour l'utilisation des templates.

Cas d'utilisation Les cas d'utilisation présentés sont des problèmes typiques qui devront être implémentables par la solution proposée.

Conception du langage Présente le langage `XTiger` en détail, sa spécification concrète et le compare aux langages déjà existants.

Les processus d'édition avec `XTiger` Décrit les mécanismes nécessaires pour l'édition des templates `XTiger` et ses instances.

Implémentation Présente l'implémentation des mécanismes introduits dans le chapitre précédent sur le logiciel Amaya, qui aussi décrit pour mieux comprendre les problèmes rencontrés et les méthodes appliquées.

Conclusion Finalement, le chapitre étudie d'abord quels avantages et inconvénients offre le langage `XTiger` par rapport aux outils déjà existants. Après, il donne quelques voies par où le travail pourrait se poursuivre et finit par un rappel des connaissances principales.

1.1 Objectifs du stage

Les objectifs du stage peuvent être séparés en deux parties principales, la conception du langage et l'implémentation des mécanismes pour l'édition à l'aide des templates sur le logiciel Amaya.

1.1.1 Objectifs pour la conception du langage

Le premier des objectifs du stage est la conception d'un langage qui permettra la création de templates plus puissants que ceux qu'on trouver actuellement et qui permettront une édition plus aisée des instances générées.

Nous pouvons résumer les propriétés que le langage doit présenter en cinq points suivants.

Simplicité

Cette simplicité s'applique aux deux processus où l'utilisateur intervient, c'est-à-dire, l'édition des templates et l'édition des instances. L'objectif est qu'un utilisateur sans connaissance informatique soit capable d'éditer tous les deux après une période de formation la plus courte possible.

Il faut tenir compte que l'utilisateur ne devra jamais s'affronter directement au langage, mais il travaillera avec des outils d'édition qui vont l'abstraire de tous les détails du langage.

Grand pouvoir d'expression

Le langage doit être capable d'exprimer n'importe quelle structure et le nombre le plus grand de contraintes possibles. Quand le choix doit être fait entre la simplicité et le pouvoir d'expression, par le but de base du projet, ce sera la simplicité qui primera.

Généralité

Nous voulons pouvoir créer des templates pour n'importe quel langage *XML*. Cela concerne les langages qui ont une structure définie grâce à une *DTD* un *XML Schema*, etc. mais aussi ceux qui n'en ont pas. Pour ces derniers, le template pourra être vu comme une définition de sa structure.

Validité

Quand le langage cible est défini par une *DTD* ou d'autres langages de Schéma, nous voulons assurer que le document obtenu après édition à l'aide d'un template soit valide par rapport à la structure qui y est définie. En plus, toutes les contraintes données dans le template devront aussi être accomplies par le document final.

Réutilisation de l'existant

En accord avec la fonctionnalité des templates d'uniformisation des documents (et de son contenu) nous voyons que c'est très important de pouvoir utiliser une même structure dans plusieurs templates. Par exemple, la structure qui définit une adresse le plus répandue elle sera, le plus pratique elle sera.

Pour cela, nous devons fournir la possibilité d'utiliser des ressources externes sous forme de librairies partagées. L'idée est de pouvoir utiliser des structures définies par d'autres personnes ou entités.

1.1.2 Objectifs pour la partie implémentation

Le stage étant orienté à un résultat théorique, contient aussi une partie d'implémentation de l'édition d'instances à l'aide d'un template `XTiger`. Cette implémentation permettra, entre autres, de vérifier la faisabilité du processus et de montrer d'une façon très claire de quelle façon elle peut être implémentée.

Le développement consistera à ajouter une fonctionnalité sur l'éditeur-navigateur *Amaya* en utilisant la librairie *Thot*. Tous les deux sont présentés dans le chapitre consacré à la phase d'implémentation.

Chapitre 2

État de l'art

Dans cette section nous présentons les solutions existantes dans le domaine qui nous concerne. Nous allons classer les outils disponibles selon l'approche qu'ils utilisent pour la création des documents à l'aide de templates.

2.1 Les approches

Nous présentons quatre approches différentes :
- Création automatique des instances à partir d'un template et des données
- Création des instances par transformation d'un document
- Création d'instances à l'aide de *wizards*[1].
- Édition directe des instances.

2.1.1 Instanciation d'un template à partir des données

Cette approche est utilisée quand nous voulons créer un ensemble de documents qui ont une partie commune et une partie variable. La partie commune formera le template, et la partie variable sera représentée comme des données complémentaires. Ces données peuvent être sous une forme quelconque, par exemple dans une base de données, un document XML ou un fichier binaire de format indifférent.

[1] Un *wizard*, ou assistant, consiste en une suite de dialogues qui guident l'interaction de l'utilisateur. Dans notre cas, il aide l'utilisateur à guider l'instanciation

Cette approche est très utilisée pour montrer des données stockées dans une base de données au travers d'une application web générant des pages `HTML`. Dans la figure 2.1 nous pouvons en voir un schéma. Les instances sont le résultat de la combinaison du template avec les données propres à chaque cas. En conséquence nous pouvons utiliser n'importe quel langage de programmation pour l'implémenter.

Le template peut être explicite, sous la forme d'un document avec des marqueurs pour l'inclusion des données, ou implicite. Les templates sont implicites quand ils font partie du code d'instanciation.

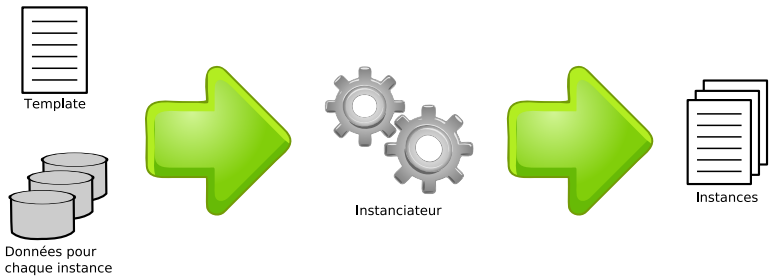

Fig. 2.1 – Instanciation à partir des données

Cette approche est largement utilisée dans des plateformes comme *Microsoft.NET* avec `ASP`[14], *J2EE* avec `JSP`[6], ou encore avec `PHP`[17]. Tous ces outils présentent un document template avec des marqueurs propres pour indiquer les données à inclure.

L'approche est assez voisine de l'utilisation des requêtes aux serveurs web. L'idée principale est d'utiliser CGI[10] sur le serveur web pour exécuter des programmes qui réalisent l'instanciation. Ces programmes peuvent être écrits dans n'importe quel langage de programmation, les plus utilisés étant C, Perl et d'autres langages de script.

2.1.2 Transformation d'un document vers un autre

Nous identifions une approche consistent à transformer les documents vers des documents finales plus uniformes. Cette approche est assez proche de la précédente, mais ici les données d'entrée sont déjà des documents. Ces documents sont transformés d'après un ensemble de règles qui sont donnés dans une feuille de transformation (ou style).

Nous pouvons voir cette approche comme un cas particulier du précédent ou le template ne montre pas une structure explicite mais des règles de transformation.

Cette approche, dont le schéma est montré dans la figure 2.2, est utilisée entre autres par XSL[11] et CSS[24].

`XSL` nous permet de générer un nouveau document à partir d'un document d'origine en `XML`. Le document d'origine est un document `XML` quelconque auquel nous appliquons un ensemble de transformations exprimées en `XSL` pour obtenir un nouveau document. Ce document résultat n'est pas forcement `XML`, même si cela est le cas en général.

Avec `CSS` nous ne créons pas vraiment un nouveau document, mais nous le décorons pour pouvoir l'afficher d'une façon déterminée par les règles de styles présentes dans la feuille de style `CSS`.

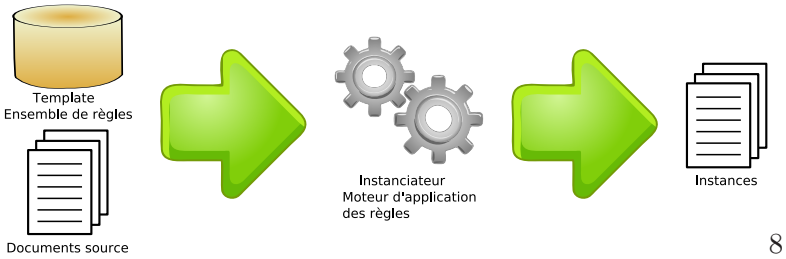

FIG. 2.2 – Instanciation à partir d'un document

2.1.3 Instanciation à l'aide de *wizards*

Dans les deux approches précédentes le processus d'instanciation n'est pas interactif. L'approche d'instanciation de templates à l'aide de *wizards* ajoute la possibilité d'interagir avec l'utilisateur. Ces interactions ont comme but principal l'obtention des données à inclure dans le résultat. Dans les approches précédentes, elles seraient stockées sous forme de fichier ou d'autre support accessible au moment de l'instanciation.

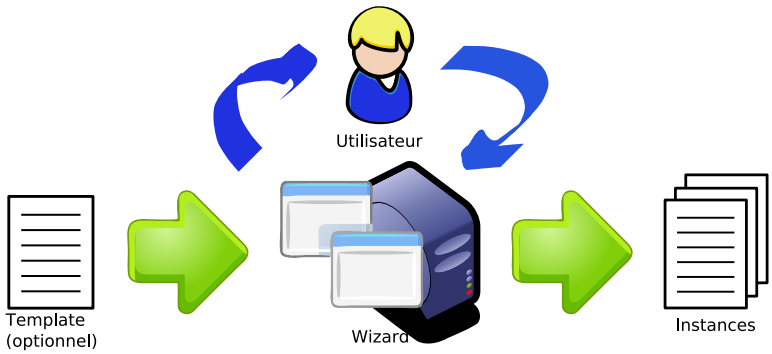

FIG. 2.3 – Instanciation à l'aide de *wizards*

Cette approche, illustrée dans la figure 2.3, est utilisée dans des applications très variées pour faciliter la création d'objets. Par exemple, nous trouvons des *wizards* dans la suite *Microsoft Office*[20][25][8] et aussi dans *OpenOffice*[2], pour la création de documents, de tables, des bases de données complètes, etc. La figure 2.4 montre un wizard pour la création d'étiquettes de présentation.

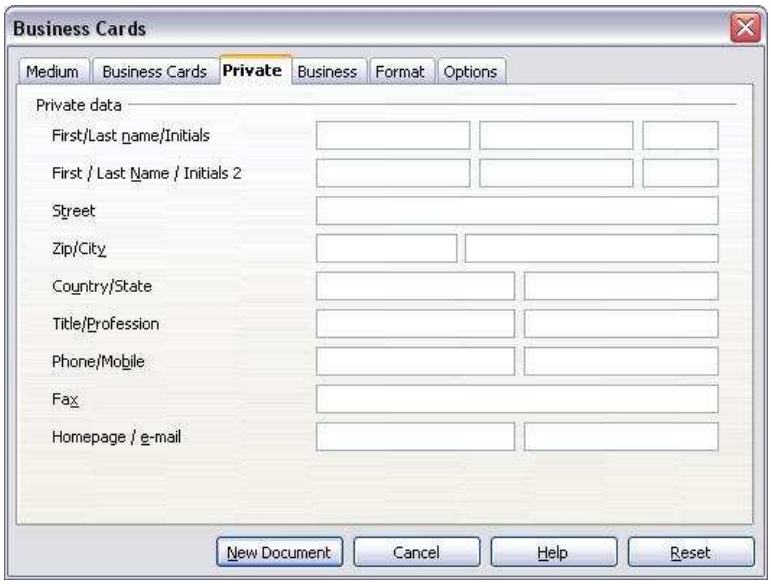

FIG. 2.4 – Un wizard pour la création d'étiquettes de présentation en *OpenOffice*.

Comme dans le cas présenté dans 2.1.3, kes *wizards* peuvent utiliser des documents templates pour générer les documents finals ou créer de toute pièce le document. Quand il existe un document template que l'on complète avec les données obtenues, nous disons que le template est explicite, car il préexiste sous forme de document. Par contre, quand tout le processus de création de l'instance est codé dans le programme du wizard, nous disons que le template est implicite.

2.1.4 Édition directe des instances

Finalement, c'est l'approche la plus simple pour l'utilisateur et, en même temps, la plus complexe à implémenter. Cette approche, schématisée dans la figure 2.5, est celle qui donne le plus de liberté à l'utilisateur. L'édition commence sur une instance créée à partir d'un template par le processus de l'instanciation. Cette instance est un document qui n'est pas forcement complet. L'utilisateur édite alors cette ébauche jusqu'à obtenir le document

voulu.

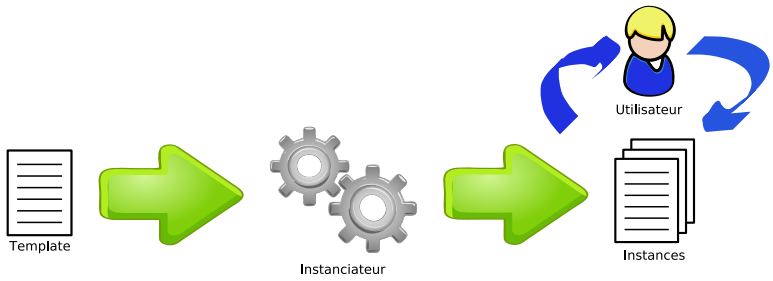

FIG. 2.5 – Instanciation et édition des instances

La problématique de cette approche est la liberté de l'utilisateur qui peut conduire à des documents résultants invalides ou non conformes au template correspondant.

Pour éviter cela, l'édition doit être contrôlée. Et c'est précisément ce contrôle de l'utilisateur qui, en s'opposant à la liberté nécessaire pour permettre n'importe quel document conforme au template d'origine, fait l'intérêt de cette approche dans notre perspective.

Nous pouvons trouver dans le marché différents outils qui utilisent cette approche avec des rapports contrôle/liberté très différents. Dans la liste suivante, nous donnons quelques exemples de ces rapports contrôle/liberté.

Liberté totale
: *Microsoft Word*[20] et d'autres outils similaires comme *OpenOffice.org Writer* fournissent la création de documents à partir de templates. Ces outils ont déjà été présentés car ils offrent aussi la possibilité d'utiliser des wizards. Une fois l'instance initiale créée l'utilisateur peut tout faire, même effacer complètement le contenu du document et recommencer de zéro. Ces applications n'assurent donc aucune propriété sur les instances des templates.

Liberté totale ou nulle
: *Macromedia Dreamweaver*[18] permet un contrôle de l'édition de l'utilisateur limité à deux concepts :

  – les structures répétables, où l'utilisateur peut répéter le contenu autant de fois qu'il le veut (zéro fois compris), et
  – les zones libres, qui ne posent aucune restriction sur l'édition.

2.2 Langages de validation XML

Les structures des documents XML sont normalement décrites à l'aide d'un langage de schéma ou de validation XML, comme XML Schema[28], DTD[11] ou RelaxNG[29]. Ces langages ont été conçus pour la validation des documents. Ils décrivent un sous-langage de XML et leur utilité et de pouvoir vérifier qu'un document donné appartenant au langage décrit est valide ou non. Lee et Chu [15] font une étude comparative de six langages de schéma qui présentent de façon très claire et formatrice les propriétés standard des langages de schéma.

Valider un document et le créer sont deux processus très différents mais qui, en même temps, partagent quelques aspects. Notamment, tous les deux sont liés à la grammaire d'un langage.

Mais la validation ne fait que vérifier si un document appartient à un langage, alors que la création d'un document se base sur une description de la structure pour générer des documents qui appartiennent au langage.

Le rapport entre les langages de validation et `XTiger` se compose de deux points.

- Les règles définies dans le schéma d'un langage doivent être prises en compte par les outils d'éditions des instances des templates `XTiger`.
- Un template `XTiger` peut éviter l'utilisation de plusieurs schémas pour un même document (pour l'utilisation de microformats, par exemple).

Finalement, les schémas sont des objets séparés des documents finals, alors que `XTiger` se mêle aux instances pour gérer sa structure plus intuitivement.

2.3 Résumé

Nous avons présenté diverses approches pour l'utilisation de templates avec quelques exemples d'applications qui les utilisent. Maintenant nous pouvons exprimer plus clairement l'objectif de notre travail. Nous voulons concevoir un langage qui permettra d'implémenter une fonctionnalité de templates dans *Amaya* basée sur l'approche présentée en section 2.1.4.

La partie innovatrice du langage consiste en deux aspects :

- Un contrôle plus fin de l'édition des instances, passant du "éditable/non éditable" à un ensemble de structures plus puissantes basées principalement sur les idées des langages de validation vues dans 2.2.
- Un enrichissement sémantique de l'édition XML par addition de composants à un niveau d'abstraction plus élevé. De la même façon que

dans les langages de validation nous utilisons des éléments de plus haut niveau (par exemple, une figure peut être vue comme une image, un objet ou encore comme du texte plat). Nous utiliserons ces idées non pas pour l'analyse des documents mais pour sa construction.

Chapitre 3

Cas d'utilisation

Cette section présente quelques cas d'utilisations typiques des templates en général. Elle nous permet de mieux introduire le domaine et de mieux cibler la solution souhaitée.

3.1 Templates pour structurer les instances

Une des utilisations des templates est de limiter les possibilités d'édition des instances pour uniformiser l'ensemble avec une structure plus ou moins figée. Cette section présente quelques cas d'utilisation de templates dans ce but.

3.1.1 Formulaires

Les formulaires sont très un bon exemple de template car la différence entre le template original et ses instances est très claire. Le template étant le formulaire vide, une instance le formulaire rempli.

Pour remplir un formulaire nous pouvons exprimer plusieurs contraintes. Le premier type de contrainte est celui qui décrit les zones où l'utilisateur peut écrire, c'est à dire, sont les champs à remplir. En outre, nous pouvons contraindre les valeurs possibles dans un champ à un domaine particulier.

La figure 3.1 montre un formulaire d'inscription. Ce formulaire contient des champs verts à remplir, et il est interdit d'éditer quoi que ce soit en dehors de ces champs. De plus, nous voulons que dans le champ *date de naissance* il y est bien une date, dans le champ *téléphone* une chaîne numérique

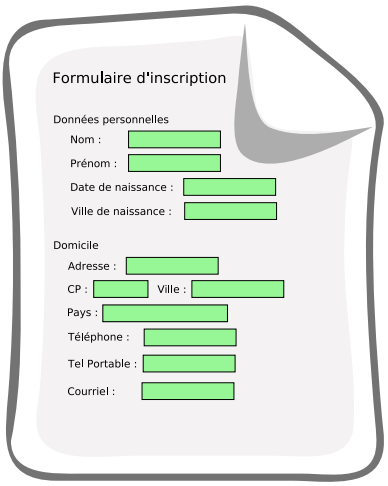

FIG. 3.1 – Le formulaire visualisé par l'application d'édition d'instances

qui corresponde à un format de numéro téléphonique et dans le champ *courriel* une adresse de courrier électronique syntaxiquement valide.

3.1.2 Formulaires avec répétitions

On peut trouver dans de nombreux formulaires des champs qui se répètent plusieurs fois. Normalement il s'agit de ne compléter qu'une partie des répétitions et de laisser en blanc celles dont on n'a pas besoin. On trouve ce cas par exemple dans un formulaire de type CV, où nous pouvons introduire plusieurs diplômes ou expériences professionnelles.

Ce cas doit être donc aussi traité par `XTiger`. Plus concrètement, dans une répétition d'éléments nous pouvons vouloir limiter le nombre d'éléments minimum et maximum. Par exemple, dans un formulaire comme celui de la figure 3.1 nous pourrions permettre autant de numéros de téléphone que l'utilisateur veut nous fournir, mais en exiger au moins un.

De la même façon que pour les répétitions, nous pouvons définir des sections qui sont optionnelles. Ces parties sont définies de façon identique à une répétition de 0 à 1 éléments. Par exemple, dans un CV, nous pouvons ajouter une section pour les références professionnelles qui peut être optionnelle. En conséquence, il y aura des instances qui auront la section de références et d'autres qui ne l'auront pas.

3.1.3 Formulaires où plusieurs types de réponse sont possibles

Un formulaire comme celui montré dans la figure 3.1 devrait accepter pour le champ date plusieurs formats possible date, par exemple "20/05/06", "20-5-2006", "20 mai 2006", ou encore "vingt mai, deux mille six".

Ce type d'options doit être modélisable en `XTiger`.

3.2 Templates pour contraindre le contenu

Dans les cas précédents, les possibilités d'édition des instances sont très limitées. Cela peut s'avérer très pratique pour quelques applications mais se révéler beaucoup trop contraignant pour d'autres cas. Cette section présente des cas d'utilisation qui modélisent des templates beaucoup moins restrictifs que les précédents. La philosophie est différente, si dans la section précédente nous donnons ce que nous autorisions, maintenant nous donnerons ce qui ne peut pas être utilisé.

3.2.1 Un article de recherche

Les articles de recherche suivent une structure traditionnellement établie qui est, à son niveau le plus haut, assez stricte. Comme montré dans la figure 3.2, un article a normalement un titre, une liste d'auteurs, un paragraphe d'*abstract*, le corps de l'article et une section de bibliographie.

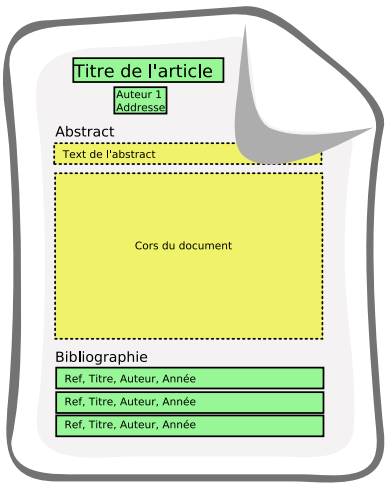

Fig. 3.2 – Un article de recherche et ses composants principaux

Nous pouvons considérer que le titre, les auteurs et la bibliographie peuvent être modélisés avec les outils précédents. Mais comment pouvons-nous modéliser le contenu du corps ou celui de l'*abstract* ? Ces zones montrées en jaune et entourées en pointillé sont d'une nature très différent de celles que l'on a trouvées jusqu'à présent.

Dans ces zones, aucune structure commune n'est exigée. L'éditeur doit pouvoir agir comme il le ferait s'il n'y avait pas de template.

3.2.2 Du texte sans images

Dans certains cas, nous voulons permettre une structure quelconque mais en ajoutant des contraintes. Notamment, nous pouvons restreindre les types d'éléments qui peuvent être utilisés dans une zone. Par exemple, dans un article de journal nous pouvons interdire l'insertion d'images dans le corps du document. Ce type de contrainte doit aussi être géré par notre langage.

3.3 Templates pour structurer les données

Une autre fonction des templates `XTiger` est d'uniformiser le format des données introduites en utilisant des éléments réutilisables. Cette section présente quelques cas d'utilisation qui exploitent cette fonctionnalité des templates.

3.3.1 Un livre de programmation

Nous pouvons penser que la création d'un template de livre de programmation dont nous n'allons créer qu'une seule instance n'est pas pertinente. En effet, définir la structure complète du template peut s'avérer aussi coûteux que la création de l'instance-elle-même.

Néanmoins, nous pouvons créer des templates très simples au niveau structurel. Par exemple, nous pouvons seulement définir la structure des chapitres, sections et parties, et enrichir le template avec des composants pour les objets les plus utilisés. Dans le cas du livre de programmation, comme montré dans la figure 3.3, nous pouvons créer des composants pour les exemples, les remarques, etc.

Ces composants nous permettront une édition plus aisée du document en assurant que tous les exemples et remarques suivront une structure uniforme et que les applications qui dépendront d'eux n'auront pas de problème

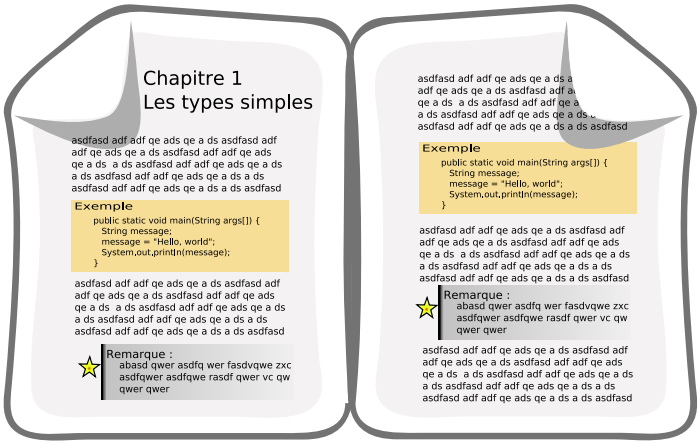

FIG. 3.3 – Un livre de programmation avec exemples et remarques

d'incompatibilité. Notamment, nous pourrons créer des feuilles de style CSS et des scripts web pour un affichage plus dynamique.

3.3.2 Les microformats

Les microformats peuvent être vus comme une généralisation des composants montrés jusqu'à présent. Il s'agit de spécifications d'un format commun pour la description de données d'utilisation très répandue. Par exemple, nous utiliserons un microformats pour les dates (hCalendar), pour les adresses (hCard), pour les liens vers d'autres pages web (XFN), etc. Nous utiliserons aussi les microformats basiques pour créer des microformats décrivant des éléments de plus grande taille, voire des documents complets, comme des présentation de slides (Slidy) ou des CV (hResume).

Pour son utilisation dans `XTiger`, ils est essentiel que les définitions puissent être réutilisées par plusieurs documents sans avoir besoin de redéfinir quoi que ce soit. Le langage doit donc fournir les fonctionnalités nécessaires pour la création des librairies d'éléments et son utilisation par les différents templates développés avec.

hCard

Nous voudrions pour ce microformat proposer plusieurs composant respectant sa spécification. L'idée est donc de permettre à l'utilisateur d'insérer une de ces cartes de présentation et de le guider pendant l'introduction des données.

XFN

Ce microformat, comme la plupart d'entre eux, repose sur l'utilisation des attributs. `XFN` définit une liste possible de valeurs pour l'attribut `rel` de l'élément `a` de `HTML`. Pour pouvoir modéliser ce microformat nous pouvons créer autant de composants que de combinaisons possibles de valeurs. Mais cette solution devient très vite fastidieuse autant pour l'utilisateur que pour le créateur des templates.

Nous voulons donc que `XTiger` permette une gestion plus puissante des attributs, tel que le nombre de valeurs admis (est-ce que c'est un token ou une liste), les valeurs possibles, par défaut, etc.

Chapitre 4

Conception du langage `XTiger`

Ce chapitre présente le langage `XTiger` d'une façon très générale, en justifiant tous les choix de conception et de design qui ont amené à la grammaire XML présentée dans le chapitre suivant.

Pour commencer nous énumérons les propriétés visées par notre langage. Après, dans la section 4.2 nous donnons une description générale du langage.

4.1 Les propriétés

Comme nous avions déjà présenté dans le chapitre d'introduction, le langage `XTiger` est axé sur deux applications principales, à savoir

- l'édition structurée de documents avec des contraintes ajoutées sur le langage cible, et
- l'ajout de sémantique au langage cible.

Les propriétés que nous avons en tête pour le design du langage sont principalement la simplicité d'utilisation, la validité des résultats et la réutilisation.

4.1.1 Simplicité

Pour la simplicité d'utilisation nous ne visons pas que le langage en lui-même, mais aussi la façon dont les utilisateurs vont finalement utiliser le langage. Plus concrètement, nous considérons indispensable l'utilisation d'un outil pour la création de nouveaux templates et l'édition des instances. Nous pourrons envisager en fait d'utiliser l'outil d'édition des instances pour

créer des nouveaux templates à l'aide d'un méta-template. Ces outils doivent cacher la syntaxe du langage à l'utilisateur final de la même façon qu'un outil graphique cache la syntaxe de représentation du graphique comme SVG.

4.1.2 Généricité

Nous ne faisons aucune hypothèse sur le langage cible. Nous pouvons créer des templates pour n'importe quel langage XML, qu'il ait une DTD qui le définit formellement (comme c'est le cas pour XHTML, SVG ou encore MathML) ou que ce soit un langage sans DTD et qui justement profitera des templates pour définir les contraintes du langage sur les documents produits.

En outre, toute structure doit pouvoir être décrite à l'aide de XTiger. De la même forme, nous voulons que le langage soit capable d'exprimer le plus grand nombre de contraintes possibles. Quand la puissance du langage rentre en opposition avec sa simplicité, nous valorisons la simplicité comme la propriété plus importante même si parfois des compromis entre les deux seront faits.

4.1.3 Validité

Les instances et les templates XTiger sont des documents bien formés. La validité des instances est achevée en enlevant tous les éléments XTiger.

La validité des instances est définie par rapport au langage cible et sa définition sous forme de DTD mais aussi par rapport aux règles introduites par le template utilisé. Cette validité est assurée par quelques uns des processus qui composent l'édition des documents à l'aide des templates montrés dans la figure 4.1.

La validité doit être vérifiée pour les templates et pour les instances. Et la validité des templates est vérifiée

à l'édition des templates. Nous vérifions que les templates suivent bien la syntaxe de XTiger.

à l'instanciation. Les templates sont revérifiés au moment de l'instanciation car nous considérons que les templates peuvent aussi être modifiés à la main ou par des applications qui ne sont pas tout à fait correctes par rapport à la syntaxe XTiger.

La validité des instances est vérifiée

statiquement après l'édition d'un template. Cette vérification assure que toutes les structures définies par le template correspondent bien à une structure valide du langage cible. Par exemple, ici nous pouvons

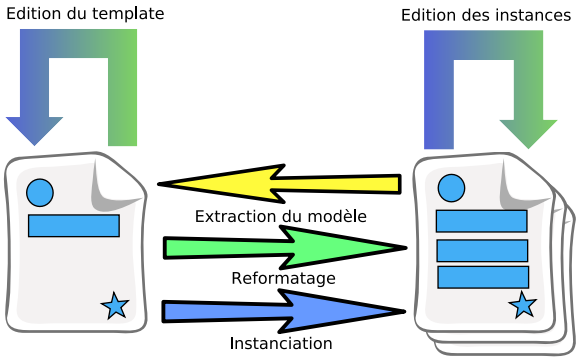

FIG. 4.1 – Les processus de l'édition de documents à l'aide des templates.

vérifier que le contenu d'un élément `ul` de `XHTML` est une série non vide d'éléments `li`.

dynamiquement pendant l'édition des instances. L'application d'édition d'instances (*Amaya* dans notre cas) ne doit fournir à l'utilisateur que les options qui donnent comme résultat un document valide. Par exemple, si l'utilisateur est en train d'éditer le contenu d'un paragraphe `p` l'application doit s'assurer de ne fournir aucun composant contenant des paragraphes `p`, car la `DTD` de `XHTML` interdit cela.

En plus la validité doit être conservée par les processus de reformatage. Le reformatage peut être vu comme une transformation d'un document vers un autre suite à une évolution entre deux versions de templates. C'est donc cette transformation (qui pourrait être présentée sous forme de feuille de transformation *XSL*) qui doit pour toute instance valide du template précédent créer une instance valide du nouveau template.

Finalement, pour l'extraction d'un template à partir des instances, il est préférable que les instances soient des documents valides, mais nous pouvons envisager la création de templates qui ignoreraient ou répareraient les possibles erreurs dans les documents.

4.1.4 *(Ré)*utilisabilité

Un des aspects les plus importants pour que le langage soit utilisable comme modèle pour les microformats est que les structures (voire les formats) puissent être définies une seule fois et utilisées autant de fois qu'on en a besoin. C'est pour ça, et parce que la reutilisabilité est de plus en plus une partie très important de tout outil informatique, que le langage `XTiger` fournit le support pour le partage de structures définies entre documents.

Les librairies sont des documents qui définissent des structures partageables. Les librairies et leur utilisation sont présentées dans la section 5.2.

Le processus de reformatage présenté dans la section précédente est aussi un moyen de réutiliser l'existant. Dans ce cas-là, les nouvelles instances réutilisent les instances du template avant changement.

Le dernier aspect de la réutilisation est la combinaison que nous faisons des règles induites par la DTD du langage cible avec celles définies par le template. Cela nous permet de ne pas avoir besoin d'exprimer les contraintes déjà introduites dans la DTD XHTML quand nous créons un template pour un type de document en XHTML. Toutes les contraintes seront utilisées ensemble au moment de l'édition des instances.

4.2 Description du langage XTiger

Cette section présente les grands traits du langage XTiger. Dans la première partie nous justifions les choix faits dans la conception générale du langage pour après présenter les conséquences que ces choix provoquent.

XTiger est un langage XML très simple. En fait, il n'est formé que de huit éléments, et ces éléments ont un nombre très limité d'attributs, voire aucun pour certains.

4.2.1 Forme du langage

Le premier choix à faire pour la conception du langage était la forme que le langage devait prendre. Nous avons travaillé avec trois options :

- l'utilisation de commentaires XML.
- l'utilisation de l'attribut class de HTML.
- l'utilisation d'attributs d'un espace de noms spécial dans les éléments du langage cible.
- l'utilisation d'un langage XML dont les éléments se mêleraient au langage cible dans les templates et les instances.

La première solution, l'utilisation de commentaires XML n'est pas adapté à la validité que l'on souhaite pouvoir vérifier, et d'autre part le but des commentaires XML n'a pas de rôle sémantique sinon que de documentation du code source. La solution est donc rejetée même si elle est utilisée par d'autres applications comme *Dreamweaver*.

L'utilisation de l'attribut class impose une contrainte très forte, il faut que tous les éléments du langage cible aient cet attribut-là. Cela veut dire que

nous restreignons l'utilisation de XTiger au langage XHTML. Cette solution s'oppose donc à la généricité du langage et elle est donc rejetée.

Les deux dernières solutions se basent sur la définition d'un nouveau langage et un nouvel espace de noms[27]. Nous décidons d'utiliser un nouveau langage avec un nouvel espace de noms car cela nous permet de maintenir la validité des instances vis à vis du langage cible tout en conservant la possibilité de vérifier la conformité XML du template, chose impossible si nous utilisions des commentaires.

Le choix est donc entre l'utilisation d'attributs d'un espace de noms spécial appliqués aux éléments du langage cible, et l'utilisation d'éléments de l'espace de noms insérés parmi les éléments du langage cible.

L'utilisation d'un langage composé seulement d'attributs présente des avantages vis à vis des applications travaillant sur le langage cible, notamment si elles ne savent pas sauter les éléments des espaces de noms inconnus. Malheureusement, limiter le langage à un ensemble d'attributs réduit la puissance du langage et rend sa validation très compliquée. Notamment, la validation n'est plus possible à l'aide d'une DTD ou d'un Schéma XML. En fait, ce problème est équivalent à celui de la validation des microformats, présenté récemment par Eric van der Vlist [30] et Norman Walsh[31].

Nous optons donc pour une solution à base d'un langage avec des éléments dans un nouveau espace de noms. Le mélange d'espaces de noms dans un même document produit des documents que l'on appelle composites. Les documents composites sont de plus en plus utilisés pour combiner les langages XHTML, SVG et MathML.

En plus d'une validation plus simple, la possibilité d'introduire des éléments et leurs attributs augmente grandement le pouvoir d'expression du langage.

4.2.2 Structure générale du langage

Les templates XTiger sont en fait des documents *presque valides* du langage cible. Par *presque valide* nous exprimons le fait que certaines contraintes peuvent ne pas être remplies, comme par exemple les attributs obligatoires peuvent ne pas avoir une valeur dans le template. Ces documents presque valides assurent par contre que les instances crées à partir d'eux seront des documents valides du langage (par rapport au schéma de validation du langage cible et aux restrictions exprimées dans le template).

Un template inclut deux catégories d'éléments du langage XTiger, l'entête et le corps.

L'entête du template, qui est d'utilisation optionnelle, est définie par l'élément `t:head`, présenté dans 5.2.1. L'entête contient des déclarations de types qui pourront être utilisés dans la suite du template.

Les éléments de corps du template définissent les contraintes sur la structure des instances et utilisent les types qui ont été définis par l'entête. Par défaut, le contenu du template ne sera pas éditable dans les instances. C'est grâce aux éléments `XTiger` du corps que nous allons permettre l'édition de certaines zones ou de certains éléments.

Nous pouvons voir qu'un template qui n'a pas d'entête et dont le corps ne contient pas d'éléments du langage `XTiger` est en fait un document valide du langage cible.

4.2.3 Réutilisation des schémas existants

Les templates rajoutent des restrictions sur la structure et le contenu de ses instances, mais il est très important de conserver les contraintes définies par le schéma de validation du langage (DTD, XML Schéma, etc.).

Ces contraintes héritées seront utilisées au moment de la validation du template mais aussi par l'outil d'édition d'instances, qui ne devra proposer à l'utilisateur que les options valides à chaque instant. La réutilisation de ces contraintes est un thème récurrent pendant tout le travail réalisé et il est aussi présent dans nombreux points de ce rapport.

Chapitre 5

Spécification concrète

Ce chapitre présente les éléments du langage en se basant sur un des besoins exprimés dans le chapitre correspondant aux cas d'utilisation. Ces besoins sont :

- la création de zones de structure libres et le contrôle des éléments qui peuvent y apparaître,
- la création de champs à remplir et le contrôle des types du contenu,
- la gestion des zones répétables et optionnelles,
- la possibilité de faire un choix entre plusieurs types pour un champ déterminé
- et la gestion des attributs.

Pour cela, nous commençons par présenter une hiérarchie de types dans laquelle nous pouvons répertorier tous les types que nous trouvons dans les templates. Postérieurement, nous présentons les éléments utilisés pour chacun des besoins de la liste ci-dessus en justifiant tous les choix faits pendant sa conception.

5.1 L'univers de types de `XTiger`

Tous les éléments que nous avons trouvés pendant l'analyse du domaine peuvent être classés dans quelques catégories. Ces catégories sont

- les valeurs atomiques (ou indivisibles),
- les éléments du langage cible (comme l'élément `h1` de `XHTML`) et
- les structures réutilisables ou composants.

5.1.1 Les types simples

Les *types simples* sont ceux auxquels les valeurs atomiques appartiennent. Ces types sont inspirés des types simples définis dans des langages de programmation (C, Java, etc) mais aussi des types définis par les langages de validation XML (`RelaxNG` et `XML Schéma`).

Nous avons décidé de fournir seulement trois types prédéfinis, qui sont

number pour les nombres entiers ou réels,

boolean pour les valeurs booléennes (vrai et faux) et

string pour les chaînes de caractères.

A ces types, l'utilisateur pourrait vouloir ajouter des types définis pour ses besoins. Cela nécessite un mécanisme de définition de nouveaux types qui n'est pas disponible pour le moment. C'est une évolution possible comme le propose le système de déclaration de nouveaux types de `XML Schéma` et `RelaxNG`.

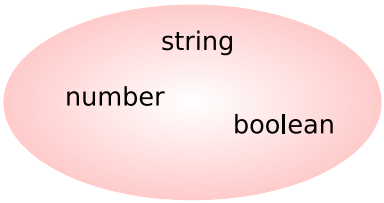

FIG. 5.1 – Types simples

5.1.2 Les éléments du langage cible

Tous les templates définis à l'aide du langage XTiger portent sur des documents XML. Ce document, qui peut utiliser un ou plusieurs langages XML (c'est le cas des documents composites), est une structure d'arbre où les nœuds sont du texte, des attributs ou des éléments du langage cible. Ces éléments du langage cible, qui peuvent apparaître dans les instances mais aussi dans les templates, sont classés dans la catégorie *éléments XML*.

Comme il est montré dans la figure 5.1.2, deux langages différents (ex. `XHTML` et `SVG`) peuvent potentiellement avoir deux éléments avec le même nom (ex. `a`). Deux éléments avec même nom peuvent donc avoir une sémantique tout à fait différente, et il nous faut clairement distinguer les uns des autres. Pour cela, il existe les espaces de noms `XML` décrits dans [27]. Tous les éléments seront donc identifiés par leur espaces de noms qui sera utilisé en cas d'ambiguïté.

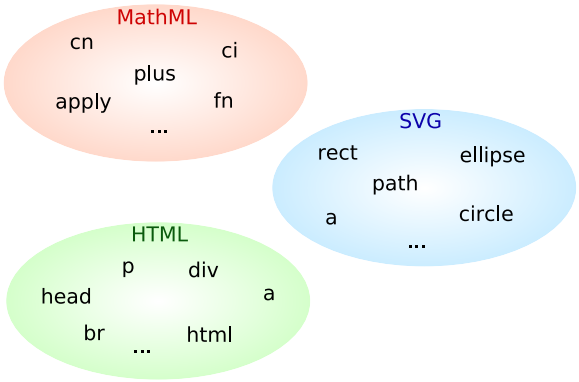

FIG. 5.2 – Éléments XML

Les types d'éléments XML sont définis soit par le schéma ou la DTD auquel le template fait référence, soit à partir du contexte de sa première utilisation dans le template. Au moment de la première utilisation nous définissons son espace de noms comme étant le même que celui de son ascendant le plus proche n'appartenant pas à XTiger.

5.1.3 Les composants

Avant, nous avons montré le besoin de réutiliser des sous-structures que nous avons appelées composants. Ces composants forment aussi une catégorie différente à celles des types simples et des éléments XML que nous appellerons la catégorie des *composants*.

L'élément component

L'élément t:component nous permet de définir un bout de template qui pourra être utilisé ailleurs. Un composant est composé par un nom qui doit être unique et une structure qui le caractérise.

Le nom du composant est défini par l'attribut name du t:component et la structure correspond au contenu de l'élément.

Par exemple, nous pouvons créer un composant très simple avec le code de la figure 5.1.3. Ce composant est identifié par la chaîne HelloWorld et sa structure est un élément p du langage XHTML avec le contenu textuel *Hello world !*.

Cet élément peut se trouver dans une section de déclaration mais peut aussi faire partie du corps du template. Dans ce dernier cas, nous disons que

```
<t:component name="HelloWorld">
  <p>Hello world!</p>
</t:component>
```

FIG. 5.3 – Un composant très simple

c'est une déclaration *inline* du composant. La sémantique des déclarations *inline* est celle d'une utilisation du composant à l'endroit où se trouvait la déclaration.

5.1.4 Les unions

Très souvent, nous avons besoin d'utiliser un ensemble de types à plusieurs endroits. Par exemple, nous pouvons permettre tous les éléments de XHTML à en endroit, ou bien seulement un des modules définis par la modularisation de XHTML [4]. En plus de ces modules nous pouvons définir des liste avec des types appartenant à des catégories différentes. Ces listes forment aussi une catégorie appelée catégorie des *unions*.

Les unions sont alors des ensembles de types, et chacun de ces types peut être soit un type simple, soit un élément `XML`, soit un composant, soit une union aussi. Le mécanisme choisit pour la création de nouvelles unions est la différence d'ensembles. Ce mécanisme nous permet d'exprimer d'une façon très claire et compacte les unions les plus communes, en conservant la possibilité de définir des unions plus compliquées en appliquant plusieurs fois le même mécanisme.

Pour rendre plus facile la création de nouvelles unions, `XTiger` fournit quatre unions prédéfinies. Ces unions sont les suivantes :

anySimple comprend tout les types simples, en absence d'un mécanisme de création de nouveaux types simples, cette union contient uniquement `number`, `string` et `boolean`.

anyElement comprend tous les éléments XML qui ont était utilisés par le template ou qui sont définis par un schéma ou DTD auquel le template fait référence.

anyComponent comprend tous les composants.

any comprend absolument tous les types, ou de façon équivalente, est l'union des trois unions précédentes.

Notons qu'il n'y a pas d'union `anyUnion`. Cela est tout simplement parce qu'elle serait équivalent à `any`.

L'élément `union`

L'élément `t:union` permet la création de nouvelles unions. Ces unions sont aussi identifiées par un nom et définissent un ensemble de types. Pour le calcul de l'ensemble de types, nous utilisons les attributs `include` et `exclude` qui donnent la liste d'éléments à ajouter et à exclure de l'ensemble résultat. Les deux attributs contiennent une liste de types de n'importe quelle catégorie.

Dans le code de la figure 5.1.4 nous définissons une union pour le module Heading de `XHTML` 1.1 [4]. La nouvelle union appelée *Heading*, contient les éléments `XHTML` `h1`, `h2`, `h3`, `h4`, `h5` et `h6`.

```
<t:union name="Heading"
         include="h1 h2 h3 h4 h5 h6"/>
```

FIG. 5.4 – Une union pour le module Heading de XHTML.

De la même façon que nous avons crée l'union précédente, nous pouvons créer une union pour tous les éléments `XHTML`. Supposons que cette union s'appelle *Html*, alors nous pouvons définir l'union de tous les éléments `HTML` mais sans images avec le code de la figure 5.1.4.

```
<t:union name="HtmlNoImages"
         include="Html"
         exclude="img"/>
```

FIG. 5.5 – Une union pour code html sans image.

5.2 Les entêtes et les librairies

Comme nous avons déjà dit, les templates XTiger peuvent avoir un entête où ils déclarent des types à l'aide des éléments montrés précédemment (`t:union` et `t:component`). Ces types sont alors utilisables par les éléments qui apparaîtront dans le corps du document.

Pour ce type de déclaration, nous utiliserons l'élément `t:head`. Les déclarations faites dans un template ne sont utilisables que par le template lui-même. C'est pour cela que nous ajoutons le concept de librairie. Les librairies sont des documents qui ne contiennent que des déclarations XTiger, c'est l'équivalent d'un élément `t:head`. La seule différence avec le `t:head` d'un

template est que l'élément qui englobe toutes les déclarations dans les librairies est un élément `t:library`.

5.2.1 L'élément `head`

L'élément `t:head` contient une liste de déclarations. Ces déclarations peuvent être des éléments `t:union`, `t:component` ou des liens vers des librairies utilisées par le template. Pour importer une librairie nous utilisons l'élément `t:import`.

L'élément `t:head`, s'il existe, il est toujours fils d'un élément du langage cible (c'est-à-dire qu'il n'est pas la racine du document) et il n'y a jamais un autre élément du langage XTiger avant lui, ou en utilisant la syntaxe `XPath` [7], il y a aucun élément de l'espace de noms de `XTiger` dans l'axe *preceding* de l'élément `t:head`. Cela implique aussi qu'il est unique.

Cet élément n'a pas d'attributs.

5.2.2 L'élément `import`

L'élément import indique l'importation d'une librairie. Le seul attribut qu'il a est une *url* pour l'adresse de la librairie à importer, appelé `src`. L'élément est toujours vide.

5.2.3 L'élément `library`

Les librairies sont des documents XML dont la racine est l'élément `t:library`. Cet élément est identique à l'élément `t:head`, il n'a pas d'attributs et ses fils sont des déclarations. Une librairie peut aussi contenir des `t:import` vers d'autres librairies.

Les librairies nous permettent de partager les déclarations des types les plus utilisés de façon très simple. Cela est une fonctionnalité vitale pour pouvoir travailler avec des microformats.

5.3 Le corps du template

Le corps d'un template peut contenir tous les éléments qui ne font pas partie de l'entête. Il est destiné à générer des instances grâce au processus d'instanciation, expliqué dans la section 6.2.4.

Le corps du template est formé par
– des éléments du (des) langage(s) cible(s),
– des éléments XTiger décrivant la structure des instances et
– des `t:component`.

Cette section présente les types d'éléments `XTiger` avec lesquels nous pouvons décrire la structure des instances.

5.3.1 Les champs

Les champs de formulaire sont des zones complètement éditables, qui peuvent avoir une valeur par défaut et qui peuvent restreindre leur contenu en fixant un type de données.

Normalement ces types de données sont des **types simples**, car les données d'un formulaire peuvent être classées comme des chaîne de caractères, des nombres, etc. Mais cela est un cas particulier, car nous pouvons envisager l'utilisation d'un microformat pour les dates. Les dates seront alors définies par un `component` et non un **type simple**.

Dans le cas des composants, le contenu du champ ne sera éditable que dans les éléments de la sous-structure qui définissent un champ éditable.

Pour décrire ce type d'utilisation nous définissons l'élément `t:use`.

L'élément `use`

L'élément `t:use` nous permet l'utilisation ponctuelle d'un type. En suivant l'exemple du formulaire, nous pouvons définir les dates comme des chaînes de caractères (fig. 5.3.1) ou comme un composant qui définit le format avec plus de détail avec un jour, un mois et un an (fig. 5.3.1). Notons que le contenu du `t:use` est traité comme le contenu par défaut du champ, et qu'il peut donc être modifié.

```
<t:use id="birthdate" types="string">6 juin 1983</t:use>
```

Fig. 5.6 – Un champ de formulaire pour la date de naissance en utilisant une chaîne de caractères dont la valeur par défaut est "6 juin 1983".

Remarquons que si nous utilisons le type *string* nous accepterons n'importe quelle valeur, alors qu'avec le type *date* seulement les chaînes du type $x/y/z$ où x, y et z sont des nombres.

```
<!--Définition d'un composant pour les dates -->
<t:component name="date" title="A date with dd/mm/yyyy format">
  <t:use id="day" types="number">6</t:use> /
  <t:use id="month" types="number">6</t:use> /
  <t:use id="year" types="number">1983</t:use>
</t:component>

<!-- Utilisation du composant -->
<t:use id="birthdate" types="date"/>
```

FIG. 5.7 – Un champ de formulaire pour la date de naissance en utilisant un composant spécialisé avec un valeur par défaut "6/6/1983".

L'élément `use` et les éléments `XML`

Le prochain pas est la généralisation de l'utilisation de `t:use` avec les éléments `XML` (et en conséquence aussi avec les unions). L'utilisation de `t:use` avec un élément XML a une sémantique particulière.

La différence entre l'utilisation de l'élément `XML` lui-même et l'utilisation d'un élément `t:use` avec l'élément `XML` est que, dans le premier cas les attributs ne sont pas éditables, alors que dans le deuxième cas l'élément est obligatoire mais ses attributs sont éditables.

Un élément `t:use` peut avoir un contenu dans le template. Il est traité comme le contenu par défaut, mais il pourra être modifié dans l'instance, c'est-à-dire qu'il pourra contenir tout ce qui est permis par la `DTD` du langage cible.

Notons la différence entre le code des figures 5.8 et 5.9.

```
<h1>
  <t:use id="title" types="string">The title</t:use>
</h1>
```

FIG. 5.8 – Un élément h1 dont les attributs ne sont pas éditables et le contenu doit être une chaîne de caractères

Le code de la figure 5.8 correspond à un élément `h1` qui n'est absolument pas éditable incluant un contenu de type `string` éditable avec le texte par défaut *"The Title"*.

La figure 5.9 montre une solution beaucoup plus simple. Nous indiquons seulement l'utilisation d'un élément `h1`. Tous ses attributs sont éditables et son contenu est défini comme celui permis par la `DTD` de `XHTML`.

```
<t:use id="title" types="h1"/>
```

Fig. 5.9 – Un élément h1 dont les attributs sont éditables avec un contenu libre

Pour rendre l'édition des instances plus simple, nous ajoutons l'attribut `currentType` à `t:use`. Cet attribut indique quel type est utilisé à un moment. Sa valeur doit être un des types autorisés par l'attribut types. Il doit être utilisé au template pour les `t:use` où il y a plus d'un type possible à choisir et du contenu par défaut. Dans ce cas, il faut indiquer le type du contenu par défaut.

L'attribut est affecté pendant l'instanciation seulement si l'attribut `types` n'autorise qu'un type (après substituer les unions par les types qu'elles contiennent). Dans ce cas, le seul type possible est utilisé pour l'attribut `currentType`. Autrement, l'attribut reste inchangé, en considérant comme une erreur qu'il y ait un contenu par défaut si `currentType` est indéfini.

5.3.2 Les zones à structure libre

Comme nous avons vu dans les cas d'utilisation, parfois nous avons besoin de laisser une grande liberté à l'auteur du document. Nous avons déjà vu que cela peut être fait en utilisant des éléments XML comme le type d'un `t:use`. Mais ce mécanisme ne suffit pas pour certaines utilisations, comme par exemple, le cas où nous voulons empêcher l'utilisateur d'ajouter des images.

Pour traiter ces cas où l'édition est très libre, mais restreinte par le type d'éléments permis, nous utilisons l'élément `t:bag`.

L'élément `bag`

L'élément `t:bag` définit une zone à structure libre où l'édition est limitée par l'ensemble des types permis et par la `DTD` du langage. L'ensemble de types permis par le `t:bag` est donné par l'attribut `types`. Cet attribut contient une liste de types (simples, composants, éléments XML et unions) qui pourront être utilisés tant qu'ils ne soit pas interdit par la `DTD`.

Dans la figure 5.10, le code définit une zone qui n'est restreinte que par la propre `DTD` du langage, car l'union prédéfinie `any` inclut tous les types. Nous pouvons aussi utiliser l'union `HtmlNoImages`, définie dans la figure 5.1.4,

```
<t:bag id="sample" types="any"/>
```

FIG. 5.10 – Une zone complètement libre

pour créer une zone de structure libre en empêchant l'utilisateur d'ajouter des images comme montre la figure 5.11.

```
<t:bag id="sample" types="HtmlNoImages">
   Here you can do anything (following the DTD) but putting images.
</t:bag>
```

FIG. 5.11 – Une zone à structure libre et sans images avec un contenu par défaut.

5.3.3 La gestion des zones répétables et optionnelles

Dans plusieurs cas d'utilisation nous voyons le besoin de répéter une structure un nombre de fois parfois limité. C'est pour cela que `XTiger` fournit deux éléments très similaires qui gèrent les répétitions et les zones optionnelles.

Nous disons qu'ils sont très similaires parce qu'une zone optionnelle est conceptuellement synonyme d'une zone qui peut être répétée entre zéro et une fois. Alors, pourquoi avoir deux éléments différents ? La réponse est très simple, pour l'utilisateur une zone répétable et une zone optionnelle sont des objets sémantiquement différents, et les mélanger rendrait le langage moins claire.

L'élément `repeat`

L'élément `t:repeat` indique que son contenu est répétable. Le nombre de répétitions permises sont données à l'aide de ses attributs `minOccurs` et `maxOccurs`, pour le nombre minimal et maximal respectivement. Les valeurs par défaut sont 0 et $*$ respectivement, où $*$ représente l'infini.

Par exemple, supposons que nous avons un composant pour les références bibliographiques et qu'il s'appelle `BibRef`. Alors pour créer la bibliographie à la fin d'un document nous pouvons utiliser le code de la figure 5.12.

En plus des attributs présentés, l'attribut `currentOccurs` est utilisé pour l'édition des instances. Cet attribut indique le nombre de répétitions courante de son contenu. Il est donc toujours entre `minOccurs` et `maxOccurs`. Il

```
<h2>Bibliographie</h2>

<t:repeat minOccurs="1">
   <t:use types="BibRef"/>
</t:repeat>
```

FIG. 5.12 – Une bibliographie comme une répétition de références bibliographiques.

peut être utilisé au template pour indiquer le nombre de répétitions initiales. Sa valeur par défaut est le la valeur de `minOccurs`.

L'élément `option`

L'élément `t:option` est équivalent à un élément `t:repeat` avec la valeur 0 pour l'attribut `minOccurs` et 1 pour `maxOccurs`. Il n'a donc pas besoin de ces attributs.

Pour continuer avec l'exemple de la bibliographie présenté dans la figure 5.12, maintenant nous pouvons indiquer que la bibliographie est elle même optionnelle, pour cela nous utiliserons le code la figure 5.13. Si la bibliographie n'est pas utilisée, il n'y a pas de titre. Si elle est utilisée, alors le titre est engendré et il est obligatoire d'ajouter au moins une référence bibliographique.

```
<t:option>
   <h2>Bibliographie</h2>

   <t:repeat minOccurs="1">
      <t:use types="BibRef"/>
   </t:repeat>
</t:option>
```

FIG. 5.13 – Une bibliographie optionnelle

Pour mieux gérer l'édition des instances, nous ajoutons l'attribut `checked` à `t:option`. Cet attribut booléen indique si le contenu de l'option est ajouté ou pas. Il peut être utilisé par le template pour indiquer l'état initial. Sa valeur initial est `true`, en conséquence les zones optionnelles sont montrées par défaut.

5.3.4 Les attributs

`XTiger` ne considère pas seulement les éléments du langage cible, mais aussi les attributs. Jusqu'à présent nous avons vu que les attributs d'un éléments ne sont éditables que s'ils font partie d'un `t:use` ou un `t:bag`. A l'aide de l'élément `t:attribute` nous avons un contrôle beaucoup plus fin sur les droits d'édition mais aussi sur les valeurs des attributs du langage cible.

L'élément `attribute`

L'élément `t:attribute` a été crée en inspiration directe de celui de *XML Schéma* [28]. Un élément `t:attribute` parle d'un attribut et d'un seul, duquel il indique le nom. Il indique si l'attribut est d'utilisation obligatoire, optionnelle ou s'il est interdit. Il peut aussi donner une valeur par défaut ou imposer une valeur fixe à cet attribut. Par exemple, la figure 5.14 montre une division où l'attribut `class` peut être édité.

```
<div>
   <t:attribute name="class" use="optional"/>
   ...
</div>
```

FIG. 5.14 – Une division dont l'attribut `class` est éditable. L'utilisateur peut définir une valeur pour l'attribut, alors que sans cet élément `attribute` aucun attribut pourrait être utilisé.

L'élément `t:attribute` peut aussi indiquer le type des valeurs de l'attribute correspondant. Les types possibles sont `string` pour les chaînes de caractères, `boolean` pour les valeurs booléennes, `number` pour les nombres et `list` pour les listes de chaînes de caractères séparés par espaces.

```
<a>
   <t:attribute name="rel" type="list"
                values="friend acquaintance contact ..."/>
   ...
</a>
```

FIG. 5.15 – Un lien XHTML utilisant `XFN`.

Au lieu de donner un type, nous pouvons spécifier un ensemble de valeurs possibles que l'attribut peut prendre à l'aide de l'attribut `values`. Si nous donnons ces valeurs et un type `list`, l'attribut peut utiliser plusieurs valeurs parmi l'ensemble. Par exemple, la figure 5.15 montre un lien qui suit le

microformat XFN[26]. L'attribut `rel` est obligatoire, et il contient une liste des valeurs définies par XFN, c'est-à-dire *friend*, *acquaintance*, *contact*, etc.

La figure 5.16 montre une division qui peut avoir un attribut `class` (ce n'est pas obligatoire). Si elle l'utilise, les seules valeurs possibles sont *example* et *code*.

```
<div>
   <t:attribute name="class" values="example code" use="optional"/>
   <t:bag id="content" types="any"/>
</div>
```

FIG. 5.16 – Une division où nous pouvons modifier l'attribut `class`, en affectant la valeur *example* ou *code*.

La figure 5.17 donne un code assez similaire, la seule différence est que si dans 5.16 nous ne pouvons modifier que l'attribut `class`, dans 5.17 nous pouvons modifier tous les attributs, car l'élément `div` est lié au `t:use`. Néanmoins, la valeur de l'attribut `class` est limitée aux valeurs *example* et *code*.

```
<t:use types="div">
   <t:attribute name="class" values="example code" use="optional"/>
</t:use>
```

FIG. 5.17 – Une division où nous pouvons modifier tous les attributs, mais où l'attribut `class` est limité aux valeurs *example* et *code*.

Les contraintes des attributs ne peuvent jamais être contraires à celles indiquées par la DTD du langage, par exemple, regardons la figure 5.18. L'élément `t:attribute` indique que l'attribut `alt` est optionnel, mais cela rentre en contradiction avec la DTD de XHTML qui dit qu'il est obligatoire. En conséquence, l'application d'édition des instances doit considérer que l'attribut est obligatoire et ignorer l'`optional` du `t:attribute`.

```
<t:use types="img">
   <t:attribute name="alt" use="optional"/>
</t:use>
```

FIG. 5.18 – Contrainte contradictoire avec la DTD de XHTML. L'attribut `alt` est obligatoire, et l'élément `attribute` essaye de le rendre optionnel.

Chapitre 6

Production de documents avec XTiger

Ce chapitre présente les processus qui participent à la production de documents à l'aide de XTiger. Ces processus vont de la création des templates jusqu'à l'édition des instances.

Les processus utilisent, produisent et modifient des documents. Nous introduisons d'abord les ressources impliquées dans les processus. Ensuite nous présentons en détail ces processus.

6.1 Les ressources

Pour la production d'un document à l'aide de XTiger nous travaillons avec les ressources suivants.

Les templates. Ces documents sont écrits dans le langage cible, mais ils peuvent être non valides à cause essentiellement des éléments XTiger qu'ils contiennent (cf. section 4.1.3). Ils sont identifiés par l'extension '.xtd', les sigles de *XTiger Template Definition*.

Les librairies. Les librairies sont des documents du langage XTiger qui contiennent des parties en langage cible, notamment dans les définitions des composants. Elles sont identifiées par l'extension '.xld', les sigles de *XTiger Library Definition*.

Les ressources externes. Les ressources externes sont toutes celles qui sont utilisées par les templates et les instances sans qu'elles soient liées à l'aide de `XTiger`. Dans le cas des templates `XHTML` nous trouverons des images, des feuilles de style `CSS` [24] [19], des fichiers `javascript`[1], etc.

Pour les documents `XML` hors des langages les plus répandus, nous pouvons aussi trouver des feuilles de transformation `XSLT` [9] qui les convertissent vers les langages standards du web.

Ces ressources ne concernent pas directement le langage `XTiger`, mais elles sont importantes dans le processus de production des instances.

Les instances. Ce sont des documents du langage cible valides, une fois enlevés les éléments `XTiger`. Ces éléments `XTiger` servent à restreindre la structure des instances et à contrôler l'édition de son contenu. Toutes les instances gardent un lien vers leur template originaire sous la forme d'une instruction de traitement[1].

6.2 Les processus

Les ressources présentées précédemment sont créées, utilisées ou modifiées par un ensemble de processus. Ces processus peuvent être classés en processus de création, d'édition et de mise à jour ou de transformation.

Les processus de création sont ceux qui rendent une nouvelle ressource comme résultat. Nous avons processus de création pour les templates, les instances, les librairies et les ressources externes.

Les processus d'édition sont ceux qui permettent à l'utilisateur de modifier à sa demande une ressource. Nous avons processus d'édition pour les templates et les instances.

Finalement, nous avons un processus pour la misa à jour des instances suite aux modifications des templates et un processus pour convertir les instances en documents valides classiques.

6.2.1 Création d'un template

Les templates peuvent être créés à partir de zéro, mais normalement il est plus intéressant de se baser sur les instances. Nous trouvons alors deux

[1] Processing Instruction dans la version anglaise

façons de générer un template à partir d'un ou des instances : la génération à la main et la génération automatique.

Création manuelle à partir d'une instance

En nous basant sur une instance, nous pouvons ajouter des éléments XTiger pour définir quelles zones sont éditables, optionnelles, répétables, etc. De la même façon nous pouvons sélectionner des bouts de structure et les définir comme nouveaux composants. Ces composants seront alors ajoutés dans l'en-tête du template.

Cette façon de générer un template est très intuitive pour un utilisateur sans expérience. Dans certains cas il faudra ajouter d'autres mécanismes, notamment pour la gestion des concepts les plus abstraits comme les unions et les types simples.

Création automatique à partir de plusieurs instances

Nous pouvons envisager la génération automatique des templates à partir d'un ensemble significatif d'instances. Ce processus pourrait se baser sur les résultats achevés dans l'extraction de schémas à partir d'un ensemble de documents XML présenté dans [21]. En outre, il faudrait ajouter un mécanisme pour pouvoir réutiliser les définitions faites dans d'autres librairies pour qu'on puisse détecter l'utilisation de types déjà existants.

Ce processus serait très pratique pour les utilisateurs sans expérience qui pourraient, de façon automatique, générer des templates à partir des documents qu'ils ont créés auparavant. Par contre, il est prévisible que les résultats auront une sémantique beaucoup moins riche que celle des templates créés manuellement.

Nous proposons donc un processus mixte qui commencerait par la création automatique du template, et permettrait après à l'auteur d'éditer le template pour ajouter la sémantique nécessaire.

6.2.2 Création des librairies

Les librairies ne sont qu'un ensemble de déclarations qui pourraient être faites directement dans l'en-tête des templates. Ces déclarations sont déplacées vers un document à part pour être partagées. Pour sa création, le plus simple est donc de les créer à partir des déclarations générées par la création du template.

Dans le cas des microformats, le plus simple serait de créer tous les composants dans un template et de les exporter vers une librairie. Ensuite, le template utilisé pourrait être effacé, car il n'aurait plus d'intérêt.

6.2.3 Création des ressources externes

Les ressources externes peuvent être créées en ignorant l'utilisation de `XTiger` pour la production des autres documents.

Par contre, il est conseillé de les créer (ou les adapter) une fois que le template est prêt pour en tirer meilleure partie de l'uniformité des instances.

6.2.4 Création des instances

Les instances sont crées à partir d'un template. Le processus de création d'une instance à partir d'un template est connu comme *instanciation*. L'instanciation produit un document du langage cible valide par rapport aux contraintes exprimées par un template mais aussi par le schéma ou la DTD du langage cible.

L'instanciation se fait en deux étapes : la première est la liaison de l'instance vers le template et la deuxième est l'initialisation du contenu de l'instance à partir du template.

Pour lier l'instance au template, le processus ajoute une instruction de traitement (PI), comme celle montrée dans le code suivant, indiquant la situation du template à partir duquel l'instance a été générée. Ce lien sera utilisé à chaque édition de l'instance pour accéder aux déclarations du template utilisé.

```
<?xtiger template="url du template" ?>
```

Cette instruction de traitement permet aussi de distinguer les documents instance des documents classiques du langage cible. Quand nous trouvons cette instruction, nous devons figer les parties non éditables et appliquer les contraintes exprimées par les balises `XTiger`.

L'élément `t:head` et tout son contenu n'est pas recopié dans l'instance. Nous ne trouverons donc pas des éléments `t:union` ni `t:import` dans l'instance. Par contre, nous pouvons encore trouver des éléments `t:component`

comme des déclarations *inline* (cf. 5.1.3). Ces déclarations sont substituées par des éléments `t:use` du type défini par le `t:component`.

Pour les éléments `t:use`, nous initialisons l'attribut `currentType`, s'il n'est pas défini par le template et qu'il n'y ait qu'un seul type possible. Ensuite nous adaptons le contenu courant en fonction du `currentType` :

si c'est un type simple nous copions le contenu du `t:use` du template dans celui de l'instance. S'il n'a pas de contenu, le `t:use` de l'instance sera aussi vide.

si c'est un composant nous copions le contenu du `t:use` du template dans celui de l'instance en vérifiant que la structure est bien celle du type indiqué (voir la figure 6.1).

Cela permet de donner une valeur par défaut à chaque `t:use` différente de celle donnée dans la déclaration du composant.

Si le contenu du `t:use` du template est vide, le contenu du `t:use` de l'instance est l'instanciation du contenu du `t:component` qui définit le type dans `currentType`. Si `currentType` n'est pas défini, le contenu du `t:use` dans l'instance est vide.

si c'est un élément `XML` nous créons d'abord un élément du type indiqué par `currentType` et l'insérons comme seul fils du `t:use` dans l'instance. Cet élément ne sera pas effaçable, mais se attributs pourront par défaut être édités. Néanmoins, les élément `t:attribute` fils du `t:use` portent sur l'élément créé, en ignorant tous ceux qui rentrent en contradiction avec la `DTD` du langage. Nous ignorons notamment, les attributs qui portent sur des attributs qui n'existent pas pour le type `currentType` (fig. 6.2).

Le contenu du `t:use` du template est copié comme contenu de l'élément fils du `t:use` dans l'instance. Ce contenu est complètement éditable, comme si nous nous trouvions dans un `t:bag` de type `any`.

L'attribut on le change pas de place

Les éléments `t:bag` ne sont pas modifiés. Le contenu du `t:bag` de l'instance est l'instanciation du contenu de celui du template. Néanmoins, nous vérifions que le contenu donné correspond bien aux restrictions du `t:bag` et qu'il est valide au sens du langage cible.

Quand nous trouvons un élément `t:repeat`, nous lui initialisons son attribut `currentOccurs`, avec la valeur `minOccurs`. Ensuite, nous répétons le contenu du `t:repeat` le nombre de fois indiqué par `currentOccurs`.

Les éléments `t:option` voient leur attribut `checked` initialisé à la valeur `true` s'il n'est pas défini dans le template. Après, le contenu du `t:option` est inclus dans l'instance ou pas selon la valeur de `checked`.

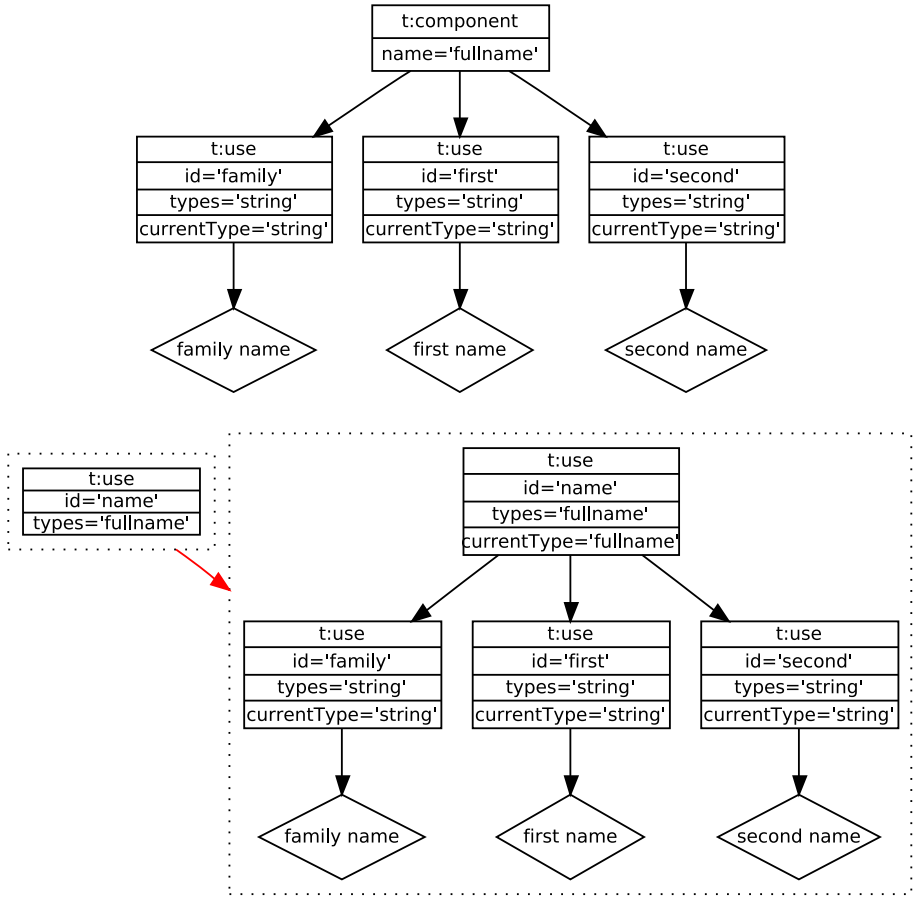

FIG. 6.1 – Schéma d'instanciation d'un élément `use` d'un composant défini en haut. À gauche le template, à droite l'instance. Seulement les losanges sont éditables.

Finalement, quand les éléments `t:attribute` indiquent une valeur par défaut ou fixe et qu'ils ne rentrent pas en contradiction avec la définition du langage cible, l'attribut est créé avec la valeur correspondante pour l'élément concerné. Autrement, aucune modification est faite.

6.2.5 Édition des instances

Une fois le processus d'instanciation fini, nous obtenons un document du langage cible valide (une fois enlevées les éléments `XTiger`) mais qui est, dans la plupart des cas, incomplet. L'auteur doit donc éditer cet instance pour arriver au document souhaité.

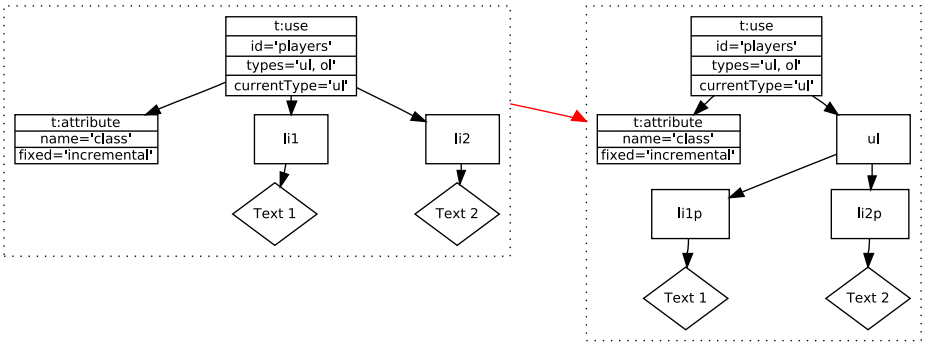

FIG. 6.2 – Schéma d'instanciation d'un élément `use` d'un élément XML avec un `attribute` et un contenu défini dans le template.

L'édition de documents structurés est un processus très complexe [22]. C'est pour cela que nous n'envisageons pas de présenter en détail tous les aspects existants, mais nous expliquons seulement les nouveaux aspects liés à l'utilisation des templates.

Le document es figé sauf dans les parties rendues éditables par les balises XTiger. Plus concrètement, les éléments qui permettent à l'auteur de réaliser quelque action sont les éléments `t:repeat`, `t:option`, `t:attribute`, `t:bag` et `t:use`. Nous présentons à présent quel type d'action ils permettent de réaliser.

`t:repeat`
: Cet élément permet la répétition de la structure décrite par son contenu un nombre de fois borné inférieurement par son attribut `minOccurs` et supérieurement par `maxOccurs`.

`t:option`
: Cet élément exprime l'optionnalité de son contenu, c'est-à-dire que son contenu peut être complètement retiré du document. Dans ce cas, l'élément `t:option` est présent dans l'instance, mais son contenu est vide.

`t:attribute`
: Cet élément modifie les contraintes existantes sur un des attributs de son père direct. Si cet élément est un `t:use` d'un type élément XML, ce sera sur ce dernier que le `t:attribute` s'appliquera et non pas sur le `t:use`.

`t:bag`
: Cet élément permet l'édition libre de son contenu en limitant les types qui sont permis dans sa descendance. En fait, l'édition est libre jusqu'à ce que nous utilisions un composant. Dans ce cas, l'édition est

contrainte par la structure du composant et non par celle du `t:bag` ; néanmoins, le composant peut encore être supprimé.

`t:use`
: La sémantique de cet élément dépend du type utilisé. Ce type est indiqué par l'attribut `currentType` de l'élément. Si le type est :

  un type simple
  : le contenu est complètement éditable.

  un élément XML
  : le premier fils du `t:use` (en ignorant les `t:attribute`) ne peut être que d'un des types permis. Ses attributs sont éditables sauf si les élément `t:attribute` du `t:use` indiquent autrement. Son contenu est complètement éditable.

  un composant
  : tout le contenu est figé, sauf pour les éléments qui indiquent le contraire.

Pour rendre l'édition plus simple pour l'auteur nous ajoutons un attribut `title` qui contient une description en langue naturelle de l'élément concerné. Ce n'est pas nécessaire de gérer l'internationalisme des langues car un template n'est que dans une seule langue, et nous considérons cette langue comme celle de l'auteur.

La façon d'implanter tous ces mécanismes sur une application d'édition pour des documents structurés est présentée dans le chapitre 7.

6.2.6 Édition des templates

L'édition des templates ne fait pas partie du but du stage. Néanmoins, nous considérons que le meilleur choix est d'intégrer l'édition des templates aux différents outils d'éditions des instances. Cela s'explique par l'approche très pragmatique du stage. Plus proche sera l'édition des templates soit de celle des instances, plus facile cela sera pour l'auteur de comprendre.

En outre, une des façons la plus simple de créer des templates est de se baser sur les instances. Il est donc important de pouvoir passer d'une façon transparente d'un document classique d'un langage vers un template. Ce passage sera beaucoup plus simple s'il se passe dans le cadre d'un seul outil d'édition.

Par exemple, pour le cas de l'édition `XHTML` sur `Amaya`, nous envisageons la création d'une fonctionnalité pour convertir un document `XHTML` en un template (basiquement il faut changer l'extension du fichier à *'.xtd'*).

L'édition serait donc intégrée à Amaya et se ferait suivant les mêmes principes que l'édition XHTML mais en ajoutant des fonctionnalités pour les éléments XTiger. Après avoir fini le template, il pourra être utilisé, encore dans Amaya, pour créer une nouvelle instance qui aura à nouveau l'extension *'.html'*.

6.2.7 Mise à jour des instances

L'édition des templates reste possible même après avoir été utilisés pour la création d'instances. Cela implique qu'après la modification d'un template, les instances déjà existantes ne seront potentiellement plus concordantes avec le template.

Nous voudrions donc pouvoir mettre à jour les vieilles instances pour les rendre conformes à la nouvelle version du template. Il est important de remarque que cette transformation n'est pas toujours possible, car deux versions d'un même template peuvent être complètement différentes et même incompatibles.

Nous laissons donc ce processus comme une question ouverte et nous nous limitons à une partie du problème. Cette partie est l'identification des changements de base comme les ajouts, les changements de position et la suppression des parties éditables. Ce problème est aussi traitée par le logiciel *Dreamweaver*.

Pour faire cela, nous associons un identificateur à tous les éléments XTiger qui peuvent apparaître dans le corps du template sauf `t:attribute`. Tous ces éléments auront donc un attribut `id` qui les identifie sans ambiguïté dans le template.

Les instances vont pouvoir se resynchroniser avec les nouvelles version du template grâce à ces identificateurs. Plus concrètement nous pouvons détecter et synchroniser après ces changements :

- Élément supprimé. Nous le détectons comme un *id* qui disparaît d'une version à l'autre. Pour synchroniser nous supprimons du document les éléments avec l'*id* correspondant.
- Élément ajouté. Nous le détectons comme un *id* qui n'existait dans la version précédente. Pour synchroniser nous instancions ce nouvel élément et nous l'insérons à sa place dans l'instance.
- Élément déplacé. Nous le détectons quand un élément avec un id n'est plus dans le même emplacement que dans la version précédente. Pour synchroniser l'instance nous replaçons l'élément concerné dans le nouvel endroit.

Même si nous avons un aperçu de comment nous pouvons développer ce processus, ce n'est qu'un ébauche à étudier plus en profondeur.

6.2.8 Transformation des instances en documents classiques

Comme nous avons déjà présenté, les instances ne sont pas tout à fait des documents valides. Pour devenir valides il faut enlever tous les éléments du langage `XTiger`. Ce justement le but de ce processus, qui enlève toute trace du langage XTiger pour rendre les instances complètement valides.

Concrètement, le processus enlève tous les éléments `XTiger` de l'instance et enlève l'instruction de traitement (PI) qui la lie à son template.

Chapitre 7

Implémentation de l'édition des instances

Ce chapitre présente la partie implémentation du stage. D'abord nous présentons *Amaya*, le logiciel auquel nous avons ajouté des fonctionnalités pour traiter les templates `XTiger`. Ce logiciel est construit sur une librairie pour gérer des documents structurés appelée *Thot*, dans la section 7.2 nous présentons la librairie, pour présenter ensuite la relation entre les deux logiciels dans la section 7.3.

Finalement nous présentons les deux processus qui ont était implémentés
- l'instanciation dans la section 7.4.2 et
- l'édition des instances dans la section 7.4.3

et comment ils ont été implémentés et intégrés sur *Amaya*.

7.1 *Amaya*

Amaya est un éditeur Web, développé conjointement avec le W3C, pour éditer et publier très simplement des pages contenant du texte (en `HTML` ou `XHTML`), du graphique (en `SVG`) et des expressions mathématiques (en `MathML`), le tout avec des feuilles de style `CSS`.

7.1.1 Navigateur et éditeur de pages Web

Amaya est à la fois un navigateur et un outil auteur qui utilise le Web comme un espace de travail uniforme et dynamique. L'utilisateur d'*Amaya* peut à tout moment prendre l'initiative de modifier, copier/coller, mettre à

jour les informations de la page Web visualisée et re-publier immédiatement cette page sur le serveur Web (avec la méthode `HTTP/PUT`), pour peu qu'il ait les droits d'accès nécessaires. L'utilisateur n'a pas besoin d'avoir une bonne connaissance des langages de balisage utilisés. Il peut créer des liens hypertexte par simple clic. Il peut copier/coller entre deux pages une structure complexe comme une table ou une liste. Si cette structure contient des liens, ces liens sont préservés.

7.1.2 `XHTML` (HyperText Markup Language)

Amaya est un outil auteur Web qui produit du code `XHTML` valide. Il permet aussi à ses utilisateurs de contrôler les profils `XHTML` utilisés. La conformité des éléments et des attributs utilisés est vérifiée au chargement des documents et les menus de création s'adaptent au profil de document choisi. La validité et le contrôle de conformité sont tout particulièrement utiles pour produire de l'information destinée à des équipements mobiles (téléphones cellulaires, PDA, etc.)

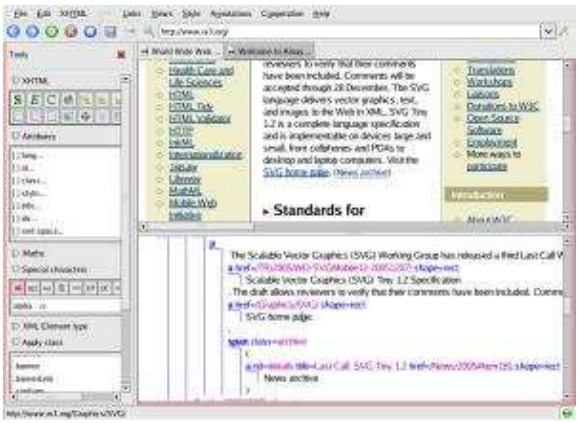

FIG. 7.1 – L'édition peut se faire dans la vue formatée ou la vue structure

Amaya inclut un support complet pour les documents `(X)HTML`

- Tous les éléments et les attributs `HTML` sont présentés dans des menus contextuels
- Un mécanisme de transformation performant permet de réorganiser la structure du document (par exemple, un ensemble de paragraphes peut être transformé en une liste, etc.).
- *Amaya* aide à gérer les liens hypertexte (création de liens relatifs, copier/coller de morceaux de document incluant des liens).
- Il inclut un éditeur de table capable d'insérer/copier/coller des colonnes et des lignes de tables, et d'étendre/restreindre des cellules.

– *Amaya* fournit un correcteur orthographique capable de travailler avec plusieurs dictionnaires pour les documents multilingues.
– Les documents peuvent être vus et édités via de multiples vues : formatée, vue structure, vue source, etc.

7.1.3 CSS (Cascading Style Sheets)

Amaya inclut un support auteur pour CSS 2

– *Amaya* fournit un mécanisme simple pour ajouter du style (fontes, couleurs, espacements, etc.) à des documents Web.
– Il permet aussi de tester des feuilles de style externes et de les associer facilement aux documents
– Les fichiers CSS peuvent être téléchargés, édités et publiés avec *Amaya* aussi simplement que les autres documents.
– *Amaya* fournit une commande qui permet de voir quelles propriétés CSS sont effectivement appliquées à la sélection et quelle feuille de style CSS les a produites.

7.1.4 MathML et SVG (Scalable Vector Graphics)

Amaya permet d'éditer et de publier très simplement des graphiques SVG (graphiques structurés) et des expressions mathématiques en MathML, ainsi que des documents composites.

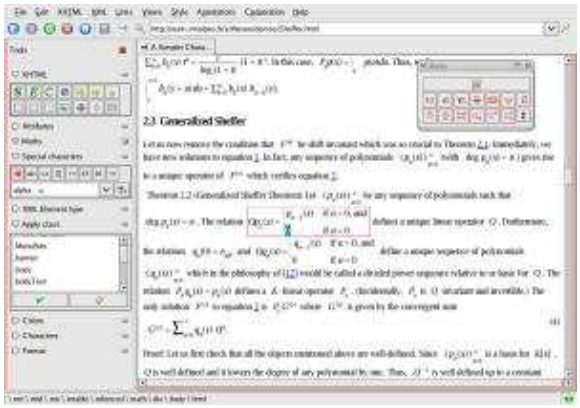

FIG. 7.2 – Édition mathématique

Par exemple, il peut traiter de façon uniforme et sans faire appel à des plug-ins ou à différents éditeurs un document XHTML incluant des graphiques SVG avec des éléments XHTML et des expressions MathML à l'intérieur de ces graphiques.

– Tout le balisage de présentation de MathML peut être aisément édité avec *Amaya*.
– Un analyseur de texte permet d'engendrer une structure MathML à partir d'un texte simple. Par exemple, 'a+b' produira
<mi>a</mi><mo>+</mo><mi>b</mi>
– L'éditeur de tableaux sait gérer les matrices MathML aussi bien que les tables HTML.
– La fonction de recherche/remplacement et le correcteur orthographique travaillent sur les textes inclus dans toutes les structures XML du document (HTML, SVG, etc.).

7.1.5 XML

Aujourd'hui *Amaya* gère quelques langages de balisage XML connus, comme XHTML, MathML et SVG. Les tout récents développements d'*Amaya* concernent l'édition de documents XML génériques (documents édités sans DTD ni schéma XML). Ce travail est en cours, mais *Amaya* permet déjà d'éditer le texte et les attributs des documents XML et d'améliorer leur présentation en créant et en appliquant des feuilles de style CSS.

7.1.6 Annotations

Amaya permet de créer et de lire des annotations de type Annotea. Les annotations sont des commentaires, notes explications, ou d'autres types de remarques externes qui peuvent être attachées à n'importe quel document Web ou partie de document que l'on a sélectionnée sans avoir à toucher au document.

Quand l'utilisateur lit un document Web, il ou elle peut charger les annotations qui lui sont attachées depuis un ou plusieurs serveurs d'annotation sélectionnés ou stockées dans son espace local. Distribution et contributions

Amaya est un projet Open Source couvert par la licence logicielle du W3C. Le code source et des versions pré-compilées pour un certain nombre de plates-formes (Linux, Windows NT/2000/XP, Windows 98) peuvent être téléchargés librement depuis le Web. Une base CVS permet aux personnes qui veulent contribuer de suivre de plus près les derniers développements du logiciel.

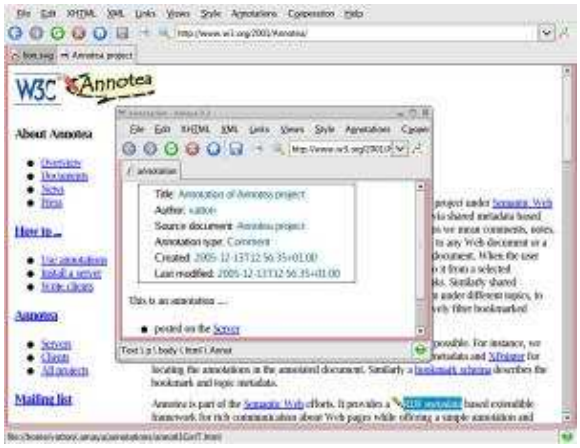

FIG. 7.3 – Le crayon indique qu'li y a une annotation et la sélection donne la portée de l'annotation.

7.2 *Thot* : un éditeur de documents structurés

Thot permet de créer, de modifier et de consulter de façon interactive des documents qui respectent des modèles. Grâce à ces modèles, on obtient des documents homogènes et l'utilisateur peut se concentrer sur l'organisation et le contenu des documents qu'il traite, sans s'occuper de formatage ou de typographie, ces fonctions étant prises en charge par le système.

Thot effectue également d'autres traitements pour l'utilisateur, comme les numérotations, le maintien des références croisées, la gestion des index, la correction orthographique, etc.

Thot permet la production de documents dans de nombreux formats comme `Postscript`, `LaTeX`, ou `HTML`.

7.2.1 Édition structurée

Dans *Thot*, un document est représenté par sa structure logique, c'est-à-dire son organisation en éléments comme des titres, chapitres, sections, paragraphes, notes, figures, etc. Ces éléments forment des structures hiérarchiques qui rendent compte des relations d'inclusion et d'ordre entre les éléments. Le texte ainsi que d'autres éléments de base (symboles, graphiques, images) constituent les éléments terminaux de ces structures hiérarchiques.

La structure logique est contrainte par un schéma de structure, qui spécifie principalement les types des éléments utilisables et les relations qui

peuvent les relier. Chaque type de document est défini par un schéma de structure et il est possible de définir de nouveaux types de documents.

La structure logique d'un document est construite par l'éditeur *Thot*, sous le contrôle de l'utilisateur. L'éditeur assure que chaque document qu'il traite respecte le modèle de son schéma de structure et, pour cela, il n'autorise que les opérations qui conduisent à une structure logique conforme au schéma de structure. Il utilise également le schéma de structure pour guider l'utilisateur ou pour engendrer automatiquement certaines parties de la structure du document.

7.2.2 Système intégré

Thot est un système intégré et extensible. Il permet de traiter avec le même outil et dans le même document non seulement du texte structuré, mais aussi du graphique, des tableaux complexes, des expressions mathématiques, etc. Cette liste n'est pas exhaustive : les utilisateurs peuvent ajouter d'autres types d'informations, en définissant les modèles adéquats.

7.2.3 Système ouvert

Thot est un système ouvert. Il peut échanger des documents avec d'autres systèmes, par l'intermédiaire d'un outil d'exportation paramétrable. Par exemple, il est possible d'exporter des documents en `LaTeX` et en `HTML`.

Thot peut aussi s'intégrer dans d'autres applications, à travers son interface de programmation et son mécanisme d'appels externes.

7.3 Intégration de XTiger, *Amaya* et *Thot*

Amaya délègue à Thot tout le traitement des documents structure tels que le parsing, l'affichage des documents où l'exportation vers plusieurs formats de fichier. Comme il est montré par le diagramme de la figure 7.3, l'intégration d'Amaya et Thot est basée sur deux points

- Amaya utilise l'API[1] de Thot pour indiquer les actions que Thot doit réaliser sur les documents structurés qu'il gère.
- Un fichier d'application lie les événements produits sur les éléments des documents structurés avec les callback correspondants d'Amaya. Ce fichier nommé `Template.A` est adjoint de l'annexe B.1.

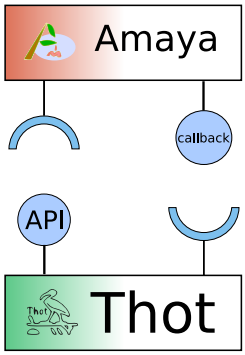

FIG. 7.4 – Diagramme basique de l'architecture pour l'intégration d'Amaya et Thot.

Par exemple, Amaya fait appel à la libraire Thot pour charger et montrer un document. Mais si nous cliquons sur un élément `t:use` c'est Thot qui détecte l'événement et qui fait appel au callback que nous l'avons indiqué dans le fichier d'application.

7.3.1 Fichiers Thot de description des Templates

En plus du fichier d'application, nous devons fournir une description de la structure des templates. Pour cela nous utilisons quatre fichiers :

Structure logique

La structure logique du template est décrite à l'aide du langage S de Thot dans le fichier appelé `Template.S`. Le contenu de ce fichier est inclus dans l'annexe B.2.

Dans ce fichier nous énumérons les éléments du langage ainsi que leurs attributs et les relations hiérarchiques qui existent entre eux. Nous pouvons le comparer à une `DTD XML`.

Présentation

La façon de présenter graphiquement les éléments du langage est décrite à l'aide du langage P de Thot dans le fichier appelé `TemplateP.P`. Le contenu de ce fichier est inclus dans l'annexe B.4.

Le fichier contient une description graphique des éléments du langage pour les différentes vues d'Amaya. Par exemple, nous indiquons que l'élément `t:option` est affiché avec un cadre orange. Nous pouvons le comparer à l'utilisation du langage `CSS` pour l'affichage des documents `XML`.

[1] Application Programming Interface ou Interface de Programmation d'Application

Code source

La façon de sauvegarder les éléments XTiger sous forme de code source est indiquée à l'aide du langage T dans le fichier `TemplateT.T`. Le contenu de ce fichier est inclus dans l'annexe B.3.

Thot représente en interne les documents structurés sous la forme de structures d'arbres et les externalise par défaut dans des fichiers en utilisant un format qui lui est propre. Les schémas de transformation (ou fichiers T) permettent d'externaliser les structures d'arbres sous un autre format. C'est ce langage que nous utilisons pour pouvoir externaliser les structures XTiger au format XML.

Noms des éléments

Le nom lisible des éléments est indiqué dans le fichier `Template.en`, inclus dans l'annexe B.5. Cela nous permet d'utiliser des noms différents dans les fichiers précédents à ceux que l'utilisateur visualise.

Par exemple, si nous avons un attribut et un élément qui s'appellent pareil, pour éviter l'ambiguïté il faut utiliser des noms différents. Ce sera alors dans ce fichier `Template.en` que nous indiquerons qu'ils ont le même nom. Nous pouvons aussi rendre invisible des éléments pour l'utilisateur en les donnant un nom vide[2].

7.4 Implémentation des processus de production XHTML

Cette section présente la façon dont nous avons implémenté les processus basiques permettant l'édition de documents `XHTML` à l'aide des template `XTiger`.

Ces processus sont l'instanciation et l'édition. Ils partagent un processus commun qui est le chargement d'un template. Ce processus auxiliaire est présenté d'abord pour mieux comprendre le résultat complet.

7.4.1 Le chargement des templates

Le premier pas pour créer une nouvelle instance à partir d'un template ou d'en éditer une déjà existante est de lire la définition des types. Quand nous lisons un type nous gardons toute son information dans un dictionnaire qui indexe tous les types à partir de son nom.

Plus concrètement, nous avons un ensemble de structures où chaque structure représente un template ou une librairie. Cela nous permet de

[2]Le nom vide est représenté par le symbole `\0` dans le fichier `Template.en`

n'avoir à charger le template ou la librairie qu'une seule fois quand l'utilisateur édite plusieurs instances, et ainsi de limiter l'effort de chargement au minimum.

Chacune de ces structures que nous appelons environnements de déclaration, ou plus simplement environnements, contient quatre dictionnaires. Chacun de ces dictionnaires contient des déclarations d'une nature différente. Nous avons donc :

- un dictionnaire de types simples,
- un dictionnaire de composants,
- un dictionnaire d'éléments XML et
- un dictionnaire d'unions.

Pour les types simples, pour le moment, nous ne mémorisons que son nom. Ce mécanisme est implémenté avec un propos de rendre l'application plus facilement extensible. Nous considérons que l'extension de XTiger pour accepter la définition de types simples serait très utile, donc il nous semble convenable d'en prendre soin du debout.

Les déclarations des composants sont aussi identifiées par son nom. Elles contiennent aussi la structure du composant en la forme en la que `Thot` stocke les documents internement. Cela nous permet de l'insérer plus facilement que si nous gardions le code `XHTML`.

Les éléments XML sont identifiés par son nom qualifié, c'est-à-dire par son nom précédé de son espace de noms. Quand l'espace de noms n'est pas donné explicitement il est obtenu en appliquant la recommandation, il est donc l'espace de noms du père.

Finalement, les unions sont identifiées par son nom et gardent un dictionnaire de pointeurs vers les déclarations des types qu'elles contiennent.

Tous ces quatre dictionnaires sont mutuellement disjoint par rapport aux noms des éléments. C'est-à-dire qu'à tout moment il ne peut y avoir qu'une déclaration pour un nom. Quand nous trouvons une déclaration avec un nom déjà présent aux dictionnaires nous enlevons celle qui existait avant et nous insérons la nouvelle déclaration. Ce mécanisme est égal à celui des feuilles de style CSS[24].

Les templates et les librairies forment des hiérarchies qui peuvent être représentées par de DAG[3] comme nous montrons dans la figure 7.5.

Dans la figure nous voyons que la librairie M définit les types a et b. Le template C les utilise en ajoutant un nouveau type e. Par contre, la librairie L utilise M en redéfinissant le type a en a' et en ajoutant le type

[3]Directed Acyclic Graph : Graph dirigé sans cycles

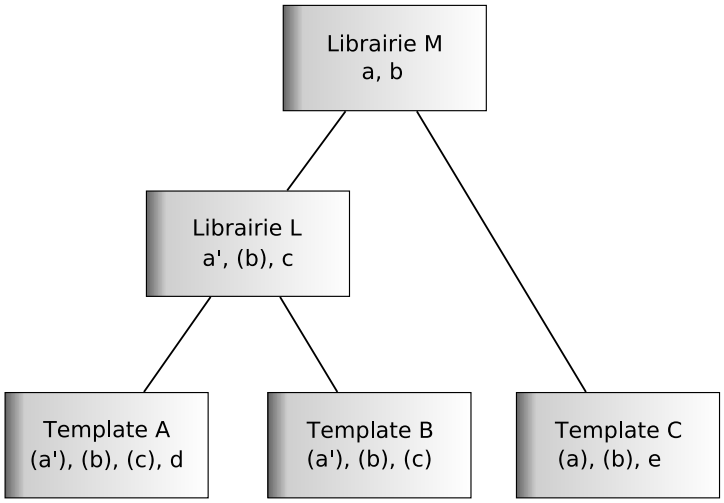

FIG. 7.5 – Hiérarchie d'importation de librairies. Toutes les arêtes sont dirigées vers le haut dans la relation *"imports"*.

c. Finalement, le template A utilise L en ajoutant un type d, et B l'utilise telle qu'elle.

Nous avons le choix de garder cette structure de librairies ou de ne garder qu'une version aplatie. Cette version aplatie ne contiendrait que les types qui sont accessibles pour un template concret. Par exemple, pour le template A de la figure 7.5 nous aurions les types *a'*, *b*, *c* et *d*, mais les librairies L et M ne seraient pas conservées en mémoire.

L'autre option est de garder toute la structure telle qu'elle. Cette solution est plus rapide si nous utilisons plusieurs fois les mêmes librairies, car elles ne seraient chargées qu'une seule fois. Par contre, la recherche d'un type dans un dictionnaire pourrait nous amener à rechercher récursivement dans tous se successeurs.

Nous avons choisit d'utiliser une option mixte. Cette solution garde la structure sans l'aplatir pour éviter de charger plusieurs fois une librairie. Mais en même temps nous copions les déclarations des librairies dans l'environnement des templates et librairies de ceux qui la utilisent. Avec cette solution l'accès aux types est celui de l'accès à un élément dans un dictionnaire et les librairies ne sont rechargées que si c'est nécessaire.

Après le chargement d'un template, nous disposons de toutes ses déclarations. Ces déclarations sont accessibles grâce à l'adresse du template, qui l'identifie dans un dictionnaire de templates qui fait partie d'*Amaya*.

7.4.2 L'instanciation

L'instanciation est le processus qui à partir d'un template créé un document du langage cible. Pour cela, il faut d'abord que le template soit chargé comme il est indiqué dans 7.4.1.

Ce processus est déclenché après confirmation du dialogue montré dans la figure 7.6.

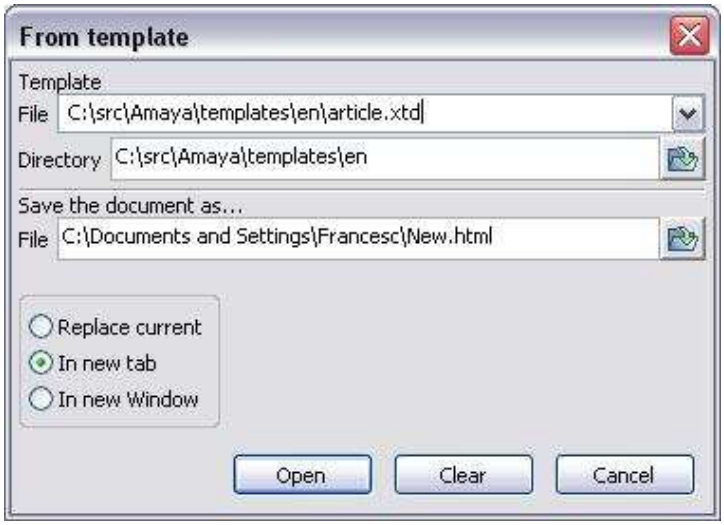

FIG. 7.6 – Dialogue pour la création de nouvelles instances d'un template.

Comme nous avons déjà indiqué dans la section 6.2.4, nous devons ajouter une instruction de traitement[4] pour pouvoir plus tard retrouver le template à partir de l'instance. Cette instruction est insérée en utilisant les fonctions de Thot pour les modifications de la structure.

Ensuite nous devons traiter tous les éléments XTiger que nous trouvons dans le template comme nous avons expliqué dans la section 6.2.4. Nous enlevons l'élément `t:head`, nous créons du contenu par défaut pour les `t:use`, etc. Ce traitement est réalisé récursivement sur toute la structure.

Finalement, nous n'avons qu'a sauvegarder cette nouvelle instance dans le fichier indiqué par l'utilisateur dans le dialogue montré dans la figure 7.6.

Problèmes rencontrés lors de l'implémentation

Le premier des problèmes que nous avons trouvé est un problème très commun. Les `DTD` des langages décrivent la grammaire d'un langage mais

[4]Processing instruction en anglais

pas comment celui peut être mélangé à d'autres langages. Cela pose des problèmes d'intégration avec des nouveaux langages. Ishikawa [12] a créé une DTD pour les éléments utilisant `XHTML`, `SVG` et `MathML`. Avec `XTiger` ce n'est pas nécessaire, car les éléments `XTiger` peuvent apparaître à n'importe quel point du document, nous optons donc de les ignorer tout simplement.

Dans notre cas, *Amaya* contient des contraintes sur le langage `XHTML` qui portent sur des structures très figées. Par exemple, dans les figure 7.8 nous montrons ce problème. Si *Amaya* ne prend pas en compte que l'élément `t:use` est d'un autre espace de noms particulier (celui de `XTiger`), au lieu d'ignorer l'élément, il considère comme une erreur le fait que le père direct d'un `li` ne soit pas une liste `ul` ou `ol`. Un élément `ul` est ajouté comme fils du `t:use` et père du `li` pour rendre le code valide vis-à-vis des contraintes XHTML. Si Amaya ignorait les éléments `XTiger` pour la validation du `XHTML` ce problème n'apparaîtrait pas, car le père du `li` serait bien un `ul`.

```
<p>Un use qui englobe un item de la liste</p>
<ul>
   <t:use types="li">
      <li>Cet item est dans le use</li>
   </t:use>
   <li>Pas celui-là.</li>
</ul>
```

Fig. 7.7 – Code problématique : *Amaya* considère comme une erreur d'avoir un élément différent de `ul` comme père de `li`.

Nous avons dû modifier le code d'Amaya pour que cela ne se produise pas, en ignorant tous les éléments de l'espace de noms de `XTiger`.

7.4.3 L'édition d'instances

L'édition d'une instance se divise en deux parties. D'abord nous chargeons l'instance pour ensuite permettre une édition contrôlée du document en suivant les contraintes exprimés par le template et par le langage cible lui-même.

Le chargement des instances

Le premier pas du chargement d'une instance est de récupérer le template dont elle est issue. L'adresse de ce template se trouve dans une instruction de traitement du type `<?xtiger template="..." ?>`. Si cette instruction

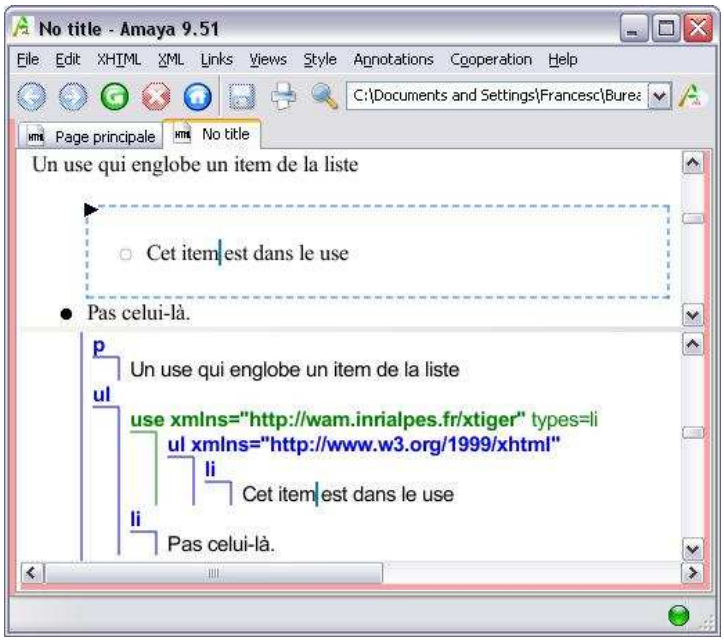

FIG. 7.8 – Résultat du chargement du code de la figure 7.7. Un `ul` a été inséré comme père du premier `li`.

n'est pas présente dans le document, ce n'est pas une instance et tous les éléments XTiger que nous trouvons doivent être ignorés.

Une fois vérifié que le document est bien une instance, nous chargeons le template correspondant s'il n'était pas encore chargé. Ensuite nous pouvons commencer le chargement proprement dit de l'instance.

Nous lions tous les éléments `t:use` et `t:bag` aux déclarations correspondantes aux types auxquels ils font référence par nom. Cela nous permettra une édition plus rapide, car l'accès aux définitions sera immédiate.

En même temps que nous lions les types à leur déclaration nous définissons quels éléments sont éditables et lesquels sont figés. Nous commençons par dire que tout le document est figé pour ensuite libérer les zones éditables correspondantes.

Pendant tout ce traitement nous vérifions la validité de l'instance. C'est-à-dire, l'instance doit
- être un document valide vis à vis du langage cible,
- être conforme aux contraintes exprimés dans le template et
- respecter les contraintes données par les éléments XTiger.

Cette dernière vérification est nécessaire car l'utilisateur peut avoir modifié l'instance avec des outils autres qu'*Amaya*.

L'édition à l'aide des templates `XTiger` ajoute des fonctionnalités nouvelles par rapport à l'édition `XHTML`. Ces fonctionnalités doivent être disponibles pour l'utilisateur de la façon la plus homogène possible avec les mécanismes déjà existants pour faciliter son apprentissage.

Les nouvelles fonctionnalités dont l'utilisateur peut se servir pour éditer les instances sont les suivantes.

L'édition des attributs

Les attributs peuvent être édités à l'aide du même mécanisme que celui utilisé lors de l'édition des documents XHTML classiques. Ce mécanisme consiste à montrer sur un panneau latéral tous les attributs de l'élément sélectionné, en cliquant sur un attribut nous pouvons lui donner une valeur qui peut être restreinte d'après la DTD du langage.

Par exemple, la figure 7.9 montre ce panneau. Dans la figure, l'élément sélectionné est un en-tête `H1`. Nous montrons l'attribut `dir`, qui d'après la `DTD` de `XHTML` ne peut avoir que les valeurs `ltr` ou `rtl`[5]. Nous voyons que les valeurs sont montrées comme seules valeurs possibles pour l'utilisateur.

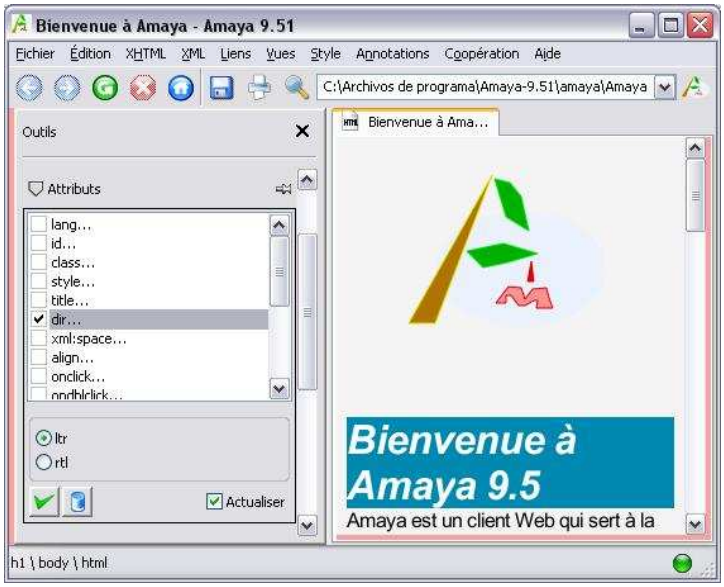

FIG. 7.9 – La barre latérale pour l'édition d'attributs peut être utilisée pour l'édition restreinte des attributs

[5] `ltr` (left-to-right) et `rtl` (right-to-left) sont utiles pour déterminer le sens d'écriture de la langue utilisée

C'est justement ce mécanisme que nous pouvons augmenter pour gérer les nouvelles contraintes. Nous voulons donc pouvoir :

- *Interdire un attribut.* L'attribut est montré en grisé, pour montrer qu'il n'est pas disponible.
- *Limiter les valeurs possibles.* Les valeurs sont choisies parmi une liste comme celle montré dans la figure 7.9.
- *Typer les valeurs.* Les valeurs peuvent être contrôlées *a posteriori* et nous pouvons communiquer une erreur à l'utilisateur si la valeur rendue n'est pas du type défini.
- *Forcer l'utilisation d'un attribut.* Amaya sait déjà signaler ce type d'erreurs quand un attribut obligatoire d'un élément est omis. Le point vert au coin inférieur droit devient rouge pour indiquer qu'il y a des problèmes de ce type dans le document.

Le choix du type d'un élément `use`

Les éléments `t:use` permettent à l'utilisateur faire un choix parmi plusieurs types à utiliser. Amaya fournit un panneau latéral pour l'édition XML qui donne la liste de types `XML` que l'utilisateur peut créer à un endroit précis du document. Ce panneau, montré par la figure 7.10, pourrait nous être utile pour fournir un mécanisme de sélection du type pour les `t:use`.

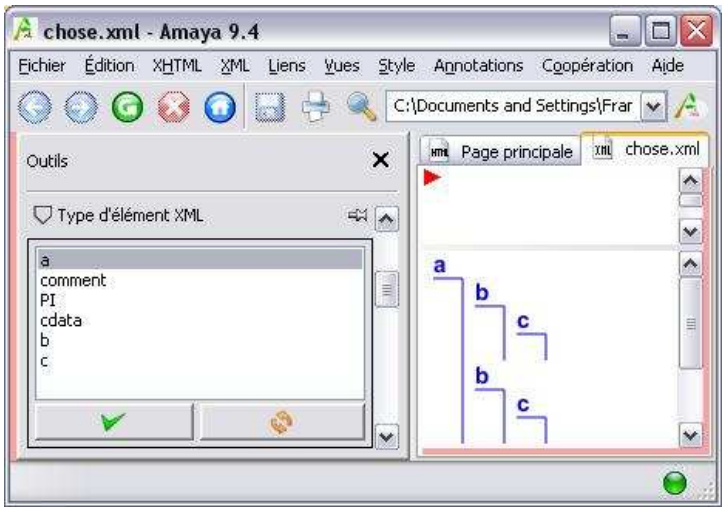

FIG. 7.10 – Le panneau de types d'éléments XML qui permet d'insérer des nouveaux éléments dans un document XML.

Néanmoins, cette solution n'est pas la meilleure. Nous pouvons imaginer la taille de cette liste pour un `t:use` de type `any`. Elle serait trop longue pour que l'utilisateur puisse l'utiliser d'une façon confortable. Notre solution est d'utiliser un menu contextuel hiérarchique attaché à l'élément `t:use`.

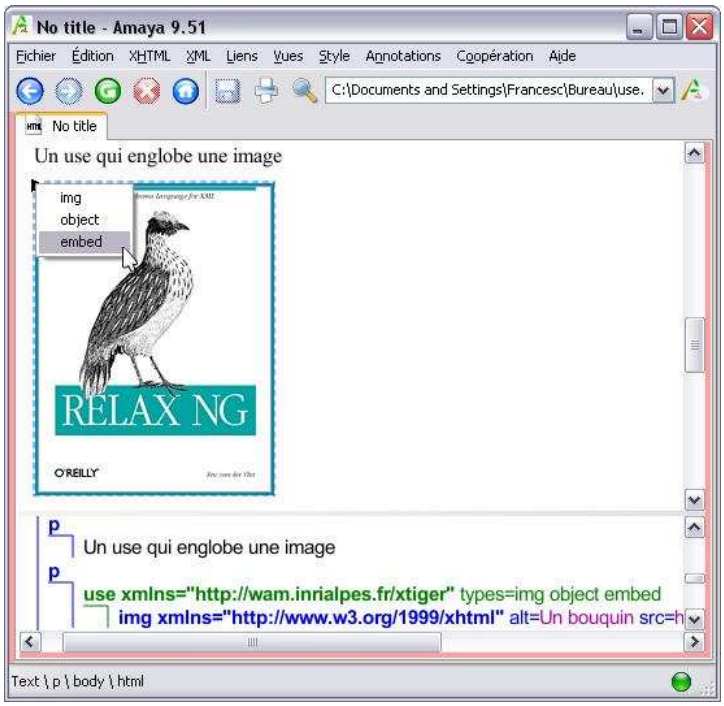

FIG. 7.11 – Le menu des éléments `use` permet de choisir le type utilisé parmi les autorisés.

Le menu que nous proposons (fig. 7.11) est une hiérarchie de menus. Chaque union est un sous-menu et contient les types qui en font partie. Cela permet de choisir d'une façon plus efficace le type recherché.

L'utilisation de composants dans des `bag`

Nous affrontons deux problèmes pour l'édition du contenu des `t:bag` : la limitation des éléments du langage cible utilisables et la proposition des composants du template que l'utilisateur peut insérer.

Limitation des mécanismes disponibles Pour limiter les éléments disponibles nous devons contrôler l'utilisation des mécanismes correspondants. Amaya offre à l'utilisateur deux types de mécanismes d'édition :

les mécanismes d'édition directes

Ces mécanismes sont ceux qui permettent l'édition directe des éléments à l'aide basiquement du clavier. Par exemple, nous pouvons créer des nouveaux paragraphes, diviser un paragraphe en deux, répéter des

éléments d'une liste, etc. en utilisant la touche `ENTER`, supprimer des éléments avec la touche `SUPR`, etc.

les mécanismes d'édition indirectes
Ces mécanismes permettent une édition indirecte des éléments via des menus, des buttons, etc. Par exemple, nous pouvons créer des en-têtes à l'aide du menu `XHTML` → `En-tête` → `Titre niveau 1 (h1)`. Mais nous pouvons aussi le créer à l'aide du bouton `H1` dans la barre d'outils `XHTML`.

Quand une fonctionnalité n'est pas disponible, si le mécanisme pour l'activer est direct nous ne pouvons pas éviter que l'utilisateur essaye de l'utiliser. Dans ce cas, il nous faut notifier *a posteriori* l'impossibilité d'utiliser la fonctionnalité.

Par contre, quand le mécanisme pour déclencher la fonctionnalité est indirect, nous pouvons éviter que l'utilisateur l'utilise en désactivant le menu et le bouton correspondant.

Proposition des composants Les composants définis par les templates ne sont pas disponibles à travers d'aucun menu d'Amaya. Nous devons donc ajouter un mécanisme pour que l'utilisateur puisse les insérer aux endroits où ils sont autorisés.

Pour cela nous pouvons soit enrichir la barre de menus d'Amaya, soit utiliser la barre de types d'éléments montrée dans la figure 7.10, soit proposer un menu contextuel lié au `t:bag`.

La première solution n'est pas pratique car en plus d'ajouter un menu à Amaya, premièrement, ce menu est inutile la plupart du temps. Deuxièmement, la liste des composants n'est pas disponible en permanence et cela peut être moins confortable pour l'utilisateur.

Une alternative serait de proposer un menu contextuel sur le bouton droit à partir de la position courante. Ce mécanisme pose les mêmes problèmes que le précédent.

Le menu contextuel comme dans le cas du `t:use` (fig. 7.11) pose principalement deux problèmes. Premièrement, il oblige l'utilisateur à aller jusqu'à en haut du `t:bag` pour insérer un composant. Il se peut que le haut du `t:bag` ne soit pas visible du point d'insertion. Deuxièmement, l'utilisateur pourrait cliquer sur le menu contextuel d'un `t:bag` alors que le point d'insertion courant n'est pas dans ce `t:bag`.

Nous proposons donc d'utiliser le panneau latéral de types d'éléments. Ce panneau, qui n'est pas utilisé pour l'édition `XHTML`, peut être utilisé

pour montrer les composants disponibles en fonction du point d'insertion. Néanmoins, nous proposons une amélioration du panneau actuel pour pouvoir gérer les unions. Cela consiste à convertir la liste en un arbre repliable.

Chapitre 8

Conclusion

L'objectif du stage était de concevoir un langage pour la description de templates `XML` et de mettre en place un système de templates `XHTML` dans Amaya. À l'issue de ces six mois, d'une part un nouveau langage appelé XTiger, pour *eXtensible Templates for the Interactive and Guided Edition of Resources*, est créé. D'autre part, un prototype partiel pour l'édition des instances `XHTML` à l'aide de templates `XTiger` a été implémenté dans *Amaya*.

8.1 Résultats du stage

XTiger, notre nouveau langage, a été créé pour aider les auteurs créer des pages web sémantiquement riches en se basant sur les concepts du XHTML sémantique et des microformats. Le langage est né de la combinaison de plusieurs propriétés observées dans plusieurs langages différents, comme les langages de définition de template, les langages de schéma XML, et les langages de transformation.

XTiger permet la création de pages web sémantiquement plus riches, mais il assure aussi que les instances produites seront valides. Le fait d'avoir un langage simple mais avec une description formalisée rend possible le calcul de plusieurs propriétés des documents, la validité entre autre, et facilite l'implémentation des divers processus.

Nous avons pris le compromis de rendre le langage le plus simple possible. Nous avons donc créé un langage de seulement neuf éléments. Le langage fonctionne sur une hiérarchie de types où nous trouvons des types simples, des éléments XML, des composants et enfin des unions des trois précédents.

Nous pouvons créer des nouveaux composants (`t:component`) et des nouvelles unions (`t:union`). Pour cela, nous devons utiliser des déclarations, qui peuvent appartenir au document (`t:head`) ou former des ressources à part (`t:library`) utilisés par les templates (`t:import`).

Dans le corps du template nous pouvons définir des zones répétables (`t:repeat`) et optionnelles (`t:option`). Nous pouvons aussi créer des zones où les types définis au préalable peuvent être utilisés (`t:use` et `t:bag`). Nous pouvons enfin contrôler l'utilisation des attributs du langage cible (`t:attribute`).

Malgré la simplicité du langage, il permet un contrôle de l'édition pour une très grande variété de structures, depuis les structures très générales des grands documents jusqu'aux détails les plus petits nécessaires aux microformats.

Pour montrer l'utilité du langage nous en avons implémenté quelques fonctionnalités dans le logiciel d'édition Amaya. Même si nous n'avons pas réalisé une implémentation complète nous avons analysé les problèmes que cela pourrait poser et nous avons proposé des solutions.

Les fonctionnalités implémentées jusqu'à présent sont la création d'instances à partir d'un template et le chargement des instances dans l'application. Nous avons aussi implémenté l'interface utilisateur, mais nous n'avons pas encore terminé l'implémentation des traitements liés aux éléments `XTiger`.

8.2 Travaux futurs

D'abord, nous continuons le stage pour compléter l'implémentation du prototype d'édition des instances dans Amaya. Ensuite, nous considérons quelques améliorations du langage qui, en conservant sa simplicité, le rendraient plus puissant.

Nous proposons l'ajout d'un mécanisme de création de types simples similaire à celui existant dans `XML Schéma` ou dans `RelaxNG`. Nous envisageons aussi la fusion des types simples et des types des attributs. Pour ces deux améliorations, la seule modification à apporter au langage serait l'ajout de nouveaux éléments pour la déclarations des types simples et la redéfinition de l'élément `t:attribute`.

Pendant le stage nous nous sommes rendus compte que la génération de contenu était une partie importante qu'il faudrait exploiter dans notre langage. Par exemple, les références bibliographiques d'un article devraient générer toutes seules le contenu à partir des information de l'œouvre référencée.

Pour cela nous proposons l'ajout de trois nouveaux éléments. Le premier des éléments est `t:copy` qui permet de définir une zone dont le contenu est la copie d'une autre. La structure copiée est définie à l'aide du langage `XPath`[7]. Les deux autres éléments permettent la création de contenu grâce à l'utilisation de variables `t:var` et de références `t:ref`, qui pourront être utilisées dans les expressions `XPath` du `t:copy`.

8.3 Connaissances tirées du stage

Ce stage m'a permis de participer à un projet de recherche de grande taille comme Amaya. Les différences entre un projet de ce type et un projet orienté industrie sont énormes. D'abord on est beaucoup plus libre d'expérimenter et donc de se tromper que dans les projets industriels. Cependant, l'autonomie est très importante pour arriver à un bon résultat. Cela peut paraître banal, mais la motivation et la capacité de prendre du recul sont deux qualités très importantes pour un chercheur.

J'ai decouvert l'architecture d'Amaya. Elle est très intéressante car on utilise des librairies très variées (*Thot*, *openGL*, *libWWW*, ...) mais aussi parce qu'elle est portable. C'est-à-dire qu'elle est disponible dans plusieurs systèmes d'exploitation. La création de ce type d'applications est complexe et demande un très grand effort sur partie de vérification des modifications.

En revanche, dans les projets industriels que j'ai fait jusqu'à présent l'ambiance était beaucoup plus dynamique, et le droit à l'erreur beaucoup moins répandu. Ce stage m'a servi donc pour me faire une très bonne idée du monde de la recherche et, plus concrètement, du travail de chercheur.

De plus, j'ai compris l'intérêt des normes développées tantôt par le W3C, tantôt par d'autres organismes de plus petite taille. Ces normes sont présentes sous la forme de langages comme `XML`, `XHTML`, `SVG`, etc., mais aussi sous la forme de conventions de taille plus modeste : les *microformats*.

J'ai decouvert l'utilité des *microformats* pour l'introduction de sémantique dans le web. Ils forment un chemin très pragmatique vers le web sémantique, en utilisant ce qui existe déjà. Cette approche est très bien adaptée à la réalité car, non seulement permet un développement plus rapide, mais aussi une diffusion beaucoup plus effective.

Les utilisations possibles des *microformats* sont infinies. Ils vont très probablement devenir un point très important dans le développement du web, comme l'est `AJAX`[5] à présent. Il faut donc profiter de cette nouvelle vague technologique et faire d'*Amaya* le premier éditeur `XHTML` offrant des fonctionnalités intéressantes pour les microformats.

Bibliographie

[1] *Javascript : the definitive guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[2] *OpenOffice.org2.x User Guide*. 2005. http ://documentation.openoffice.org/manuals/index.html.

[3] Worldwide internet users top 1 billion in 2005. *Computer Industry Almanac Inc.* (2006). http ://www.c-i-a.com/pr0106.htm.

[4] Altheim, M., Boumphrey, F., Dooley, S., McCarron, S., Schnitzenbaumer, S., and Wugofski, T. Xhtml modularization 1.1, February 2006. http ://www.w3.org/TR/xhtml-modularization/.

[5] Asleson, R., and Schutta, N. T. *Foundations of Ajax (Foundation)*. Apress, Berkely, CA, USA, 2005.

[6] Bergsten, H. *Javaserver Pages*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[7] Clark, J., and DeRose, S. Xml path language (xpath), November 1999. http ://www.w3.org/TR/xpath.

[8] Feddema, H. *Microsoft Access Version 2002 Inside Out*. Microsoft Press, Redmond, WA, USA, 2001.

[9] Fitzgerald, M. *Learning XSLT*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2003.

[10] Gundavaram, S. *CGI programming on the World Wide Web*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[11] Harold, E. R., and Means, W. S. *XML in a Nutshell : A Desktop Quick Reference*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[12] ISHIKAWA, M. An xhtml + mathml + svg profile, August 2000. http ://www.w3.org/TR/XHTMLplusMathMLplusSVG/xhtml-math-svg.html.

[13] Khare, R. Microformats : The next (small) thing on the semantic web ? *IEEE Internet Computing 10*, 1 (2006), 68–75.

[14] Kittel, Michael A., L. G. T. *ASP .NET Cookbook*. O'Reilly, 2004.

[15] Lee, D., and Chu, W. W. Comparative analysis of six xml schema languages. *SIGMOD Rec. 29*, 3 (2000), 76–87.

[16] Lee, T. B. Semantic web road map. http ://www.w3.org/DesignIssues/Semantic.html.

[17] Lerdorf, R. J., Tatroe, K., Kaehms, B., and McGredy, R. *Programming Php*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[18] Lowery, J. W. *Dreamweaver MX Bible*. John Wiley & Sons, Inc., New York, NY, USA, 2002. Foreword By-David Deming.

[19] Meyer, E. A. *Cascading Style Sheets : The Definitive Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[20] Millhollon, M., and Murray, K. *Microsoft Word Version 2002 Inside Out*. Microsoft Press, Redmond, WA, USA, 2001.

[21] Min, J.-K., Ahn, J.-Y., and Chung, C.-W. Efficient extraction of schemas for xml documents. *Inf. Process. Lett. 85*, 1 (2003), 7–12.

[22] Quint, V., and Vatton, I. Techniques for authoring complex xml documents. In *DocEng '04 : Proceedings of the 2004 ACM symposium on Document engineering* (New York, NY, USA, 2004), ACM Press, pp. 115–123.

[23] Riley, D. Number of blogs now exceeds 50 million worldwide. *The Blog Herald* (2005). http ://www.blogherald.com/2005/04/14/number-of-blogs-now-exceeds-50-million-worldwide/.

[24] Schmitt, C., and Meyer, E. A. *Designing CSS Web Pages*. Pearson Education, 2002.

[25] Stinson, C. *Microsoft Excel Version 2002 Inside Out*. Microsoft Press, Redmond, WA, USA, 2001.

[26] Tantek Çelik, E. M., and Mullenweg, M. Xhtml friends network. http ://gmpg.org/xfn.

[27] Tim Bray, Dave Hollander, A. L. R. T. Namespaces in xml 1.1. World Wide Web Consortium. http ://www.w3.org/TR/xml-names11/.

[28] van der Vlist, E. *XML Schema*. O'Reilly Media, Inc., 2002.

[29] van der Vlist, E. *RELAX NG*. O'Reilly Media, Inc., 2003.

[30] van der Vlist, E. Validating microformats, May 2006. http ://eric.van-der-vlist.com/blog/2277_Validating_microformats.item.

[31] Walsh, N. Validating microformats, May 2006. http ://norman.walsh.name/2006/04/13/validatingMicroformats.

Annexe A

DTD du langage XTiger

```
<!ELEMENT head ((component | union | import)*)>

<!ELEMENT component ANY>
<!ATTLIST component
   name              NMTOKEN           #REQUIRED
   title             CDATA             #REQUIRED>

<!ELEMENT union EMPTY>
<!ATTLIST union
   name              NMTOKEN           #REQUIRED
   include           NMTOKENS          #REQUIRED
   exclude           NMTOKENS          #IMPLIED
   title             CDATA             #REQUIRED>

<!ELEMENT import EMPTY>
<!ATTLIST import
   src               CDATA             #REQUIRED>

<!ELEMENT option ANY>
<!ATTLIST option
   id                CDATA             #REQUIRED
   checked           (true|false)      #IMPLIED "true">

<!ELEMENT repeat ANY>
<!ATTLIST repeat
   id                CDATA             #REQUIRED
   minOccurs         CDATA             #IMPLIED "0"
   maxOccurs         CDATA             #IMPLIED "*"
   currentOccurs     CDATA             #IMPLIED
   title             CDATA             #REQUIRED>

<!ELEMENT use ANY>
<!ATTLIST use
   id                CDATA             #REQUIRED
   types             NMTOKENS          #REQUIRED
   currentType       NMTOKEN           #REQUIRED
   title             CDATA             #REQUIRED>

<!ELEMENT bag ANY>
<!ATTLIST bag
```

```
   id                CDATA              #REQUIRED
   types             NMTOKENS           #REQUIRED
   title             CDATA              #REQUIRED>

<!ELEMENT attribute EMPTY>
<!ATTLIST attribute
   name              NMTOKEN            #REQUIRED
   type              (number, string, list)
                                        #IMPLIED "string"
   use               (required, optional, prohibited)
                                        #IMPLIED "required"
   default           CDATA              #IMPLIED
   fixed             CDATA              #IMPLIED
   values            CDATA              #IMPLIED
   title             CDATA              #REQUIRED>
```

Annexe B

Fichiers de description pour Thot

B.1 Descripteur d'application Template.A

```
APPLICATION Template;

DEFAULT
   BEGIN
          ElemActivate.Pre -> DoubleClick;
          ElemClick.Pre -> SimpleClick;
          ElemRClick.Pre -> SimpleRClick;
          ElemLClick.Pre -> SimpleLClick;
   END;

ELEMENTS

   useMenu :
      BEGIN
         ElemClick.Pre -> UseMenuClicked;
      END;
   bagMenu :
      BEGIN
         ElemClick.Pre -> BagMenuClicked;
      END;

END
```

B.2 Descripteur de structure Template.S

```
      { Structure Schema for XTiger Templates
        Francesc Campoy Flores     March 2006 }

STRUCTURE Template;
```

```
DEFPRES TemplateP;

ATTR
  { generic attributes for internal use }
  Unknown_attribute = TEXT;        { to store an unknown attribute }

CONST

  C_CR = '\12';

STRUCT

  Template = CASE OF
      head;
      repeat;
      option;
      useEl;
      bag;
      attribute;
    END;

  head = LIST OF (
    Declaration = CASE OF
      component;
      union;
      import;
    END
   );

  component ( ATTR name = TEXT; title = TEXT ) =
    ANY - ( head, union, import );

  union ( ATTR name; includeAt = TEXT; exclude = TEXT; title ) =
    CONSTANT C_CR; {It is always empty}

  import ( ATTR src = TEXT ) = CONSTANT C_CR; {It is always empty}

  folder =
  BEGIN
    ANY - ( head, union, import, component );
  END;

  option ( ATTR id = TEXT; checked = TEXT ) =
  BEGIN
    ANY - ( head, union, import, component );
    optionMenu;
  END;

  repeat ( ATTR id; minOccurs = TEXT; maxOccurs = TEXT;
           currentOccurs = TEXT; title ) =
  LIST OF ( Repetition =
    CASE OF
      useEl;
      option;
      repeat;
      bag;
      folder;
      repeatMenu;
      END
    );
```

```
  useEl ( ATTR id; types = TEXT; currentType = TEXT; title ) =
  BEGIN
    ANY;
    useMenu;
  END - ( head, union, import, component );

  bag ( ATTR id; types; title ) =
  BEGIN
    ANY;
    bagMenu;
  END - ( head, union, import, component );

  useMenu      = CONSTANT C_CR; {Just a button}
  bagMenu      = CONSTANT C_CR; {Just a button}
  repeatMenu   = CONSTANT C_CR; {Just a button}
  optionMenu   = CONSTANT C_CR; {Just a button}

  attribute ( ATTR
    name;
    type      = integerVal, decimal, string, listVal;
    useAt     = TEXT;
    defaultAt = TEXT;
    fixed     = TEXT;
    values    = TEXT;
    title ) = CONSTANT C_CR; {It is always empty}

EXCEPT
    {Hiding unkown and language attributes}
    Unknown_attribute   : Invisible;
    Language            : Invisible;

    {Element buttons are not editable}
    useMenu            : NoCut, NoCreate, NoMove, NoResize, NoSelect;
    bagMenu            : NoCut, NoCreate, NoMove, NoResize, NoSelect;
    optionMenu  : NoCut, NoCreate, NoMove, NoResize, NoSelect;
    repeatMenu  : NoCut, NoCreate, NoMove, NoResize, NoSelect;

END
```

B.3 Descripteur de traduction TemplateT.T

```
    { Translation Schema for XTiger Templates
      Francesc Campoy Flores       March 2006 }


TRANSLATION Template;

LINELENGTH 78;

BUFFERS
  ElemPrefixBuffer (variable);
  AttrPrefixBuffer (variable);

CONST
  DoubleQuote = '"';
  CloseTagNL = '>\12';
  CloseEmptyTagNL = '/>\12';
  tagend = '>\12';
```

```
VAR
  SOT : '<' ElemPrefixBuffer;   { Start of opening tag }
  SCT : '</' ElemPrefixBuffer;  { Start of closing tag }
  SAN : ' ' AttrPrefixBuffer;   { Start of attribute name }
  SUA : '';                     { Start of unknown attribute }

RULES
  Template: begin
    Use HTMLTX for HTML;
    Use XLinkT for XLink;
    Use SVGT for SVG;
    Use MathMLT for MathML;
  end;

  head:
    begin
    create SOT;
    create 'head';
    create Attributes;
    create tagend;
    indent +2;
    indent -2 after;
    create SCT after;
    create 'head>\12' after;
  end;

  component:
    begin
    create SOT;
    create 'component';
    create Attributes;
    create tagend;
    indent +2;
    indent -2 after;
    create SCT after;
    create 'component>\12' after;
  end;

  union:
    begin
    create SOT;
    create 'union';
    create Attributes;
    create tagend;
    indent +2;
    indent -2 after;
    create SCT after;
    create 'union>\12' after;
  end;

  import:
    begin
    create SOT;
    create 'import';
    create Attributes;
    create CloseEmptyTagNL;
  end;

  repeat:
    begin
    create SOT;
```

```
    create 'repeat';
    create Attributes;
    create tagend;
    indent +2;
    indent -2 after;
    create SCT after;
    create 'repeat>\12' after;
  end;

  option:
    begin
    create SOT;
    create 'option';
    create Attributes;
    create tagend;
    indent +2;
    indent -2 after;
    create SCT after;
    create 'option>\12' after;
  end;

  useEl:
    begin
    create SOT;
    create 'use';
    create Attributes;
    create '>';
    indent +2;
    indent -2 after;
    create SCT after;
    create 'use>' after;
  end;

  bag:
    begin
    create SOT;
    create 'bag';
    create Attributes;
    create tagend;
    indent +2;
     indent -2 after;
    create SCT after;
    create 'bag>\12' after;
  end;

  attribute:
    begin
    create SOT;
    create 'attribute';
    create Attributes;
    create CloseEmptyTagNL;
  end;

ATTRIBUTES

  name :
    begin
      create SAN;
      create 'name=';
      create DoubleQuote;
      create name;
      create DoubleQuote;
```

```
  end;

title :
  begin
    create SAN;
    create 'title=';
    create DoubleQuote;
    create title;
    create DoubleQuote;
  end;

includeAt :
  begin
    create SAN;
    create 'include=';
    create DoubleQuote;
    create includeAt;
    create DoubleQuote;
  end;

exclude :
  begin
    create SAN;
    create 'exclude=';
    create DoubleQuote;
    create exclude;
    create DoubleQuote;
  end;

src :
  begin
    create SAN;
    create 'src=';
    create DoubleQuote;
    create src;
    create DoubleQuote;
  end;

minOccurs :
  begin
    create SAN;
    create 'minOccurs=';
    create DoubleQuote;
    create minOccurs;
    create DoubleQuote;
  end;

maxOccurs :
  begin
    create SAN;
    create 'maxOccurs=';
    create DoubleQuote;
    create maxOccurs;
    create DoubleQuote;
  end;

types :
  begin
    create SAN;
    create 'types=';
    create DoubleQuote;
    create types;
```

```
    create DoubleQuote;
  end;

currentType :
  begin
    create SAN;
    create 'currentType=';
    create DoubleQuote;
    create currentType;
    create DoubleQuote;
  end;

type :
  begin
    create SAN;
    create 'type=';
    create DoubleQuote;
    create type;
    create DoubleQuote;
  end;

useAt :
  begin
    create SAN;
    create 'use=';
    create DoubleQuote;
    create useAt;
    create DoubleQuote;
  end;

defaultAt :
  begin
    create SAN;
    create 'default=';
    create DoubleQuote;
    create defaultAt;
    create DoubleQuote;
  end;

fixed :
  begin
    create SAN;
    create 'fixed=';
    create DoubleQuote;
    create fixed;
    create DoubleQuote;
  end;

values :
  begin
    create SAN;
    create 'values=';
    create DoubleQuote;
    create values;
    create DoubleQuote;
  end;

Unknown_attribute:
  begin
    if Template begin
      remove;
    end;
```

```
      if NOT Template begin
        create SAN;
    create Unknown_attribute;
      end;
    end;

END
```

B.4 Descripteur de présentation TemplateP.P

```
        { Presentation Schema for Templates.
        Francesc Campoy Flores  May 2006 }

PRESENTATION Template;

VIEWS
   Formatted_view,
   Structure_view merge with Structure_view;

#define     STRUCT_INDENT       1.5
#define     TAG_Font            Helvetica
#define     TAG_Color           DarkGreen3
#define     TAG_Weight          Bold
#define     TAG_Visibility      7
#define     TAG_FONTSIZE        11 pt
#define     ATTR_Font           Helvetica
#define     ATTR_Style          Roman
#define     ATTR_Weight         Normal
#define     ATTR_Visibility     6
#define     INVALID_Style       Roman
#define     INVALID_Weight      Normal
#define     INVALID_Visibility  7
#define     INVALID_Font        Helvetica
#define     ATTR_Color          DarkGreen3
#define     ATTRVALUE_Color     DarkMagenta2
#define     INVALID_Color       Red

DEFAULT BEGIN
        Visibility              :   Enclosing =;
        VertRef                 :   * . Left ;
        HorizRef                :   Enclosed . HRef;
        Height                  :   Enclosed . Height;
        Width                   :   Enclosed . Width;
        VertPos                 :   HRef = Previous . HRef;
        HorizPos                :   Left = Previous . Right;
        LineBreak               :   No;
        Font                    :   Enclosing =;
        Style                   :   Roman;
        Weight                  :   Enclosing =;
        Size                    :   Enclosing =;
        Indent                  :   0;
        Adjust                  :   Left;
        LineSpacing             :   1;
        Depth                   :   Enclosing =;
        UnderLine               :   Enclosing =;
        Thickness               :   Enclosing =;
        LineStyle               :   Enclosing =;
        LineWeight              :   Enclosing =;
```

```
        Background              :   Enclosing =;
        Foreground              :   Enclosing =;
        FillPattern             :   Enclosing =;
        IN Structure_view           BEGIN
            VertPos             :   Top = Previous . Bottom;
            HorizPos            :   Left = Enclosing . Left;
        END;
    END;

BOXES

    AttrName                    :   BEGIN
        Content                 :   (Text ' ' AttributeName Text '=');
        Visibility              :   0;
        IN Structure_view           BEGIN
            HorizPos            :   Left = Previous . Right;
            VertPos             :   HRef = Previous . HRef;
            Size                :   TAG_FONTSIZE;
            Font                :   ATTR_Font;
            Style               :   ATTR_Style;
            Weight              :   ATTR_Weight;
            Visibility          :   ATTR_Visibility;
            Foreground          :   ATTR_Color;
        END;
    END;

    AttrValue                   :   BEGIN
        Content                 :   (AttributeValue);
        Visibility              :   0;
        IN Structure_view           BEGIN
            HorizPos            :   Left = Previous . Right;
            VertPos             :   HRef = Previous . HRef;
            Size                :   TAG_FONTSIZE;
            Font                :   ATTR_Font;
            Style               :   ATTR_Style;
            Weight              :   ATTR_Weight;
            Visibility          :   ATTR_Visibility;
            Foreground          :   ATTRVALUE_Color;
        END;
    END;

    AttrNameAndValue            :   BEGIN
        Content : (Text ' ' AttributeName Text '=' attributeValue);
        Visibility              :   0;
        IN Structure_view           BEGIN
            HorizPos            :   Left = Previous . Right;
            VertPos             :   HRef = Previous . HRef;
            Size                :   TAG_FONTSIZE;
            Font                :   ATTR_Font;
            Style               :   ATTR_Style;
            Weight              :   ATTR_Weight;
            Visibility          :   ATTR_Visibility;
            Foreground          :   ATTR_Color;
        END;
    END;

    ElementName                 :   BEGIN
        Content                 :   (ElemName);
        Visibility              :   0;
        IN Structure_view           BEGIN
            Size                :   TAG_FONTSIZE;
            Font                :   TAG_Font;
```

```
        Weight              :   TAG_Weight;
        Visibility          :   TAG_Visibility;
        Foreground          :   TAG_Color;
    END;
END;

VerticalLine                :   BEGIN
    Content                 :   Graphics 'W';
    Visibility              :   0;
    IN Structure_view           BEGIN
        VertPos             :   Top = Creator . Top;
        Height              :   Creator . Height;
        Width               :   1.1;
        LineWeight          :   1 px;
        Visibility          :   TAG_Visibility;
        Foreground          :   TAG_Color;
    END;
END;

UnknownAttrValue            :   BEGIN
    Content                 :   (AttributeValue);
    Visibility              :   0;
    IN Structure_view           BEGIN
        HorizPos            :   Left = Previous . Right;
        VertPos             :   HRef = Previous . HRef;
        Size                :   TAG_FONTSIZE;
        Font                :   INVALID_Font;
        Style               :   INVALID_Style;
        Visibility          :   INVALID_Visibility;
        Foreground          :   INVALID_Color;
    END;
END;

Unknown_End                 :   BEGIN
    Content                 :   Text '>';
    Visibility              :   0;
    IN Structure_view           BEGIN
        HorizPos            :   Left = Previous . Right;
        VertPos             :   HRef = Previous . HRef;
        Size                :   TAG_FONTSIZE;
        Style               :   INVALID_Style;
        Visibility          :   INVALID_Visibility;
        Foreground          :   Creator =;
    END;
END;

UseMenu_Gif                 :   BEGIN
    Content                 :   Picture 'xtigeruse.gif';
    HorizPos                :   Left = Enclosing . Left;
    VertPos                 :   Top  = Enclosing . Top;
    MarginTop               :   0 px;
    MarginBottom            :   0 px;
    MarginLeft              :   0 px;
    MarginRight             :   0 px;
    Depth                   :   Creator - 2;
    IN Structure_view           BEGIN
        Visibility          :   0;
    END;
END;

BagMenu_Gif                 :   BEGIN
    Content                 :   Picture 'xtigerbag.gif';
```

```
        HorizPos                :   Left = Enclosing . Left;
        VertPos                 :   Top  = Enclosing . Top;
        MarginTop               :   0 px;
        MarginBottom            :   0 px;
        MarginLeft              :   0 px;
        MarginRight             :   0 px;
        Depth                   :   Creator - 2;
        IN Structure_view           BEGIN
            Visibility          :   0;
        END;
    END;

    OptionMenu_Gif              :   BEGIN
        Content                 :   Picture 'xtigeroption.gif';
        HorizPos                :   Left = Enclosing . Left;
        VertPos                 :   Top = Enclosing . Top;
        MarginTop               :   0 px;
        MarginBottom            :   0 px;
        MarginLeft              :   0 px;
        MarginRight             :   0 px;
        Depth                   :   Creator - 2;
        IN Structure_view           BEGIN
            Visibility          :   0;
        END;
    END;

    RepeatMenu_Gif              :   BEGIN
        Content                 :   Picture 'xtigerrepeat.gif';
        HorizPos                :   Left = Enclosing . Left;
        VertPos                 :   Top = Enclosing . Top;
        MarginTop               :   0 px;
        MarginBottom            :   0 px;
        MarginLeft              :   0 px;
        MarginRight             :   0 px;
        Depth                   :   Creator - 2;
        IN Structure_view           BEGIN
            Visibility          :   0;
        END;
    END;

RULES

    head                        :   BEGIN
        Visibility              :   0;
        CreateBefore                (ElementName);
        CreateWith                  (VerticalLine);
        IN Structure_view           BEGIN
            HorizPos : Left     =   Enclosing . Left + STRUCT_INDENT;
            VertPos  : Top      =   Previous  . Bottom;
            Visibility          :   Enclosing =;
            Width               :   Enclosing . Width - STRUCT_INDENT;
        END;
    END;

    component                   :   BEGIN
        Visibility              :   0;
        CreateBefore                (ElementName);
        CreateWith                  (VerticalLine);
        IN Structure_view           BEGIN
            HorizPos            :   Left = Enclosing . Left
                                    + STRUCT_INDENT;
            VertPos             :   Top  = Previous  . Bottom;
```

```
        Visibility          :   Enclosing =;
        Width               :   Enclosing . Width - STRUCT_INDENT;
    END;
END;

union                       :   BEGIN
    Visibility              :   0;
    CreateBefore                (ElementName);
    CreateWith                  (VerticalLine);
    IN Structure_view           BEGIN
        Visibility          :   Enclosing =;
        Width               :   Enclosing . Width - STRUCT_INDENT;
        HorizPos : Left     =   Enclosing . Left + STRUCT_INDENT;
        VertPos : Top       =   Previous  . Bottom;
    END;
END;

attribute                   :   BEGIN
    Visibility              :   0;
    CreateBefore                (ElementName);
    CreateWith                  (VerticalLine);
    IN Structure_view           BEGIN
        HorizPos : Left     =   Enclosing . Left + STRUCT_INDENT;
        VertPos  : Top      =   Previous  . Bottom;
        Visibility          :   Enclosing =;
        Width               :   Enclosing . Width - STRUCT_INDENT;
    END;
END;

folder                      :   BEGIN
    Width                   :   auto{Enclosing . Width};
    Height                  :   auto{Enclosing . Height};
    {Display                :   Block;}
    MarginTop               :   3 px;
    VertPos                 :   Top     =   Previous . Bottom;
    HorizPos                :   Left    =   Enclosing . Left;
    BorderTopWidth          :   1 px;
    BorderLeftWidth         :   1 px;
    BorderBottomWidth       :   1 px;
    BorderRightWidth        :   1 px;
    BorderTopStyle          :   Dashed;
    BorderLeftStyle         :   Dashed;
    BorderBottomStyle       :   Dashed;
    BorderRightStyle        :   Dashed;
    BorderTopColor          :   Violet;
    BorderLeftColor         :   Violet;
    BorderBottomColor       :   Violet;
    BorderRightColor        :   Violet;
    PaddingRight            :   2 px;
    PaddingLeft             :   2 px;
    PaddingTop              :   2 px;
    PaddingBottom           :   2 px;
    IN Structure_view           BEGIN
        BorderTopStyle      :   None;
        BorderLeftStyle     :   None;
        BorderBottomStyle   :   None;
        BorderRightStyle    :   None;
        PaddingRight        :   0 px;
        PaddingLeft         :   0 px;
        PaddingTop          :   0 px;
        PaddingBottom       :   0 px;
        BorderTopWidth      :   0 px;
```

```
        BorderLeftWidth     :   0 px;
        BorderBottomWidth   :   0 px;
        BorderRightWidth    :   0 px;
    END;
END;

bagMenu                     :   BEGIN
    CreateFirst                 (BagMenu_Gif);
    VertPos                 :   Top  =  Enclosing . Top  - 5 px;
    HorizPos                :   Left =  Enclosing . Left - 5 px;
    HorizRef                :   * . Bottom;
    MarginTop               :   0 px;
    MarginBottom            :   0 px;
    MarginRight             :   0 px;
    MarginLeft              :   0 px;
    IN Structure_view           BEGIN
        Visibility          :   0;
    END;
END;

bag                         :   BEGIN
    CreateBefore                (ElementName);
    CreateWith                  (VerticalLine);
    Width                   :   auto{Enclosing . Width};
    Height                  :   auto{Enclosing . Height};
    {Display                :   Block;}
    VertPos    : Top        =   Previous . Bottom;
    HorizPos   : Left       =   Enclosing . Left;
    BorderTopWidth          :   1 px;
    BorderLeftWidth         :   1 px;
    BorderBottomWidth       :   1 px;
    BorderRightWidth        :   1 px;
    BorderTopStyle          :   Dashed;
    BorderLeftStyle         :   Dashed;
    BorderBottomStyle       :   Dashed;
    BorderRightStyle        :   Dashed;
    BorderTopColor          :   Green;
    BorderLeftColor         :   Green;
    BorderBottomColor       :   Green;
    BorderRightColor        :   Green;
    PaddingRight            :   2 px;
    PaddingLeft             :   2 px;
    PaddingTop              :   2 px;
    PaddingBottom           :   2 px;
    IN Structure_view BEGIN
        BorderTopWidth      :   0;
        BorderLeftWidth     :   0;
        BorderBottomWidth   :   0;
        BorderRightWidth    :   0;
        PaddingRight        :   0;
        PaddingLeft         :   0;
        PaddingTop          :   0;
        PaddingBottom       :   0;
        HorizPos  : Left    =   Enclosing . Left + STRUCT_INDENT;
        Width               :   Enclosing . Width - STRUCT_INDENT;
        VertPos   : Top     =   Previous . Bottom;
    END;
END;

useMenu                     :   BEGIN
    CreateFirst                 (UseMenu_Gif);
    VertPos                 :   Top = Enclosing . Top - 5 px;
```

```
    HorizPos: Left          =   Enclosing . Left - 5 px;
    HorizRef                :   * . Bottom;
    MarginTop               :   0 px;
    MarginBottom            :   0 px;
    MarginRight             :   0 px;
    MarginLeft              :   0 px;
    IN Structure_view           BEGIN
        Visibility          :   0;
    END;
END;

useEl                       :   BEGIN
    CreateBefore                (ElementName);
    CreateWith                  (VerticalLine);
    Width                   :   auto{Enclosing . Width};
    Height                  :   auto{Enclosing . Height};
    LineBreak               :   no;
    {Display                :   Block;}
    VertPos  : Top          =   Previous . Bottom;
    HorizPos : Left         =   Enclosing . Left;
    BorderTopWidth          :   1 px;
    BorderLeftWidth         :   1 px;
    BorderBottomWidth       :   1 px;
    BorderRightWidth        :   1 px;
    BorderTopStyle          :   Dashed;
    BorderLeftStyle         :   Dashed;
    BorderBottomStyle       :   Dashed;
    BorderRightStyle        :   Dashed;
    BorderTopColor          :   Blue;
    BorderLeftColor         :   Blue;
    BorderBottomColor       :   Blue;
    BorderRightColor        :   Blue;
    PaddingRight            :   2 px;
    PaddingLeft             :   2 px;
    PaddingTop              :   2 px;
    PaddingBottom           :   2 px;
    IN Structure_view           BEGIN
        BorderTopWidth      :   0;
        BorderLeftWidth     :   0;
        BorderBottomWidth   :   0;
        BorderRightWidth    :   0;
        PaddingRight        :   0;
        PaddingLeft         :   0;
        PaddingTop          :   0;
        PaddingBottom       :   0;
        HorizPos : Left     =   Enclosing . Left + STRUCT_INDENT;
        Width               :   Enclosing . Width - STRUCT_INDENT;
        VertPos  : Top      =   Previous . Bottom;
    END;
END;

repeatMenu                  :   BEGIN
    CreateFirst                 (RepeatMenu_Gif);
    VertPos  : Top          =   Enclosing . Top - 11 px;
    HorizPos : Left         =   Enclosing . Left - 11 px;
    HorizRef                :   * . Bottom;
    MarginTop               :   0 px;
    MarginBottom            :   0 px;
    MarginRight             :   0 px;
    MarginLeft              :   0 px;
    IN Structure_view           BEGIN
        Visibility: 0;
```

```
    END;
END;

repeat                      :   BEGIN
    CreateBefore                (ElementName);
    CreateWith                  (VerticalLine);
    Width                   :   auto{Enclosing . Width};
    {Display                :   Block;}
    MarginTop               :   3 px;
    VertPos  : Top          =   Previous . Bottom;
    HorizPos : Left         =   Enclosing . Left;
    BorderTopWidth          :   2 pt;
    BorderLeftWidth         :   2 pt;
    BorderBottomWidth       :   2 pt;
    BorderRightWidth        :   2 pt;
    BorderTopStyle          :   Dotted;
    BorderLeftStyle         :   Dotted;
    BorderBottomStyle       :   Dotted;
    BorderRightStyle        :   Dotted;
    BorderTopColor          :   Purple;
    BorderLeftColor         :   Purple;
    BorderBottomColor       :   Purple;
    BorderRightColor        :   Purple;
    PaddingRight            :   3 px;
    PaddingLeft             :   8 px;
    PaddingTop              :   8 px;
    PaddingBottom           :   3 px;
    IN Structure_view           BEGIN
        BorderTopWidth      :   0;
        BorderLeftWidth     :   0;
        BorderBottomWidth   :   0;
        BorderRightWidth    :   0;
        PaddingRight        :   0;
        PaddingLeft         :   0;
        PaddingTop          :   0;
        PaddingBottom       :   0;
        HorizPos : Left     =   Enclosing . Left + STRUCT_INDENT;
        VertPos  : Top      =   Previous . Bottom;
        Width               :   Enclosing . Width - STRUCT_INDENT;
    END;
END;

optionMenu                  :   BEGIN
    CreateFirst                 (OptionMenu_Gif);
    VertPos  : Top          =   Enclosing . Top - 5 px;
    HorizPos : Left         =   Enclosing . Left - 5 px;
    HorizRef                :   * . Bottom;
    MarginTop               :   0 px;
    MarginBottom            :   0 px;
    MarginRight             :   0 px;
    MarginLeft              :   0 px;
    IN Structure_view           BEGIN
        Visibility          :   0;
    END;
END;

option                      :   BEGIN
    CreateBefore                (ElementName);
    CreateWith                  (VerticalLine);
    Width                   :   auto{Enclosing . Width};
    {Display                :   Block;}
    MarginTop               :   0;
```

```
        VertPos  : Top          =   Previous . Bottom;
        HorizPos : Left         =   Enclosing . Left;
        BorderTopWidth          :   1 pt;
        BorderLeftWidth         :   1 pt;
        BorderBottomWidth       :   1 pt;
        BorderRightWidth        :   1 pt;
        BorderTopStyle          :   Dashed;
        BorderLeftStyle         :   Dashed;
        BorderBottomStyle       :   Dashed;
        BorderRightStyle        :   Dashed;
        BorderTopColor          :   Orange;
        BorderLeftColor         :   Orange;
        BorderBottomColor       :   Orange;
        BorderRightColor        :   Orange;
        PaddingRight            :   2 px;
        PaddingLeft             :   2 px;
        PaddingTop              :   2 px;
        PaddingBottom           :   2 px;
        IN Structure_view           BEGIN
            BorderTopWidth      :   0;
            BorderLeftWidth     :   0;
            BorderBottomWidth   :   0;
            BorderRightWidth    :   0;
            PaddingRight        :   0;
            PaddingLeft         :   0;
            PaddingTop          :   0;
            PaddingBottom       :   0;
            Width               :   Enclosing . Width - STRUCT_INDENT;
            HorizPos : Left     =   Enclosing . Left + STRUCT_INDENT;
            VertPos  : Top      =   Previous . Bottom;
        END;
    END;

ATTRIBUTES

    name :
    BEGIN
        CreateBefore(AttrName);
        CreateBefore(AttrValue);
    END;

    title :
    BEGIN
        CreateBefore(AttrName);
        CreateBefore(AttrValue);
    END;

    includeAt :
    BEGIN
        CreateBefore(AttrName);
        CreateBefore(AttrValue);
    END;

    exclude :
    BEGIN
        CreateBefore(AttrName);
        CreateBefore(AttrValue);
    END;

    src :
    BEGIN
        CreateBefore(AttrName);
```

```
    CreateBefore(AttrValue);
END;

minOccurs :
BEGIN
    CreateBefore(AttrName);
    CreateBefore(AttrValue);
END;

maxOccurs :
BEGIN
    CreateBefore(AttrName);
    CreateBefore(AttrValue);
END;

types :
BEGIN
    CreateBefore(AttrNameAndValue);
END;

currentType :
BEGIN
    CreateBefore(AttrName);
    CreateBefore(AttrValue);
END;

type :
BEGIN
    CreateBefore(AttrName);
    CreateBefore(AttrValue);
END;

useAt :
BEGIN
    CreateBefore(AttrName);
    CreateBefore(AttrValue);
END;

defaultAt :
BEGIN
    CreateBefore(AttrName);
    CreateBefore(AttrValue);
END;

fixed :
BEGIN
    CreateBefore(AttrName);
    CreateBefore(AttrValue);
END;

values :
BEGIN
    CreateBefore(AttrName);
    CreateBefore(AttrValue);
END;

Unknown_attribute:
BEGIN
    if TEXT_UNIT
        CreateBefore(UnknownAttrValue);
    if PICTURE_UNIT
        CreateBefore(UnknownAttrValue);
```

```
        if Unknown_namespace
            CreateAfter(UnknownAttrValue);
        if Unknown_namespace and LastAttr
            CreateAfter(Unknown_End);
        if NOT TEXT_UNIT and NOT PICTURE_UNIT
            and NOT Unknown_namespace
            CreateBefore(UnknownAttrValue);
    END;

END
```

B.5 Descripteur de langage Template.en

```
extension

presentation

TemplateP   :   Standard presentation

export

TemplateT   :   XML

translation

C_CR        :   \0
TEXT_UNIT   :   \0
useMenu     :   \0
bagMenu     :   \0
repeatMenu  :   \0
optionMenu  :   \0
folder      :   \0
includeAt   :   include
integerVal  :   integer
listVal     :   list
useAt       :   use
defaultAt   :   default
useEl       :   use
```

Annexe C

Exemples de templates
XTiger

C.1 Un article recherche

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml"
 xmlns:t="http://wam.inrialpes.fr/xtiger" xml:lang="en" lang="en">
<head>
  <title>Report</title>
  <meta http-equiv="content-type" content="text/html; charset=UTF-8" />
  <meta name="description" content="Report" />
  <meta name="keywords" content="report" />
  <meta name="template_id" content="xtiger-report-en-0.1" />
  <link rel="stylesheet" type="text/css" href="report.css" />
</head>

<body>
<t:head>
  <t:component name="refbook">
    <p class="refbook"><t:repeat minOccurs="1">
      <span class="bibauthor"><t:use types="string">
      </t:use>
      </span></t:repeat>
    <span class="title"><t:use types="string">
      Book title</t:use>
    </span> <span class="pub"><t:use types="string">
      Publisher</t:use>
    </span></p>
  </t:component>
  <t:component name="refarticle">
    <p class="refarticle"><t:repeat minOccurs="1">
      <span class="bibauthor"><t:use types="string">
      </t:use>
      </span></t:repeat>
    <span class="title"><t:use types="string">
      Article title</t:use>
    </span> <span class="conf"><t:use types="string">
      Conference</t:use>
```

```
    </span></p>
  </t:component>
  <t:component name="refreport">
    <p class="refreport"><t:repeat minOccurs="1">
      <span class="bibauthor"><t:use types="string">
      </t:use>
      </span></t:repeat>
    <span class="title"><t:use types="string">
      Report title</t:use>
    </span> <span class="url"><t:use types="string">
      url</t:use>
    </span></p>
  </t:component>
  <t:union name="bibref" include="refbook refarticle refreport">
  </t:union>
  <t:component name="author">
    <p class="author"><t:option>
      <img class="portrait" /> <t:attribute name="alt"
      default="picture of the author"/>
      <t:attribute name="width" use="optional"/>
    </t:option>
    <strong class="name"><t:use types="string">
      Name</t:use>
    </strong><br />
    <address class="address">
      <t:use types="string">
        Address's body</t:use>

    </address>
    </p>
  </t:component>
</t:head>
<t:use types="h1">

  <h1>REPORT Title</h1>
</t:use>
<t:repeat minOccurs="1">
  <t:use types="author">
  </t:use>
</t:repeat>

<div class="abstract">
<h2>ABSTRACT</h2>
<t:repeat>
  <t:use types="p">

    <p>Paragraph ...</p>
  </t:use>
</t:repeat>
</div>
<t:repeat minOccurs="1">

  <div class="section">
  <h2><t:use types="string">
    SECTION title ...</t:use>
  </h2>
  <t:repeat>
    <t:bag types="anyElement">
      <p>Section content ...</p>
    </t:bag>
  </t:repeat>
  </div>
```

```
</t:repeat>

<h2>REFERENCES</h2>
<t:repeat minOccurs="5" maxOccurs="20">
  <t:use types="bibref">
  </t:use>
</t:repeat>
</body>
</html>
```

C.2 Un Curriculum Vitae

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:t="http://wam.inrialpes.fr/xtiger"
      xml:lang="en" lang="en">
  <head>
    <title>CV</title>
    <meta http-equiv="content-type"
          content="text/html; charset=UTF-8" />
    <meta name="description" content="CV" />
    <meta name="keywords" content="employment" />
    <meta name="template_id" content="amaya-cv-en-0.1" />
    <link rel="stylesheet" type="text/css" href="cv.css" />
  </head>
  <body>
    <p class="info">
      <strong class="name">
        <t:use types="string"> Name</t:use>
      </strong>
      <br />
      <span class="birth_day">Date of birth:
        <t:use types="number"> ../../..</t:use>
      </span>
    </p>
    <address class="address">
      <t:use types="string"> Address's body</t:use>
      <br /> Tel:
      <t:use types="string"> number</t:use>
      <br />
      <a class="email">
        <t:attribute name="href" default="mailto:email@serveur" />
         e-mail:
        <t:use types="string"> email@server </t:use>
      </a>
    </address>
    <map id="menu" title="Menu" class="menu">
      <a class="entry" href="#Objectives">Objectives</a>
      <a class="entry" href="#Education">Education and training</a>
      <a class="entry" href="#Employment">Employment & placements</a>
      <a class="entry" href="#Language">Languages</a>
      <a class="entry" href="#Skill"> Skills</a>
      <a class="entry" href="#Interest">Interests</a>
    </map>
    <div class="body">
      <div class="field">
        <h1 id="Objectives">Objectives</h1>
        <t:use types="p" />
        <p>Objectives description.</p>
```

```
    </div>
    <div class="field">
      <h1 id="Education">Education and training</h1>
      <t:repeat minoccurs="1">
        <t:use types="dl" />
      </t:repeat>
      <dl>
        <dt class="date">2004 - 2005</dt>
        <dd class="content">
          <strong class="label"> Activity description </strong>
          <span class="comment">Comments</span>
          <span class="location">Location..</span>
        </dd>
      </dl>
    </div>
    <div class="field">
      <h1 id="Employment">Employment and placements</h1>
      <t:repeat minoccurs="1">
        <t:use types="dl" />
      </t:repeat>
      <dl>
        <dt class="date">June-July 2004</dt>
        <dd class="content">
          <strong class="label">Description</strong>
          <span class="location">Location</span>
          <a href="http://url">Associated URL</a>
        </dd>
      </dl>
    </div>
    <div class="field">
      <h1 id="Languages">Languages</h1>
      <t:repeat minoccurs="1">
        <t:use types="dl" />
      </t:repeat>
      <dl>
        <dt>English</dt>
        <dd>Level's description</dd>
      </dl>
    </div>
    <div class="field">
      <h1 id="Skill">Skills</h1>
      <t:repeat minoccurs="1">
        <t:use types="p" />
      </t:repeat>
      <p>Description of knowlegdes</p>
    </div>
    <div class="field">
      <h1 id="Interest">Interests</h1>
      <t:repeat minoccurs="1">
        <t:use types="p" />
      </t:repeat>
      <p>Other informations</p>
    </div>
  </div>
  <!-- Fin du Corps -->
  <div class="footer">
    <hr />
    <p>
      <a href="http://validator.w3.org/">
        <img src="http://www.w3.org/Icons/valid-xhtml10"
             alt="Valid XHTML 1.0!" height="31" width="88" />
      </a>
```

```
        <a href="http://www.w3.org/Amaya">
          <img src="http://www.w3.org/Amaya/Icons/w3c-amaya.gif"
               alt="W3C-Amaya" height="31" width="88" />
        </a>
      </p>
      <address>
        <!--$date=-->2006-05-02
        <!--$-->
      </address>
    </div>
  </body>
</html>
```