L'apport des technologies XML et Objets pour un générateur d'environnements : SmartTools

Carine Courbis — Pascal Degenne — Alexandre Fau
Didier Parigot

INRIA Sophia Antipolis - Projet OASIS
2004, route des Lucioles BP 93
France, 06902 Sophia-Antipolis cedex

http://www-sop.inria.fr/oasis/SmartTools/

Prénom.Nom@sophia.inria.fr

RÉSUMÉ. *SmartTools est un générateur d'environnements de développement basé sur les technologies objets et XML. Grâce à une technique de génération automatique à partir de spécifications, SmartTools permet de développer très rapidement des environnements spécialisés pour des langages de programmation ou pour des langages métiers. En particulier, certaines de ces spécifications (DTD, Schema) sont directement issues des technologies du W3C, ce qui donne l'accès à un grand nombre de langages métiers. D'autre part, SmartTools s'appuie sur les technologies objets : implantation en Java, utilisation du patron visiteur, de la programmation par aspects, de la distribution des objets et composants. La combinaison de ces technologies permet de proposer, à moindre coût, une plate-forme de développement ouverte, interactive, uniforme et évolutive.*

ABSTRACT. *SmartTools is a development environment generator, based on object technologies and XML. Thanks to a process of automatic generation from specifications, SmartTools makes it possible to quickly develop environments dedicated to programming languages and domain-specific languages. More precisely, some specifications are directly coming from W3C technologies, which are an open source of varied emerging domain-specific languages. SmartTools is based on object technologies: visitor pattern, aspect-oriented programming, distribution of objects and components, written in Java. This contributes to the design and implementation, at minimal cost, of a development platform which is open, interactive, uniform, and most important prone to evolve.*

MOTS-CLÉS : *technologies XML, transformation de programme, ingénierie logicielle, environnement interactif, patron visiteur, programmation par aspects.*

KEYWORDS: *XML technologies, program transformation, software engineering, interactive environment, visitor design pattern, aspect programming.*

1. Introduction

La qualité du logiciel et sa capacité à évoluer, ainsi que la rapidité du développement, sont des soucis majeurs pour les industriels. Un logiciel bien conçu doit pouvoir s'adapter rapidement aux demandes des clients et aux nouvelles technologies pour pouvoir lutter contre la concurrence. Il doit aussi être capable d'échanger des données très variées avec d'autres applications, particulièrement depuis l'avènement d'internet.

Adopter des formats de données standardisés facilite l'échange d'informations entre logiciels. Le W3C (*World Wide Web Consortium*) [W3C] élabore des spécifications [1] pour les formats de données (XML - *eXtensible Markup Language*), les langages (XSL - *eXtensible Stylesheet Language*, SVG - *Scalable Vector Graphics*) et les protocoles (SOAP - *Simple Object Access Protocol*) liés à internet. Il donne aussi la possibilité aux concepteurs de décrire les structures des données échangées en utilisant les formalismes DTD (*Document Type Definition*) ou Schema. Les concepteurs définissent des langages dits métiers (par opposition aux langages de programmation) très variés et liés à un domaine d'application : télécommunications, mais aussi finance, assurances, transports, etc. Toutes les techniques liées aux langages de programmation peuvent être employées pour les langages métiers d'autant plus que ces derniers ont souvent une syntaxe et une sémantique plus simples. Mais les concepteurs et les utilisateurs de langages métiers n'ont pas forcément de compétences approfondies sur les techniques issues de la programmation (analyse, compilation, interprétation, etc). Il y a donc un besoin d'outils pour faciliter l'utilisation de ces techniques. De plus, de telles applications (liées à l'internet) nécessitent un développement rapide, des possibilités d'intégration, une utilisation facile et un affichage multi support.

Les objectifs de la plate-forme SmartTools s'inscrivent parfaitement dans cette nouvelle problématique de conception rapide et simplifiée de langages métiers pour l'échange et/ou le traitement d'informations. Plus précisément, à partir d'une description d'un langage (DTD ou Schema), la plate-forme génère un environnement de développement contenant un analyseur d'une forme concrète du langage (*parser*), l'afficheur associé (*pretty-printer*), un éditeur syntaxique et un ensemble de fichiers Java facilitant l'écriture de traitements sémantiques (analyses, transformations).

L'originalité et l'innovation de notre approche peuvent se synthétiser en cinq points qui seront développés dans les sections suivantes.

1. Accepter en entrée des formats non propriétaires définis par le W3C (DTD et Schema) et profiter ainsi des nombreux développements réalisés autour de XML. Ainsi, le coût et le temps de développement de l'outil peuvent être fortement réduits. Notre innovation consiste à proposer des traitements sur des documents XML, en utilisant une méthodologie de programmation basée sur le patron visiteur (*visitor design pattern*) [GAM 95, PAL 96, PAL 98], issu de la programmation par objets.

---

1. Les spécifications du W3C (XML et DTD, DOM, XSL et XSLT, Schema, BML, MathML, SVG, XPath, XHTML, SOAP et WSDL) sont accessibles sur le site du W3C (`http://www.w3c.org`).

2. Proposer une programmation par aspects [KIC 96, KIC 97, BOU 01] au-dessus de la technique des visiteurs ne requérant pas de transformation de code. Cette approche dynamique a l'intérêt d'être beaucoup plus simple dans sa mise en œuvre que les approches plus classiques et généralistes [LIE 97]. Mais surtout, elle aura certainement un grand intérêt dans le cadre d'applications web pour traiter les problèmes de reconfiguration, d'adaptation et de sécurité des composants.

3. Posséder une architecture logicielle modulaire [BER 98] (avec des composants indépendants) et extensible pour assurer une bonne évolution de l'outil. Nos choix ont été confirmés avec la réalisation quasiment naturelle d'une version répartie et surtout par la facilité d'ajout de nouveaux composants et d'interconnexion avec d'autres plates-formes comme par exemple .NET [MIC 01], avec le protocole SOAP.

4. Fournir une interface utilisateur conviviale. L'innovation de notre approche consiste à traiter tous les aspects d'affichage, y compris l'interface utilisateur, selon le même modèle. Il se dégage ainsi une approche homogène et uniforme ayant un fort potentiel de réutilisation tant pour SmartTools que pour les environnements produits. Un autre avantage important est que les techniques utilisées permettent d'exporter les vues graphiques vers d'autres supports dont le web.

5. Auto-utiliser l'outil pour le développer ; ainsi les techniques proposées sont directement testées. Par exemple, tous les langages de description propres à SmartTools ont été développés grâce à l'outil. Chaque environnement produit réutilise les composants de SmartTools.

Cet article souhaite montrer les passerelles établies dans SmartTools entre différentes familles technologiques (langages, objets, XML), et comment l'adéquation de ces technologies a permis de construire un système ouvert et évolutif. L'innovation de notre système vient principalement de leur mise en commun. C'est dans ce sens qu'il n'existe pas de système comparable à SmartTools même si chaque élément pris séparément se retrouve dans bien d'autres outils ou travaux de recherche (générateurs d'environnements [BOR 88, KLI 93, REP 84], compilateurs [JOU 90, WIL 94] pour Java [GAG 98, PAL 97]). Cet article justifie notre démarche en présentant les avantages et avancées obtenues par la mise en commun de ces technologies. Il insiste sur l'intérêt d'utiliser les technologies XML et les composants existants dans le cadre d'un tel développement. Ce travail peut être considéré comme les prémices d'une utilisation des technologies objets et langages pour le web sémantique, en particulier les travaux sur la sémantique des langages de programmation.

L'article se décompose en six sections. La première section introduit les formalismes de base (syntaxe abstraite), les liens avec les formalismes équivalents du W3C et les outils associés (éditeur structuré). La deuxième section présente les outils pour la programmation des traitements sémantiques comme la programmation par visiteur ou la programmation par aspects. La troisième section donne un aperçu de l'architecture du système organisée autour d'un bus logiciel (contrôleur de messages). La quatrième section décrit notre approche uniforme pour la conception et la réalisation des interfaces graphiques et de l'interface utilisateur de SmartTools. La cinquième sec-

tion présente quelques applications de notre outil. La sixième section compare notre approche vis-à-vis de travaux similaires et décrit les perspectives de nos travaux.

2. Syntaxe abstraite et outils

Tous les outils de SmartTools sont basés sur la notion de syntaxe abstraite étendue et fortement typée (AST [2] - *Abstract Syntax Tree*) que nous allons définir dans cette section. Cette notion de syntaxe abstraite est bien connue et est couramment utilisée dans de nombreux générateurs d'environnements ou de compilateurs [BOR 88, JOU 90, KLI 93]. Cette section décrit le langage de définition d'AST, l'implantation des arbres manipulés, les passerelles réalisées pour importer d'autres formats de définition d'AST et enfin les différents outils générés.

2.1. *Langage de définition de syntaxe abstraite*

Les concepts importants de la définition d'une syntaxe abstraite sont les constructeurs (opérateurs) et les types. Les constructeurs sont regroupés dans des ensembles nommés : les types. Les fils (paramètres) des constructeurs sont typés. La partie gauche de la figure 1 montre la définition incomplète de notre langage jouet : `tiny` [3]. Par exemple, le constructeur `affect` est de type `Statement` et possède deux fils : le premier de type `Var` et le second de type `Exp`.

Il existe trois catégories de constructeurs :

– atomique sans fils ou feuille (par exemple `var`);

– d'arité fixe et de types différents (`affect`);

– d'arité variable (liste) à type fixe (`statements`).

La deuxième catégorie de constructeurs permet de définir des fils optionnels, obligatoires ou de liste. Par exemple, `op(A[] aList, B? bSon, C cSon)` indique que le premier fils du constructeur `op` est une liste de `A`, le deuxième est optionnel de type `B` et le troisième est obligatoire de type `C`. Dans ce cas, les contraintes sont que les types des fils soient disjoints deux à deux pour que le système sache à quel fils se réfère le nœud courant. Il est aussi possible de déclarer des informations associées aux constructeurs sous forme d'annotations typées plus communément appelées attributs. Par exemple, la figure 2 montre les attributs du constructeur `affect` utiles pour la génération d'un analyseur syntaxique et de l'afficheur associé.

---

2. Dans le reste de cet article, nous utiliserons cette abréviation pour désigner un arbre de syntaxe abstraite.
3. Langage utilisé comme fi l d'Ariane au cours de cet article.

```
Formalism of tiny is
Root is %Top;

Top = program(Decls declarations,
              Statements statements);
Decls = decls(Decl[] declarationList);
Statements = statements(Statement[]
                         statementList);
Statement =
  affect(Var variable, Exp value),
  while(ConditionExp cond,
        Statements statements),
  if(ConditionExp cond,
     Statements statementsThen,
     Statements statementsElse);
Var = var as STRING;
Exp =
   %ArithmeticOp,
   var,
   int as STRING,
   true(),
   false();
...
End
```

```
<!ENTITY % Top 'program'>
<!ENTITY % Decls 'decls'>
<!ENTITY % Statements 'statements'>
<!ENTITY % Statement 'if|while|affect'>
<!ENTITY % Var 'var'>
<!ENTITY % Exp 'false|int|var|true|
                %ArithmeticOp;'>

<!ELEMENT program  (%Decls;,
                    %Statements;)>
<!ELEMENT decls  (%Decl;)*>
<!ELEMENT statements  (%Statement;)*>
<!ELEMENT affect  (%Var;, %Exp;)>
<!ELEMENT while  (%ConditionExp;,
                  %Statements;)>
<!ELEMENT if  (%ConditionExp;,
               %Statements;,
               %Statements;)>
<!ELEMENT var    (#PCDATA)>
<!ELEMENT int    (#PCDATA)>
<!ELEMENT true EMPTY>
<!ELEMENT false EMPTY>
...
```

Figure 1. *Une partie de la défi nition d'AST de notre langage jouet (`tiny`) avec notre langage interne (à gauche) et son équivalence en DTD (à droite)*

```
affect(Var variable, Exp value)
  with attributes {
    fixed String separator1 = "=",
    fixed String afterOp = ";",
    fixed String styleS1 = "kw"
  }
```

```
<!ELEMENT affect (%Var;, %Exp;)>

<!ATTLIST affect
  separator1     CDATA #FIXED '='
  afterOp        CDATA #FIXED ';'
  styleS1        CDATA #FIXED 'kw'
>
```

Figure 2. *Défi nition du constructeur `affect` avec les sucres syntaxiques utiles à la génération d'un analyseur syntaxique et de l'affi cheur associé ; à gauche avec notre langage interne et à droite en DTD*

2.2. Implantation au-dessus de l'API DOM

Nous souhaitons utiliser le plus possible les composants logiciels existants issus des standards du W3C, comme par exemple l'API DOM (*Document Object Model*) de manipulation d'arbres XML. Cette API manipule des nœuds de type uniforme `org.w3c.dom.Node`. Mais l'utilisation du patron visiteur (cf. paragraphe 3.1) nécessite une structure fortement typée. Dans notre cas, cela signifie que le type de chaque nœud dépend du constructeur auquel il est associé. Nous avons étendu et complété cette API afin de travailler sur des arbres fortement typés. Par exemple, un nœud `affect` sera une instance de la classe `tiny.ast.AffectNodeImpl` qui étend la classe de base `org.w3c.dom.Node` comme le montre la figure 3. L'avantage de construire un arbre typé est que sa cohérence est garantie par le vérificateur de types de Java. Les classes (`AffectNodeImpl, etc.`) sont automatiquement générées par SmartTools à partir de la définition d'AST (cf. figure 1). Par constructeur, SmartTools génère une classe et une interface (la figure 4 montre l'interface générée pour

le constructeur `affect`) et une interface par type ; celle-ci est implantée par tous les constructeurs qui sont inclus dans ce type.

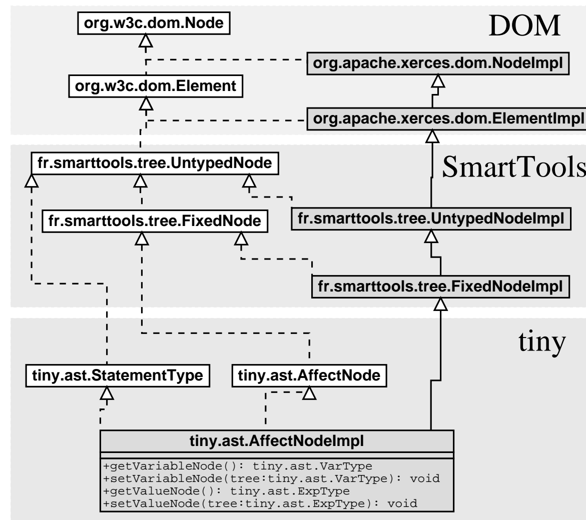


Figure 3. *Schéma du graphe d'héritage du constructeur* `affect`

```
package tiny.ast;

public interface AffectNode extends EVERYType, StatementType  {
   public tiny.ast.VarType getVariableNode();
   public void setVariableNode(tiny.ast.VarType node);
   public tiny.ast.ExpType getValueNode();
   public void setValueNode(exp.ast.ExpType node);

   // Attributes for affect operator
   public java.lang.String getSeparator1Attr();
   public java.lang.String getAfterOpAttr();
   public java.lang.String getStyleS1();
}
```

Figure 4. *Interface* `AffectNode` *générée*

Chaque classe Java décrivant un constructeur étend une implantation de DOM. Ces classes contiennent les méthodes d'accès (par exemple, `getVariableNode`) et de modification (`setVariableNode`) des fils et des annotations (`getSeparator1Attr`). Le nommage des fils des constructeurs (`statementList` pour le constructeur `statements`) est utilisé pour la génération des noms des accesseurs (dans ce cas `setStatementListNode` et `getStatementListNode`).

Dans la version 2 de l'API DOM, les attributs ne peuvent être que de type String. Comme il est parfois nécessaire lors d'un calcul de conserver des objets de type plus complexe dans les nœuds, nous avons ajouté la possibilité d'avoir des attributs de type autre que String mais ils sont volatiles. Ils n'apparaissent pas dans le format XML du document (programme) et sont perdus lors de la sauvegarde (*sérialisation*). Donc les attributs sont soit de type String à valeur constante, obligatoire ou optionnelle, soit de type quelconque mais volatiles (ce sont des attributs de travail seulement utiles pour des calculs sémantiques).

Cette couche au-dessus de DOM est compatible avec l'utilisation de tous les outils liés aux technologies XML comme les moteurs de transformation XSLT ou le mécanisme de références des chemins XPath (*XML Path language*) ; elle peut aussi utiliser les services proposés par l'API DOM dont la représentation de l'arbre au format XML.

Cette couche permet d'obtenir les informations contenues dans la définition d'AST (type du constructeur attendu, arité du constructeur, etc.), d'ajouter la notion de numéro de fils, de maintenir la cohérence de l'arbre s'il est modifié et de gérer des attributs volatiles de type complexe. Les classes des constructeurs n'héritent pas directement de DOM mais d'une des trois classes faiblement typées regroupant les informations communes à la catégorie du constructeur (feuille, liste ou variable à types différents). Avec ce typage faible, il est possible de décrire des traitements génériques [4] (par exemple, pour construire une représentation graphique de l'arbre) qui ne reposent que sur la catégorie des constructeurs.

2.3. Passerelles pour importer d'autres formalismes (DTD, Schema)

Il est important que les concepteurs de langages puissent définir leurs langages (définition d'AST) en utilisant directement les formats proposés par le W3C (DTD, Schema) et pas nécessairement le format propriétaire de SmartTools.

Le principal problème rencontré lors de la réalisation de l'application d'importation de DTD a été d'inférer les types nécessaires aux outils sémantiques de SmartTools. En effet, il n'existe pas explicitement de notion de type (ensemble d'éléments) dans une DTD. Avec la notion d'entité paramétrée, il est possible de définir un groupement d'éléments mais seulement à des fins de factorisation. Dans une première approche, on peut supposer que les parties droites des définitions d'éléments ne soient composées que par des références à des entités paramétrées.

Par exemple, seule la première de ces deux définitions d'éléments est acceptée :
```
<!ELEMENT while ((%ConditionExp;), (%Statements;))>)
<!ELEMENT while ((true|false|var|equal|notEqual), (statements))>.
```

Les éléments (<!ELEMENT while ...) sont vus comme des définitions de constructeur et leurs parties droites ne devraient être composées que de références vers des entités paramétrées (%ConditionExp;) pour indiquer le type de leurs fils. Afin de

4. Ces traitements ne seront pas détaillés dans cet article.

traiter le plus de DTD possibles, il est nécessaire de définir un algorithme d'inférence de type. Par exemple pour la deuxième définition, on infère un type qui regroupe l'ensemble des éléments qui définissent une expression conditionnelle (comme l'entité `%ConditionExp;`).

Pour les Schema, la notion de type est explicitement présente, mais il existe des mécanismes d'extension ou de restrictions (de type) que nous devons prendre en compte lors de la traduction des Schema vers notre formalisme.

2.4. Outils générés

Le format XML d'un langage est essentiellement un format d'échange de données entre applications, pas vraiment adapté à l'édition et la manipulation directes. Pour contourner ce problème, le concepteur peut définir une «vraie» syntaxe concrète pour son langage (voir la figure 5), donc écrire un analyseur syntaxique et l'afficheur associé. Mais cette tâche demande des compétences en techniques d'analyse syntaxique. Dans les cas simples (syntaxe non ambiguë, sans notion de priorité des constructeurs arithmétiques), cette tâche peut être automatisée. Le concepteur doit juste indiquer en supplément dans la définition d'AST les expressions régulières et les sucres syntaxiques (cf. figure 2 attributs `afterOp` et `separator1`) enrobant les constructeurs. Avec ces informations (ajoutées aux constructeurs), notre outil peut produire la spécification d'un analyseur syntaxique pour le générateur ANTLR [ANTa] composée d'une partie lexicale et d'une partie syntaxique avec les fonctions de construction d'arbre correctement typées. Il serait très facile de l'adapter à d'autres formats d'analyseur syntaxique LL(k) écrits en Java dont JavaCC [JAV]. Par contre, il faudrait modifier l'algorithme de génération pour d'autres méthodes d'analyse syntaxique, comme la méthode LALR du générateur CUP [CUP]. Notre outil génère aussi une spécification de l'afficheur associé (cf. partie 5.3) décrivant comment représenter les constructeurs. Cette possibilité est utilisée pour deux de nos langages internes (`xprofile` et `Xpp` présentés respectivement en 3.1 et 5.3).

La figure 6 présente toutes les spécifications qui peuvent être générées à partir d'une définition d'AST :

– l'ensemble de classes et d'interfaces décrivant les constructeurs et les types (cf. 2.2),

– les classes de base utiles pour définir des analyses sémantiques (voir section suivante),

– un analyseur syntaxique et l'afficheur associé si des informations complémentaires sont ajoutées à la définition du langage,

– un fichier de ressources minimal qui contient des informations utiles à l'analyseur syntaxique et à l'éditeur structuré dédié au langage,

– la DTD équivalente pour valider les documents XML produits ou le Schema.

```
int table
int i
int resultat
{
   table = 2;
   i = 1;

   while (i!=10)
   {
      resultat = (i*table);
      i = (i+1);
   }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<program>
 <decls>...</decls>
 <statements>
  <affect>
   <var>table</var>
   <int>2</int>
  </affect>
  <affect>...</affect>
  <while>
   <notEqual>
    <var>i</var>
    <int>10</int>
   </notEqual>
   <statements>...</statements>
  </while>
 </statements>
</program>
```

Figure 5. *Programme tiny (table de multiplication de 2) écrit en utilisant la syntaxe concrète de tiny (à gauche) ou en XML (à droite)*

SmartTools offre naturellement un éditeur structuré spécialisé pour chaque langage. C’est un composant générique disponible en standard pour chaque langage.

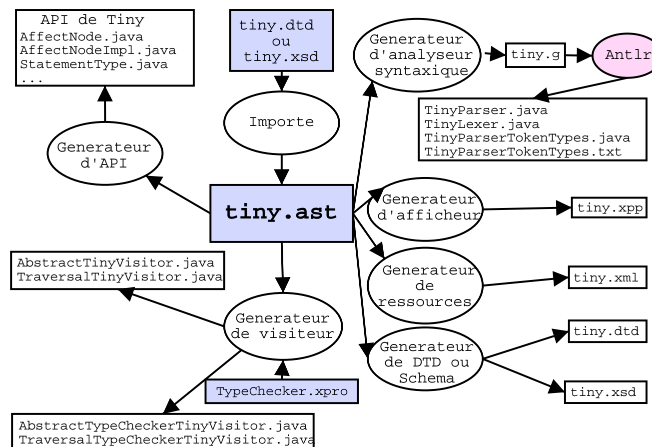


Figure 6. *L’ensemble des spécifications générées à partir d’une définition d’AST*

3. Traitements sémantiques

Cette section présente les outils utiles à la conception d’applications (traitements sémantiques) sur des ASTs.

3.1. *Le patron visiteur*

Tous ces outils dérivent du patron de conception visiteur et de la programmation par aspects. Pour mieux comprendre la conception des outils, nous rappelons brièvement l'idée de base de la technique de programmation du patron visiteur.

Pour ajouter un traitement dans une hiérarchie de classes modélisant un AST (voir Figure 3), il suffit d'introduire une nouvelle méthode dans la classe de base. Pour définir le comportement pour un nœud de type `N`, il faut implanter cette méthode dans la classe `N`. Le traitement est donc «éclaté» dans chacune des classes représentant un nœud de l'AST. Ainsi, si `M` traitements sont définis sur un AST, chaque classe contiendra ces `M` méthodes. Il est problématique d'avoir un code réparti sur toutes les classes pour chaque traitement : maintenance de code difficile, lisibilité réduite, etc. Le patron visiteur a été introduit pour résoudre partiellement ce problème. Les méthodes communes à un traitement sont regroupées en une seule classe, dénommée visiteur qui correspond à un traitement sémantique spécifique. Cette classe contient une méthode `visit(N)` pour chaque classe `N` de l'AST. Pour mettre en œuvre cette technique, il faut que chaque classe de l'AST soit équipée d'une méthode générique `accept(visiteur)` qui délègue l'exécution à la méthode `visit(N)` appropriée dans l'objet visiteur. L'article de J. Palsberg et B. Jay [PAL 98] a servi de point de départ à nos travaux.

A partir de la définition d'AST, SmartTools génère automatiquement des fichiers Java, `AbstractVisitor` et `TraversalVisitor`, implantant une des variantes du patron visiteur. Le visiteur abstrait, `AbstractVisitor`, déclare toutes les méthodes `visit` (une par constructeur). Le visiteur de parcours, `TraversalVisitor`, hérite du visiteur abstrait en implantant toutes les méthodes `visit` de façon à effectuer un parcours en profondeur de l'arbre. Ce visiteur peut être étendu par héritage et ses méthodes `visit` surchargées pour réaliser une nouvelle analyse.

Il est aussi possible de personnaliser les signatures de ces méthodes `visit` à l'aide d'un fichier de description dénommé `xprofile`. La granularité de cette personnalisation se situe au niveau des types : il faut définir un profil pour chaque type contenu dans la définition d'AST et non pas un profil par constructeur. Pour un type donné, il sera possible de préciser le type Java de retour, le nom de la méthode et le nombre, les types Java et les noms des paramètres. L'utilisation de cette possibilité évite les défauts de la technique des visiteurs : un code illisible à cause des nombreuses coercitions de type (*casts*) en retour des méthodes `visit` ou sur les arguments (vus auparavant comme un objet de type `java.lang.Object`) et l'usage de variables globales.

La Figure 7 présente une partie du fichier de personnalisation de l'évaluateur de `tiny` qui parcourt l'arbre et fait évoluer les valeurs des variables. A partir de cette spécification, le système génère automatiquement les visiteurs abstraits et de parcours correctement typés et ayant les arguments voulus. On peut remarquer que les noms des méthodes `visit` sont `eval`, `evalBoolean` ou `evalInteger`, que les types de retour sont différents (par exemple de type `Boolean` pour l'évaluation d'une condition cf. ligne 15). Pour comparer, la Figure 9 montre la même méthode `visit` que la Figure

8 : il n'y a plus de coercitions de type en `Boolean` en ligne 2 et l'argument `env` est correctement typé.

```
1  XProfile EvalTinyVisitor;
2  Formalism tiny;
3    import tiny.visitors.TinyEnv;
4    import java.lang.Boolean;
5    import java.lang.Integer;
6
7  Profiles
8    Object eval(%Top, TinyEnv env);
9    Object eval(%Decls, TinyEnv env);
10   Object eval(%Decl, TinyEnv env);
11   Object eval(%Statements, TinyEnv env);
12   Object eval(%Statement, TinyEnv env);
13   Object eval(%Exp, TinyEnv env);
14   Integer evalInteger(%ArithmeticExp, TinyEnv env);
15   Boolean evalBoolean(%ConditionExp, TinyEnv env);
16   Boolean evalBoolean(%ConditionOp, TinyEnv env);
17   Object eval(%ArithmeticOp, TinyEnv env);
18   Object eval(%Var, TinyEnv env);
19   ...
```

Figure 7. *Partie du fi chier de personnalisation de l'évaluateur de* `tiny`.

```
1  public Object visit(WhileNode node, Object env) throws VisitorException {
2     Boolean cond = (Boolean)visit(node.getCondNode(),env);
3
4     if (cond.booleanValue()) {//tantque condition vérifiée
5        visit(node.getStatementsNode(), env); //execute le bloc
6        visit(node, env); //ré-exécute la visite sur ce noeud -> récursion
7     }
8     return null;
9  }
```

Figure 8. *Evaluation du constructeur* `while` *sans profi l.*

```
1  public Object eval(WhileNode node, TinyEnv env) throws VisitorException {
2     Boolean cond = evalBoolean(node.getCondNode(),env);
3
4     if (cond.booleanValue()) {//tantque condition vérifiée
5        eval(node.getStatementsNode(), env); //execute le bloc
6        eval(node, env); //ré-exécute la visite sur ce noeud -> récursion
7     }
8     return null;
9  }
```

Figure 9. *Evaluation du constructeur* `while` *avec un profi l.*

Le langage `xprofile` permet également de préciser le parcours à effectuer dans l'arbre (du nœud de départ vers les nœuds d'arrivées) pour un visiteur. De cette façon, seuls les nœuds présents sur les chemins choisis sont visités. Cela permet de réduire de

façon significative le temps d'exécution des visiteurs. Une analyse de dépendance de graphe sur la définition d'AST est effectuée pour générer les visiteurs correspondants à ce parcours.

L'introduction des signatures des méthodes interdit l'utilisation de la variante de base des visiteurs (utilisant un appel explicite à une méthode `accept`) ; sinon il faudrait avoir une méthode `accept` par signature dans les classes des nœuds. Il est donc nécessaire d'utiliser la réflexivité de Java pour trouver la méthode `visit` à appeler en fonction du type du nœud et des arguments. Ce problème est bien connu, surtout dans le cadre de Java et correspond aux travaux de recherche sur les multi-méthodes [MIL 99]. Nous utilisons donc une variante des visiteurs basée sur l'introspection de Java. Une méthode générique (appelée `invokeVisit`) est exécutée à chaque appel d'une méthode `visit` pour trouver la méthode à appeler. Nous avons utilisé dans un premier temps une implantation des multi-méthodes pour Java [FOR 00] qui correspondait à notre problème. Cependant, l'ensemble des méthodes `visit` (ou signatures) est connu d'avance (ensemble borné) dans notre cadre. L'implantation du mécanisme d'invocation des méthodes a donc été remplacée par une version plus simple utilisant une table d'indirection pour rechercher la méthode `visit` adéquate. Cette table, pré-calculée lors de la génération des visiteurs, indique pour chaque couple `(type, constructeur)` la référence de la méthode à appeler.

En fait, notre approche est une spécialisation de celle des multi-méthodes. Nous avons comparé les deux approches (multi-méthodes et génération d'une table) et les performances sont équivalentes en temps d'exécution. L'intérêt de l'approche par génération d'une table d'indirection, outre sa simplicité de mise en œuvre, est de permettre d'associer d'autres traitements comme l'introduction d'aspects à chaque appel d'une méthode `visit`.

3.2. *Programmation par aspects*

Il a suffi de modifier légèrement la méthode `invokeVisit` pour exécuter du code avant et après les appels effectifs aux méthodes `visit`. Ainsi, on obtient une programmation par aspects [KIC 97] spécifique à nos visiteurs sans transformation de programme, contrairement aux premiers outils d'AOP (*Aspect-Oriented Programming*) [asp]. Nos points de jonction sont limités à avant et après une méthode `visit`. On peut définir ces aspects sur un constructeur, sur un type de nœud ou sur tous les nœuds. La Figure 10 présente le code d'un aspect permettant de tracer toutes les méthodes `visit` appelées. Plusieurs aspects différents peuvent être branchés sur un même visiteur. Ils seront alors exécutés séquentiellement dans l'ordre de branchement. Ce branchement (mais aussi le débranchement) peut se faire dynamiquement et à tout moment pendant l'exécution d'un visiteur. On peut donc modifier dynamiquement le comportement d'un visiteur par ajout ou retrait d'aspects.

Par exemple, cette technique est employée pour fournir un mode générique d'exécution pas-à-pas graphique (dit *mode debug*) à nos visiteurs. Il suffit de brancher sur

```
package fr.smarttools.debug;
import fr.smarttools.vtp.visitorpattern.Aspect;
import fr.smarttools.vtp.Type;

public class TraceAspect implements Aspect {
   public void before(Type t, Object[] param) {
      // param[0] est le noeud courant
      System.out.println ("Debut visit sur " + param[0].getClass());
   }
   public void after(Type t, Object[] param) {
      System.out.println ("Fin visit sur " + param[0].getClass());
   }
}
```

Figure 10. *Code d'un aspect traçant les méthodes `visit` appelées.*

chaque appel de méthode `visit` un aspect standard qui gère la communication entre la fenêtre de dialogue (fenêtre de *debug*) et l'utilisateur.

L'utilisation conjointe des visiteurs et des aspects fournit une technique simple et puissante de développement d'outils d'analyses dédiés à un langage. Actuellement, nous travaillons sur une extension de cette technique pour découpler le parcours des traitements (actions sémantiques). L'idée est de pouvoir spécifier les actions indépendamment d'un parcours. Ainsi, au lieu de coder le parcours dans les méthodes `visit`, on construit un objet sachant effectuer ce parcours et les actions ne sont décrites qu'avec des aspects. L'intérêt de cette extension est de permettre la conception de traitements par composition d'aspects. Avec cette extension, le vérificateur de types de `tiny` a été décomposé en deux aspects : l'un pour l'analyse de nom et l'autre pour la vérification.

4. L'architecture de SmartTools

La motivation principale de cette architecture est de construire un outil avec des composants logiciels ayant des fonctionnalités bien spécifiques comme le modèle "modèle-vue-contrôleur" de Smalltalk. Les communications (échanges de messages) entre ces divers composants sont majoritairement de type asynchrone : l'émetteur ne reste pas bloqué en attente du résultat du récepteur. Comme le nombre de composants peut devenir important, un mécanisme d'aiguillage des événements (contrôleur de messages) est nécessaire pour gérer l'ensemble des communications.

SmartTools comporte un ensemble de composants qui échangent des messages (événements typés) de manière asynchrone à travers le contrôleur de messages. Le comportement d'un composant est défini par l'ensemble des types de messages qu'il peut recevoir et émettre. Un composant doit, tout d'abord, s'enregistrer auprès du contrôleur de messages et indiquer les types de messages qu'il pourra traiter.

Le contrôleur de messages a la responsabilité de gérer le flux de messages et de les aiguiller pour les délivrer à leur(s) destinataire(s). Il s'agit d'un bus logiciel spécifiquement développé pour les besoins de SmartTools.

Les principaux composants sont brièvement décrits ci-dessous :

– **Document**
Chaque document contient un AST. Dans la Figure 11, `Document 1` et `Document 2` correspondent à des ASTs sur lesquels un utilisateur travaille. `Document IG` joue un rôle particulier : c'est l'AST correspondant à la structure de l'interface graphique de SmartTools.

– **Vue**
Chaque vue est un composant indépendant qui montre le contenu d'un document selon le type d'affichage. Par exemple, certaines vues vont afficher l'AST sous forme textuelle avec une syntaxe colorée, d'autres vont en donner une représentation graphique.

– **Interface utilisateur**
Le module d'interface utilisateur a pour rôle de créer des vues et de gérer les différents menus et la barre d'outils.

– **Les gestionnaires d'analyseurs syntaxiques et de documents**
Le premier composant choisit l'analyseur syntaxique approprié en fonction de l'extension du fichier reçu. Puis il exécute cet analyseur pour produire l'AST. Le gestionnaire de documents construit des composants document à partir d'ASTs reçus et les connecte au contrôleur de messages.

– **Base**
La base est un composant qui contient toutes les définitions de ressources utilisées par SmartTools : définitions de styles, de menus, de couleurs, de fontes, etc. Ces définitions sont stockées dans la base sous forme d'ASTs.

Pour fixer les idées, la Figure 11 montre une configuration possible de ces divers types de composants.

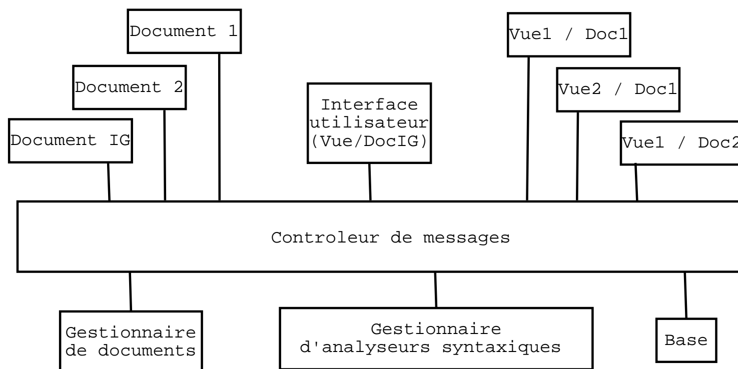


Figure 11. *L'architecture de SmartTools.*

Il est important de préciser que la communication entre composants passe obligatoirement par l'échange de messages à travers notre bus logiciel. Cette contrainte est un moyen simple pour imposer que chaque composant soit «proprement écrit». Ainsi un composant n'a pas de référence directe sur des objets appartenant à d'autres composants. Avec cette discipline de programmation, les composants peuvent être facilement exportés ou importés.

L'ensemble des types de messages est extensible. C'est l'une des techniques proposées pour étendre les fonctionnalités de SmartTools afin de répondre à des besoins spécifiques ou pour intégrer SmartTools dans des environnements de travail déjà existants.

Les messages sont constitués de deux parties : une partie concernant les informations nécessaires à l'acheminement du message et une partie concernant les données transportées. La première partie est commune à tous les types de messages et est donc gérée par une classe abstraite dont ils héritent. La deuxième partie est spécifique à chaque type de message et contient les données utiles. Les données les plus couramment échangées entre ces composants sont des arbres. Le format XML comme protocole d'arbre est naturellement un bon support pour ces échanges. Ce format possède l'avantage d'être proche de la structure des messages SOAP. En effet, il est possible de placer les informations liées à l'acheminement du message (type d'action, identifiant du module expéditeur et éventuellement identifiant du module destinataire) dans l'entête (balise `HEADER`) d'un message SOAP. Les données utiles peuvent être placées dans le corps du message (balise `BODY`). C'est ainsi que le contrôleur de messages de SmartTools a été doté de filtres capables d'importer ou d'exporter des messages en respectant les spécifications SOAP. Cela offre à SmartTools la capacité d'échanger des messages à travers un réseau avec des modules qui ne sont pas forcément écrits en Java mais qui veulent bénéficier de certaines fonctionnalités de SmartTools.

5. Environnement interactif

Cette section présente les concepts et mécanismes qui ont permis la réalisation d'une interface graphique conviviale, exportable et aisément configurable pour la plateforme SmartTools.

5.1. *Modèle document/vues*

L'interface graphique est basée sur le modèle document/vues qui s'intègre particulièrement bien dans l'architecture de SmartTools. Le composant document s'occupe des traitements (visiteurs, persistance, etc) sur un AST et les composants vues des représentations graphiques ou textuelles de cet AST. Ce modèle nous apporte une bonne séparation des fonctionnalités et la possibilité d'avoir plusieurs types de vues sur un même document (voir Figure 12). Il est conçu de manière à conserver en permanence l'isomorphisme entre les vues et le document.

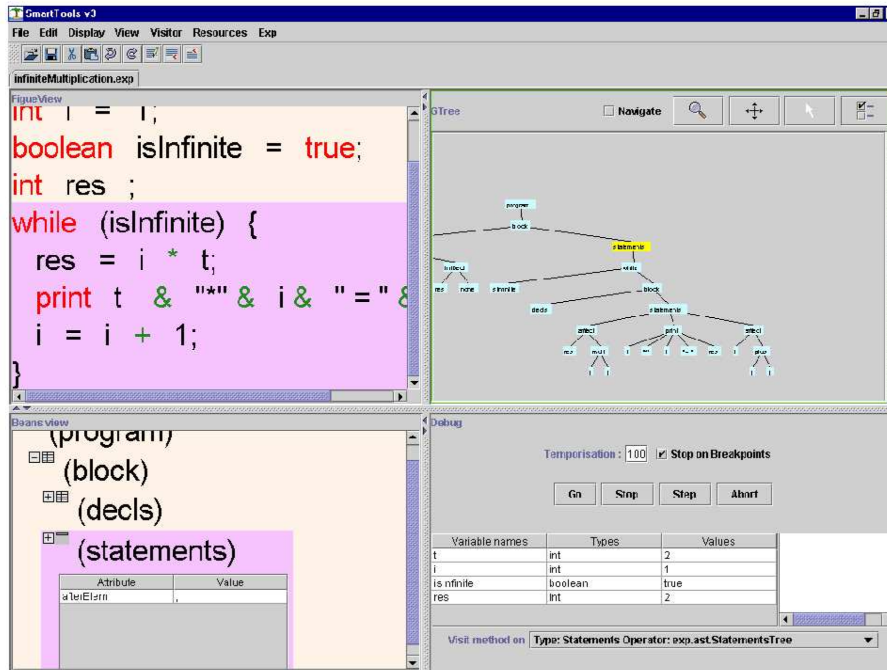


Figure 12. *Exemple d'interface utilisateur composée de quatre vues différentes du même AST : la vue textuelle (syntaxe concrète) située en haut à gauche, la vue sous forme d'arbre graphique à droite, la vue sous forme de menus et d'attributs en bas à gauche, et enfi n la vue du mode debug d'un visiteur d'évaluation.*

Comme tous les composants de SmartTools, un document et ses vues s'échangent des messages dont le contenu est codé dans un format XML. Cela implique qu'il n'y ait pas de référence directe entre les nœuds de l'AST et les objets graphiques de ses vues. L'isomorphisme est alors garanti par échanges de messages (voir Figure 13) contenant trois types d'informations : l'action, son emplacement et éventuellement le sous-arbre modifié. L'action est exprimée par le type du message, l'emplacement par un chemin absolu sous forme de XPath et le sous-arbre par sa représentation en XML.

5.2. *Construction des vues et de l'interface graphique*

Les vues sont construites en appliquant une transformation à la représentation XML d'un arbre, puis en interprétant son résultat avec un afficheur approprié.

Cette transformation peut être effectuée soit à l'aide d'un visiteur après avoir reconstruit un arbre à partir des données XML, soit par l'utilisation d'un moteur XSLT.

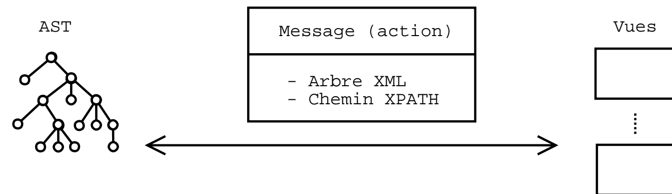


Figure 13. *Communication entre le document et ses vues.*

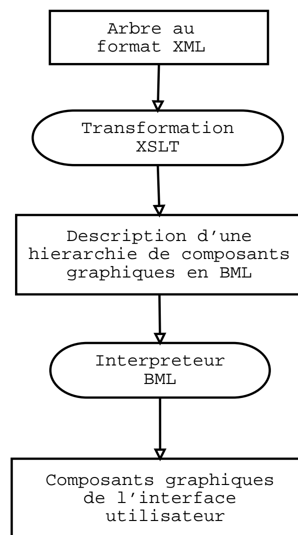


Figure 14. *Processus de transformation.*

C’est cette deuxième technique (voir Figure 14) qui a été retenue dans SmartTools pour plusieurs raisons :

– elle évite d’avoir une copie du document côté vue ;

– elle n’oblige pas à utiliser Java pour la transformation et l’affichage ;

– elle permet l’envoi des transformations à travers le réseau pour déplacer la construction de vues côté client ;

– il s’agit d’une recommandation du W3C et non d’une technique propriétaire.

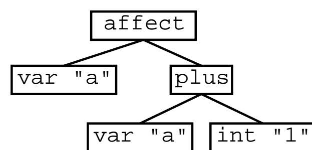
Le paragraphe 6.1 montre comment cette technologie facilite l’exportation de vues vers un navigateur Web.

La mise en page des vues et l’organisation de l’interface graphique sont décrites par un AST sur lequel on peut appliquer les concepts de création des vues. L’interface graphique est donc une vue particulière de cet AST. Ce dernier peut aisément être rendu persistant, et l’interface devient facilement configurable.

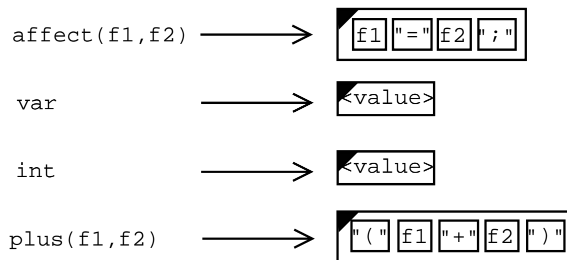
Les composants graphiques utilisés pour la version Java des vues sont basés sur la bibliothèque `javax.Swing`. Le moteur XSLT ne produisant que du format texte, il faut disposer d'un format de description des objets et de l'interpréteur associé pour les construire. Le format BML (*Bean Markup Language*) [IBM] répond parfaitement à ce besoin.

Nous allons détailler la procédure de transformation et la technique de marquage des nœuds qui permettent de maintenir les vues isomorphes à l'AST.

Prenons en exemple l'expression `a=a+1` correspondant à l'AST :

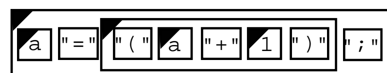


Pour obtenir une vue textuelle (syntaxe concrète) de cet arbre, les transformations suivantes ont été définies :



Chaque règle de transformation indique comment transformer un constructeur en une hiérarchie de composants graphiques. Les composants graphiques correspondants à des nœuds de l'AST sont marqués pour les différencier des sucres syntaxiques (=, +, etc). Cette technique permet de calculer la correspondance entre les composants graphiques et les nœuds de l'AST : seuls les composants marqués sont pris en compte lors du calcul du chemin.

Dans notre exemple, la transformation de l'arbre va produire la hiérarchie de composants graphiques suivante :



La vue générique (cf. la Figure 12 à gauche en bas) a été réalisée en créant une nouvelle fonctionnalité de formatage qui hérite d'un seul composant `Swing`. Cette vue montre le contenu d'un AST, de manière interactive, avec les attributs de chaque nœud dans des tables que l'on peut cacher ou afficher à volonté. Les calculs d'alignement et de mise en page sont effectués par les composants `Swing`. La création et l'intégration de nouveaux composants sont ainsi simplifiés.

5.3. *Le langage Xpp*

Cette procédure d'affichage présente cependant plusieurs inconvénients. Tout d'abord, BML et XSLT sont des langages XML peu lisibles et très redondants. Ensuite, XSLT autorise des transformations d'arbres ascendantes (sur les ancêtres du nœud sélectionné) et descendantes (sur les sous-arbres). Dans SmartTools, seules les transformations sur les sous-arbres doivent être autorisées pour conserver un calcul de chemin cohérent. Ainsi, si un nœud ou l'un de ses descendants est modifié lors de l'édition structurée, seul le réaffichage de ce nœud est nécessaire et non celui de l'arbre entier. Ces contraintes nous ont amenés à définir un langage de filtrage (*pattern-matching*) appelé Xpp, traduit ensuite en XSLT. Ses fonctionnalités sont proches de celles de XSLT mais sa syntaxe est beaucoup plus simple et concise (voir Figures 15 et 16). Le langage Xpp est un préprocesseur à XSLT. De plus, dans Xpp, seules les transformations descendantes sont autorisées, ce qui respecte ainsi la contrainte d'incrémentalité.

Les fichiers Xpp consistent en un ensemble de règles de transformations et de fonctions. Les règles de transformation (voir Figure 15) sont constituées de deux parties :

– en partie gauche, le filtre décrivant le nom du nœud recherché avec éventuellement des conditions sur ses fils ou attributs ;

– en partie droite, le code décrivant le formatage et les sucres syntaxiques souhaités.

```
Rules

affect(x, y) -> h(x, label("="), y, label(";"));
```

Figure 15. *Exemple de règle Xpp.*

Ces règles permettent d'identifier les nœuds d'un arbre correspondant à un certain motif, et de définir pour ces nœuds une mise en forme générale (alignement horizontal ou vertical, indentation, espacement, etc) indépendante du format de sortie (BML, HTML ou texte), comme le montre la Figure 17.

Chaque format de sortie possède une feuille de style définissant les fonctions de formatage (`label`, `h`, etc). Le processus de transformation s'effectue en deux étapes :

– La première étape transforme le fichier Xpp du langage avec la feuille de style du format de sortie souhaité ;

– Puis, la feuille de style obtenue est appliquée au document.

La feuille de style de chaque format de sortie définit la traduction de fonctions de mise en forme ; par exemple, la fonction d'alignement horizontal, `h`, en BML, HTML ou texte. Il est possible d'étendre Xpp en ajoutant de nouvelles fonctions de mise en forme. Il suffit de définir le résultat de ces fonctions dans les trois fichiers XSLT, pour pouvoir les utiliser ensuite dans Xpp.

```
<xsl:template match="affect[*[1]][*[2]][count(*)=2]">
  <xsl:variable name="x" select="./*[1]"/>
  <xsl:variable name="y" select="./*[2]"/>
  <bean class="fr.smarttools.view.GNodeContainer">
     <add>
       <xsl:apply-templates select="$x"/>
     </add>
     <add>
       <bean class="fr.smarttools.view.FJLabel">
         <args>
           <string>=</string>
         </args>
       </bean>
     </add>
     <add>
       <xsl:apply-templates select="$y"/>
     </add>
     <add>
       <bean class="fr.smarttools.view.FJLabel">
         <args>
           <string>;</string>
         </args>
       </bean>
     </add>
  </bean>
</xsl:template>
```

Figure 16. *Règle de la Figure 15 exprimée en XSLT.*

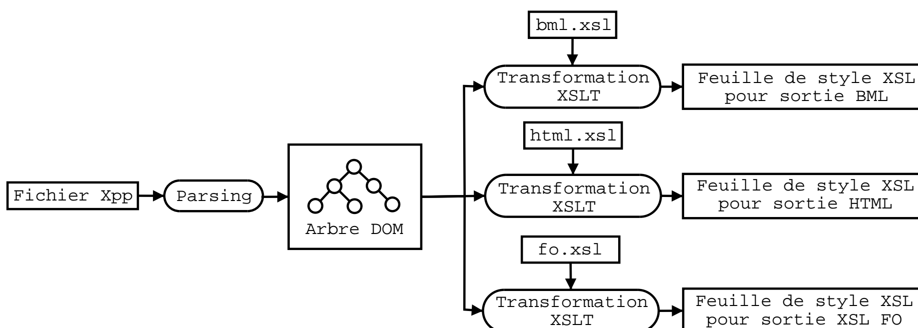


Figure 17. *Processus de génération des feuilles de style décrites par le fi chier Xpp pour chaque format de sortie.*

Le but est de mettre à disposition des utilisateurs un ensemble de boîtes d'affichage (horizontales, verticales). Ceux-ci n'ont plus qu'à décrire l'affichage sous forme de règles pour chaque opérateur de leur langage sans se soucier de l'implantation des boîtes. L'utilisateur garde cependant la possibilité de réécrire ou d'ajouter des fonctions si l'ensemble des fonctions prédéfinies ne lui suffit pas.

6. Applications

Cette section présente une partie des applications réalisées grâce à SmartTools. Il convient tout d'abord de noter que tous ses langages internes ont été construits et définis en l'utilisant. Nos techniques de programmation par visiteur et afficheur sont largement utilisées au sein de SmartTools, à tous les niveaux : environ 40% de son code source est automatiquement généré.

6.1. *Une application d'interconnexion avec un afficheur Web*

Grâce à l'architecture modulaire et à l'utilisation des technologies XML, toute vue de SmartTools peut être envoyée sur un navigateur Web, à travers le protocole HTTP. L'idée de base consiste à installer un mécanisme clients/serveur par type de vue. Côté client, l'interpréteur BML construit les objets graphiques. Côté serveur, la transformation est appliquée pour produire une description BML d'une vue. Nous avons expérimenté ces concepts par l'utilisation d'*applets* pour les clients et de *servlet* connectée au bus SmartTools pour le serveur (voir Figure 18). On peut généraliser cette approche en utilisant les *Web Services* et le protocole SOAP. Ainsi, nous avons présenté la sélection d'un nœud dans l'AST sous la forme d'un *Web Service*. Puis dans la plate-forme .NET, nous avons écrit un client .NET en C# qui fait appel à ce service de sélection par le protocole SOAP [VAR 01]. Cette expérience nous a montré qu'il était facile de connecter SmartTools à des environnements hétérogènes.

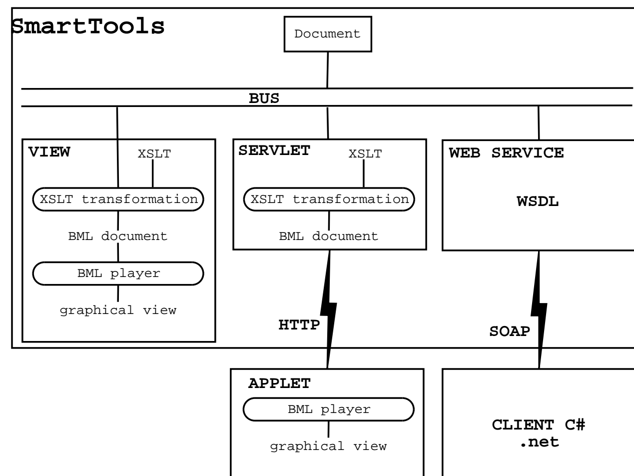


Figure 18. *Différent types d'accès à SmartTools.*

6.2. *Environnement dédiés*

Accepter le format DTD en entrée de notre outil présente l'intérêt de pouvoir produire un environnement minimal (voir Figure 6) pour n'importe quel langage défini avec une DTD. L'exemple type a été la réalisation d'un environnement pour l'outil Ant [ANTb] qui est un «système de `make`» écrit en Java dont les règles de dépendances sont exprimées en XML. L'utilisation de la DTD de Ant a permis de construire cet environnement avec un coût de développement quasiment nul.

Un autre outil a été développé pour la visualisation de formules mathématiques en utilisant la norme SVG (*Scalable Vector Graphics*) qui offre un rendu graphique de très bonne qualité. Une conversion de MathML 2.0 (*Mathematical Markup Language*) vers SVG a été réalisée avec des visiteurs et l'affichage proprement dit effectué grâce à l'intégration d'une vue de *Batik* [Bat] dans SmartTools.

Un environnement plus complexe (pour Java Card, langage de programmation des cartes à puce) a aussi été élaboré dans le cadre d'un contrat industriel avec CP8 Schumberger et a montré que le passage à l'échelle était possible avec SmartTools.

7. Discussion et Perspectives

En comparant notre approche à d'autres travaux, nous allons, sur chaque point important de notre démarche, motiver nos choix, donner les perspectives futures et souligner les principales innovations. Comme notre outil supporte et utilise différentes familles technologiques, une comparaison exhaustive et complète s'avère une tâche difficile et fastidieuse. Il nous a semblé beaucoup plus instructif de préciser pour chaque famille quels sont les problèmes ou les verrous technologiques connus. A partir de ces indications, nous expliquerons plus particulièrement en quoi notre démarche nous semble être la plus appropriée pour résoudre ces problèmes ; puis pourquoi nos choix permettent de mieux s'adapter aux futures évolutions ou sauts technologiques.

Édition Structurée et environnement interactif

Dans le domaine des éditeurs structurés [REP 84, BOR 88, KLI 93], les langages métiers définis à l'aide des formalismes XML sont certainement de meilleurs candidats que les langages dits de programmation, où les éditeurs professionnels et spécialisés (*Visual Studio*™, *JBuilder*™, *Visual Age*™, etc) sont des concurrents manifestes à l'approche générique. Mais il est important, même pour ces langages métiers [DEU 98] beaucoup moins exigeants en terme d'édition libre (possibilité d'écrire du code à la volée), de proposer des outils d'affichage ouverts et extensibles à de nouvelles bibliothèques de composants graphiques. Nous avons montré qu'il est relativement simple de construire au-dessus de l'outil de transformation XSLT, un mécanisme d'affichage de vues avec les contraintes particulières liées à l'édition structurée. Nous montrons aussi que cette approche (avec BML) permet une exportation aisée des vues

graphiques à travers le réseau. Cela montre encore une fois tout l'avantage d'utiliser diverses familles technologiques : la bibliothèque `Swing` pour le graphisme, l'outil XSLT pour les transformations, XML et BML pour la *sérialisation*. Nous n'avons pas encore exploité toutes les possibilités de nos outils d'affichage. L'utilisation des technologies XML est un atout indéniable de notre approche.

Passerelle vers les formalismes DTD et Schema

Nos efforts pour accepter les formalismes du W3C sont certes motivés par notre souci d'élargir le champ d'applications de SmartTools. L'intérêt est de proposer pour ce type d'applications (langages) nos outils d'édition, d'affichage et/ou de description sémantique. Notre approche de génération automatique du couple analyseur syntaxique et afficheur semble envisageable pour des langages métiers simples. Elle serait certainement trop complexe pour des langages de programmation. Cette génération devrait rendre de grands services dans ce contexte de petits langages métiers. Les formalismes du W3C avaient comme vocation initiale de mieux structurer les informations issues des documents. Aujourd'hui, ils sont aussi utilisés pour la manipulation de données de natures différentes. Par exemple, le format XMI (*XML Metadata Interchange*) [XMI] est utilisé pour la représentation des diagrammes UML (*Unifi ed Modeling Language*) [UML], formalisme très proche des langages de programmation. Nous sommes convaincus qu'il y aura un besoin important d'outils puissants pour décrire des traitements complexes pour ces langages métiers. Il serait dommage de réinventer des outils de manipulation dans ce nouveau contexte alors qu'il existe déjà, certes dans d'autres communautés scientifiques, des travaux de recherche et des développements parfaitement adaptés à cela.

Outils sémantiques

Le succès grandissant de la notion de patrons de conception [GAM 95] montre que les concepts de programmation par objets ne sont pas suffisants et que chaque type d'application ou problématique demande des solutions appropriées [NIE 95]. En particulier, le patron visiteur a suscité un ensemble de travaux de recherche [PAL 96], qui ont tous comme objectif de trouver le meilleur compromis entre la lisibilité et l'efficacité. L'un des autres soucis de ce patron est la composition de visiteurs [KUI 01].

Dans [COR 98], nous avions déjà remarqué certaines similitudes sur cette problématique pour des familles de technologies différentes (grammaires attribuées [ROU 94, HED 01], programmation polytypique [JEU 96] et programmation adaptative [PAL 95, LIE 97]). La programmation adaptative suit cette même problématique de séparation des concepts (parcours et sémantique). La programmation par aspects [KIC 96] a aussi été introduite pour la séparation des parties fonctionnelle et non-fonctionnelle (applicative ou de services) d'une application. Les approches par transformation de programme utilisées pour la programmation par aspects ont montré leurs limites [BOU 01]. Il est clair qu'il existe des liens très forts avec les travaux de recherche sur la réflexi-

vité [MAL 96a, MAL 96b] pour les langages à objets, en particulier la notion de MOP (*Meta-Object Protocole*) [KIC 91]. Les mécanismes mis en jeu dans ces approches totalement dynamiques ne sont pas simples d'utilisation. Ils demandent de comprendre la sémantique sous-jacente des langages à objets.

L'originalité de notre approche est de partir d'une spécification déclarative de la structure des objets et d'effectuer une génération de code source enrichie par les mécanismes de programmation par aspects ou adaptative. L'intérêt est d'une part d'éviter les problèmes d'efficacité par une génération de code source et de cacher à l'utilisateur la complexité des mécanismes mis en jeu. De plus, notre approche de programmation par aspects a le mérite d'être mise en œuvre très simplement par une extension naturelle du patron de conception visiteur.

Architecture modulaire

Pour l'architecture de notre logiciel, nous avons volontairement pris la solution de centraliser les problèmes de communication et de connexion au sein d'un bus logiciel [REI 96, BER 98]. La technologie XML joue pleinement son rôle en terme de protocole de communication des données complexes. Mais cette démarche a des limites car les composants perdent la possibilité d'être intégrés dans d'autres environnements sans être forcément rattachés au bus. En effet, le bus est le seul moyen pour les composants de communiquer vers l'extérieur. Par exemple, dans [BER 98], les auteurs ont poussé cette voie à son paroxysme, puisqu'ils spécifient la sémantique des communications à l'aide d'un langage de coordination au sein même de l'objet bus. A l'inverse, l'approche qui consiste à enrichir chaque composant par de nouveaux services se heurte essentiellement aux trois problématiques suivantes :

– avoir des composants ouverts à de nouveaux services non prévus initialement (au moment de la phase de conception) ;

– maîtriser l'ensemble des composants, ce qui impose naturellement la construction d'un référentiel de l'état du système ;

– prendre en compte les diverses technologies de composants (CORBA, EJB, COM, Web Services).

Actuellement nous étudions plusieurs voies possibles pour faire évoluer notre architecture dans le but de rendre les composants indépendants du bus logiciel. Dans un premier temps, nous voulons que l'interface de communication des composants soit indépendante du choix de la technologie de composant effectivement choisie. Pour cela, la communication sera déléguée à un objet qui prendra en compte le choix de la technologie de composant effectivement utilisée. A partir d'une description de haut niveau des interfaces des composants et de la technologie de composant choisie, SmartTools générera l'implantation de cet objet de communication.

Applications

Bien que notre outil soit en cours de développement, il présente de nombreuses perspectives d'utilisation dans des domaines d'application très variés. Ces perspectives peuvent être regroupées en six catégories :

– les langages métiers de SmartTools :
L'outil utilise les technologies objets et XML pour la description de ses langages (définition d'AST, Xpp, xprofile) et composants, pour ses fichiers de configuration (scripts de lancement), et pour le paramétrage des vues graphiques. Cette utilisation intensive valide nos choix et est susceptible d'engendrer de nouveaux champs d'application. Par exemple, nous avons découvert que l'interface graphique pouvait être traitée comme un AST.

– les langages de programmation ou métiers :
Le champ d'application traditionnel est la conception d'environnements pour des langages de programmation ou métiers. Par exemple un environnement pour Java est en cours de réalisation dans le but de réaliser des analyses statiques complexes utilisant les techniques de programmation par visiteurs.

– les langages de méta-modélisation :
Depuis l'émergence d'outils de méta-modélisation autour d'UML (MOF - *Meta-Object Facility*, OCL - *Object Constraint Language*), des analogies entre ces méta-langages et les langages de programmation commencent à être identifiées. L'établissement de passerelles est déjà à l'étude entre l'AGL de Softeam (*Objecteering*™) et SmartTools. Nos techniques de programmation par aspect pourraient être utilisées pour la description de la sémantique de ces modèles (*Action Semantic* d'UML).

– les langages du W3C :
Pour la réalisation des passerelles entre les formalismes du W3C et nos langages de spécification, des environnements de programmation pour les DTD et les Schema ont été développés. Dans le cadre d'un contrat industriel, un environnement de transformation pour le format XHTML va être réalisé. Tous ces exemples montrent que les formalismes du W3C sont potentiellement des champs d'application pour SmartTools. Son approche générique est un atout indéniable pour la conception rapide d'environnements.

– les langages de description de composant :
Les équipes de recherche sur le thème des composants (en particulier les travaux autour de la nouvelle norme CORBA (CCM - *CORBA Component Model*) ou les plates-formes comme *ObjectWeb*) sont des utilisateurs potentiels de SmartTools. Des environnements de développement peuvent être générés pour leurs langages métiers (par exemple le langage IDL - *Interface Definition Language*) et des générateurs (compilateurs) spécifiés à l'aide de nos techniques. Pour la description de nos composants, nous pensons utiliser le format WSDL (*Web Service Definition Language*) avec des générateurs spécifiques selon la technologie de composant choisie (CORBA, Web Services, EJB).

– les systèmes d'ingénierie ontologique (RDF - *Resource Description Framework*) du Web Sémantique :

Le thème de recherche du Web Sémantique regroupe tout un ensemble de concepts très variés dont le lien avec SmartTools semble être l'analogie entre les systèmes d'ingénierie ontologique [CRA 01] et les notions de syntaxe abstraite (système de type). Même s'il est trop tôt pour parler d'application spécifique pour le Web Sémantique avec SmartTools, il nous semble raisonnable de penser que nos techniques (visiteur ou programmation par aspects) pourront être utilisées pour certaines applications de ce domaine.

Conclusion

Le souci majeur du développement est certainement la réutilisation au sens large du terme. Nous espérons que le lecteur a été convaincu que notre outil, par sa conception, est un exemple type de logiciel centré sur l'idée de la réutilisation grâce à son approche générique. L'utilisation de standards tant en terme de composants logiciels ou bibliothèques (le succès de Java vient principalement de la richesse des bibliothèques et des outils proposés) qu'en terme de spécifications (les formalismes du W3C) est la garantie d'une facilité d'évolution. L'une des caractéristiques importantes de notre approche est que l'outil s'appuie sur une description abstraite des objets pour générer automatiquement une grande partie des applications, aussi bien pour l'interface utilisateur, les afficheurs, les composants sémantiques et bientôt pour les interfaces de communications. L'un des grands avantages de cette phase de génération est de faciliter la prise en compte des futures évolutions des technologies et des fonctionnalités non prévues initialement.

Finalement, notre outil est notre propre champ d'expérimentation des concepts introduits. La richesse et la variété des techniques, et le champ des applications susceptibles d'être traitées démontrent l'intérêt de notre approche générique. Cette approche par génération à partir de modèles est conforme à la nouvelle stratégie de conception [BéZ 01], *Model Driven Architecture* (MDA), définie par l'OMG (*Object Management Group*). En effet, les phases de génération de code source à partir de modèles décrits avec les technologies XML sont un moyen de garantir une évolution aisée de l'outil SmartTools et des environnements produits.

Remerciements

Nous avons beaucoup profité des avis de Colas Nahaboo, Thierry Kormann et Stéphane Hillion d'Ilog à propos des technologies XML. Ce projet fut partiellement financé par Bull CP8/Schlumberger (Dyade) avec la participation de Claude Pasquier et Microsoft Research, et est maintenant partiellement financé par le W3C dans le cadre du contrat européen QUESTION-HOW[5].

5. QUality Engineering Solutions via Tools, Information and Outreach for the New Highly-enriched Offerings from W3C (voir `http://www.w3.org/2001/qh/` pour plus d'information)

8. Bibliographie

[ANTa] « ANTLR - ANother Tool for Language Recognition », http://www.antlr.org/.

[ANTb] « Apache Ant - The Jakarka Project (Apache) », http://jakarta.apache.org/ant.

[asp] « AspectJ - Aspect-Oriented Programming (AOP) for Java », http://aspectj.org.

[Bat] « Batik SVG Toolkit (Apache) », http://xml.apache.org/batik/.

[BER 98] BERGSTRA J., KLINT P., « The discrete time ToolBus – A software coordination architecture », *Science of Computer Programming*, vol. 31, n° 2-3, 1998, p. 205–229.

[BOR 88] BORRAS P., CLÉMENT D., DESPEYROUX T., INCERPI J., KAHN G., LANG B., PASCUAL V., « CENTAUR: the System », *SIGSOFT Software Eng. Notes*, vol. 13, n° 5, 1988, p. 14–24.

[BOU 01] BOURAQADI-SAÂDANI N. M. N., LEDOUX T., « Le point sur la programmation par aspects », *Technique et Sciences Informatiques*, vol. 20, page 505 à 528, Hermès, 2001.

[BéZ 01] BÉZIVIN J., « From Object Composition to Model Transformation with MDA », *TOOLS USA*, IEEE TOOLS-39, 2001, page 2001.

[COR 98] CORRENSON L., DURIS E., PARIGOT D., ROUSSEL G., « Schéma générique de développement par composition », *Approches Formelles dans l'Assistance au Développement de Logiciel AFADL'98*, Poitiers - Futuroscope, 1998.

[CRA 01] CRANEFIELD S., « UML and the Semantic Web », *Proceedings of the International Semantic Web Working Symposium (SWWS)*, 2001, http://www.semanticweb.org/SWWS/program/full/paper1.pdf.

[CUP] « CUP - LALR Parser Generator for Java », http://www.cs.princeton.edu/ appel/modern/java/CUP/.

[DEU 98] VAN DEURSEN A., KLINT P., « Little Languages: Little Maintenance ? », *Journal of Software Maintenance*, , 1998.

[FOR 00] FORAX R., DURIS E., ROUSSEL G., « Java Multi-Method Framework », *International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'00)*, 2000.

[GAG 98] GAGNON E., HENDREN L., « SableCC - An Object-Oriented Compiler Framework », *In Proceedings of TOOLS 1998 - 26th International Conference and Exhibition*, August 1998, p. 140-154.

[GAM 95] GAMMA E., HELM R., JOHNSON R., VLISSIDES J., *Design Patterns*, Addison Wesley, Reading, MA, 1995.

[HED 01] HEDIN G., MAGNUSSON E., « JastAdd—a Java-based system for implementing front ends », VAN DEN BRAND M., PARIGOT D., Eds., *Electronic Notes in Theoretical Computer Science*, vol. 44, Elsevier Science Publishers, 2001.

[IBM] IBM, « Bean Markup Language », http://www.alphaworks.ibm.com/formula/bml.

[JAV] « Java Compiler Compiler (JavaCC) - The Java Parser Generator (Sun) », http://www.webgain.com/products/java_cc.

[JEU 96] JEURING J., JANSSON P., « Polytypic Programming », LAUNCHBURY J., MEIJER E., SHEARD T., Eds., *Advanced Functional Programming*, vol. 1129 de *Lect. Notes in Comp. Sci.*, Springer-Verlag, 1996, p. 68–114.

[JOU 90] JOURDAN M., PARIGOT D., JULIÉ C., DURIN O., LE BELLEC C., « Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System », *Conf. on Program-*

ming Languages Design and Implementation, White Plains, NY, june 1990, p. 209–222, Published as *ACM SIGPLAN Notices*, 25(6).

[KIC 91] KICZALES G., DES RIVIERES J., *The Art of the Metaobject Protocol*, MIT Press, Cambridge, MA, USA, 1991.

[KIC 96] KICZALES G., « Aspect-Oriented Programming: A Position Paper From the Xerox PARC Aspect-Oriented Programming Project », MUEHLHAUSER M., Ed., *Special Issues in Object-Oriented Programming*, 1996.

[KIC 97] KICZALES G., LAMPING J., MENHDHEKAR A., MAEDA C., LOPES C., LOINGTIER J.-M., IRWIN J., « Aspect-Oriented Programming », AKŞIT M., MATSUOKA S., Eds., *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, vol. 1241 de *Lecture Notes in Computer Science*, p. 220–242, Springer-Verlag, New York, NY, 1997.

[KLI 93] KLINT P., « A Meta-Environment for Generating Programming Environments », *ACM Transactions on Software Engineering Methodology*, vol. 2, n° 2, 1993, p. 176–201, Springer-Verlag.

[KUI 01] KUIPERS T., VISSER J., « Object-Oriented Tree Traversal with JJForester », VAN DEN BRAND M., PARIGOT D., Eds., *Electronic Notes in Theoretical Computer Science*, vol. 44, Elsevier Science Publishers, 2001.

[LIE 97] LIEBERHERR K. J., ORLEANS D., « Preventive Program Maintenance in Demeter/Java », *Proceedings of the 19th International Conference on Software Engineering*, ACM Press, 1997, p. 604–605.

[MAL 96a] MALENFANT J., COINTE P., « Aspect-Oriented Programming versus Reflection », rapport, 1996, Ecole des Mines de Nantes.

[MAL 96b] MALENFANT J., DONY C., COINTE P., « A Semantics of Introspection in a Reflective Prototype-Based Language », *Lisp and Symbolic Computation*, vol. 9, n ° 2/3, 1996, p. 153–180.

[MIC 01] MICROSOFT, « The .NET platform », 2001, http://www.microsoft.com/net/.

[MIL 99] MILLSTEIN T., CHAMBERS C., « Modular Statically Typed Multimethods », GUERRAOUI R., Ed., *Proceedings ECOOP'99*, LCNS 1628, Lisbonne, Portugal, 1999, Springer-Verlag, p. 279–303.

[NIE 95] NIERSTRASZ O., TSICHRITZIS D., Eds., *Object-Oriented Software Composition*, Prentice-Hall, 1995.

[PAL 95] PALSBERG J., XIAO C., LIEBERHERR K., « Efficient Implementation of Adaptive Software », *ACM Transactions on Programming Languages and System*, vol. 17, n° 2, 1995, p. 264–292.

[PAL 96] PALSBERG J., PATT-SHAMIR B., LIEBERHERR K., « A New Approach to Compiling Adaptive Programs », NIELSON H. R., Ed., *European Symposium on Programming*, Linköping, Sweden, 1996, Springer Verlag, p. 280–295.

[PAL 97] PALSBERG J., TAO K., « Java Tree Builder », 1997, http://www.cs.purdue.edu/jtb.

[PAL 98] PALSBERG J., JAY C. B., « The Essence of the Visitor Pattern », *COMPSAC'98, 22nd Annual International Computer Software and Applications Conference*, Vienna, Austria, 1998.

[REI 96] REISS S. P., « Simplifying Data Integration: The Design of the Desert Software Development Environment », *Proceedings of the 18th International Conference on Software Engineering*, IEEE Computer Society Press, 1996, p. 398–407.

[REP 84] REPS T., TEITELBAUM T., « The Synthesizer Generator », *ACM SIGSOFT/SIGPLAN Symp. on Practical Software Development Environments*, p. 42–48, ACM

press, Pittsburgh, PA, april 1984, Joint issue with Software Eng. Notes 9, 3.Published as ACM SIGPLAN Notices, volume 19, number 5.

[ROU 94] ROUSSEL G., PARIGOT D., JOURDAN M., « Coupling Evaluators for Attribute Coupled Grammars », FRITZSON P. A., Ed., *5th Int. Conf. on Compiler Construction (CC' 94)*, vol. 786 de *Lect. Notes in Comp. Sci.*, Edinburgh, 1994, Springer-Verlag, p. 52–67.

[UML] « UML - Unifi ed Modeling Language », http://www.uml.org.

[VAR 01] VARIAMPARAMBIL J. G., « Getting SmartTools and VisualStudio.NET to talk to each other using SOAP and web services », rapport, 2001, INRIA, http://www-sop.inria.fr/oasis/SmartTools/publications/Joseph/report.pdf.

[W3C] « http://www.w3.org/ », The World Wide Web Consortium.

[WIL 94] WILSON R. P., FRENCH R. S., WILSON C. S., AMARASINGHE S. P., ANDERSON J.-A. M., TJIANG S. W. K., LIAO S.-W., TSENG C.-W., HALL M. W., LAM M. S., HENNESSY J. L., « SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers », *SIGPLAN Notices*, vol. 29, n° 12, 1994, p. 31-37.

[XMI] « XMI - XML Metadata Interchange (OMG) », http://www.omg.org/technology/documents/formal/xmi.htm.