UNIVERSITÉ DE NICE SOPHIA-ANTIPOLIS - UFR SCIENCES
École Doctorale STIC

CONTRIBUTION À LA PROGRAMMATION GÉNÉRATIVE

APPLICATION DANS LE GÉNÉRATEUR SMARTTOOLS : TECHNOLOGIES XML, PROGRAMMATION PAR ASPECTS ET COMPOSANTS

THÈSE

Présentée pour obtenir le titre de
DOCTEUR EN SCIENCES
DE L'UNIVERSITÉ DE NICE SOPHIA-ANTIPOLIS

Spécialité INFORMATIQUE

par

Carine COURBIS

Thèse dirigée par Didier PARIGOT et Isabelle ATTALI à l'INRIA Sophia-Antipolis
A soutenir publiquement le 10 décembre 2002 à 11 heures
à l'INRIA devant le jury composé de

Président	Michel	RIVEILL	Université de Nice Sophia-Antipolis
Rapporteurs	Jean	BÉZIVIN	Université de Nantes
	Jean-Marc	JÉZÉQUEL	Université de Rennes - IRISA
Examinateur	Daniel	DARDAILLER	W3C
Directeurs	Didier	PARIGOT	INRIA Sophia-Antipolis
	Isabelle	ATTALI	INRIA Sophia-Antipolis

Table des matières

Table des figures

Liste des acronymes

AOP	Aspect-Oriented Programming
API	Application Programming Interface
ARC	Action de Recherche Coopérative
AST	Abstract Syntax Tree
BML	Bean Markup Language
BSML	Bioinformatic Sequence Markup Language
CCM	CORBA Component Model
CML	Chemical Markup Language
CORBA	Common Object Request Broker Architecture
CSS	Cascading Style Sheet
DCOM	Distributed Component Object Model
DDML	Document Definition Markup Language
DOM	Document Object Model
DSL	Domain Specific Language
DT4DTD	DataTypes for DTDs
DTD	Document Type Definition
EBNF	Extensible Backus Naur Form
EJB	Entreprise JavaBeans
HTML	Hyper Text Markup Language
MDA	Model-Driven Architecture
MOF	Meta-Object Facility
MOP	Meta-Object Protocol
MathML	Mathematical Markup Language
OCL	Object Constraint Language
OFX	Open Financial eXchange
OMG	Object Management Group
OMT	Object Modeling Technique
OOSE	Object-Oriented Software Engineering

PDA	Personal Digital Assistant
PIM	Plateform-Independant Model
PPML	Pretty Printing Meta Language
PSM	Platform-Specific Model
QoS	Quality of Service
RDF	Resource Description Framework Schema
RELAX	Regular Language Description for XML
RNTL	Réseau National de recherche et d'innovation en Technologies Logicielles
SAX	Simple API for XML
SGBD	Système de Gestion de Base de Données
SGML	Standard Generalized Markup Language
SMIL	Synchronized Multimedia Integration Language
SOAP	Simple Object Access Protocol
SOX	Schema for Object-oriented XML
SVG	Scalable Vector Graphics
TREX	Tree Regular Expression for XML
UML	Unified Modeling Language
URI	Uniform Resource Identifier
VTP	Virtual Tree Processor
W3C	World Wide Web Consortium
XML	Extensible Markup Language
XPath	XML Path language
XSLT	Extensible Stylesheet Language Transformation

Introduction

Contexte et présentation du problème

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. En quelques années, l'informatique confinée aux domaines scientifiques s'est démocratisée. Ses utilisateurs ont maintenant des besoins, des connaissances et des domaines d'activité différents. De plus, la pression du marché impose des temps de développement de logiciel plus courts et des coûts plus faibles.

Cette évolution a bouleversé la manière de concevoir et de réaliser les logiciels. Il n'est plus possible de les développer entièrement sauf dans les domaines tels que la défense ou les transports où la surêté de l'ensemble du code doit être vérifiée. Les technologies propriétaires sont à bannir car elles isolent les logiciels et freinent leurs évolutions. Les exigences vis-à-vis des logiciels ont aussi été modifiées à cause des disparités de connaissances des utilisateurs et des programmeurs, des contraintes de temps et d'argent, et des nécessités d'adaptation rapide aux besoins du marché. Les logiciels doivent satisfaire les exigences suivantes :

- conviviaux grâce à une interface utilisateur interactive ;
- faciles à utiliser avec peu de compétences informatiques et basés sur des techniques bien connues ou des standards ;
- ouverts grâce à un format d'échange de données standard utilisé pour communiquer entre les composants et avec les applications externes ;
- à implémentation modulaire et flexible, basée sur des composants génériques et réutilisables.

Pour prendre en compte ces bouleversements, de nouvelles techniques ont émergé. Tout d'abord, il y a eu la programmation par objets avec les notions d'encapsulation et d'héritage propices à la modularité, la réutilisation et l'extensibilité du code.

Mais ce style de programmation est apparu insuffisant pour prendre en compte des préoccupations transversales aux classes. Pour pallier ce problème, la programmation dite par aspect (AOP - *Aspect-Oriented programming*) a vu le jour permettant de gérer, de manière modulaire, ces préoccupations en les séparant du code de base (*separation of concerns*) ; la plus connue des implémentations étant AspectJ [3, 49] de Kiczales. Il est ainsi très facile d'étendre le code de base avec de nouvelles fonctionnalités sans le modifier directement (technique non invasive). Cette technique jeune soulève de nombreuses questions : où tisser les aspects ? Comment composer plusieurs aspects ? Quelle technique d'implémentation choisir entre la transformation de programme, la réflexivité ou la génération de code adapté ?

L'approche objet a aussi une granularité très fine, peu adaptée aux systèmes complexes. Le concept de composant a été introduit afin d'encapsuler plusieurs objets proposant des services et de pouvoir facilement y associer du code non fonctionnel tel que la communication entre composants, la persistance ou la politique de sécurité. Avec cette approche par composants, il est facile de déployer et même de répartir une application. Il est aussi possible de construire une application uniquement par assemblage de composants, ce qui devrait réduire les coûts et les temps de développement. Quatre principaux modèles de composants ont émergé : les EJBs (*Entreprise JavaBeans*) [7] de Sun, CCM (*CORBA Component Model*) [4] de l'OMG (*Object Management Group*), DCOM (*Distributed Component Object Model*) de Microsoft et les *Web Services* du W3C (*World Wide Web Consortium*). Avec l'avènement des *Web Services*, cette possibilité de création d'applications par assemblage de composants attire de nombreux industriels. Mais il reste encore de nombreuses interrogations : comment faire communiquer des composants de modèles différents ? Comment découvrir les composants ayant les services et la qualité de service (QoS) souhaités ? Quelle sécurité adopter pour les transactions ?

Avec l'effervescence liée à la naissance du Web, il y a eu une volonté de standardiser les langages et protocoles, ce qui a conduit à la création du W3C [15]. Ce consortium a proposé un nouveau format de données, XML (*Extensible Markup Language*), issu de SGML (*Standard Generalized Markup Language*) afin de simplifier et d'uniformiser les échanges d'informations entre applications, indépendamment de tout langage et plate-forme. Ce consortium a ensuite élaboré des spécifications[1] de langages, de protocoles et d'APIs (*Application Programming Interface*) tels que le langage XSLT (*Extensible Stylesheet Language Transformation*) pour effectuer des transformations, le protocole SOAP (*Simple Object Access Protocol*) pour échanger des messages ou l'API DOM (*Document Object Model*) pour manipuler les documents sous forme d'arbres. Utiliser des technologies standardisées et pour lesquelles il existe de nombreux outils simplifie le développement, l'évolution et rend les applications ouvertes.

Ces bouleversements ont aussi été pris en compte au niveau des spécifications des logiciels, avec l'apparition de nouvelles méthodes d'analyse et de conception à base de modèles

[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).

et à production de code. La plus célèbre et récente de ces méthodes de modélisation de systèmes à objets est UML (*Unified Modeling Language*) [14] de l'OMG qui réunit et unifie les notations et les sémantiques des méthodes Booch de Grady Booch, OMT (*Object Modeling Technique*) de Jim Rumbaugh et OOSE (*Object-Oriented Software Engineering*) de Ivar Jacobson. Avec ce langage graphique, les différentes dimensions d'un logiciel complexe peuvent être spécifiées, assemblées, visualisées et documentées. Ces différentes modélisations sont ensuite utilisées pour générer automatiquement la trame du code du logiciel cible. Récemment, l'OMG a introduit une nouvelle approche d'écriture de spécifications et de développement d'applications, nommée MDA[2] (*Model-Driven Architecture*) [67, 69, 26]. Cette approche prône l'utilisation de modèles indépendants de toute plate-forme et technologie (PIM - *Plateform-Independant Model*) qui sont ensuite transformés vers un ou plusieurs modèles de plate-forme spécifique (PSM - *Platform-Specific Model*). La partie métier (PIM) est ainsi séparée de la partie technologique cible (PSM) et le passage du PIM vers un ou plusieurs PSMs s'effectue par des règles de transformation. Cette approche par abstraction permet de mieux se concentrer sur la partie «intelligente» et la rend pérenne car utilisable avec les technologies futures.

Contribution

Un changement radical des méthodologies de conception et de développement d'application est nécessaire pour aisément prendre en compte ces différentes approches ou facettes (AOP, composant, MDA, etc).

Cette thèse jette les bases d'une nouvelle manière de programmer où ces facettes sont automatiquement intégrées aux spécifications ou modèles abstraits de l'application lors de phases de génération de code source [32]. Cette idée a été appliquée, à différents niveaux, lors de la réalisation d'un outil nommé SmartTools, aussi bien pour la représentation de données ou d'environnements interactifs que pour les traitements sémantiques ou l'architecture. Cette thèse défend cette idée et montre, de manière pragmatique, les apports de ces différentes facettes et comment les combiner. Nos objectifs avec cet outil s'inscrivent parfaitement dans cette nouvelle problématique, aussi bien pour sa réalisation que pour les environnements qu'il produit ; son but étant d'aider à la création d'outils tels que des éditeurs, des afficheurs (*pretty-printers*), des analyseurs syntaxiques (*parsers*) ou des traitements sémantiques (analyses, transformations) pour les langages de programmation ou métiers. Plus précisément, à partir d'une description d'un langage (DTD ou XML Schema), il génère un environnement minimal doté d'un éditeur guidé par la syntaxe, d'un ensemble de fichiers Java facilitant l'écriture de traitements sémantiques, d'afficheurs génériques et d'un analyseur syntaxique XML ayant les fonctions de construction d'arbres de ce langage. Cet environnement peut, ensuite, être enrichi d'autres outils.

L'originalité et l'innovation de notre approche, dans le contexte de l'outil, peuvent se synthétiser en cinq points :

[2] Des informations de cette approche sont disponibles à http://www.omg.org/mda ou http://www.sciences.univ-nantes.fr/info/lrsg/Recherche/mda/

1. Fournir une interface utilisateur construite à partir d'un modèle. 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 outils produits. Un autre avantage important est que les techniques utilisées permettent d'exporter les vues graphiques vers d'autres supports dont le Web.
2. Accepter et utiliser des formats non propriétaires définis par le W3C et profiter ainsi des nombreux développements réalisés autour de XML. De cette manière, le coût et le temps de développement de l'outil peuvent être fortement réduits. Notre innovation consiste à proposer des traitements sémantiques sur des documents XML, en utilisant une méthodologie de programmation basée sur le patron de conception visiteur (*visitor design pattern*) [37, 61, 62], issu de la programmation par objets.
3. Proposer une programmation par aspects [23, 47, 49] 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 [53]. 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.
4. Posséder une architecture logicielle modulaire [21] (avec des composants indépendants) et extensible pour assurer une bonne évolution de l'outil. Ainsi, il est facile d'ajouter de nouvelles fonctionnalités (importation de nouveaux composants) ou de partager notre savoir-faire (exportation de nos composants) avec d'autres plates-formes.
5. Auto-utiliser l'outil pour le développer ; ainsi, les techniques proposées sont directement testées. Par exemple, tous nos langages internes ont été développés grâce à l'outil. Chaque environnement produit réutilise les composants existants.

Justificatifs

Sur chaque point important énoncé ci-dessus, nous allons expliquer notre démarche et justifier en quoi elle est originale par rapport aux travaux existants.

Édition structurée et environnement interactif

Dans le domaine des éditeurs structurés [22, 50, 65], 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 (*Eclipse*, *Sun ONE* anciennement *Forte*, *Visual Studio*™, *JBuilder*™, *Visual Age*™, etc) sont des concurrents manifestes à l'approche générique. Mais il est important, même pour ces langages métiers [72] 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 allons montrer 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. Cette approche (avec BML - *Bean Markup Language*) rend aussi l'exportation aisée des vues graphiques à travers le réseau. Le mélange de diverses familles technologiques – la bibliothèque `Swing` pour le graphisme, l'outil XSLT pour les transformations, BML pour la *sérialisation* et CSS (*Cascading Style Sheet*) pour l'application de styles – apporte une solution efficace à faible coût de développement et de maintenance. Développer nos propres algorithmes de placement d'objets graphiques, langages et moteurs nous aurait demandé beaucoup d'énergie et de temps pour un résultat qui, bien que peut-être plus efficace (car conçus pour et non adaptés), aurait été non évolutif et non ouvert. Notre solution profite ainsi des avantages de chaque technologie et surtout de leurs futures évolutions. L'utilisation des technologies XML est un atout indéniable de notre approche.

Passerelle vers les formalismes DTD et XML Schema

Adopter des formats de données standardisés facilite l'échange d'informations entre logiciels. Le W3C donne la possibilité aux concepteurs de décrire les structures des données échangées en utilisant les formalismes DTD (*Document Type Definition*) ou XML 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. 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-supports tel que l'écran d'un PDA - *Personal Digital Assistant* -, d'un téléphone mobile, un éditeur de texte ou un navigateur Internet.

Nos efforts pour accepter les formalismes du W3C sont certes motivés par notre souci d'élargir le champ d'applications de SMARTTOOLS mais aussi par notre volonté de faciliter son utilisation. De cette manière, les utilisateurs ne sont pas contraints à apprendre nos langages internes. L'outil accepte, en entrée, aussi bien notre propre langage de définition d'AST (*Abstract Syntax Tree*) qu'une DTD et bientôt un XML Schema. 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.

En suivant la même approche, établir une passerelle vers le méta-langage d'UML, le MOF (*Meta Object Facility*), semble aussi naturel et sera expliqué, en perspectives, dans la conclusion.

Outils sémantiques

Le succès grandissant de la notion de patrons de conception [37] montre que les concepts de programmation par objets ne sont pas suffisants et que chaque type d'application ou problématique demande des solutions appropriées [60]. En particulier, le patron visiteur, base de nos outils sémantiques, a suscité un ensemble de travaux de recherche [62], 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 [51].

Certaines similitudes sur cette problématique avaient déjà été remarquées dans [27] mais pour des familles de technologies différentes (grammaires attribuées [41, 66], programmation polytypique [43] et programmation adaptative [63, 53]). La programmation adaptative suit cette même problématique de séparation des concepts (parcours et sémantique). La programmation par aspects [47] 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 [23]. Il est clair qu'il existe des liens très forts avec les travaux de recherche sur la réflexivité [54, 55] pour les langages à objets, en particulier la notion de MOP (*Meta-Object Protocol*) [48]. 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 objet.

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 des visiteurs enrichie par les mécanismes de programmation par aspects ou adaptative. L'intérêt est d'une part d'éviter les problèmes d'efficacité par cette génération et de cacher à l'utilisateur la complexité des mécanismes mis en jeu. De plus, notre approche de programmation par aspects, adaptée à nos besoins, a le mérite d'être d'une mise en œuvre très simple, par une extension naturelle du patron de conception visiteur. Avec tous ces mécanismes «cachés», une analyse peut, très facilement, être étendue, soit par héritage, soit par l'ajout d'aspects et son code est simple.

Architecture à composants

Pour l'architecture de notre logiciel, nous avons préféré suivre une approche proche de MDA en concevant notre propre modèle abstrait de composants, dédié à nos besoins (métier) et projetable vers des modèles spécifiques concrets. Les raisons d'un tel choix sont les suivantes :

- la difficulté de choisir une technologie de composants pérenne et adaptée à notre métier ;
- l'identification claire des besoins en définissant ce modèle indépendamment de toute technologie.

Avec cette approche, les besoins spécifiques sont clairement identifiés. Alors qu'une utilisation directe d'un modèle existant, non adapté à nos besoins, aurait caché les spécificités de nos composants. Lors de la phase de projection de notre modèle vers CCM, EJB ou les *Web Services*, nous nous aussi sommes aperçus que certains de nos besoins, dont l'extensibilité de services des composants, auraient été difficilement exprimables. De plus, grâce à cette technique de projection, il est très facile, par définition de nouvelles règles de transformation, d'exporter nos composants vers une plate-forme ayant un nouveau modèle de composant.

Une architecture par composants pour un tel outil (à génération de code) est nécessaire afin d'établir une nette séparation entre le code du noyau, des outils génériques et des langages. De cette façon, seuls les composants utiles peuvent être chargés en mémoire. Cette modularité est aussi souhaitable pour, aisément, exporter ou importer des composants.

Le lien conducteur de cette thèse est la programmation générative qui fédère toutes ces technologies hétéroclites, simplifie leur usage et facilite les évolutions. Cette génération, à partir de modèles, correspond à un sous-ensemble de l'approche MDA.

Depuis trois ans, l'outil est développé de manière incrémentale et a fortement évolué passant d'une version proche de Centaur [22, 68] (version 1) avec aïoli [70], VTP (*Virtual Tree Processor*), PPML (*Pretty Printing Meta Language*) à une version ouverte basée sur des techniques non-propriétaires et sur les technologies XML et donc n'ayant plus rien de commun avec Centaur. Comme l'outil est en perpétuelle évolution (développement en cours de la version 4), certains aspects énoncés dans ce manuscrit pourraient être erronés lors de la diffusion de cette version. De plus, certaines de nos expériences (visiteurs génériques) effectuées en version 3 et langages traités n'ont pas encore été portés.

Plan du manuscript

Chapitre 1 : Présentation générale de SMARTTOOLS (page 9)

Ce chapitre donne une vue d'ensemble de l'outil [29, 64], cible de nos expérimentations. Cette vue permet d'appréhender les besoins et rouages de l'outil mais aussi son évolution puisque cette présentation date d'un an et demi (version 3). Elle montre comment l'approche par génération de code et l'usage de technologies standardisées avantagent le développement et l'évolution d'un tel logiciel.

Chapitre 2 : Présentation des technologies XML (page 31)

Ce chapitre présente diverses technologies XML telles que XML, DTD et XML Schema, XSLT et XPath, BML, CSS, SOAP et WSDL. Son objectif est de fournir aux néophytes un aperçu rapide de leurs utilisations et avantages pour mieux comprendre les choix réalisés dans SMARTTOOLS. Il est indépendant de tout chapitre et donc sa lecture peut être effectuée dans n'importe quel ordre ou même omise par les personnes connaissant ces technologies.

Chapitre 3 : Outils syntaxiques (page 47)

Ce chapitre explique pourquoi nous avons créé notre propre formalisme de définition de langage, nommé ABSYNT, et présente ses notions de base. C'est le cœur du système sur lequel repose tous les autres outils. Comme SMARTTOOLS est adapté pour les langages métiers, nous établissons ou avons établi des passerelles avec leurs langages de définition, DTD et XML Schema. Ainsi l'outil est ouvert et ses utilisateurs ne sont pas contraints à apprendre notre format interne. Ce chapitre présente aussi comment spécifier des afficheurs avec notre langage de syntaxe concrète, nommé CoSynt, ou directement avec XSLT et les analyseurs syntaxiques associés.

Chapitre 4 : Outils sémantiques : visiteurs et aspects (page 75)
Ce chapitre présente différentes manières [30] d'implémenter des analyses sémantiques lisibles et facilement extensibles, ayant comme point de départ le patron de conception visiteur. Il est ainsi possible de configurer le parcours et les signatures des méthodes `visit` (évitant les coercitions de type) et de les enrichir, à l'exécution, d'aspects. Une des techniques très prometteuses découple même le parcours du code de l'analyse sémantique permettant la composition d'analyses. Ce chapitre évoque également comment des visiteurs génériques, indépendants de tout langage, pourraient être réalisés.

Chapitre 5 : Architecture par composants (page 99)
Ce chapitre explique pourquoi nous avons préféré créer notre propre modèle de composants plutôt qu'en utiliser un existant tel que CORBA ou EJB. Ce modèle est décrit [28] ainsi que les principaux composants de l'outil et sa mise en œuvre. Les avantages principaux de nos composants sont leur adéquation aux besoins, leur extensibilité de services et leur capacité à être exportés vers d'autres technologies [74].

Conclusion (page 123)
En conclusion, nous présentons les champs possibles d'application d'un tel outil et montrons comment l'approche MDA est utilisée, à différents niveaux, dans SmartTools et dans les environnements produits.

Annexe A : SmartTools mini-HowTo for designers (page 131)
Ce tutorial, destiné à des concepteurs de langages, explique comment créer un environnement minimal d'un langage avec la version 4 de SmartTools puis comment l'enrichir d'analyses sémantiques, d'afficheurs ou d'analyseurs syntaxiques.

Annexe B : Exemple de visiteurs (page 139)
Cette annexe donne, pour comparer, les exemples d'un visiteur configuré et d'un visiteur découplé de la même analyse sémantique, la vérification de type du langage Tiny.

Cet outil est le fruit du travail d'une équipe soudée composée principalement de *Didier Parigot* – chargé de recherche à l'INRIA Sophia-Antipolis –, d'*Alexandre Fau*, de *Pascal Degenne* – ingénieurs experts à l'INRIA puis au W3C –, et de moi-même. Toutes les idées présentées dans cette thèse sont issues de nos séances de «tempêtes de cerveaux» stimulantes et ont été validées par leur utilisation dans l'outil. Cette thèse correspond au travail de cette équipe

Chapitre 1

Présentation générale de SMARTTOOLS

Introduction

Ce chapitre permet d'avoir une vue d'ensemble de SMARTTOOLS. Il introduit brièvement tous les concepts utilisés pour sa construction et son utilisation. Tous ces concepts seront ensuite développés et actualisés dans les chapitres suivants. En effet, nous avons préféré baser ce chapitre, contrairement aux autres, dans la version précédente (version 3) afin de montrer l'évolution de conception de l'outil et à cause de la difficulté que nous aurions rencontrée à rédiger une présentation générale de la version en cours de développement.

Ce chapitre se décompose en cinq 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 message). La quatrième section décrit notre approche uniforme pour la conception et la réalisation des interfaces graphiques et de l'interface utilisateur de SMARTTOOLS. Enfin la cinquième présente quelques applications de notre outil.

CHANGEMENTS DANS LA VERSION 4
Les principales différences entre la version 3 et la version 4 portent essentiellement sur l'architecture du système, les informations de présentation des

vues graphiques et la définition de la structure d'un arbre de syntaxe abstraite. Les modifications suivantes ont pour but de rendre l'outil et les environnements produits plus ouverts, flexibles et proches des technologies XML :
- L'architecture n'est plus centrée autour du bus logiciel (décrit dans le paragraphe 1.3 à la page 20) mais utilise une approche par communication directe entre composants facilitant l'importation et l'exportation de composants.
- Les informations de préférences de style des éléments (couleur, positionnement, fonte) composant une vue ne sont plus gérées au niveau de la transformation de l'arbre de syntaxe abstraite en arbre de syntaxe concrète mais directement au niveau de l'arbre de syntaxe concrète en employant CSS (*Cascading Style Sheet*). Cette séparation entre le contenu et la présentation rend les vues ajustables sur mesure par l'utilisateur final de l'outil.
- La définition de structure des arbres a été enrichie des notions de fils optionnels ou tableaux afin d'être en meilleure adéquation avec les documents XML traités.

Des remarques en fin de paragraphe résument les éventuelles différences pour actualiser ce chapitre et les phrases inconsistantes ont été mises au passé.

1.1 Syntaxe abstraite et outils

Tous les outils de SMARTTOOLS sont basés sur la notion de syntaxe abstraite étendue et fortement typée (AST - *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 [22, 44, 50]. Cette section décrit le langage de définition d'AST, l'implémentation des arbres manipulés, les passerelles réalisées pour importer d'autres formats de définition d'AST et enfin les différents outils générés.

1.1.1 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 1.1 montre la définition incomplète de notre langage jouet : TINY [1]. 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 existait trois catégories de constructeurs :
- atomique sans fils ou feuille (par exemple `var`) ;
- d'arité fixe (`affect`) ;
- d'arité variable (liste) à type fixe (`statements`).

Il est aussi possible de déclarer des informations associées aux constructeurs sous forme d'annotations typées plus communément appelées attributs. Par exemple, des sucres syn-

[1] Langage utilisé comme fil d'Ariane au cours de ce manuscrit.

```
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>
...
```

FIG. 1.1: Une partie de la définition d'AST de notre langage jouet TINY avec notre langage interne (à gauche) et son équivalence en DTD (à droite)

taxiques (Figure 1.2 pour le constructeur `affect`) pouvaient être spécifiés sous forme d'attributs pour la génération d'analyseurs syntaxiques et d'afficheurs associé.

```
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'
>
```

FIG. 1.2: Définition du constructeur `affect` avec les sucres syntaxiques utiles à la génération d'un analyseur syntaxique et de l'afficheur associé ; à gauche avec notre langage interne et à droite en DTD

CHANGEMENTS DANS LA VERSION 4

Afin d'accepter un plus large éventail de définitions de langages ou de documents, les notions de fils optionnel et de fils tableau ont été rajoutées en version 4. Ce changement était nécessaire pour être plus proche des structures des documents XML et perdre moins d'informations de typage. Ainsi, il est maintenant possible d'écrire `op(A[] aList, B? bSon, C cSon, D? dSon)` qui indique que le premier fils du constructeur `op` est un tableau de `A`, le deuxième optionnel de type `B`, le troisième obligatoire de type `C` et le quatrième optionnel de type `D`. La contrainte de cette catégorie de constructeurs est que les types des fils optionnels ou tableaux placés devant un fils requis soient disjoints entre eux et aussi disjoints avec le type du fils requis. Dans notre exemple, il est impératif d'avoir $A \cap B =$, $A \cap C =$ et $B \cap C =$ et il est possible d'avoir $C \cap D \neq$. Cette contrainte est nécessaire afin de savoir à quel fils affecter un nœud. Un tel constructeur, en version 3, aurait été spécifié `op(T[] tList)` avec `T = %A, %B, %C, %D`, ce qui aurait impliqué une granularité de typage de l'arbre moins fine. La catégorie des constructeurs de liste a ainsi disparu puisqu'elle se trouve incluse, par construction, dans cette nouvelle catégorie de constructeurs à arité de type (fils) fixe et de nœuds variable.

Pour augmenter la lisibilité des définitions de structures, les syntaxes abstraite et concrète de ce langage (renommé ABSYNT au lieu d'AST, nom qui prêtait à confusion) ont aussi été légèrement modifiées. Une définition se décompose maintenant en trois parties : la première contenant les informations de typage, la deuxième les attributs (avec factorisation possible d'un attribut sur plusieurs constructeurs) et la troisième des informations supplémentaires utiles pour les calculs sémantiques mais non incluses dans les documents XML.

Afin de ne plus «polluer» les documents XML d'attributs réservés à la génération d'afficheurs et d'analyseurs syntaxiques, un nouveau langage, nommé COSYNT, (voir section 3.4 page 69) a été défini. Ainsi toutes les informations relatives à une syntaxe concrète d'un langage (sauf les informations de style gérées dans un fichier CSS) sont maintenant stockées dans un document COSYNT et non dans la définition ABSYNT du langage.

1.1.2 Implémentation 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 1.2.1 page 16) 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 1.3. 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 1.1). Par constructeur, SMARTTOOLS génère une classe et une interface (la Figure 1.4 montre l'interface générée pour le constructeur `affect`) et une interface par type ; celle-ci est implémentée par tous les constructeurs qui sont inclus dans ce type.

Chaque classe Java décrivant un constructeur étend une implémentation 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 avions ajouté la possibilité d'avoir des attributs de type autre

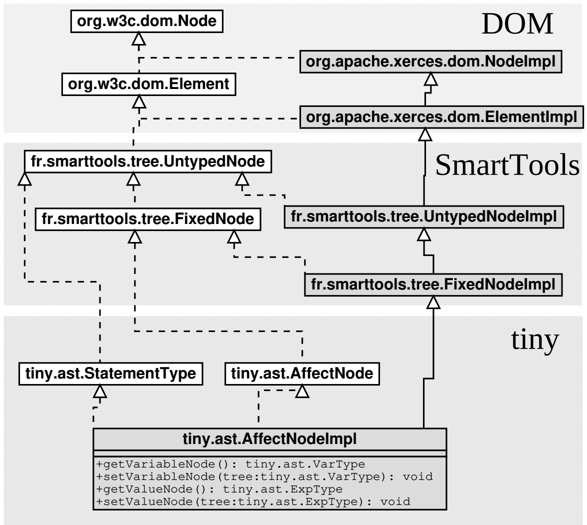


FIG. 1.3: Schéma du graphe d'héritage du constructeur `affect`.

```
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();
}
```

FIG. 1.4: Interface `AffectNode` générée

que `String` mais ils étaient volatiles. Ils n'apparaissaient pas dans le format XML du document (programme) et étaient perdus lors de la sauvegarde (*serialisation*). Donc les attributs étaient soit de type `String` à valeur constante, obligatoire ou optionnelle, soit de type quelconque mais volatiles (c'étaient 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 (par exemple, pour construire une représentation graphique de l'arbre) qui ne reposent que sur la catégorie des constructeurs (voir paragraphe 4.4 page 96).

1.1.3 Passerelles pour importer d'autres formalismes (DTD, XML 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, XML 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 traiter le plus de DTDs 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 XML 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 XML Schema vers notre formalisme.

1.1.4 Outils générés

Le format XML d'un langage est essentiellement un format d'échange de données entre applications, pas vraiment adapté pour l'édition et la manipulation directes. Pour contourner ce problème, le concepteur peut définir une «vraie» syntaxe concrète à son langage (voir la Figure 1.5), 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 pouvait être automatisée. Le concepteur devait juste indiquer en supplément dans la définition d'AST les expressions régulières et les sucres syntaxiques (cf. Figure 1.2 attributs `afterOp` et `separator1`) enrobant les constructeurs. Avec ces informations (ajoutées aux constructeurs), notre outil pouvait produire la spécification d'un analyseur syntaxique pour le générateur ANTLR [1] composée d'une partie lexicale et d'une partie syntaxique avec les fonctions de construction d'arbre correctement typées. Il aurait été très facile de l'adapter à d'autres formats d'analyseur syntaxique LL(k) écrits en Java dont JavaCC [10]. Par contre, il aurait fallu modifier l'algorithme de génération pour d'autres méthodes d'analyse syntaxique, comme la méthode LALR du générateur CUP [5]. Notre outil générait aussi une spécification de l'afficheur associé (cf. partie 1.4.3 page 25) décrivant comment représenter les constructeurs. Cette possibilité était utilisée pour deux de nos langages internes (ViProfile et Xpp présentés respectivement en 1.2.1 et 1.4.3).

```
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>
```

Fig. 1.5: Programme Tiny (table de multiplication de 2) écrit en utilisant la syntaxe concrète de Tiny (à gauche) ou en XML (à droite).

La Figure 1.6 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. paragraphe 1.1.2),
– les classes de base utiles pour définir des analyses sémantiques (voir section suivante),
– un analyseur syntaxique et l'afficheur associé si des informations complémentaires étaient ajoutées à la définition du langage,
– un fichier de ressources minimal qui contenait 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 XML Schema.

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

Ces éléments syntaxiques seront mieux détaillés et actualisés dans le chapitre 3.

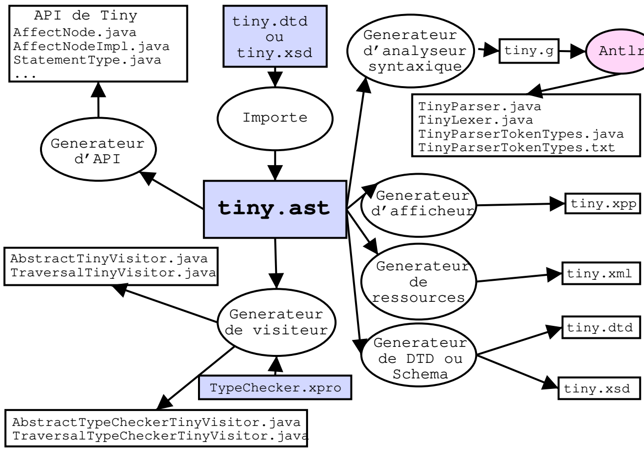


FIG. 1.6: L'ensemble des spécifications générées à partir d'une définition d'AST

1.2 Traitements sémantiques

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

1.2.1 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 1.3), 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 [61] 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 VIPROFILE. 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 1.7 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 1.9 montre la même méthode `visit` que la Figure 1.8 : 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   ...
```

FIG. 1.7: Partie du fichier de personnalisation de l'évaluateur de TINY

Le langage VIPROFILE 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

```
public Object visit(WhileNode node, Object env) throws VisitorException {
   Boolean cond = (Boolean)visit(node.getCondNode(),env);

   if (cond.booleanValue()) {//tantque condition vérifiée
      visit(node.getStatementsNode(), env); //execute le bloc
      visit(node, env); //ré-exécute la visite sur ce noeud -> récursion
   }
   return null;
}
```

FIG. 1.8: Evaluation du constructeur `while` sans profil

```
public Object eval(WhileNode node, TinyEnv env) throws VisitorException {
   Boolean cond = evalBoolean(node.getCondNode(),env);

   if (cond.booleanValue()) {//tantque condition vérifiée
      eval(node.getStatementsNode(), env); //execute le bloc
      eval(node, env); //ré-exécute la visite sur ce noeud -> récursion
   }
   return null;
}
```

FIG. 1.9: Evaluation du constructeur `while` avec un profil

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 était 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 [59]. Nous utilisions 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 avions utilisé dans un premier temps une implantation des multi-méthodes pour Java [35] 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 avait 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, indiquait pour chaque couple `(type, constructeur)` la référence de la méthode à appeler.

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

CHANGEMENTS DANS LA VERSION 4
L'implémentation de notre mécanisme de visiteurs a été améliorée grâce à l'ajout d'un mécanisme de cache des méthodes `visit` permettant d'éviter toute réflexivité.

1.2.2 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 [49] spécifique à nos visiteurs sans transformation de programme, contrairement aux premiers outils d'AOP (*Aspect-Oriented Programming*) [3]. 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 1.10 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.

```
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());
   }
}
```

FIG. 1.10: Code d'un aspect traçant les méthodes `visit` appelées

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 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. Nous avons aussi réalisé 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.

1.3 L'architecture de SmartTools

Cette section présentait brièvement l'architecture modulaire de SMARTTOOLS [2]. La motivation principale de cette architecture était 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 est majoritairement de type asynchrone : l'émetteur ne reste pas bloqué en attente du résultat du récepteur. Comme le nombre de composants pouvait devenir important, un mécanisme d'aiguillage des événements (contrôleur de messages) était nécessaire pour gérer l'ensemble des communications.

SMARTTOOLS était donc composé d'un ensemble de composants qui échangeaient des messages (événements typés) de manière asynchrone à travers le contrôleur de messages. Le comportement d'un composant était défini par l'ensemble des types de messages qu'il pouvait recevoir et émettre. Un composant devait, tout d'abord, s'enregistrer auprès du contrôleur de messages et indiquer les types de messages qu'il pouvait traiter.

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

Les principaux composants étaient les suivants :

- **Document**
  Chaque document contient un AST. Dans la Figure 1.11, `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 choisissait l'analyseur syntaxique approprié en fonction de l'extension du fichier reçu. Puis il exécutait cet analyseur pour produire l'AST. Le gestionnaire de documents construisait des composants document à partir d'ASTs reçus et les connectait au contrôleur de messages.
- **Base**
  La base était un composant qui contenait toutes les définitions de ressources utilisées par SMARTTOOLS : définitions de styles, de menus, de couleurs, de fontes, etc. Ces définitions étaient stockées dans la base sous forme d'ASTs.

Pour fixer les idées, la Figure 1.11 montrait une configuration possible de ces divers

[2] Attention, l'architecture a beaucoup évoluée entre les versions 3 et 4.

types de composants.

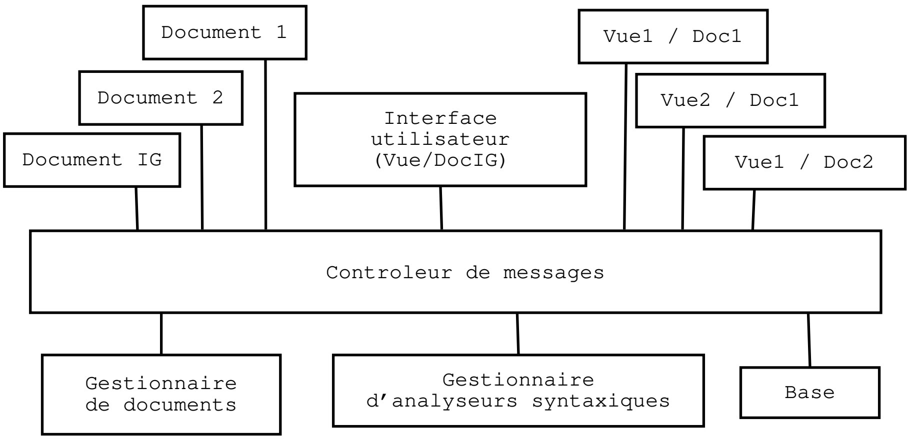


FIG. 1.11: L'architecture de SMARTTOOLS

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

L'ensemble des types de messages était extensible. C'était 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 étaient 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 était commune à tous les types de messages et était donc gérée par une classe abstraite dont ils héritaient. La deuxième partie était spécifique à chaque type de message et contenait les données utiles. Les données les plus couramment échangées entre ces composants étaient des arbres. Le format XML comme protocole d'arbre était naturellement le bon support pour ces échanges. Ce format possèdait l'avantage d'être proche de la structure des messages SOAP. Dans un tel message, 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`), puis les données utiles dans le corps du message (balise `BODY`). C'est ainsi que le contrôleur de messages de SMARTTOOLS avait été doté de filtres capables d'importer ou d'exporter des messages en respectant les spécifications SOAP. Cela offrait à SMARTTOOLS la capacité d'échanger des messages à travers un réseau avec des modules qui n'étaient pas forcément écrits en Java mais qui voulaient bénéficier de certaines fonctionnalités de SMARTTOOLS.

CHANGEMENTS DANS LA VERSION 4

L'architecture a complétement été modifiée afin de faciliter l'évolution de l'outil qui était dépendant du bus logiciel. En effet, il était impossible d'exporter un de nos «composants» sans le bus logiciel qui gérait toutes les communications. Seules des personnes connaissant très bien l'outil pouvaient ajouter ou extraire des composants. De plus, les descriptions des fonctionnalités des composants étaient masquées. Il était donc nécessaire d'avoir de vrais composants indépendants du médium de communication, pouvant dialoguer directement entre eux par échanges de services et facilement connectables. Nous avons défini notre propre modèle de composants comme expliqué dans le chapitre 5 en parfaite adéquation avec nos besoins mais surtout avec la possibilité de projeter leur descriptif vers des technologies plus connues telles que CCM, EJB ou les *Web-Services*. Les principaux types de composants sont les documents - un par langage traité - et les composants de visualisation dont l'interface graphique, la vue d'édition structurée et la vue générique ; les «composants» base, gestionnaires d'analyseurs syntaxiques et de documents, et le bus ont ainsi disparu. Tous ces nouveaux composants sont extensibles dynamiquement par ajout de services au cours de leur cycle de vie.

1.4 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 plate-forme SmartTools.

1.4.1 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 1.12). Il est conçu de manière à conserver en permanence l'isomorphisme entre les vues et le document.

Comme tous les composants de SmartTools, un document et ses vues s'échangeaient des messages dont le contenu était codé dans un format XML. Cela impliquait qu'il n'y avait pas de référence directe entre les nœuds de l'AST et les objets graphiques de ses vues. L'isomorphisme était alors garanti par échanges de messages (voir Figure 1.13) contenant trois types d'informations : l'action, son emplacement et éventuellement le sous-arbre modifié. L'action était 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.

1.4.2 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. C'est cette

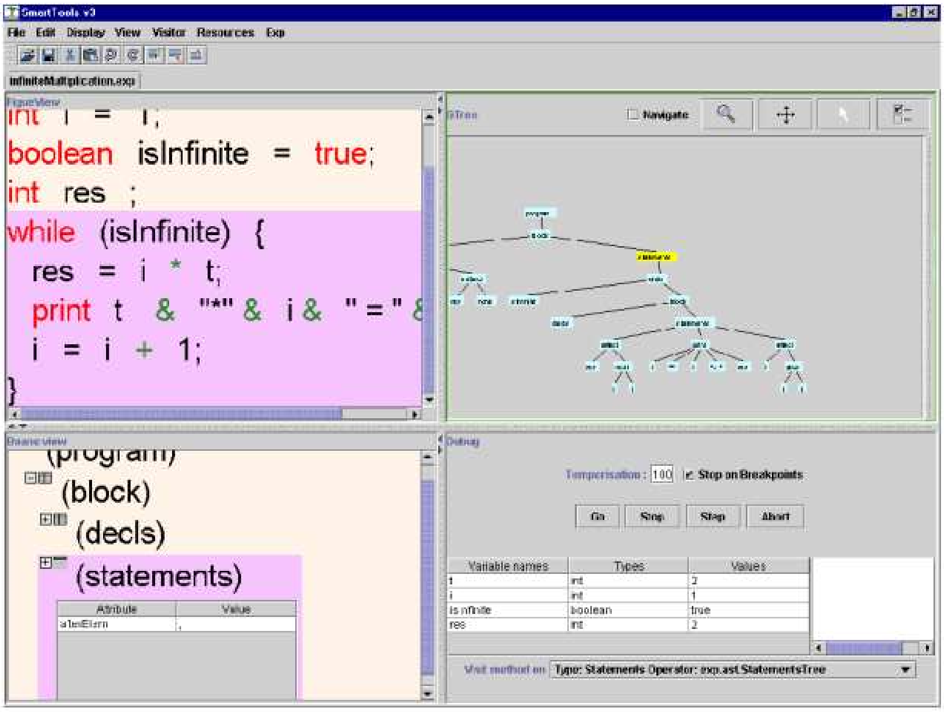


FIG. 1.12: Exemple d'interface utilisateur composée de quatre vues différentes du même AST : la vue textuelle (syntaxe concrète) située en haut à gauche, la vue sous forme d'arbre graphique à droite, la vue sous forme de menus et d'attributs en bas à gauche, et enfin la vue du *mode debug* d'un visiteur d'évaluation.

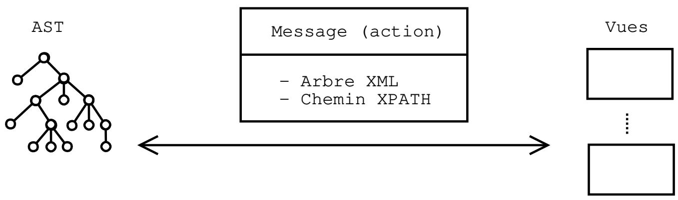


FIG. 1.13: Communication entre le document et ses vues

deuxième technique (voir Figure 1.14) 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 1.5.1 montre comment cette technologie facilite l'exportation de vues vers un navigateur Web.

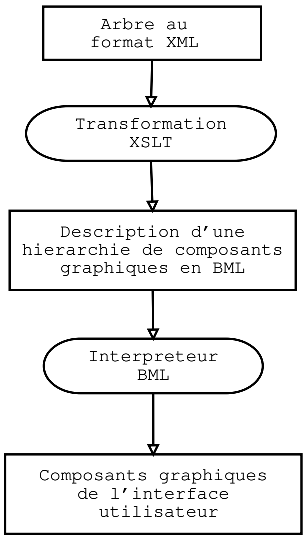
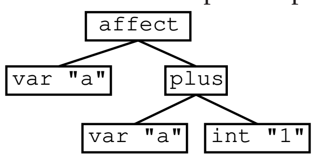


FIG. 1.14: Processus de transformation.

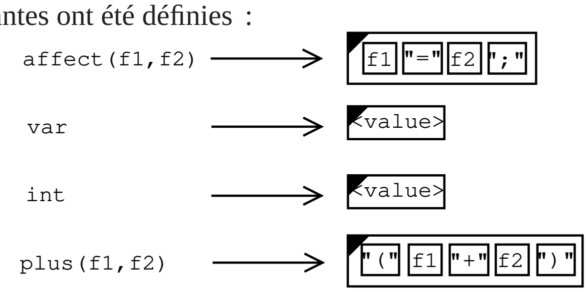
Les composants graphiques utilisés pour la version Java des vues sont basés sur la bibliothèque `Swing`. Le moteur XSLT ne produisant que du format texte, il faut disposer d'un format de description des objets et de l'interpréteur associé pour les construire. Le format BML (*Bean Markup Language*) [42] 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 :

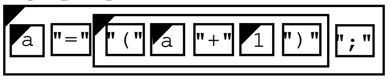


Pour obtenir une vue textuelle (syntaxe concrète) de cet arbre, les transformations suivantes ont été définies :



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 1.12 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.

CHANGEMENTS DANS LA VERSION 4

Le processus de transformation (Figure 1.14) a été enrichi par l'application d'une feuille de style CSS après la transformation XSLT. De cette façon, l'utilisateur final peut lui-même changer les styles (fontes, couleurs, positionnement, etc) à appliquer pour l'affichage de ses documents. Dans la version 3, ces informations étaient figées lors de la transformation XSLT et donc difficilement modifiables.

Nous nous sommes aussi aperçus que la composition de l'interface graphique constituait un AST et que son organisation et la mise en pages des vues étaient une vue possible de ce document. Toutes les techniques d'arbre et d'afficheurs, points forts de l'outil, pouvaient ainsi être réutilisées à moindre coût et pour une meilleure qualité logicielle ; il y a moins de code à écrire et à maintenir spécifique à l'interface graphique, et plus de tests dûs à la réutilisation. Dans la version 4, l'interface dépend donc d'un document (de langage LML) et est obtenue suivant le même processus de transformation que les vues (XSLT, BML). L'effet de bord de cette technique est d'obtenir gratuitement la sauvegarde de l'état de l'interface (persistance) et sa possible configuration selon les langages.

1.4.3 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 avaient amenés à définir un langage au-dessus de XSLT, nommé Xpp, ayant une syntaxe plus simple et concise (voir Figures 1.15 et 1.16). Le langage Xpp était un préprocesseur à XSLT. De plus, dans Xpp, seules les transformations descendantes étaient autorisées, ce qui respectait ainsi la contrainte d'incrémentalité.

Les fichiers Xpp consistaient en un ensemble de règles de transformations et de fonctions. Les règles de transformation (voir Figure 1.15) étaient 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(";"));
```

FIG. 1.15: Exemple de règle 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>
```

FIG. 1.16: Règle de la Figure 1.15 exprimée en XSLT.

Ces règles permettaient 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 1.17.

Chaque format de sortie possédait une feuille de style définissant les fonctions de formatage (`label`, `h`, etc). Le processus de transformation s'effectuait en deux étapes :
– La première étape transformait le fichier Xpp du langage avec la feuille de style du format de sortie souhaité ;
– Puis, la feuille de style obtenue était appliquée au document.

La feuille de style de chaque format de sortie définissait la traduction de fonctions de mise en forme ; par exemple, la fonction d'alignement horizontal, `h`, en BML, HTML ou texte. Il était possible d'étendre Xpp en ajoutant de nouvelles fonctions de mise en forme.

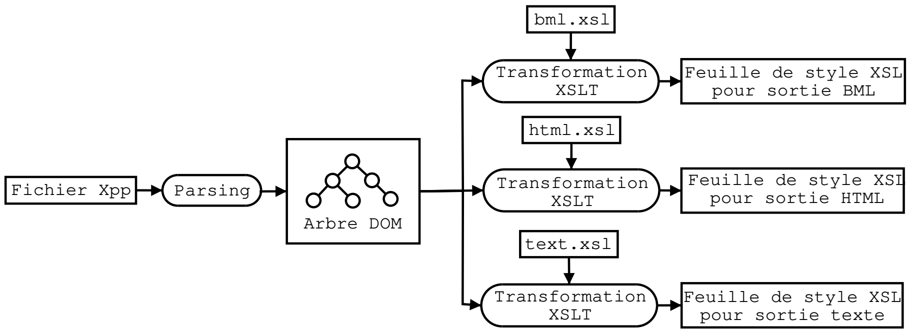


FIG. 1.17: Processus de génération des feuilles de style décrites par le fichier Xpp pour chaque format de sortie.

Il suffisait de définir le résultat de ces fonctions dans les trois fichiers XSLT, pour pouvoir les utiliser ensuite dans Xpp.

Le but était de mettre à disposition des utilisateurs un ensemble de boîtes d'affichage (horizontales, verticales). Ceux-ci n'avaient 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 gardait cependant la possibilité de récrire ou d'ajouter des fonctions si l'ensemble des fonctions prédéfinies ne lui suffisait pas.

CHANGEMENTS DANS LA VERSION 4

Ce langage n'a pas été porté car il présente moins d'intérêt comme les décisions de style pour les vues sont gérées au niveau d'une feuille de style CSS et non à la transformation XSLT. Notre priorité était de visualiser les documents sous forme de vues graphiques et non de pages Web (HTML), documents PDF ou texte ascii.

Nous avons conçu le langage COSYNT (voir paragraphe 3.4 page 69) pour décrire des syntaxes concrètes et des afficheurs aux langages traités. Il contient les divers sucres syntaxiques, les informations utiles à la génération d'analyseurs syntaxiques (valeurs des *lookaheads* et les expressions régulières de l'analyseur lexical), les noms des objets graphiques à utiliser et leur emplacement. A partir d'une telle description, il est possible de générer les règles de transformation XSLT pour obtenir un afficheur, l'analyseur syntaxique associé et un fichier CSS de base.

1.5 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 30% de son code source est automatiquement généré (55% en comptant les différents langages traités).

1.5.1 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 avions expérimenté ces concepts par l'utilisation d'*applets* pour les clients et de *servlet* connectée au bus SMARTTOOLS pour le serveur (voir Figure 1.18). On peut généraliser cette approche en utilisant les *Web Services* et le protocole SOAP. Ainsi, nous avions présenté la sélection d'un nœud dans l'AST sous la forme d'un *Web Service*. Puis dans la plate-forme .NET, nous avions écrit un client .NET en C# [73] qui faisait appel à ce service de sélection par le protocole SOAP. Cette expérience nous avait montré qu'il était possible de connecter SMARTTOOLS à des environnements hétérogènes.

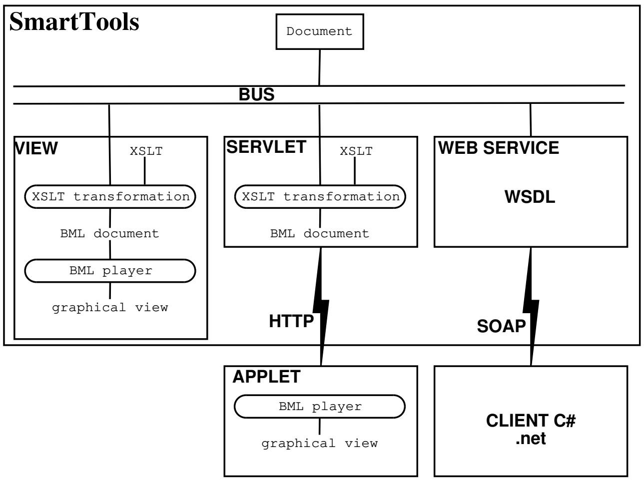


FIG. 1.18: Différent types d'accès à SMARTTOOLS

1.5.2 Environnements dédiés

Accepter le format DTD en entrée de notre outil présente l'intérêt de pouvoir produire un environnement minimal (voir Figure 1.6, page 16) pour n'importe quel langage défini avec une DTD. L'exemple type avait été la réalisation d'un environnement pour l'outil Ant [2] 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 avait permis de construire cet environnement avec un coût de développement quasiment nul.

Un autre outil avait 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 avait été réalisée avec des visiteurs et l'affichage proprement dit effectué grâce à l'intégration d'une vue de *Batik* [8] dans SMARTTOOLS.

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

CHANGEMENTS DANS LA VERSION 4
La conclusion décrit les possibles champs d'application de SMARTTOOLS.

Conclusion

Les points forts de cette version (version 3) sont les suivants :

– l'introduction, avec succès, des formalismes liés à XML pour éviter l'usage exclusif de langages ou moteurs propriétaires et ouvrir de nouvelles perspectives à l'outil. L'utilisation combinée de XSLT, BML et de la bibliothèque `Swing` a fait disparaître le code relatif à la création et à l'affichage d'arbres de boîtes (bibliothèque FIGUE [71], langage PPML [70]). Les gains sont multiples : l'acquisition d'une bibliothèque de boîtes (`Swing`) beaucoup plus riche et facilement extensible, l'absence de maintenance de code, l'évolution de nos techniques liées à celle de ces outils (donc gratuite pour nous) et la facilité d'apprentissage de l'outil pour des débutants familiers au monde XML.
  L'acceptation du format DTD comme définition de structure d'arbres ouvre aussi un nouvel horizon (et donc de nouveaux utilisateurs) à l'outil : celui des langages métiers (ou DSL - *Domain Specific Language*).
– des arbres correctement typés (nécessaires aux traitements sémantiques) basés au dessus-de l'API DOM. Toutes les fonctionnalités de cette API peuvent ainsi être utilisées dont celle de la sélection XPath auxquelles ont été ajoutées nos fonctionnalités spécifiques dont l'accès à la description de la structure (le formalisme). Les nœuds des arbres instancient des classes automatiquement générées à partir de la définition de structure ;
– des traitements sémantiques efficaces à base du patron de conception visiteur et de programmation par aspect.

Malheureusement, son architecture était trop figée statiquement et dépendante du bus logiciel - point névralgique de toute communication entre «composants» - ce qui était un obstacle à l'évolution de l'outil. La connexion de nouveaux composants ou l'extraction de nos composants sans le bus étaient des tâches difficiles à réaliser. Ces inconvénients étaient majeurs pour un tel outil dont la finalité est d'aider à la création de nouveaux environnements à composants hétéroclites. Cela justifie pleinement notre décision de ré-écrire complètement l'outil pour avoir une architecture à composants ouverte et flexible, tout en conservant les points forts de la version précédente.

Les objectifs de cette nouvelle version sont donc les suivants, rangés selon l'ordre d'importance :

1. Faciliter la création, l'ajout et la génération de composants ainsi que le déploiement d'une application ;
2. Se rapprocher et utiliser au maximum les spécifications et outils issus du W3C ;
3. Rendre l'interface graphique configurable et sa gestion plus aisée et uniforme ;

4. Permettre la configuration des styles à l'utilisateur final et ne plus les figer au niveau du concepteur des vues.

Chapitre 2

Présentation des technologies XML

L'objectif de ce chapitre est de présenter brièvement les technologies XML [16] utilisées dans SMARTTOOLS, telles que DTD et XML Schema, XSLT et XPath, BML, CSS, SOAP et WDSL. Certains aspects de ces technologies sont expliqués pour aider à la compréhension des choix de conception et du fonctionnement de notre outil. La lecture de ce chapitre, bien que facultative, est fortement recommandée aux néophytes de ces technologies. Les chapitres suivants présupposent de cette connaissance et n'incluent donc aucun autre rappel pour éviter toute dispersion.

XML

XML (*Extensible Markup Language*) est un langage à balise extensible créé en 1996 par le W3C (*World Wide Web Consortium*) comme langage pivot à tout échange d'information entre applications. C'est un sous-ensemble (ou version «allégée») de la norme SGML (*Standard Generalized Markup Language* ou ISO 8879), utile pour décrire des documents, mais restreint aux besoins du Web. Bien qu'aussi flexible que ce méta-langage et avec la même puissance d'expression, il est plus simple d'usage et donc aussi bien accepté qu'HTML (*Hyper Text Markup Language*) qui, lui, n'est qu'une instance de SGML. XML est en train de devenir la LINGUA FRANCA de l'informatique, utilisé aussi bien dans le monde des documents - le Web qui était son but initial, la documentation (pas seulement technique comme SGML) et la bureautique - que dans le monde des données - les SGBDs (*Système de Gestion de Base de Données*), la gestion et les applications scientifiques.

Contrairement à HTML, XML est extensible (balises - *tags* - non fixées), peut définir une structure de données complexe (à plusieurs niveaux d'imbrication) et la conformité structurelle de ses documents peut être validée. Ses utilisateurs peuvent choisir librement les attributs et les noms des balises (parmi les caractères unicode) qui qualifient sémantiquement et non graphiquement leurs données ; les balises ne correspondent plus à des informations de présentation. XML est donc un *méta-langage* pour créer des documents structurés à base d'éléments imbriqués et d'attributs, ne contenant aucune information de présentation. Cette séparation entre les données/documents et les traitements/présentations donne une plus grande réutilisabilité et longévité aux données.

La suite de ce paragraphe présente trois notions importantes : celle de document bien-formé - correct vis-à-vis de la syntaxe XML -, de document valide - conforme à une définition de structure - et d'espace de nom utile pour donner un nom unique à chaque balise et ainsi éviter des conflits de nom lors de la combinaison de différentes définitions de structures. Pour compléter les connaissances sur XML, les interfaces de programmation les plus usuelles sont très brièvement introduites.

Document bien formé

Par son format simple, non ambigu, indépendant de toute plate-forme et générique, XML facilite les échanges de données de types très différents entre applications hétérogènes et les traitements par des outils standardisés. Ce format texte, assez verbeux mais facilement compressible, est régi par des règles strictes pour assurer que tout document XML soit *bien formé*. Les principales règles sont les suivantes :

- tout document doit commencer par un entête (*prolog*) - indiquant, au minimum, le numéro de version d'XML - et n'avoir qu'un seul élément, la racine de la structure arborescente des données ;
- toute utilisation de fragment de contenu (entité) externe est proscrite à moins qu'une définition de structure du document (DTD) ne soit précisée (par l'ajout de la ligne <!DOCTYPE ...> entre l'entête et l'unique élément). Toute définition récursive d'entité est aussi interdite ;
- tout nom d'élément ou d'attribut doit commencer par une lettre ou un souligné (*underscore*) suivi, optionnellement, par des lettres, chiffres, traits d'union, points, deux points ou soulignés. La chaîne xml, quelle que soit sa casse, en début des noms est interdite (réservée pour de futurs usages) ;
- toute balise ouverte doit être fermée, soit par /> si l'élément est vide, soit par la balise fermante correspondante (</*nomBalise*>) ;
- ces deux balises (ouvrante et fermante) doivent être correctement imbriquées entre les balises de l'élément parent (pas de recouvrement) et de casse identique ;
- tout attribut ne doit apparaître qu'une seule fois dans un élément, doit avoir sa valeur entourée par des guillemets et de type chaîne non *parsable* (CDATA) si aucune DTD n'est précisée ;
- les caractères inférieur (<) et esperluette (&) ou la séquence]]> doivent être remplacés respectivement par les références d'entités générales prédéfinies <, & et]]> sauf dans les CDATA.

L'exemple de la Figure 2.1 montre un document XML bien-formé avec un entête (première ligne) et une seule racine bibliographie qui n'a qu'un livre, *le dragon*.

```
<?xml version="1.0" standalone="yes" encoding="UTF-8"?>
<!-- voila un commentaire non utile -->
<bibliographie>
  <livre surnom="le dragon" année="1986"
         titre="Compilers: Principle, Techniques and Tools" édiRef="7">
    <auteur>Aho, Alfred</auteur>
    <auteur>Sethi, Ravi</auteur>
    <auteur>Ullman, Jeffrey</auteur>
  </livre>
  <éditeur nom="Addison Wesley" id="7" site="http://www.aw.com"/>
</bibliographie>
```

FIG. 2.1: Exemple de document bien formé.

Document valide

Il est aussi possible de définir la structure des données (obligatoire en SGML) et donc de spécifier un langage métier ou vocabulaire adapté à ses besoins avec une DTD ou un XML Schema. De cette manière, toute erreur de structure des documents peut être détectée, avant tout traitement par les applications, lors de l'analyse syntaxique. Il n'est donc pas nécessaire de «polluer» le code des applicatifs par des vérifications structurelles et des traitements d'erreur. Un document qui est bien formé et qui est conforme à sa définition de structure est dit *valide*.

De nombreux langages métiers ou vocabulaires sont ainsi créés dans des domaines d'activité très variés tels que les mathématiques (MathML - *Mathematical Markup Language*), la chimie (CML - *Chemical Markup Language*), la génétique (BSML - *Bioinformatic Sequence Markup Language*), la médecine, le commerce, (BizTalk ou CommerceXML), les finances (OFX - *Open Financial eXchange*), les applications sans fils (WML - *Wireless Markup Language*), les présentation multimédia (SMIL - *Synchronized Multimedia Integration Language*), le dessin vectoriel (SVG - *Scalable Vector Graphics*), etc.

Espace de nom ou *Namespace*

Il existe une pléiade de définitions de langages métiers dans des communautés totalement différentes. Une même balise peut ainsi avoir été déclarée avec des modèles de contenu complètement différents dans plusieurs langages. En combinant différents langages pour en créer un nouveau, il est possible d'obtenir des conflits de nom. Pour résoudre ce problème, le W3C a proposé de définir des espaces de nom ou *Namespaces* qui limitent la portée des noms. Par analogie, un espace de nom correspond à un *package* Java. Un élément est donc qualifier/identifier de manière unique par l'espace de nom auquel il appartient et par son nom. Il suffit de définir l'attribut `xmlns` avec une URI (*Uniform Resource Identifier*) pour préciser l'espace de nom par défaut de élément courant et de ses sous-éléments. Pour indiquer qu'un élément appartient à un autre espace, un préfixe identifiant l'espace est ajouté au nom de l'élément ainsi que le caractère séparateur deux points (`monPrefixe:monElement`). Ce préfixe n'est visible que dans l'élément où il est défini (avec `xmlns:monPrefixe="URI"`) et ses sous-éléments.C'est l'analyseur syntaxique qui le substitue par l'URI ou qui ajoute l'URI de l'espace de nom par défaut s'il existe pour obtenir des noms d'éléments uniques.

```
<?xml version="1.0" standalone="yes" encoding="UTF-8"?>
<monPrefixe:bibliographie xmlns:monPrefixe="http://www-sop.inria.fr/oasis">
   ...
</monPrefixe:bibliographie>
```

Cette notion d'espace de nom a été ajoutée après la parution des spécifications de XML et est difficilement utilisable dans une DTD car elle n'y était pas prévue.

APIs

Il existe deux APIs (*Application Programming Interface*) standardisées très connues pour traiter le format XML : SAX (*Simple API for XML*) et DOM (*Document Object Model*, spécification du W3C). Bien que ces deux APIs analysent syntaxiquement des documents XML, leurs approches et résultats sont complètement différents.

L'API SAX fonctionne sur un mode événementiel où les traitements de l'utilisateur sont effectués pendant l'analyse. Ainsi, aucun arbre représentant la structure n'est construit en mémoire, contrairement à l'API DOM. En fait, l'analyseur syntaxique, lors d'un nouvel événement (tel que l'ouverture ou la fermeture du document, l'entrée ou la sortie d'un élément ou la lecture de texte), se déroute pour exécuter le traitant (le *handler*) associé. Ces traitants, par défaut (classe `DefaultHandler`), ne font rien. Par héritage, l'utilisateur peut ainsi facilement indiquer le code à exécuter pour chaque événement, sans modifier l'analyseur syntaxique. Cette approche est très similaire de nos visiteurs découplés et à le même inconvénient : l'utilisation de piles due à la perte de la structure physique du document (voir paragraphe 4.3 page 90).

L'API DOM, pour sa part, fonctionne sur la représentation hiérarchique à base d'objets du document XML et possède de nombreuses méthodes pour y accéder ou la modifier dynamiquement. Il est, par exemple, très facile de sélectionner un ensemble des nœuds avec une expression XPath (voir paragraphe 2.2 page 43) pour y appliquer un traitement commun. Les points faibles de cette API sont qu'il est nécessaire d'analyser syntaxiquement tout le document (pour construire cet arbre) avant tout traitement, ce qui peut être long, et que la représentation en mémoire de très gros documents peut être très coûteuse et même parfois impossible.

Ces deux APIs ont été implémentées pour différentes plates-formes et langages dont C++ et Java. Souvent les implémentations de DOM utilisent, en couche basse, l'analyseur SAX avec des traitants d'événements adaptés pour construire l'arbre. Les principales bibliothèques que nous utilisons dans SMARTTOOLS sont celles du projet *Jakarta*[1] d'Apache pour Java : *Xerces* pour les analyseurs et *Xalan* pour le moteur XSLT.

Il existe de nombreux logiciels libres ou commerciaux pour faciliter l'édition et le traitement de fichiers XML dont l'outil commercial XML Spy[2], très complet au vue de la version d'évaluation. La page http://xml.coverpages.org/software.html du site d'OASIS indique les principales pages les recensant dont http://www.garshol.priv.no/download/xmltools/ pour les logiciels libres.

[1] http://jakarta.apache.org/
[2] http://www.xmlspy.com

2.1 Les langages de définitions de structure : DTD et XML Schema

Une DTD ou un XML Schema décrit la structure des données des documents XML et sert à la validation des documents lors de l'analyse syntaxique. Les *parseurs* XML validants détectent ainsi les erreurs de syntaxe (document mal formé) et de contenu (document non valide) avant tout traitement par des applications. Cette définition de structure indique les éléments pouvant appartenir à un document ainsi que leur ordre, leur nombre, leur contenu et leurs attributs.

2.1.1 DTD

De prime abord, la lecture des spécifications des DTDs est assez difficile car il y a un mélange d'informations de différents niveaux - analyses lexicale, syntaxique (document bien-formé) et sémantique (document valide) - aussi bien pour le format DTD que pour XML puisque ces spécifications sont incluses dans celles de XML [76]. Ces spécifications sont issues de SGML et n'ont pas été modifiées pour assurer une compatibilité ascendante à la plupart des gros documents SGML. Les DTDs ont donc une syntaxe particulière, malheureusement non XML. Pour accéder aux informations d'une DTD, il est donc impossible d'utiliser les APIs SAX ou DOM (qui pourtant l'utilisent en mode validant mais aucune méthode n'a été prévue). Il faut se développer ses propres outils. Seul l'aspect description/contrainte de la structure logique du document nous intéresse car proche de la notion de syntaxe abstraite, clé de voûte de SMARTTOOLS. Nous n'aborderons pas ou peu les points relatifs à la composition d'un document logique avec des entités physiques non XML.

Une DTD est une grammaire principalement composée de déclarations d'éléments et d'attributs. Une déclaration d'*élément* indique son nom ainsi que le modèle de son contenu qui peut être soit :

- vide. Souvent de tels éléments possèdent des attributs ;

```
<!ELEMENT monElement1 EMPTY>
```

- du texte (`PCDATA`) ou du texte et des éléments mélangés et d'occurrence infinie. Le texte (données d'un `PCDATA`) doit avoir un contenu *parsable* (par exemple, le caractère < doit être remplacé par `<`) ;

```
<!ELEMENT monElement2 (#PCDATA)>
<!ELEMENT monElement3 (#PCDATA|elem1|elem2|elem3)*>
```

- des éléments. Il est possible d'indiquer que le fils de l'élément est n'importe quel autre élément déclaré (`ANY`), que c'est une séquence (de caractère séparateur `,`) ou un choix (séparateur `|`) d'éléments. Pour ces deux derniers formats, il est possible d'indiquer, sous forme EBNF (*Extensible Backus Naur Form*), des contraintes d'occurrence des éléments fils et d'imbriquer d'autres séquences ou choix. Par exemple, un élément `monElement5` (dont la structure est donnée ci-dessous) aura comme fils, dans l'ordre, un élément `elem1` puis peut-être un `elem2` et enfin un à plusieurs `elem3`. Un élément `monElement7` aura un `elem1` puis, zéro à plusieurs `elem2` ou séquences d'un `elem3` suivi de un à plusieurs `elem1`.

```
<!ELEMENT monElement4 ANY>
<!ELEMENT monElement5 (elem1, elem2?, elem3+)>
```

```
<!ELEMENT monElement6 (elem1 | elem2* | elem3)>
<!ELEMENT monElement7 (elem1, (elem2 | (elem3, elem1+))*)>
```

Une déclaration d'*attributs* indique le nom de l'élément auquel ils appartiennent, puis la liste des attributs avec pour chacun son nom, l'énumération des valeurs possibles ou son type, et une indication s'il est requis (`#REQUIRED`), optionnel (`#OPTIONAL`), ou constant (`#FIXED`, sa valeur doit dans ce cas être précisée) ou la valeur par défaut à lui attribuer s'il n'est pas défini. Les types d'attributs les plus courants sont :

– `CDATA` qui indique que cet attribut est constitué de caractères non analysés syntaxiquement. Par exemple, le caractère < n'a pas et ne doit pas être protégé par l'entité générale prédéfinie `<` ;
– `ID` qui indique que cet attribut définit une clé unique pour référencer l'élément ;
– `IDREF` qui indique que cet attribut est une clé qui référence un élément existant (plusieurs éléments peuvent être référencés avec le type `IDREFS`).

L'exemple ci-dessous indique que l'élément `monElement1` a quatre attributs : un requis qui est la clé identifiant l'élément, un constant de valeur `maValeur` et deux optionnels dont l'un doit avoir sa valeur comprise parmi celles de l'énumération.

```
<!ATTLIST monElement1 monAttr1 ID #REQUIRED
                      monAttr2 CDATA #FIXED 'maValeur'
                      monAttr3 (val1|val2|val3) #OPTIONAL>
<!ATTLIST monElement1 monAttr4 CDATA #OPTIONAL>
```

Une DTD peut aussi contenir des définitions d'*entités paramètres* ou *générales* et des références à des entités paramètres. Par analogie avec le langage C, une définition d'entité est un `#define` et une référence l'utilisation de cette constante ou macro. Une entité paramètre peut être vue comme un fragment de contenu (ou composant de factorisation) de définition de structure seulement utilisables dans une DTD ; une entité générale comme un composant de factorisation de données seulement utilisable dans un document XML. Les références aux entités paramètres peuvent être mises n'importe où dans une DTD sauf dans les attributs de type `CDATA` car elles n'y seraient pas évaluées (remplacées par leur valeur) lors de l'analyse syntaxique. Cette possibilité de factorisation rend possible la réutilisation de définitions de structure comme les séquences, les choix, les déclarations d'attributs, etc. Il est, par exemple, possible d'importer une DTD dans une autre en utilisant une entité paramètre externe. L'exemple ci-dessous montre respectivement les définitions d'une entité paramètre, d'une entité générale (utilisable dans un document XML par `&monEntite-Generale;`) et l'utilisation de l'entité paramètre.

```
<!ENTITY % monEntiteParametre "(elem1 | elem2* | elem3)">
<!ENTITY monEntiteGenerale "INRIA Sophia-Antipolis, Projet OASIS">
<!ELEMENT monElement8 (elem4, %monEntiteParametre;)>
```

Document valide par rapport à une DTD

Voici, par exemple, les principales règles de validité d'un document par rapport à une DTD :

- Tout élément présent dans le document doit avoir été déclaré, une et une seule fois, dans la DTD, être correctement positionné et avoir un contenu conforme (nombre, type et ordonnancement des fils corrects) vis-à-vis de cette déclaration ;

- Tout attribut d'un élément doit avoir été déclaré et avoir une valeur conforme au type indiqué. Par exemple, cette valeur doit être une de celles proposées si c'est une énumération. Un élément ne peut avoir qu'un attribut d'identification (de type `ID`) requis ou optionnel à valeur unique (clé). Toute clé référencée par un attribut (de type `IDREF` ou `IDREFS`) doit avoir été définie auparavant. Un attribut requis doit être initialisé dans chaque occurrence de l'élément dont il dépend.
- Toute entité doit être définie dans la DTD avant d'être référencée. Les entités générales ne peuvent être utilisées que dans les documents et les entités paramètres dans les DTDs. Si le texte de remplacement d'une entité paramètre contient une balise (ouvrante), il devra aussi avoir la balise inverse (fermante) ; idem pour les parenthèses et les symboles d'inclusion paramétrée (`<![`, `]]>`). De plus, si ce texte est utilisé pour décrire le contenu d'un élément de type séquence, choix ou texte mélangé à des éléments, il doit au moins contenir un caractère non blanc et ne pas commencer ou finir par les caractères séparateurs de ces types de contenu (`|`, `,`).
- La racine des données doit correspondre au nom donné dans la ligne du `DOCTYPE`, ligne indiquant quelle est la DTD à utiliser.

L'exemple de la Figure 2.2 présente la DTD externe validant le document de la Figure 2.3. Un document peut aussi contenir une partie (avec de possibles déclarations surchargées d'éléments, d'entités ou d'attributs) ou l'ensemble de la DTD (DTD interne) qui est prioritaire vis-à-vis de la DTD externe.

```
<!ELEMENT bibliographie (livre+, éditeur*)>
<!ELEMENT livre (auteur+)>
<!ELEMENT auteur (#PCDATA)>
<!ELEMENT éditeur EMPTY>
<!ATTLIST livre titre CDATA #REQUIRED surnom CDATA #IMPLIED isbn CDATA #IMPLIED
                année CDATA #IMPLIED édiRef IDREF #IMPLIED>
<!ATTLIST éditeur id ID #REQUIRED nom CDATA #REQUIRED site CDATA #IMPLIED
                  adresse CDATA #IMPLIED>
```

FIG. 2.2: Exemple de DTD externe.

```
<?xml version="1.0" standalone="no" encoding="UTF-8"?>
<!DOCTYPE bibliographie SYSTEM=
  "http://www-sop.inria.fr/oasis/personnel/Carine.Courbis/these/biblio.dtd">
code de la Figure 2.1 sans la ligne d'entête
```

FIG. 2.3: Exemple de document valide par rapport à la DTD de la Figure 2.2.

Inconvéniants d'une DTD

Syntaxe non XML

Une DTD a sa propre syntaxe, non XML, et aucune des APIs des *parseurs* XML validants ne possède de méthodes d'accès aux données d'une DTD. Extraire ces données demande donc quelques efforts (dont la construction d'un analyseur syntaxique) ; ce n'est pas direct.

Notion d'espace de nom non reconnue

La notion d'espace de nom n'est pas, non plus, reconnue car spécifiée après la norme XML. Aucune distinction entre les parties préfixe et nom court de l'élément n'est faite ; l'ensemble est traité comme le nom de l'élément. Pour contourner ce problème, il est nécessaire d'utiliser des entités paramètres[3] à la place des noms des éléments et donc de rendre complexe l'écriture et la lecture des DTDs. Cette notion est utile pour éviter des conflits de nom lors de la création de nouveaux langages par composition (réutilisation) d'autres définitions.

Données très peu typées

De plus, les données ne peuvent pas être fortement contraintes/typées (seulement de type `PCDATA`, `CDATA`, `ID`, etc). Il est impossible d'indiquer un type de données primitif (entier positif, date, heure, chaîne), de borner les valeurs ou le nombre de caractères, de contraindre les valeurs avec une expression régulière, etc. et de donner un intervalle du nombre d'occurrences d'un élément plus restreint et précis que de 0 (ou 1) à l'infini. De telles indications permettraient de meilleures vérifications de cohérence des données, nécessaires pour éviter de corrompre l'intégrité des tables des bases de données. Dans les formulaires de saisie, des masques de saisies pourraient, par exemple, être aussi constitués. De plus, les conversions de type des données pourraient être effectuées au niveau de l'analyseur syntaxique et non dans les applications.

2.1.2 XML Schema

A cause des inconvénients des DTDs cités ci-dessus, de nouveaux formalismes concurrents de définition de classe de documents XML sont apparus tels que RELAX (*Regular Language Description for XML*), TREX (*Tree Regular Expression for XML*), plus récemment RELAX NG d'OASIS unifiant RELAX et TREX, et XML Schema du W3C qui est issu de XML Data, SOX (*Schema for Object-oriented XML*), DDML (*Document Definition Markup Language*) et DT4DTD (*DataTypes for DTDs*). Ce dernier formalisme, XML Schema, semble avoir émergé et est maintenant adopté par la plupart des industriels et des outils commerciaux liés à XML. Par exemple, XML Spy propose un éditeur spécifique pour la création et la visualisation, sous forme graphique, de tels documents. Ce formalisme est un sur-ensemble des DTDs, donc toute DTD peut être convertie en un XML Schema. En voici ses principales caractéristiques qui sont très bien présentées (avec des exemples) dans [77] et approfondies dans la norme des structures [78] et dans celle des types de données [79] ; son cahier des charges est aussi disponible [75].

Syntaxe XML

Son contenu peut ainsi être traité par les outils standards liés à XML. Il se compose principalement de déclarations d'*éléments* et de définitions de *types*. Si celles-ci sont filles

[3] Voir, par exemple, la DTD de XML Schema http://www.w3.org/2001/XMLSchema.dtd .

de l'élément racine `<xsd:schema ...>`, elles sont dites *globales* et réutilisables ; les autres sont dites *locales*.

Une déclaration d'élément contient, au minimum, le nom de l'élément et le nom d'un type prédéfini ou global, ou bien une définition interne, locale de type décrivant son contenu et ses contraintes (ce type est dit *type anonyme*). Voici, par exemple, les déclarations de trois éléments sans attributs : de type `string`[4] (équivalent à un contenu texte, `PCDATA` en DTD), de type `monTypeCodeBarre` et d'un type anonyme constitué d'une séquence de deux éléments définis en global.

```
<xsd:element name="nom" type="xsd:string"/>
<xsd:element name="code" type="monTypeCodeBarre"/>
<xsd:element name="produit">
  <xsd:complexType>
    <xsd:sequence>
       <xsd:element ref="nom"/>
       <xsd:element ref="code"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Dans un espace de nom donné, il est interdit d'avoir deux déclarations d'éléments globaux (ou définitions de types) avec le même nom et des modèles de contenu différents. Par contre, un élément local peut avoir le même nom qu'un élément global et un contenu différent, idem pour un élément global ou non et un type ou un attribut.

Les éléments globaux sont tous des racines possibles. C'est lors de la validation des documents que les noms de la racine et du XML Schema à utiliser sont précisés, non dans les documents comme pour les DTDs. De cette façon, il est possible de valider des fragments de documents. Cette caractéristique est utile, par exemple, lors de l'édition pour ne vérifier la cohérence que des données modifiées et non de la totalité du document.

Typage des données et intervalle du nombre d'occurrences précis

Il existe deux sortes de type : les *simples* qui n'ont pas d'attribut ou de sous-élément et les autres, les *complexes*.

Un type simple représente une feuille sans attribut dans l'arbre. Il peut être un type atomique, une liste d'un type atomique ou l'union d'au moins deux types atomiques ou de liste. Un type atomique est soit un type prédéfini, soit la redéfinition/restriction d'un type prédéfini à l'aide de contraintes ou *facets*[5] telles que l'expression régulière de la valeur attendue (`pattern`), l'énumération des valeurs possibles (`enumeration`), la valeur minimale possible (`minInclusive`), etc. Un des atouts de XML Schema est la richesse[6] de ses types prédéfinis tels que `string`, `float`, `date`, `boolean`, `anyURI`, `duration`, `anyType`, etc. permettant d'accroître les vérifications des données. Il possède aussi les types des attributs des DTDs[7] pour compatibilité ascendante.

[4] xsd est par convention le préfixe référant à l'espace de nom de XML Schema `xmlns:xsd=-`http://www.w3c.org/2001/XMLSchema.

[5] Il en existe 15.

[6] Il existe 44 types de données prédéfinis dont 8 spécifiques pour les attributs.

[7] Ces types sont à réserver aux attributs.

```
<xsd:simpleType name="monTypeCodeBarre">
  <xsd:restriction base="xsd:string">
     <xsd:pattern value="[A-Za-z]2-/d{5}"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="monTypeListeEntiersPositifs">
  <xsd:list itemType="xsd:positiveInteger"/>
</xsd:simpleType>
<xsd:simpleType name="monTypeCodeBarreOuEntierPositif">
  <xsd:union memberTypes="xsd:positiveInteger monTypeCodeBarre"/>
</xsd:simpleType>
```

Un type complexe représente un nœud avec des fils ou une feuille avec des attributs dans l'arbre. Il peut correspondre à différents modèles de contenu :

- vide avec des attributs.

  ```
  <xsd:complexType name="monTypePrix1">
    <xsd:attribute name="prix" type="xsd:decimal" use="required"/>
    <xsd:attribute name="monnaie" type="xsd:string" use="required"/>
  </xsd:complexType>
  ```

- un type simple avec des attributs. Etendre le type simple d'une feuille est la seule solution pour pouvoir y déclarer des attributs.

  ```
  <xsd:complexType name="monTypePrix2">
    <xsd:simpleContent>
      <xsd:extension base="xsd:decimal">
        <xsd:attribute name="monnaie" type="xsd:string" use="required"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
  ```

- des éléments (avec ou non des attributs). Comme pour les DTDs, il est possible d'indiquer une séquence de sous-éléments ou groupes (`sequence`) ou le choix d'un sous-élément ou groupe (`choice`). L'élément `any`, qui spécifie n'importe quel élément déclaré, a lui été paramétré avec l'espace de nom auquel devra appartenir l'élément ; ce qui permet de légèrement mieux contraindre la structure de données. Un nouveau modèle de contenu (l'élément `all`) est aussi disponible pour spécifier une liste de sous-éléments pouvant apparaître dans n'importe quel ordre. Ce modèle ne peut pas être imbriqué contrairement aux choix et aux séquences (donc il est toujours placé au premier niveau) et n'autorise que des éléments individuels, non des groupes tels qu'un choix ou une séquence. Les attributs de cardinalité `minOccurs` et `maxOccurs`[8] permettent de fixer précisément le nombre d'occurrence minimum et maximum, par défaut à 1 ; `unbounded` indique illimité. Pour l'élément `all`, seules les valeurs 0 et 1 sont autorisées.
- du texte et des éléments mélangés (avec ou non des attributs), précisé par l'attribut `mixed=true` dans un élément `complexType` ou `complexContent`. Contrairement aux DTDs, il est possible d'ordonnancer et de fixer le nombre des fils (avec une séquence) du mélange, de fixer le nombre de fils avec un choix ou un `all` ou de ne pas préciser le nom du fils mais l'espace de nom avec `any`.

[8] Ces attributs peuvent être placés dans les éléments suivants : `choice`, `sequence`, `all`, `element` ou `group` non global, ou `any`.

Tout type dérive de anyType, la racine de la hiérarchie des types, qui n'indique aucune contrainte sur le contenu. Un élément dont le type n'est pas précisé ou est égal à anyType peut contenir n'importe quels éléments, du texte ou un mélange d'éléments et de texte.

Les attributs sont toujours définis à la fin de la définition d'un type complexe. Ils ne peuvent être que de type simple et sont, par défaut, optionnels. L'attribut use indique si l'attribut local déclaré est réquis, optionnel ou interdit, fixed la valeur constante à lui attribuer et default[9] la valeur à lui donner s'il n'est pas défini dans la document. Il est aussi possible de n'indiquer que l'espace de nom des attributs et non leur nombre et leurs noms avec l'élément anyAttribute.

Un élément peut aussi avoir une valeur constante ou par défaut ; cette dernière sera utilisée si l'élément est vide dans le document. Un contenu nul peut aussi avoir été autorisé (avec l'attribut nillable dans la déclaration de l'élément et xsi:nil=true[10] dans le document) pour marquer explicitement l'absence de l'information ; cette fonctionnalité peut être utile, par exemple, pour les bases de données.

Factorisation de contenu ou de déclarations d'attributs

Il existe le même mécanisme que les entités paramètres des DTDs pour factoriser la définition d'une séquence, d'un choix ou de all avec l'élément group ou d'un ensemble d'attributs avec l'élément attributeGroup. Ces groupes (qui pourront être référencés à l'aide de l'attribut ref) améliorent la lisibilité et les mises à jour.

Dérivation des types possible

Cette caractéristique de restriction ou d'extension (héritage) de type est l'un des autres atouts de XML Schema. La restriction d'un type simple consiste à indiquer des contraintes ou *facets* (voir le type monTypeCodeBarre défini ci-avant). La restriction d'un type complexe consiste à réduire le nombre d'occurrences des éléments, non à supprimer les éléments d'un certain type.

L'extension d'un type simple consiste à ajouter des attributs (voir le type monTypePrix2 défini ci-avant) pour en faire un type complexe. L'extension d'un type complexe consiste à lui rajouter séquentiellement de nouveaux éléments ou attributs. Dans un document, il est possible d'utiliser un type complexe dérivé à la place du type ancêtre dans un élément à l'aide de l'attribut xsi:type.

La dérivation d'un type peut être interdite avec l'attribut final s'il est complexe ou fixed sur les contraintes d'un type simple.

Substitution d'élément, éléments et types abstraits

Il existe aussi un mécanisme pour substituer un élément (dit *élément de tête*) par un autre qui doit avoir le même type ou un type dérivé et l'attribut substitutionGroup initialisé avec le nom de l'élément de tête. Ainsi, l'élément de tête et ses possibles remplaçants peuvent être utilisés de façon interchangeable dans le document.

[9] default n'a de sens que pour un attribut optionnel.

[10] xsi est, par convention, le préfixe référant l'espace de nom du XML Schema des instances xmlns:xsi=http://www.w3c.org/2001/XMLSchema-instance.

Les éléments et les types peuvent être déclarés comme abstraits (attribut `abstract`). Ils ne pourront pas être utilisés dans les documents mais devront être respectivement remplacés par un élément de substitution ou un type dérivé.

Gestion des espaces de nom

Pour éviter tout conflit de nom, il est préférable de limiter la portée des définitions et déclarations dans un espace de nom donné. Le formalisme XML Schema est lui bien adapté à cette utilisation d'espaces de nom. Il est possible d'indiquer l'espace de nom du langage cible (attribut `targetNamespace` de l'élément `schema`) pour les définitions et déclarations globales. Ce espace peut aussi être utilisé pour qualifier les éléments et les attributs locaux (selon les valeurs des attributs `elementFormDefault` et `attributFormDefault` de l'élément `schema`, et `form`).

Inclusion, redéfinition ou importation d'autres XML Schemas

Contrairement aux DTDs, les XML Schemas possèdent des mécanismes pour créer de nouvelles définitions de structure par composition/réutilisation d'autres définitions. Par exemple, le formalisme WSDL (voir paragraphe 2.3.4 page 45) utilise/importe le formalisme XML Schema pour sa partie définition de types ; cela a simplifié sa conception, facilite sa maintenance (car il profite des outils et de l'évolution de XML Schema) et améliore sa lisibilité.

Un ou plusieurs XML Schemas peuvent être inclus (`include`) dans un nouveau XML Schema à condition qu'ils aient le même espace de nom que ce dernier. Le seul danger réside dans de possibles conflits de nom entre les différentes définitions globales[11]. Il est aussi possible d'inclure un XML Schema (de même espace de nom) mais de redéfinir (`redefine`) certaines de ses définitions globales. Un XML Schema d'un autre espace de nom peut être utilisé par importation (`import`). Dans ce cas, ses définitions n'appartiendront pas à l'espace de nom du langage cible contrairement à une inclusion.

Mécanismes d'unicité de valeur, de clés primaires et étrangères

La valeur d'un élément, d'un attribut ou d'un *n-uplet* d'éléments ou/et d'attributs peut être déclarée comme unique (`unique`) dans un certain contexte et non pour tout le document comme avec `ID`. Une expression XPath simplifiée permet de sélectionner les éléments (le contexte) dans lesquels la valeur doit être unique.

Le même mécanisme est utilisé pour définir des clés primaires (`key`) et des clés étrangères (`keyref`), utiles pour les bases de données.

L'exemple de la Figure 2.4 présente un XML Schema équivalent à la DTD de la Figure 2.2 mais plus précis en terme de type des valeurs atomiques. Il utilise aussi la notion d'espace de nom[12] et de clés primaires et étrangères à la place de `ID` et `IDREF`.

[11] Les seuls éléments qui peuvent être globaux sont : `simpleType`, `complexType`, `element`, `attribute`, `group` et `groupAttribute`.

[12] C'est pour cette raison qu'il ne peut pas valider l'exemple de la Figure 2.1.

```
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema"
            xmlns:bib="ftp://ftp-sop.inria.fr/oasis/publications/"
            targetNamespace="ftp://ftp-sop.inria.fr/oasis/publications/"
            elementFormDefault="qualified">

<xsd:complexType name="biblio_type">
  <xsd:sequence>
     <xsd:element ref="bib:livre" maxOccurs="unbounded"/>
     <xsd:element ref="bib:éditeur" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="livre_type">
  <xsd:sequence>
     <xsd:element ref="bib:auteur" maxOccurs="unbounded"/>
  </xsd:sequence>
  <xsd:attribute name="titre" type="xsd:string" use="required"/>
  <xsd:attribute name="surnom" type="xsd:string"/>
  <xsd:attribute name="isbn" type="xsd:string"/>
  <xsd:attribute name="année" type="xsd:gYear"/>
  <xsd:attribute name="édiRef" type="unsignedShort" use="required"/>
</xsd:complexType>
<xsd:complexType name="éditeur_type">
  <xsd:attribute name="nom" type="xsd:string" use="required"/>
  <xsd:attribute name="id" type="unsignedShort" use="required"/>
  <xsd:attribute name="site" type="xsd:anyURI"/>
  <xsd:attribute name="adresse" type="xsd:string"/>
</xsd:complexType>

<xsd:element name="bibliographie" type="bib:biblio_type">
  <xsd:keyref name="foo" refer="bib:cleEdi">
    <xsd:selector xpath="bib:livre"/>
    <xsd:field xpath="@édiRef"/>
  </xsd:keyref>
  <xsd:key name="cleEdi">
    <xsd:selector xpath="bib:éditeur"/>
    <xsd:field xpath="@id"/>
  </xsd:key>
</xsd:element>
<xsd:element name="livre" type="bib:livre_type"/>
<xsd:element name="auteur" type="xsd:string"/>
<xsd:element name="éditeur" type="bib:éditeur_type"/>
</xsd:schema>
```

FIG. 2.4: Exemple de XML Schema.

2.2 Langage de transformation : XSLT (et XPath)

A FINIR

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:output method="html" indent="yes"/>

  <xsl:template match="/">
```

```
    <html>
      <head>
        <title>Bibliographie</title>
      </head>
      <body bgcolor="white">
        <xsl:apply-templates select="livre"/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="livre">
    <xsl:value-of select="nom"/> - <it>
    <xsl:for-each select="auteur">
       <xsl:value-of select="."/>
       <xsl:if test="current() != last() and last() &gt; 1"> //@@@à vérifier exp xpath
          <xsl:text>, </xsl:text>
       </xsl:if>
    </xsl:for-each>
    </it><br>
  </xsl:template>
</xsl:stylesheet>
```

- Pattern : xpath *1 pas symbolique
- Template : select, for , value , apply
- partie droite n'importe quoi !

REMARQUES
comparaison avec ppml, xerces, Parler du WebSphere d'IBM (talk de Kristoffer Rose)

2.3 Autres langages

2.3.1 BML

BML (*Bean Markup Language*) [46, 80] est un langage XML de configuration de composants *JavaBeans*. Il décrit la structure d'une application de composants en indiquant comment les *beans* sont créés, configurés et instanciés. Le but premier de la création de ce langage par des ingénieurs d'IBM était de pouvoir générer du BML pour produire une application configurée en fonction des besoins de l'utilisateur à partir d'une feuille de style XSL décrivant l'interface graphique et un document XML indiquant les besoins. Ce langage peut être soit interprété, soit compilé en un programme Java (sans réflexivité).

Un document BML se compose des éléments suivants :
- `<beans>` ou `<string>` pour la création de *beans*,
- `<event-binding>` pour indiquer quel traitement associé à un événement,
- `<call-method>` pour appeller une méthode sur le *bean*,
- `<property>` ou `<field>` pour lire ou modifier une propriété ou un attribut du *bean*.
- `<add>` pour ajouter le *bean* donné au *bean* courant ;
- `<script>` pour exécuter un script BML ou Javascript.

2.3.2 CSS

2.3.3 SOAP

SOAP (*Simple Object Access Protocol*) est un protocole pour l'échange d'informations entre objets et applications dans une architecture répartie.

2.3.4 WSDL

les 3 trucs à voir : sécurité, routing et transaction.

REMARQUES
axis

Chapitre 3

Outils syntaxiques

Dans la communauté des langages de programmation, la notion d'arbre de syntaxe abstraite est usuellement employée pour représenter un programme en mémoire. Il existe une multitude de formalismes pour décrire cette notion, souvent liée à un langage de programmation sous-jacent. Il est donc normal que ces formalismes présentent de nombreuses similitudes.

Avant de décrire notre formalisme (nommé ABSYNT), nous allons énoncer les principes de notre démarche et expliquer pourquoi nous avons créé notre propre modèle de données :

1. Avoir un formalisme de description de structure de haut niveau adapté à nos besoins et indépendant de toute technologie et langage d'implémentation. Ce modèle abstrait est la clé de voûte de SMARTTOOLS, utilisé par tous ses outils. A partir d'un tel modèle abstrait de données d'un langage, on peut générer le code nécessaire à la construction des arbres de syntaxe abstraite.
2. Établir des passerelles avec les formalismes de description de documents standardisés du W3C afin d'obtenir un outil ouvert. De cette façon, l'outil accepte en entrée, non seulement son format propriétaire de description de structure, mais aussi presque toutes les DTDs et tous les XML Schemas. Ces derniers sont, tout d'abord, convertis dans ce modèle pivot avec, parfois, quelques pertes d'information de structure. Ces pertes sont principalement dues à une description plus concrète qu'abstraite des données. De plus, tout modèle abstrait peut aussi être traduit en DTD ou XML Schema. Ces passerelles ouvrent de nouveaux champs d'applications avec les langages métiers définis avec ces formalismes et facilitent aussi leur création.

   Afin d'accepter un plus large sous-ensemble des langages définis avec les formalismes du W3C, notre modèle a dû être étendu avec la possibilité de définir des constructeurs ayant des fils optionnels ou tableaux à n'importe quel emplacement. Une telle extension n'existe pas dans les autres formalismes.

3. Avoir des interfaces de programmation, pour manipuler les fils des constructeurs, indépendantes de implémentation adoptée pour représenter les arbres. Cette sur-couche de méthodes d'accès et de modifications (`get` et `set`) des fils est générée en fonction des informations contenues dans le modèle. De cette manière, l'implémentation des arbres peut être changée sans qu'aucun des traitements (utilisant ces interfaces et non des appels directs, dépendants, eux, de l'implémentation) de l'utilisateur ne doive être modifié. Seul le générateur de l'API devra être changé pour que les corps des méthodes soient en adéquation avec la nouvelle implémentation.

A partir d'une DTD, d'un XML Schema ou d'une spécification ABSYNT, l'outil génère les classes Java (l'API) associées à ce langage, utiles pour représenter ses documents/programmes en mémoire (Figure 3.1).

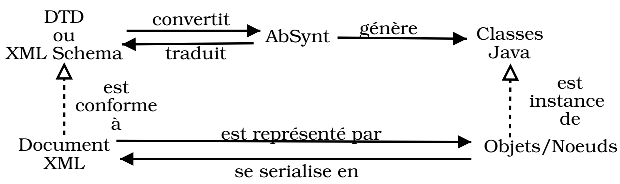


FIG. 3.1: Correspondance entre un document XML et les objets Java

A partir d'une spécification ABSYNT, on peut générer, comme le montre la Figure 3.2, l'API du langage, la DTD ou le XML Schema corespondant et des visiteurs de base ; ces derniers peuvent être personnalisés avec une spécification VIPROFILE (voir section 4.2 page 83). Les arbres de syntaxe abstraite peuvent aussi être affichés avec une forme plus lisible. Pour cela, nous proposons, à partir d'une spécification ABSYNT et d'une spécification COSYNT décrivant la syntaxe concrète et les objets graphiques à utiliser, de générer des afficheurs, écrits à l'aide de transformations XSLT. Il est aussi possible de produire l'analyseur syntaxique réciproque (spécifications ANTLR).

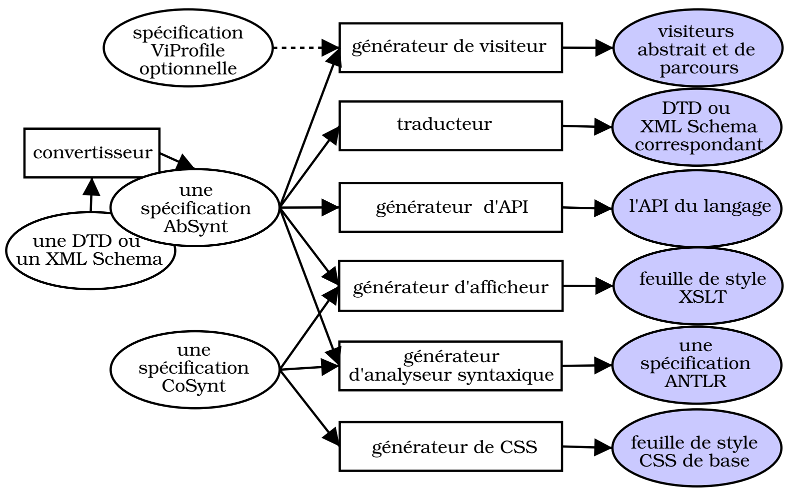


FIG. 3.2: Spécifications générées à partir d'une spécification ABSYNT ou d'une DTD

Ce chapitre se décompose en quatre sections. La première explique notre modèle abstrait de définition de langage, le langage ABSYNT. La deuxième expose comment une DTD est convertie dans notre modèle. La troisième indique comment un XML Schema pourrait être converti. Enfin la dernière section explique comment générer un afficheur (*pretty-printer*) pour visualiser les arbres de syntaxe abstraite ou un analyseur syntaxique.

3.1 Le langage de définition d'AST : ABSYNT

3.1.1 Notions de base : constructeur, type et attribut

Comme les autres langages de définition de syntaxe abstraite, ABSYNT possède les notions de *constructeur* (opérateur), de *type* et d'*attribut* (annotation). Les constructeurs représentent les nœuds des arbres et les attributs les décorations des nœuds. Afin de représenter les structures de documents XML, notre langage a été étendu avec les notions de *fils optionnels* et de *fils tableaux* (liste de nœuds d'un même type). Les types, regroupant des constructeurs en ensembles, indiquent les différents constructeurs possibles des fils. Ce langage contient aussi des définitions de données supplémentaires, utiles pour les calculs sémantiques. Une définition d'un langage en ABSYNT (voir Figure 3.3) se décompose donc en trois parties :

Définitions de types et de constructeurs

Cette partie contient les définitions de types et de constructeurs. Il existe deux catégories de constructeurs :

- les constructeurs atomiques correspondant aux feuilles (par exemple, `var`, `string` et `int` pour le langage TINY). ;
- les constructeurs à arité de types (fils) fixe et de nœuds variable (tous les autres constructeurs du langage TINY).

Ces constructeurs sont regroupés en ensembles, les types, pouvant aussi contenir des types (types inclus). Les constructeurs à arité de types fixe ont des fils typés et nommés afin de faciliter la manipulation des sous-arbres. Pour les feuilles, les seuls types possibles sont `String` et les types de base de Java.

```
Formalism of monLangage is
Root is Type;
Operator and type definitions {
Type = opAtomique as java.lang.String,
       opVariable(Type1? filsOptionnel, Type2[] filsTableau, Type3 filsRequis),
       %TypeInclus;
TypeInclus = opVariable2(); ...
}
```

Définitions d'attributs

Cette partie contient les définitions d'attributs des constructeurs et des types. Chaque attribut a un nom, peut être requis, constant ou facultatif, avoir une valeur et être de type `String` ou d'un type de base de Java. Il peut être défini sur plusieurs types et/ou constructeurs.

```
Attribute definitions {
REQUIRED attributRequis as java.lang.String in opAtomique, opVariable2;
```

```
FIXED attributConstant="2" as java.lang.Integer in %TypeInclus;
IMPLIED attributFacultatif="ardèche" as java.lang.String in opVariable;
}
```

Définitions d'informations supplémentaires utiles pour les traitements sémantiques

Cette partie contient les définitions de données additionnelles, de type complexe, pouvant être manipulées lors des traitements sémantiques. Chaque donnée est nommée, typée et peut être définie sur plusieurs constructeurs et/xou types. Contrairement aux constructeurs et aux attributs, ces informations n'apparaissent pas au niveau du document XML ; elles sont volatiles.

```
Extra data definitions {
vecteur as java.util.HashMap in opAtomique, %TypeInclus;
}
```

<u>Contraintes</u>

- Tout constructeur ou type utilisé doit être défini de manière unique (il peut être utilisé avant d'avoir été défini). Un même nom peut être utilisé pour un type, un constructeur, un attribut et une information supplémentaire[1] ;
- Les noms des fils d'un constructeur doivent être différents ;
- Tout attribut (ou information supplémentaire) ne peut être défini qu'une fois ;
- Toute définition récursive ou circulaire de type est interdite ;
- Toute définition de type doit contenir au moins un constructeur, directement ou d'un type inclus, ayant le premier fils de type différent ou non requis ;
- A cause de l'extension du langage avec les notions de fils optionnels et tableaux, il est nécessaire de contraindre les types des fils afin de pouvoir construire, sans ambiguïté, les arbres. Pour cela, il faut que les types des fils optionnels ou tableaux placés devant un fils requis soient disjoints entre eux et aussi disjoints avec le type du fils requis. Par exemple, si la définition d'un constructeur est `op(A[] a, B? b, C c, D? d)`, il faut que $A \cap B =$, $A \cap C =$ et $B \cap C =$ et il est possible d'avoir $C \cap D \neq$. Sans cette contrainte, en supposant $B \cap C = op2$, on ne saurait pas à quel fils, entre b et c, affecter un nœud de $op2$ lors de la construction des arbres.
  A cause de cette contrainte, il est impossible de définir le constructeur `if` du langage TINY de cette manière : `if(Cond c, Statement[] sThen, Statment[] sElse)`. Il est nécessaire d'utiliser un constructeur intermédiaire, `statements`.

3.1.2 Détails d'implémentation

Comme expliqué dans le paragraphe 1.1.2, l'API de manipulation d'arbres sous-adjacente est l'API DOM. Utiliser cette API standardisée permet d'être ouvert, de profiter de ses évolutions, et des implémentations existantes (par exemple, Xerces d'Apache). Cette API DOM offre de nombreux services dont la sélection d'un ensemble de nœuds par une expression XPath ou la *serialisation* de tout arbre de syntaxe abstraite en XML. Ces services sont donc disponibles sur nos arbres.

[1]Mais ce n'est pas conseillé.

```
Formalism of tiny is
Root is Top;

Operator and type definitions {
  Top = program(Decl[] declarations, Statements statements);
  Statements = statements(Statement[] statementList);
  Decl = intDecl(),
         booleanDecl();
  Statement =
      affect(Var variable, Exp value),
      while(ConditionExp cond, Statements statements),
      if(ConditionExp cond, Statements statementsThen, Statements statementsElse),
      println(StringOrVar[] stringOrVars);
  StringOrVar =
      var,
      string as java.lang.String;
  ConditionOp =
      equal(ArithmeticExp left, ArithmeticExp right),
      notEqual(ArithmeticExp left, ArithmeticExp right);
  ConditionExp =
      %ConditionOp,
      true(),
      false(),
      var;
  ArithmeticOp =
      plus(ArithmeticExp left, ArithmeticExp right),
      minus(ArithmeticExp left, ArithmeticExp right),
      mult(ArithmeticExp left, ArithmeticExp right),
      div(ArithmeticExp left, ArithmeticExp right);
  ArithmeticExp =
      %ArithmeticOp,
      int as java.lang.String,
      var as java.lang.String;
  Exp =
      %ArithmeticOp,
      var,
      int,
      true,
      false;
  Var = var;
}

Attribute definitions {
REQUIRED varName as java.lang.String in intDecl, booleanDecl;
}
```

FIG. 3.3: Définition ABSYNT du langage TINY

Avec les APIs générées des langages, il est possible de produire des arbres strictement typés (non composés de nœuds de même type : `org.w3c.dom.Node`) nécessaires pour les traitements sémantiques et ayant des méthodes de manipulation adaptés aux nœuds. Tout visiteur, n'employant que l'API de son langage, est ainsi indépendant de l'API d'implémentation choisie ; Celle-là peut être remplacée sans que le code de ces visiteurs ne soit modifié.

Comme indiqué, l'outil génère une classe et une interface par constructeur, et une interface par type ; cette dernière étant implémentée par tous les constructeurs composant ce type. En fait, définir un attribut ou une information supplémentaire sur un type correspond à définir cette donnée dans chaque constructeur dépendant de ce type. Pour éviter des conflits de noms d'interfaces entre les constructeurs et les types, les interfaces des types se terminent par `Type`, celles des constructeurs par `Node` et les classes de ces dernières par `NodeImpl`. Pour chaque nom de fils, d'attribut ou d'information supplémentaire, il existe une méthode d'accès et une méthode de modification strictement typées, se terminant, selon le cas, par `Node`, `Attr` ou `Data` ; un attribut à valeur fixe n'a pas de méthode de modification. Si tous les constructeurs (même ceux des types inclus) d'un type ont un fils de même nom, type, position et cardinalité, les signatures de ses deux méthodes sont automatiquement ajoutées à l'interface du type. Cette factorisation n'a pas encore été réalisée pour les attributs et les informations supplémentaires.

La Figure 1.4 (page 13) montre le code de l'interface du constructeur `affect` et les Figures 3.4 et 3.5 les graphes d'héritage, respectivement, d'un constructeur à arité fixe et d'un constructeur atomique. Les classes `FixedNode` et `AtomicNode` des graphes correspondent aux classes des deux catégories de constructeurs avec lesquelles il est possible de définir des visiteurs génériques (indépendants de tout langage).

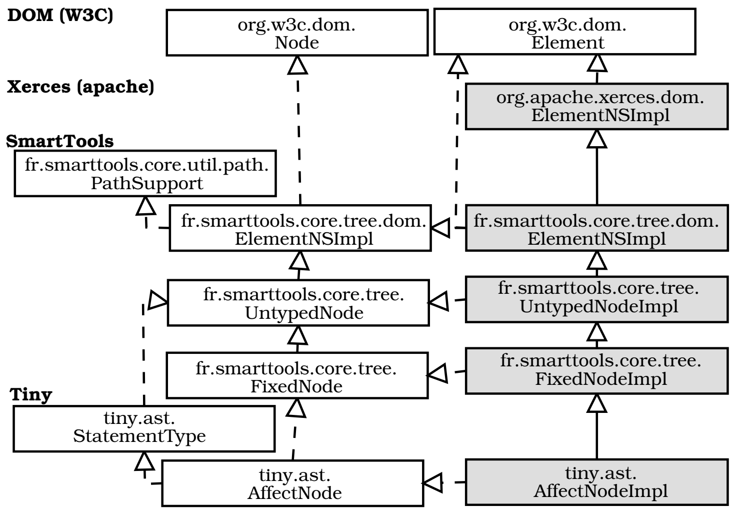


FIG. 3.4: Graphe d'héritage d'une classe d'un nœud non atomique

L'API générée relative à un langage est manipulée lors de la construction d'arbres et lors de traitements sémantiques. Un des intérêts de l'outil est qu'il n'est pas nécessaire de définir un analyseur syntaxique avec ces fonctions de construction d'arbre car l'outil accepte directement des documents XML et fait automatiquement la mise en relation nom de balise, nom de classe de nœud. Cette possibilité permet de rapidement tester les traitements sémantiques lors du développement de nouvels environnements.

3.1.3 Perspectives

Afin de pouvoir accepter de plus nombreuses définitions de documents, il faudrait étendre le langage ABSYNT des notions d'inclusion, d'importation et de redéfinition. Les notions

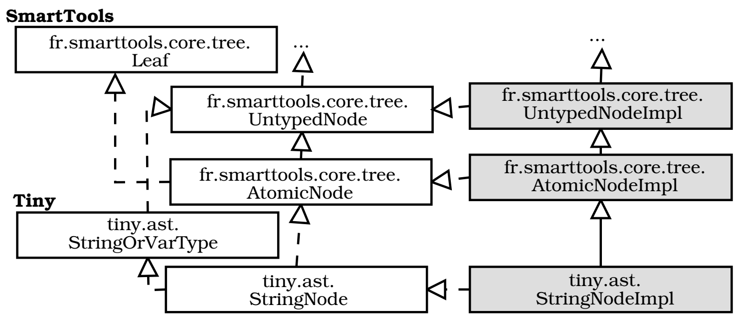


FIG. 3.5: Graphe d'héritage d'une classe d'un nœud atomique

d'inclusion et de redéfinition devraient être assez faciles à rajouter car tous les constructeurs et types résultants sont créés dans le même formalisme (même *package*) et forment un seul langage final. Alors que dans le cas d'une importation, c'est un composition de langages avec des constructeurs et des types de différents *packages*. Dans ca cas, il faudra aussi modifier la génération des visiteurs. Cette notion d'importation est liée à la notion de *namespace* des documents XML. Avec l'importation, il sera possible d'accepter un langage tel que WSDL (importe les éléments des XML Schemas)

Cette notion d'importation est certainement la plus intéressante pour définir de nouveaux langages, simplement par assemblage incrémental de composants. Cette idée a déjà été exploitée dans le système Vanilla [33] non seulement pour les langages mais aussi pour les analyseurs syntaxiques, les vérificateurs de type et les interpréteurs.

3.2 Traduction de DTD en ABSYNT

Il est important que l'outil accepte d'autres formats de description de structures de données plus connus, standardisés. De cette façon, les utilisateurs pourront choisir le format qu'ils maîtrisent le mieux. Ils ne seront pas contraints à apprendre un nouveau langage propriétaire, ABSYNT, et à ré-écrire leur description. Cette ouverture facilitera l'utilisation de SMARTTOOLS.

Comme l'outil est adapté aux langages métiers, nous avons établi des passerelles entre leurs langages de définition - DTD et XML Schema- et notre format interne. Mais évidement, d'autres passerelles pourraient être envisagées, par exemple vers des définitions de structure en Caml ou en C. En fait, bien que les syntaxes concrètes de ces formats soient différentes, les notions sous-jacentes sont très proches.

Cette section décrit quelles sont les équivalences et les transformations à effectuer pour traduire une DTD dans notre format interne. Puis, elle présente l'algorithme de traduction, ses structures de données et les résultats obtenus.

3.2.1 Rapprochement des notions

Equivalences des notions constructeur et type

Comme expliqué dans le paragraphe 2.1.1 (page 35), une DTD est majoritairement composée de définitions d'éléments, d'attributs et d'entités. Ce formalisme a été créé pour décrire la structure de documents et non de langages de programmation. Il est très proche du formalisme EBNF *Extended Backus Naur Form*, normalement utilisé pour définir les syntaxes concrètes des langages. La définition d'un élément est très similaire à une règle de production d'un analyseur syntaxique. Les expressions régulières peuvent être plus ou moins complexes, avec, par exemple, des séquences d'éléments répétées, optionnelles ou alternatives. Ces éléments complexes ne peuvent pas être directement traduits en ABSYNT mais doivent être convertis en éléments plus simples. Ces transformations font perdre des informations de structure des données rendant la définition de structure plus permissive que celle d'origine. Par conséquence, le typage d'un arbre ayant des éléments complexes sera moins strict mais son contenu sera identique (mêmes données et ordre) à celui du document d'origine et valide. Les tests de validité des données sont toujours effectués avec les DTDs d'origine, non avec les définitions de structure obtenues.

Par analogie, un élément correspond à un constructeur et un choix à un type. Comme un choix n'a pas de nom, un nom unique doit être généré pour le type correspondant. Ce nom doit avoir le plus de sens possible car il est utilisé lors des traitements sémantiques. Il en est de même pour les noms des fils des constructeurs (à l'origine des noms des méthodes d'accès et de modification de la classe du constructeur). Les notions d'attribut sont équivalentes. Seules les informations de type (`ID`, `IDREF`, les valeurs énumérées, etc) ne sont pas traitées. Mais ces dernières pourraient rendre les éditeurs plus intelligents : vérification que la chaîne mise comme `IDREF` a été définie comme `ID` ou menu proposant les valeurs de l'énumération.

Les entités paramètres sont justes des unités de factorisation. Celles dont la valeur correspond à un choix peuvent être assimilées à notre notion de type. L'avantage par rapport à un simple choix est qu'une entité est nommée et que donc son nom, ayant du sens, peut être transposé comme nom du type. Nous ne souhaitons donc pas que les références aux entités soient remplacées par leurs valeurs avant la traduction.

Le paragraphe ci-dessous décrit comment traduire les différents modèles de contenu des éléments dans notre format et les transformations à effectuer, sans se soucier des références aux entités. Les calculs de cardinalité sont indiqués ; les résultats précis sur les fils tableaux pourraient être utilisés dans la prochaine version ABSYNT. Des exemples, non formels, sont aussi proposés pour aider à la compréhension. Le paragraphe suivant indique, quant à lui, comment traiter ces références.

Modèles de contenu des constructeurs

Notations utilisées dans les exemples suivants

- $\Rightarrow$ indique la traduction d'un élément de la DTD (à gauche) vers un constructeur d'ABSYNT (à droite) ;
- $\rightarrow$ indique une transformation interne d'élément (DTD) ;
- e un élément (DTD) ou un opérateur (ABSYNT) ;
- E une entité paramètre (DTD) ou un type (ABSYNT) ;
- T un type (ABSYNT) ;
- f un nom de fils (ABSYNT) ;

c une cardinalité d'élément, de séquence, de choix pour DTD ou de fils pour ABSYNT ;

Transformations et traductions selon le modèle de contenu

EMPTY : un élément sans contenu devient un opérateur sans fils aussi.

$$e\ EMPTY \quad \Rightarrow \quad e() \tag{3.1}$$

PCDATA : un élément de modèle de contenu `PCDATA` est traduit en un constructeur atomique.

$$e\ (\#PCDATA) \quad \Rightarrow \quad e\ as\ java.lang.String \tag{3.2}$$

Juxtaposition d'éléments et de textes : un élément avec un tel modèle se traduit en un constructeur ayant un fils tableau de type regroupant tous les éléments indiqués et une feuille.

$$\begin{array}{l} \forall i \in [1,4], e_i\ un\ élément \\ e\ (\#PCDATA|e_1|e_2|e_3|e_4)^* \end{array} \quad \Rightarrow \quad \begin{array}{l} e(T[0,]\ f) \\ T = e_1, e_2, e_3, e_4, String \end{array} \tag{3.3}$$

ANY : un élément acceptant n'importe quel élément du langage comme fils se traduit par un constructeur ayant un fils de type regroupant tous les constructeurs du langage.

$$\begin{array}{r} e\ (ANY) \\ avec\ e, e_1, \ldots, e_n\ les\ éléments\ du\ langage \end{array} \Rightarrow \begin{array}{l} e(T\ f) \\ T = e, e_1, \ldots, e_n \end{array} \tag{3.4}$$

Séquence d'éléments : une séquence d'éléments requise (et placée comme modèle de contenu d'un élément) correspond exactement au contenu d'un opérateur. Chaque élément de la séquence se traduit par un type et devient un fils requis, optionnel ou tableau selon sa cardinalité. Le cas des séquences ou des choix imbriqués est décrit un peu plus loin.

$$e\ (e_1, e_2^+, e_3^*, e_4^?) \quad \Rightarrow \quad \begin{array}{l} e(T_1\ f_1, T_2[1,]\ f_2, T_3[0,]\ f_3, T_4?\ f_4) \\ avec\ \forall i \in [1,4],\ T_i = e_i \end{array} \tag{3.5}$$

Si une séquence est optionnelle, tous ses éléments deviennent optionnels ou tableaux de cardinalité $[0, \infty]$.

$$e\ (e_1, e_2^+, e_3^*, e_4^?)^? \quad \Rightarrow \quad e(T_1?\ f_1, T_2[0,]\ f_2, T_3[0,]\ f_3, T_4?\ f_4) \tag{3.6}$$

$$\begin{array}{c} selon\ les\ règles\ de\ propagation\ de\ cardinalité\ suivantes\ : \\ ? \wedge (requis \vee ?) \rightarrow ? \\ ? \wedge (+ \vee *) \rightarrow * \end{array}$$

Pour une séquence avec répétitions possibles ($c_{seq} = * \wedge +$), le constructeur correspondant aura un seul fils tableau de type regroupant tous les éléments de la séquence.

$$\left.\begin{array}{r} soit\ c_i\ la\ cardinalité\ de\ e_i \\ e\ (e_1^{c_1}, e_2^{c_2}, e_3^{c_3}, e_4^{c_4})^{c_{seq}} \\ a = 0\ si\ c_{seq} = * \\ a = \displaystyle\sum_{i=1}^{4} c_i min\ si\ c_{seq} = + \end{array}\right\} \quad \Rightarrow \quad \begin{array}{l} e(T[a,]\ f) \\ T = e_1, e_2, e_3, e_4 \end{array} \tag{3.7}$$

$$\begin{gathered} avec\ les\ cardinalités\ minimales\ suivantes\ : \\ (? \wedge *) = 0\ et\ (requis\ \wedge +) = 1 \end{gathered}$$

Avec ces deux transformations (3.6 et 3.7), la définition de structure résultante est trop permissive puisque, dans un cas, on perd l'information que toute la séquence est optionnelle et non ses fils et, dans l'autre, on perd l'ordonnancement et le nombre de chaque fils.

Choix d'éléments Tout choix d'éléments correspond à un type regroupant tous ces éléments. Pour connaître la cardinalité minimale (vs. maximale) du fils correspondant, il faut calculer le minimum (vs. maximale) des cardinalités minimales (vs. maximales) des éléments et du choix.

$$\left.\begin{aligned} soit\ c_{choix}\ la\ cardinalité\ du\ choix \\ e\ (e_1^{c1}|e_2^{c2}|e_3^{c3}|e_4^{c4})^{c_{choix}} \\ a = \min_{i=1}^{4}(c_i min) * c_{choix} min \\ b = \max_{i=1}^{4}(c_i max) * c_{choix} max \end{aligned}\right\} \Rightarrow \begin{aligned} & e(T\ f)\ si\ a = 1\ et\ b = 1 \\ & e(T?\ f)\ si\ a = 0\ et\ b = 1 \\ & e(T[a,b]\ f)\ si\ b > 1 \\ & avec\ T = e_1, e_2, e_3, e_4 \end{aligned} \tag{3.8}$$

$$\begin{gathered} avec\ les\ cardinalités\ maximales\ suivantes\ : \\ (? \wedge\ requis) = 1\ et\ (* \wedge +) = \infty \end{gathered}$$

Imbrication de séquences ou de choix

Toute séquence requise imbriquée dans une séquence est remplacée par son contenu. Cet aplatissement facilite la traduction.

$$\begin{gathered} (e_1, (e_2^+, e_3^*), e_4^?) \\ \rightarrow e\ (e_1, e_2^+, e_3^*, e_4^?) \end{gathered} \tag{3.9}$$

Si elle est optionnelle, elle est aussi remplacée par son contenu mais tous ses éléments deviennent optionnels ou tableaux de cardinalité [0,∞] (comme en 3.6).

$$\begin{gathered} e\ (e_1, (e_2^+, e_3)^?, e_4^?) \\ \rightarrow e\ (e_1, e_2^*, e_3^?, e_4^?) \end{gathered} \tag{3.10}$$

Si elle est répétitive, elle est substituée par un choix qui sera traduit par un type dans notre formalisme. Le calcul de la cardinalité minimale de ce choix est semblable à celui présenté dans la règle 3.7.

$$\begin{gathered} e\ (e_1, (e_2^+, e_3)^*, e_4^?) \\ \rightarrow e\ (e_1, (e_2|e_3)^{[a,]}, e_4^?) \end{gathered} \tag{3.11}$$

Toute séquence imbriquée dans un choix est transformée en un choix (de cardinalité maximale supérieure à 1).

$$\begin{aligned} & e\ (e_1|(e_2^?, e_3)^{c_{seq}}|e_4^?) \\ & \rightarrow e\ (e_1|(e_2|e_3)^{c_{choix}}|e_4^?) \\ c_{choix}min &= (\sum_{i=2}^{3} c_i min) * c_{seq}min \\ c_{choix}max &= (\sum_{i=2}^{3} c_i max) * c_{seq}max \end{aligned} \tag{3.12}$$

Tout choix imbriqué dans une séquence ou un choix correspond à un type. Pour faciliter la traduction, il faut «propager» les cardinalités de ses composants sur le choix. Ainsi, tout composant d'un choix aura une cardinalité [1,1].

$$\begin{aligned} & e\ (e_1, (e_2^?|e_3^+)^{c_{choix}}, e_4^?) \\ & \rightarrow e\ (e_1, (e_2|e_3)^{[a,b]}, e_4^?) \end{aligned} \tag{3.13}$$

$$les\ calculs\ de\ a\ et\ b\ sont\ équivalents\ à\ ceux\ de\ la\ règle\ 3.8\ avec\ i \in [2,3]$$

Les entités paramètres

Comme indiqué auparavant, les références aux entités paramètres ne sont pas remplacées par leurs valeurs (pour conserver les noms des choix) avant la traduction. Il faut donc les prendre en compte lors des transformations. Celles dont le contenu est un choix sont conservées et sont à traiter comme des choix dans les transformations du paragraphe précédent. Les autres sont à remplacer par leurs valeurs.

$$\begin{aligned} & E = e_2^+, e_3^* \\ & e\ (e_1, E, e_4^?) \\ & \rightarrow e\ (e_1, e_2^+, e_3^*, e_4^?) \end{aligned} \tag{3.14}$$

$$\begin{array}{rcl} E = (e_2^{c_2}|e_3^{c_3})^{c_{choix}} & & E = e_2, e_3 \\ & & e(T_1\ f_1, E\ f_2, T_4?\ f_3)\ si\ a = 1\ et\ b = 1 \\ e\ (e_1, E, e_4^?) & \Rightarrow & e(T_1\ f_1, E?\ f_2, T_4?\ f_3)\ si\ a = 0\ et\ b = 1 \\ & & e(T_1\ f_1, E[a,b]\ f_2, T_4?\ f_3)\ si\ b > 1 \end{array} \tag{3.15}$$

$$Les\ calculs\ de\ a\ et\ b\ sont\ identiques\ à\ la\ règle\ 3.8\ avec\ i\ \in [2,3].$$

3.2.2 Structures de données et algorithme de traduction

Ce paragraphe décrit les structures stockant les données extraites des DTDs et l'algorithme de traduction qui les transforme[2].

[2] Aucune des transformations n'est effectuée sur l'arbre.

Structures de données auxiliaires

Les neufs définitions suivantes correspondent à des structures de données secondaires.

$$\textit{mixte} \triangleq \textit{(nom d'élément} \mid \textit{nom d'entité paramètre)+}$$

La structure `mixte` se compose de `1` à `n` noms d'`élément` ou noms d'`entité paramètre` (types). Elle modélise une juxtaposition de textes et d'éléments.

$$\textit{enfants} \triangleq \begin{cases} \begin{cases} \textit{nom d'élément} \\ \mid \textit{choix} \\ \mid \textit{séquence} \\ \mid \textit{nom d'une entité paramétre (de modèle choix)} \end{cases} \\ \textit{cardinalité minimale} \\ \textit{cardinalité maximale} \end{cases}$$

Cette structure, `enfants`, modélise les différents contenus n'ayant que des éléments.

$$\begin{aligned} \textit{choix} &\triangleq \textit{(enfants)+} \\ \textit{sequence} &\triangleq \textit{(enfants)+} \\ \textit{notation} &\triangleq \textit{(nom)+} \\ \textit{énumération} &\triangleq \textit{(chaîne)+} \end{aligned}$$

La structure `énumération` stocke la liste des valeurs possibles d'un attribut.

$$\begin{aligned} \textit{type d'attribut} &\triangleq \textit{cdata} \mid \textit{id} \mid \textit{idref} \mid \textit{idrefs} \mid \textit{entity} \mid \textit{entities} \mid \textit{nmtoken} \mid \textit{nmtokens} \\ &\quad \mid \textit{notation} \mid \textit{énumération} \\ \textit{attribut} &\triangleq \begin{cases} \textit{nom} \\ \textit{type d'attribut} \\ \textit{requis} \mid \textit{optionnel} \mid \textit{valeur fixe ou par défaut} \end{cases} \end{aligned}$$

Structures de données principales

Les deux structures suivantes sont, respectivement, utilisées pour la table de hachage des éléments et pour celle des entités paramètres.

$$\begin{aligned} \textit{élément} &\triangleq \begin{cases} \textit{nom} \\ \textit{any} \mid \textit{empty} \mid \textit{pcdata} \mid \textit{mixte} \mid \textit{enfants} \\ \textit{(attribut)*} \end{cases} \\ \textit{entité paramètre} &\triangleq \begin{cases} \textit{nom} \\ \textit{valeur} \\ \textit{arbre} \\ \textit{(choix) ?} \end{cases} \end{aligned}$$

Pour chaque entité paramètre, on stocke son nom et sa valeur sous forme de chaîne. Cette valeur ne peut être analysée syntaxiquement qu'en fonction de son contexte d'utilisation (position dans l'arbre de sa référence) pour avoir le point d'entrée adéquat. Le résultat de cette analyse (arbre) est stocké pour être réutilisé si l'entité est référencée plusieurs fois dans le même contexte. Si c'est un choix, il est aussi stocké dans une structure adaptée aux choix qui est manipulée lors de la traduction.

Algorithme

L'algorithme se décompose en quatre phases :

1. **Analyse syntaxique de la DTD puis récupération des informations des entités paramétres, éléments et attributs, et substitution des références d'entités paramétres qui ne sont pas des choix.**
   La DTD est analysée syntaxiquement pour la représenter sous forme d'arbre de syntaxe conservant les références aux entités paramètres. Cet arbre, base de tout traitement sémantique, est parcouru pour récupérer les informations sur les éléments et les entités paramètres (tables de symboles). Les principales actions de récupération sont les suivantes :
   – Toute entité paramètre est stockée avec sa valeur.
   – Tout élément est stocké dans la table de hachage des éléments.
   – Tout attribut est ajouté à l'élément dont il dépend. Cet élément peut ne pas être encore défini (et donc mis dans la table des éléments) lors du traitement de sa ou ses listes d'attributs.
   – Toute référence à une entité paramètre est remplacée (dans les structures) par son arbre[3] sauf si c'est un choix ; la substitution 3.14 est appliquée dans le cas d'une séquence.

   A la fin de cette phase, la table des entités paramètres est «nettoyée» pour ne conserver que celles étant des choix (des futurs types dans notre formalisme).

2. **Substitution des séquences imbriquées dans les choix par des choix, propagation des cardinalités dans les choix et aplatissement des séquences imbriquées dans des séquences**
   Dans les tables des éléments et des entités, les séquences contenues dans des choix sont remplacées par des choix (transformation 3.12). Les séquences imbriquées dans une séquence sont remplacées par leur contenu (transformations 3.9, 3.10 et 3.11). Cela supprime un niveau d'imbrication inutile. Les cardinalités des composants d'un choix sont propagées au niveau de la cardinalité du choix (règle 3.13).

3. **Construction du formalisme**
   Tout élément se traduit en constructeur (de 3.1 à 3.8), tous ses composants[4] en types (application de la règle 3.15 pour les entités) et tous ses attributs en attributs de type `String`. Ainsi tout choix (contenu dans les tables des éléments ou des entités) et tout

[3]Si l'arbre n'existe pas (valeur jamais analysée) ou s'il n'est pas compatible avec le contexte d'utilisation, la valeur de l'entité doit être analysée syntaxiquement.

[4]Ici, éléments, choix ou références d'entité.

élément (fils direct) de la séquence sont convertis en des types (sauf si un type avec le même contenu contenu existe déjà). La difficulté de cette traduction est la génération de noms de type pour les choix (non contenus dans des entités paramètres) et de fils étant uniques et significatifs. Si au moins un élément a un modèle de contenu ANY, il faut aussi générer un type contenant l'ensemble des constructeurs du langage (règle 3.4). De plus, toutes les racines possibles (données par l'utilisateur) constituent aussi un type.

4. **Regroupement de fils consécutifs, optionnels ou tableau, de types non disjoints**
   Il est possible que le formalisme construit ne soit pas bien formé (voir paragraphe 3.1.1), c'est-à-dire ne respecte pas la règle qui indique que les types des fils optionnels ou tableaux placés devant un fils requis doivent être disjoints (sans constructeur commun) entre eux et aussi avec le type du fils requis. Si deux fils sont dans ce cas, il faut créer un nouveau type agrégeant les contenus de leurs types et de ceux des fils compris entre ces deux fils. Puis, ces deux fils et ceux compris entre sont remplacés par un seul fils tableau du nouveau type.

3.2.3 Implémentation et résultats

Implémentation

Nous avons, tout d'abord, spécifié la définition de structure du langage et conçu un analyseur syntaxique spécifique. En effet, aucun des analyseurs syntaxiques XML ne pouvait être utilisé puisque les DTDs ont leur propre syntaxe (non XML) et que, même s'ils les analysent en mode validant, aucun ne propose des méthodes pour y accèder. De plus, comme indiqué, nous ne souhaitons pas que les références d'entités paramètres soient remplacées avant la traduction. Cette caractéristique est importante, non seulement pour avoir des noms de types significatifs, mais aussi pour que les DTDs visualisées dans SMARTTOOLS soient identiques (au sens document, non au sens contenu) aux DTDs analysées. Sans quoi, si elles étaient éditées, elles ne contiendraient plus aucune référence aux entités paramètres rendant ces dernières inutiles[5].

Puis, nous avons réalisé un traducteur ayant un algorithme un peu plus complexe et moins efficace (trois parcours de l'arbre au lieu d'un) que celui presenté. La principale difficulté provient des entités paramètres qui peuvent être utilisées n'importe où dans les DTDs. Seuls les emplacements les plus communs ont été traités pour éviter de rendre trop complexe la définition du langage, l'analyseur syntaxique et l'implémentation de l'algorithme de traduction. De plus, il est parfois nécessaire de rajouter des caractères (une parenthèse, un supérieur, etc) à la valeur de l'entité pour être compatible avec un point d'entrée de l'analyseur syntaxique.

Résultats

Cet outil a facilité la traduction de DTDs externes de taille importante telles que MathML

[5]Le rôle d'une entité paramètre est de factoriser des définitions communes pour faciliter de futures modifications

et SVG. Certaines transformations manuelles sont parfois nécessaires, avant la traduction, pour extraire les définitions d'une DTD interne ou pour remplacer les références d'entités paramètres par leurs valeurs. Ces références sont soit des références vers des entités externes (cas du langage MathML), soit non prévues dans la définition de structure d'une DTD à cet emplacement-là, soit ayant des valeurs non compatibles avec le point d'entrée utilisé de l'analyseur syntaxique. Dans le cas d'une définition conséquante (comme MathML), il faut aussi augmenter la taille de la mémoire (tas).

3.3 Quelques indications pour traduire un XML Schema en ABSYNT

De plus en plus de nouveaux langages sont décrits avec un XML Schema, dont les notions ont été présentées dans la section 2.1.2 (page 38), et non une DTD. La principale raison de ce changement est due aux tests de validité plus exigeants imposés par les XML Schemas. Comme les utilisateurs potentiels de SMARTTOOLS sont les concepteurs de ces langages métiers, il est crucial que ce formalisme soit accepté en entrée, au même titre que les DTDs. Comme les DTDs dont ils sont les successeurs, les XML Schemas décrivent les structures de documents et ont donc aussi la possibilité de définir des éléments très complexes de structure incompatible avec notre formalisme. Ces éléments, comme pour les DTDs, devront être convertis en éléments plus simples pour être traduits en ABSYNT. De plus, comme le formalisme XML Schema est plus riche que celui de DTD ou ABSYNT, la traduction est plus complexe et la définition de structure résultante plus permissible.

3.3.1 Rapprochement des notions

Equivalences des notions constructeur et type

Tout d'abord, la notion d'*élément* (non abstrait) correspond à notre notion de constructeur, comme pour les DTDs. Tout élément non abstrait, global ou local, de nom unique est transformé en un constructeur.

Par contre, la notion de type des XML Schemas est complètement différente de la nôtre. En effet, leur notion de type complexe se rapporte au modèle de contenu d'un élément alors que la nôtre au contenu (ensemble des constructeurs possibles) d'un fils. Par exemple, l'opérateur `plus` serait spécifié, avec un XML Schema, par un élément de nom `plus` ayant un type complexe décrivant son contenu (type réutilisable pour les opérateurs binaires `moins`, `div` et `mult`).

```
<xsd:element name="plus" type="binaryOpType"/>
<xsd:complexType name="binaryOpType">
  <xsd:sequence>
    <xsd:group ref="Expression"/>
    <xsd:group ref="Expression"/>
  </xsd:sequence>
</xsd:complexType>
```

Alors que dans notre formalisme, il serait spécifié par un constructeur nommé `plus` ayant deux fils de type `Expression`.

```
plus(Expression gauche, Expression droite)
```

Notre notion de type est équivalente à deux notions en XML Schema : les *choix* et les *ensembles d'éléments substituables*. Tout choix est donc traduit en un type. Tout ensemble d'éléments substituables non abstraits[6] constitue aussi un type dans notre formalisme. Cette notion est même plus proche de notre définition de type (pas de cardinalité) mais elle ne permet pas l'appartenance multiple d'un élément ; un élément ne peut appartenir qu'à un seul ensemble de substitution contrairement à nos constructeurs qui, eux, peuvent appartenir à plusieurs types.

$$\left.\begin{array}{l} e1\ abstrait \\ e2\ substitutionGroup = "e1" \\ e3\ substitutionGroup = "e1" \end{array}\right\} \Rightarrow \quad e1 = e2, e3 \tag{3.16}$$

Le paragraphe ci-dessous explique comment traduire les différents modèles de contenu des éléments et le suivant traite des attributs.

Modèles de contenu des constructeurs

Types simples

Le modèle de contenu d'un type simple correspond à une feuille et est donc équivalent pour nous à un constructeur atomique. Il faut juste établir une table de correspondance entre les types prédéfinis de XML Schema et les types Java. Il est clair que la plupart de ces types prédéfinis seront représentés par les classes *wrappers* des types primitifs ou par la classe `String`.

$$e\ string \quad \Rightarrow \quad e\ as\ java.lang.String \tag{3.17}$$

Pour les unions de plusieurs types, il faut trouver le type parent le plus englobant (très souvent `string`). Par exemple, pour l'union des types `byte` et `int`, ce type sera `int`. Les contraintes sur ces types simples ne sont pas stockées puisque toute validation de données se fera par rapport au XML Schema d'origine et non à notre définition de structure. Dans le futur, elles pourraient être conservées pour proposer des afficheurs/éditeurs (voir section 3.4 page 69) plus intelligents. Par exemple, si l'édition d'un document se faisait via un formulaire, ce dernier pourrait comporter des champs ayant des masques de saisie conformes aux types et contraintes (de type `byte` entre 7 et 26, de type `string` de taille maximale 7) des données attendues ou des menus proposant les valeurs énumérées, au lieu d'avoir des champs de type `string`. Les afficheurs pourraient aussi représenter les données de type `anyURI` par des liens, afficher les dates dans un format textuel et les flottants avec seulement deux chiffres après la virgule, ou proposer le total ou la moyenne des valeurs des éléments de même nom.

Types complexes

[6] Si l'élément de tête de l'ensemble est abstrait, son nom peut être utilisé comme nom du type.

séquence Toute séquence fille de l'élément `complexType` et de cardinalité [1,1] correspond, sans transformation, au contenu d'un constructeur (comme la règle 3.5).

$$e\ (e_1^{[0,1]}, e_2^{[o,p]}, \ldots, e_n^{[1,1]})^{[1,1]} \quad \Rightarrow \quad \begin{array}{l} e(T_1?\ f_1, T_2[o,p]\ f_2, \ldots, T_n\ f_n) \\ avec \forall i \in [1,n],\ T_i = e_i \end{array} \tag{3.18}$$

Si la séquence est optionnelle ou à répétitions, les transformations sont identiques, respectivement, à la règle 3.6 et à la règle 3.7 de la traduction de DTD, avec les calculs de cardinalité du tableau suivants :

$$a = \sum_{i=1}^{n} c_i min * c_{seq} min \qquad b = \sum_{i=1}^{n} c_i max * c_{seq} max$$

Toute séquence, choix, groupe, `all` ou élément de cardinalité [0,0] n'est pas pris en compte lors de la traduction.

all Comme il n'existe pas de correspondant au `all` dans ABSYNT, ce modèle de contenu est transformé en un seul fils tableau de cardinalité maximale au plus égale au nombre des éléments inclus dans le `all`. Cette transformation implique une très légère perte d'information de structure puisque les éléments pourraient être répétés (ce qui est contraire à la définition d'un `all`).

$$\begin{array}{r} e(e_1^{[c_1 min,1]} - e_2^{[c_2 min,1]} - \ldots - e_n^{[c_n min,1]}) \\ a = \sum_{i=1}^{n} c_i min \end{array} \quad \Rightarrow \quad \begin{array}{l} e(T[a,\ n]\ f) \\ T = e_1, e_2, \ldots, e_n) \end{array} \tag{3.19}$$

choix Tout choix correspond à un type et les cardinalités de ses composants doivent être «propagées» au niveau du fils selon la règle 3.8.

groupe Tout groupe n'est qu'une factorisation du modèle de contenu et est à traiter comme une entité paramètre (règles 3.14 et 3.15).

type simple Tout type simple (type complexe constitué d'un type simple et d'attributs) est converti, comme expliqué auparavant, en un type Java. Les éléments utilisant ce type seront des opérateurs atomiques.

any Tout `any` paramétré par l'espace de nom courant peut être transformé en un fils de type contenant tous les constructeurs du formalisme défini (comme la règle 3.4). Un élément avec un `any` d'un autre espace de nom ou d'un espace quelconque ne peut pas encore être traduit dans SMARTTOOLS ; il manque les notions d'importation de formalisme et de nœud non typé.

élément sans type ou anyType Il est aussi possible de ne pas préciser le type du modèle de contenu ou utiliser le type `anyType` pour indiquer un contenu libre quelconque. N'importe quels éléments ainsi que du texte peuvent être utilisés. Ce genre de modèle de contenu n'a pas encore d'équivalence dans notre formalisme comme le `any` d'espace de nom quelconque.

Les imbrications de séquences et de choix sont à gérer de la même manière que pour les DTDs (règles 3.9 à 3.13).

Attributs des constructeurs

Tout attribut (non interdit) d'un élément devient attribut du constructeur correspondant. Son type simple est transformé en type Java comme expliqué auparavant.

Un groupe d'attribut constitue seulement une simplification (factorisation) d'écriture donc toute référence à un groupe d'attributs peut être remplacée par sa valeur sans que cela n'ait la moindre influence sur le formalisme cible.

Tout `anyAttribute` quel que soit son espace de nom ne sera pas pris en considération car une telle notion n'existe pas dans notre formalisme. Mais les arbres contiendront, sans discrimination, tous les attributs définis dans les documents. Seuls ceux ayant été déclarés avec `attribute` pourront être accédés ou modifiés à l'aide des méthodes générées dans la classe du constructeur correspondant ; les autres le seront seulement par les méthodes de l'API DOM.

Particularités

- **Eléments global et local de même nom et de modèles de contenu différents**
  Dans un XML Schema, des éléments locaux peuvent avoir le même nom qu'un élément global mais des modèles de contenu différents. Comme il n'y a pas cette notion de contexte/niveau dans notre formalisme, il n'est pas possible d'avoir deux constructeurs de même nom. Il faut donc unifier les modèles du contenu et les attributs pour obtenir un constructeur compatible pour ces deux éléments.

$$\left.\begin{array}{r} e(e_1^{[0,1]}, e_2^{[o,p]}, \ldots, e_n^{[1,1]}) \\ e_t(e(e_u^{[1,1]}, e_v^{[q,r]})) \\ a = \min(q, \sum_{i=2}^{n} c_i min) \\ b = \max(r, \sum_{i=2}^{n} c_i max) \end{array}\right\} \Rightarrow \begin{array}{l} e(T_1?\ f_1, T_2[a,b]\ f_2) \\ T_1 = e_1, e_u \\ T_2 = e_v, e_2, \ldots, e_n \\ e_t(T_3\ f_1) \\ T_3 = e \end{array} \tag{3.20}$$

  Très souvent, le modèle de contenu résultant se termine par un fils tableau et les attributs sont l'agrégation de leurs attributs avec transformation des requis en optionnels si non communs. En cas d'attributs de même nom et de types différents, il faut aussi modifier le type de l'attribut résultant avec le type le plus englobant.
- **Eléments avec contenu nul possible**
  Tout élément à contenu nul possible (attribut `nillable`) se traduit par un constructeur à fils optionnels (si requis ou optionnels) ou tableau de cardinalité minimale égale à 0. C'est identique à une séquence optionnelle (règle 3.6).

$$\left.\begin{array}{l} e(e_1^{[0,1]}, e_2^{[p,q]}, \ldots, e_n^{[1,1]}) \\ nillable = "true" \end{array}\right\} \Rightarrow e(T_1?\ f_1, T_2[0,q]\ f_2, \ldots, T_n?\ f_n) \tag{3.21}$$

- **Eléments et les type étendus**
  Dans un XML Schema, il est possible de déclarer un élément avec un type donné et

de le définir dans un document avec un autre type, sous-type étendu du premier. Cette extension du modèle de contenu ajoute de nouveaux fils (seulement pour les types complexes étendus) et/ou attributs à l'élément.
Pour déterminer les fils et attributs d'un constructeur, il faudra analyser, non seulement le type indiqué par l'élément, mais aussi tous ses sous-types dérivés par extension. Les fils et attributs additionnels provenant de ces sous-types devront être marqués comme optionnels. Il n'est pas nécessaire de poursuivre l'analyse dans les sous-types si le type ou tout le XML Schema est marqué comme bloqué en extension.

$$\left.\begin{array}{l} e\ type = "ct_1" \\ ct_1 = (e_1^{[0,1]}, e_2^{[o,p]}, \ldots, e_n^{[1,1]}) \\ ct_2\ extension = "ct_1" \\ ct_2 = (ct_1, e_u^{[1,1]}, e_v^{[q,r]}) \end{array}\right\} \Rightarrow \begin{array}{l} e(T_1?\ f_1, T_2[o,p]\ f_2, \ldots, \\ \quad T_n\ f_n, T_u?\ f_u, T_v[0,r]\ f_v) \end{array} \tag{3.22}$$

Il n'est pas nécessaire d'analyser les sous-types dérivés par restriction car aucun nouveau fils ou attribut n'est supprimé. Seules les cardinalités des fils peuvent être restreintes dans un sous-ensemble pour les types complexes ou des contraintes ajoutées ou restreintes pour les types simples ; ce qui n'a aucune influence pour le constructeur correspondant à l'élément.

- **Inclusion, Redéfinition et Importation de XML Schemas**
  L'inclusion ne posera pas de problème à l'importation de XML Schemas dans notre formalisme interne. Seul le découpage en module sera perdu car la composition modulaire de définition de structure n'est pas encore possible dans notre outil.
  C'est identique pour la redéfinition. Il faudra juste analyser les définitions redéfinies avant de n'inclure/analyser les définitions complémentaires.
  Les XML Schemas ayant des importations ne pourront pas être traités car la notion d'importation n'existe pas encore dans notre formalisme de définition de langages. L'ajout de cette notion qui semble, de prime abord, simple pour la génération des classes des types et les constructeurs implique aussi des modifications dans les parties sémantiques (plus particulièrement sur les parcours des arbres).

XML Schemas incompatibles avec notre formalisme

Les XML Schemas contenant les notions suivantes ne pourront pas être importés dans SmartTools :

- des `any` paramétrés avec d'autres espaces de nom que celui du formalisme cible ;
- des éléments sans type ou avec le type `anyType` ;
- ou des importations d'autres XML Schemas.

Dans un futur proche, ce dernier problème devrait être résolu car nous souhaitons ajouter la notion d'importation dans notre formalisme interne de définition de structure.

3.3.2 Structures de données et algorithme d'importation envisagés

Structures de données auxiliaires

Les sept définitions suivantes correspondent à des structures de données secondaires mais nécessaires à l'algorithme d'importation des XML Schemas.

$$\textit{informations attributs} \triangleq \begin{cases} \textit{(attribut} \mid \textit{nom d'un attribut} \mid \textit{nom d'un groupe d'attributs)*} \\ \textit{(nom d'un espace de nom)*} \end{cases}$$

Cette structure, `informations attributs`, est utilisée dans les groupes d'attributs et les types complexes.

$$\textit{modèle de contenu} \triangleq \begin{cases} \textit{all} \mid \textit{choix} \mid \textit{séquence} \mid \textit{groupe} \mid \textit{nom de groupe} \\ \textit{(cardinalité minimale) ?} \\ \textit{(cardinalité maximale) ?} \end{cases}$$

Cette structure, `modèle de contenu`, est utilisé dans les types complexes pour indiquer le modèle de contenu et ses cardinalités[7] qui sont optionnelles et par défaut à 1.

$$\begin{aligned} \textit{all} &\triangleq \textit{(nom d'élément)*} \\ \textit{choix} &\triangleq \textit{modèle de contenu imbriqué} \\ \textit{séquence} &\triangleq \textit{modèle de contenu imbriqué} \\ \textit{any} &\triangleq \textit{(nom de l'espace de nom)+} \end{aligned}$$

Ces quatre structures modélisent respectivement le contenu d'un `all`, d'un choix, d'une séquence et d'un `any`.

$$\begin{array}{c} \textit{modèle de} \\ \textit{contenu} \\ \textit{imbriqué} \end{array} \triangleq \begin{cases} \textit{(nom d'élément} \mid \textit{choix} \mid \textit{séquence} \mid \textit{groupe} \mid \textit{nom de groupe} \mid \textit{any)*} \\ \textit{(cardinalité minimale) ?} \\ \textit{(cardinalité maximale) ?} \end{cases}$$

Cette structure représente le contenu possible d'un composant d'un choix, d'une séquence ou d'un groupe.

Structures de données principales

Les six structures suivantes sont utilisées dans des tables de hachage pour stocker les informations nécessaires à la construction du formalisme. L'algorithme d'importation envisagé requiert cinq tables de hachage : une pour les éléments, une pour les attributs, une pour les groupes d'attributs, une pour les types simples ou complexes et enfin une pour les groupes de modèle de contenu.

[7] Une déclaration de `groupe` ne peut pas avoir d'informations de cardinalité

$$
\textit{élément} \triangleq \left\{ \begin{array}{l}
\textit{nom} \\
\textit{nom du type} \\
\textit{abstrait ou non} \\
\textit{nul possible ou non} \\
\textit{(nom d'un élément substituable)*}
\end{array} \right.
$$

$$
\textit{attribut} \triangleq \left\{ \begin{array}{l}
\textit{nom} \\
\textit{nom du type} \\
\textit{requis} \mid \textit{optionnel} \mid \textit{interdit} \\
\textit{valeur fixe ou par défaut}
\end{array} \right.
$$

$$
\textit{groupe d'attributs} \triangleq \left\{ \begin{array}{l}
\textit{nom} \\
\textit{informations attributs}
\end{array} \right.
$$

$$
\textit{type simple} \triangleq \left\{ \begin{array}{l}
\textit{nom} \\
\left\{ \begin{array}{l}
\textit{nom du type de base restreint} \\
\mid \textit{nom du type des items de la liste} \\
\mid \textit{(nom d'un type de l'union)+}
\end{array} \right.
\end{array} \right.
$$

$$
\textit{type complexe} \triangleq \left\{ \begin{array}{l}
\textit{nom} \\
\textit{global ou local} \\
\textit{(nom d'un type le dérivant par extension)*} \\
\textit{mélange texte possible ou non} \\
\left\{ \begin{array}{l}
\left\{ \begin{array}{l}
\left\{ \begin{array}{l}
\left\{ \begin{array}{l}
\textit{nom du type complexe de base étendu} \\
\textit{(modèle de contenu) ?}
\end{array} \right. \\
\mid \textit{nom du type simple de base dérivé} \\
\mid \textit{(modèle de contenu) ?}
\end{array} \right. \\
\textit{informations attributes}
\end{array} \right. \\
\mid \textit{nom du type complexe de base restreint}
\end{array} \right.
\end{array} \right.
$$

$$
\textit{groupe} \triangleq \left\{ \begin{array}{l}
\textit{nom} \\
\textit{all} \mid \textit{choix} \mid \textit{séquence}
\end{array} \right.
$$

Algorithme

L'algorithme envisagé se décompose en cinq phases :

1. **Analyse syntaxique du XML Schema puis récupération des informations des éléments, types, attributs, groupes d'attributs et de modèle de contenu.**
   Le XML Schema donné doit être analysé syntaxiquement puis son arbre résultant parcouru pour en extraire les informations nécessaires à la construction du formalisme sous-jacent. S'il inclut ou redéfinit d'autres XML Schemas, ces derniers doivent aussi être récursivement analysés et parcourus. Contrairement aux DTDs, il n'est pas nécessaire d'implémenter un analyseur syntaxique spécifique car les XML Schemas ont une syntaxe XML acceptée par l'analyseur de l'API DOM. Seule la définition de structure du formalisme XML Schema doit être spécifiée pour pouvoir construire et manipuler des arbres correctement typés.

- Toute redéfinition d'un composant[8] masque la définition qui lui sert de base. Seule cette redéfinition doit être stockée et sa référence à la définition de base doit être remplacée par la valeur de cette dernière.
- Tout choix, séquence, `all`, groupe ou élément de cardinalité maximale égale à 0 est à supprimer (ainsi que les références à un tel groupe ou à un groupe ayant un tel choix, séquence ou `all`).
- Tout élément global ou local est placé dans la table des éléments. Si un élément local a le même nom qu'un autre élément, il faut leur attribuer un nouveau type (au sens XML Schema). Ce type fabriqué doit proposer un modèle de contenu compatible avec ceux de ces éléments. Tout élément substituable doit être référencé dans la valeur (structure) de l'élément de tête (le «substitué») pour faciliter la création du type associé en phase 4.
- Tout type global ou local, nommé ou anonyme, est mis dans la table des types pour un traitement homogène. Il faut attribuer un nom unique (non utilisé comme nom de type) à chaque type dit anonyme. Tout type complexe dérivé par extension doit avoir son nom mis dans la structure du type de base pour faciliter l'analyse de la phase 4.
- idem pour les attributs et groupes globaux.

2. **Substitution des références aux attributs, groupes d'attributs ou de modèle de contenu séquence ou `all` par leurs valeurs**
   Dans la table des types, toute référence à un attribut ou à groupe d'attributs est remplacé par sa valeur ainsi que toute référence à un groupe de modèle de contenu séquence ou `all` ; si cette dernière avait des informations de cardinalité, il faut les reporter au modèle de contenu[9]. Seuls les groupes de modèle de contenu choix sont conservés car leurs noms pourront être utilisés comme noms de type dans notre formalisme (choix=type). Après cette phase, les tables des attributs et des groupes d'attributs ne servent plus et la table des groupes de modèle de contenu ne devrait contenir que des choix (au premier niveau).

3. **Substitution des séquences imbriquées dans les choix et les `all` par des choix et aplatissement des séquences imbriquées dans des séquences**
   Dans les tables des types et des groupes, les séquences (d'au moins deux fils) contenues dans des choix (d'au moins deux composants) et les `all` sont remplacés par des choix de cardinalité tableau en notre formalisme. De plus, toute séquence imbriquée dans une séquence est aplatie en la remplaçant directement par son contenu et en décalant les fils suivants. Ce niveau d'imbrication inutile est ainsi supprimé et cela facilitera l'importation.

4. **Construction du formalisme**
   Tout élément non abstrait est transformé en constructeur. Tout choix, tout ensemble d'éléments substituables, tout élément fils direct d'une séquence et l'ensemble des

[8] Seuls les types et les groupes peuvent être redéfinis.

[9] Ainsi, un `all` pourrait avoir une cardinalité supérieure à 1, lors de nos traitements. Mais un groupe de modèle de contenu `all` et de cardinalité supérieure à 1 semble impossible.

éléments globaux[10] constitue un type. Pour déterminer le modèle complet de contenu d'un élément à type complexe, il faut aussi analyser tous les types dérivés par extension. Ces analyses permettront d'ajouter (de manière optionnelle) les attributs et les fils possibles de l'élément.

5. **Regroupement de fils consécutifs, optionnels ou tableau, de types non disjoints** Idem que la phase 4 de l'algorithme d'importation d'une DTD (page 60.

Faire une comparaison avec JAXB (the Java Architecture for XML Binding) public draft v0.7 12 septembre 2002

3.4 Génération de parseurs et d'afficheurs

Afin de présenter les arbres de syntaxe abstraite selon des formats différents, il faut leur faire subir quelques transformations. Pour les langages de programmation, la transformation la plus connue est très certainement celle inverse de l'analyse syntaxique (l'*unparsing*) et pour les documents XML, la représentation des données dans une forme HTML plus lisible. La création de ces afficheurs (*pretty-printers*) peut être assez complexe et coûteuse en temps.

Pour aider au développement de tels afficheurs, nous avons conçu un langage de haut niveau, CoSynt, représentant les différentes transformations à effectuer sur l'arbre de syntaxe abstraite. Les principales contraintes de ce langage sont qu'il soit simple, indépendant de toute technique d'implémentation et séparant clairement les différentes préoccupations. Le but est de générer, très rapidement, à partir d'une spécification CoSynt, un ou plusieurs afficheurs.

Ces afficheurs, afin d'être ouverts, évolutifs et faciles à implémenter, maximiseront l'usage de standards. Pour ceux produisant des vues graphiques, il doit être facile d'ajouter de nouvelles boîtes (objets graphiques), de gérer leur agencement, et d'appliquer de nouvelles valeurs aux styles. La personnalisation de ces vues doit pouvoir être effectuer par l'utilisateur final (et non, à la génération de la vue comme, en version 3, avec le langage Xpp). De plus, pour ces vues, il est impératif de conserver une correspondance (isomorphisme) entre les nœuds de l'arbre de syntaxe abstraite et ceux de l'arbre de boîtes afin de répercuter toute modification de sélection ou d'édition.

En rajoutant quelques informations supplémentaires, il est possible, dans certains cas, de générer l'analyseur syntaxique associé. En effet, la phase de transformation arbre de syntaxe abstraite en arbre de syntaxe concrète est commune. Il suffit de rajouter les informations (expressions régulières) utiles au lexeur. Cette possibilité de génération d'analyseurs syntaxiques est très intéressante pour donner une forme concrète simple, non ambiguë, et lisible aux langages métiers, usuellement représentés en XML.

Le paragraphe suivant présente les informations et les transformations du langage CoSynt. Puis, les deux suivants expliquent, respectivement, comment les afficheurs et les analyseurs syntaxiques sont implémentés.

[10] C'est le type des racines possibles.

3.4.1 CoSynt ou comment décrire une forme concrète

Une spécification CoSynt est constituée de deux parties : la première relative à la *syntaxe concrète* et l'autre à l'*affichage* (voir Figure 3.6 pour une spécification CoSynt du langage Tiny).

Partie syntaxe concrète

La partie syntaxe concrète est elle-même décomposée en trois sous-parties :

- une «BNF» qui décrit quelles sont les transformations à effectuer sur l'arbre de syntaxe abstraite afin de produire un arbre de syntaxe concrète ;
- le *lookahead* de l'analyseur syntaxique ;
- les informations relatives au lexeur telles que son *lookahead* et les expressions régulières des lexèmes.

La sous-partie BNF est indépendante de toute information d'affichage. Elle indique, en fonction des nœuds de l'arbre, quels sont leurs fils, leurs attributs et les sucres syntaxiques à ajouter à l'arbre de syntaxe concrète et dans quel ordre. La partie gauche de ces règles correspond à la partie de filtrage à un seul niveau sur les noms des constructeurs avec indication des attributs utilisés et nommage des fils pour faciliter leurs manipulations. Un filtrage à plusieurs niveaux aurait rendu plus complexe le langage car cela aurait demandé l'introduction de la notion de priorité entre règles. La partie droite des règles manipule donc les fils, les attributs, les sucres syntaxiques et les lexèmes (par exemple, pour les feuilles). Les traitements des fils optionnels ou tableaux diffèrent de celui des fils requis. En effet, pour un fils optionnel, il est possible de lui ajouter, avant et après, des sucres syntaxiques eux-aussi optionnels, dépendants de la présence de ce fils. Pour les fils tableaux, il y a aussi des sucres syntaxiques optionnels, à placer avant ou après la liste des éléments s'il y a au moins un élément dans la liste. Des sucres syntaxiques peuvent aussi être placés avant, entre et après chaque élément.

Partie affichage

Le partie affichage se compose elle-aussi de trois sous-parties :

- les noms des styles à ajouter aux nœuds de l'arbre de syntaxe concrète (non leurs valeurs paramétrées par l'utilisateur final et gérées dans un fichier à part). Par défaut, chaque nœud représentant un fils aura comme nom de style le nom de son constructeur.
- les règles de boîtes additionnelles, regroupant des nœuds de l'arbre de syntaxe concrète, afin de faciliter la personnalisation du résultat.
- le ou les formats de sortie. Par exemple, si l'afficheur génère des *beans* pour une vue graphique, il faut préciser quels objets graphiques employer ; si c'est du texte, seulement les endroits où il faut retourner à la ligne (pour obtenir, une sortie textuelle plus lisible, il faudrait aussi préciser où indenter).=

La Figure 3.7 permet de visualiser la transformation d'un arbre de syntaxe abstraite du langage Tiny en arbre de syntaxe concrète puis en arbre de boîtes ou en format texte selon les règles de transformation édictées par la spécification CoSynt de la Figure 3.6.

3.4.2 Génération d'afficheurs

Afin de profiter de leurs évolutions et de leurs implémentations, nous avons choisi :

- XSLT comme langage de transformation et son moteur ;
- BML comme langage de description des objets graphiques d'une vue et son interpréteur ;
- *Swing* comme API des objets graphiques et ses algorithmes de placement ;
- CSS comme langage de description des styles à appliquer aux formats de sortie.

3.4.3 Génération de parseurs

pas notion priorité des opérateurs.

```
Cosynt for tiny is
Concrete Syntaxe {
   BNF {
      program(Decl, Stats)  : *[ #Decl ] #Stats ;
      statements(List)      : %LCURLY *[ #List ] %RCURLY ;
      while(Cond, Stats)    : "while" %LPARENT #Cond %RPARENT #Stats ;
      booleanDecl() @varName: "boolean" @varName ;
      intDecl() @varName    : "int" @varName ;
      var()                  : getValue(%VAR) ;
      string()               : getValue(%STRING) ;
      int()                  : getValue(%INT) ;
      equals(Left, Right)   : #Left "==" #Right;
      notEqual(Left, Right) : #Left "!=" #Right;
      plus(Left, Right)     : %LPARENT #Left "+" #Right %RPARENT;
      minus(Left, Right)    : %LPARENT #Left "-" #Right %RPARENT;
      mult(Left, Right)     : %LPARENT #Left "*" #Right %RPARENT;
      div(Left, Right)      : %LPARENT #Left "/" #Right %RPARENT;
      affect(Var, Exp)      : #Var "=" #Exp ";" ;
      true()                 : "true";
      false()                : "false";
      if(Cond, Then, Else)  : "if" %LPARENT #Cond %RPARENT #Then "else" #Else;
      println(Vals)         : "println" %LPARENT *[ #Vals separator "+" ] %RPARENT ";";
   }
   Parser[k = 2] {}
   Lexer[k = 1, attributes = VAR] {
      VAR     = <('a'..'z'|'A'..'Z'|'_'|'$')('a'..'z'|'A'..'Z'|'_'|'0'..'9'|'$')*> ;
      INT     = <('0')|(('1'..'9')('0'..'9')*)> ;
      STRING  = <'&quot;'('a'-'z'|'A'-'Z')*'&quot;'> ;
      RPARENT = <')'>;
      LPARENT = <'('>;
      RCURLY  = <'}'>;
      LCURLY  = <'{'>;
}}

Layout {
   Styles[default = null, sugars = keyword, attributes = null] {
      equal  : "==";
      int    : #intDecl@varName;
   }
   Transformation Rules {
      program              : child:#1 #2;
      statements           : #1 childbox:#2 #3 ;
      intDecl, booleanDecl : line:(#1 #2 #3) ;
      while                : line:(#1 #2 line:#3 #4) childbox:#5;
      if                   : line:(#1 #2 childbox:#3 #4) childbox:#5 #6 childbox:#7;
      println              : line:(#1 #2 line:#3 #4 #5);
      equals, notEqual     : line:(child:#1 #2 child:#3);
      plus, minus, mult, div : line:(#1 child:#2 #3 child:#4 #5);
      affect               : line:(child:#1 #2 child:#3 #4) ;
      true, false, var, string, int: #1;
   }
   Output {
     BML[default=fr.smarttools.core.view.Sbox,
         sugars=fr.smarttools.core.view.Slabel,
         attributes=fr.smarttools.core.view.Alabel] {
       fr.smarttools.core.view.Nlabel : #string, #var, #int ;
     }
     Text[default=sameline, sugars = null, attributes = null] {
       newline : #program[2], #statements[1], #intDecl, #booleanDecl,
                 #while[4,6], #if[4,6,7], #println, #affect;
}}}
```

FIG. 3.6: Exemple de spécification COSYNT du langage TINY

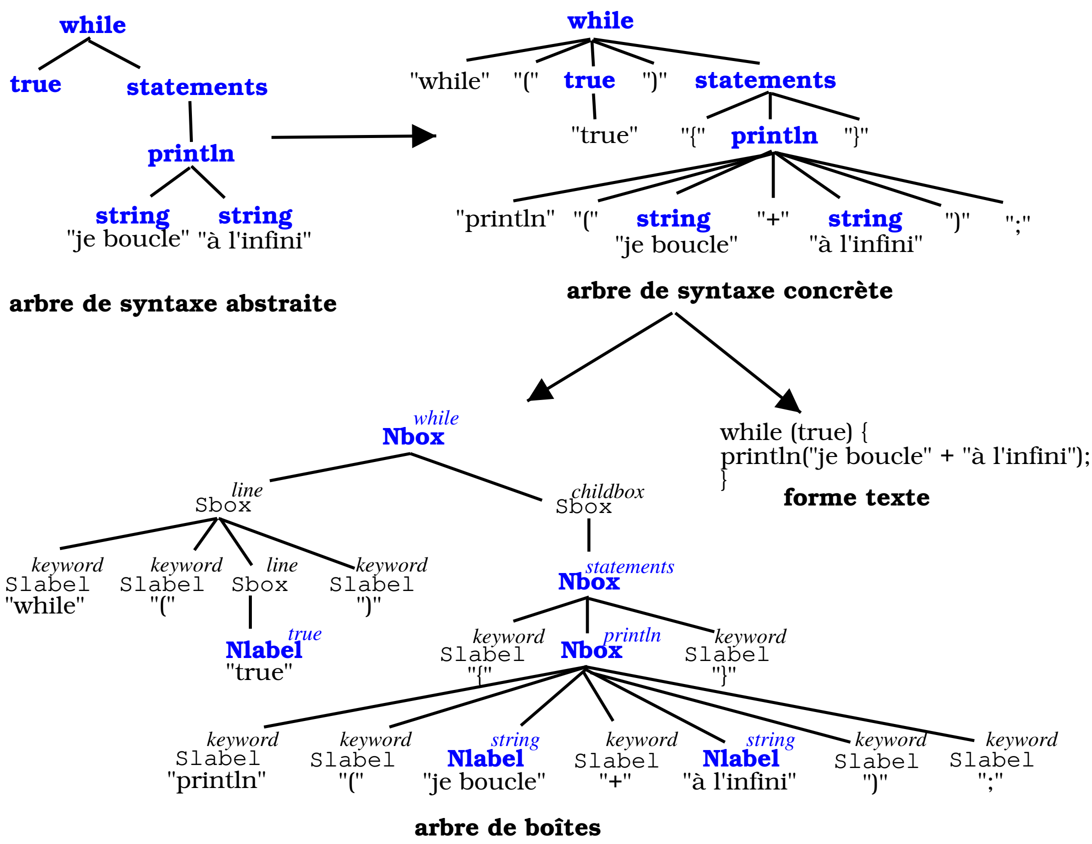


FIG. 3.7: Représentation des transformations de l'AST à l'arbre de boîtes

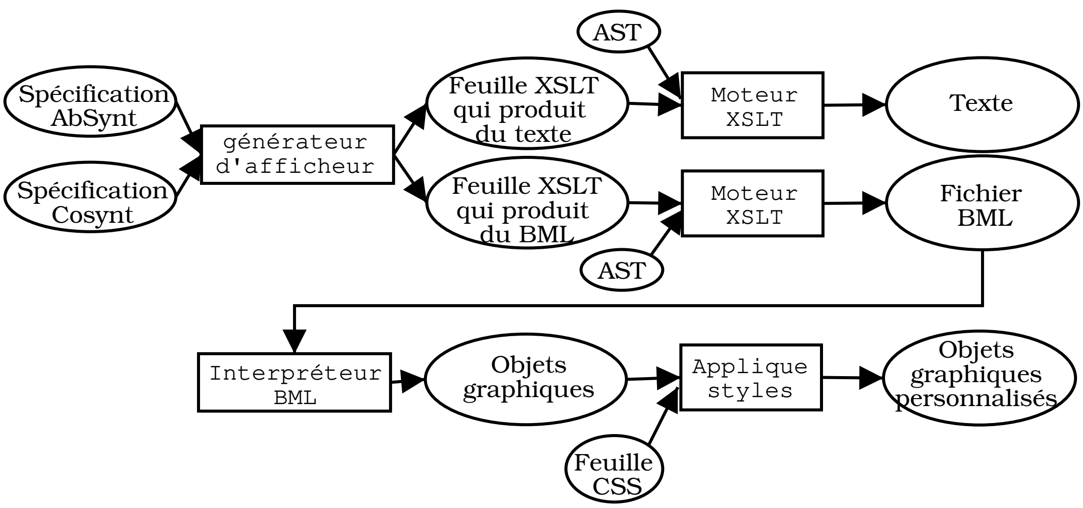


FIG. 3.8: Chaîne de génération de vues

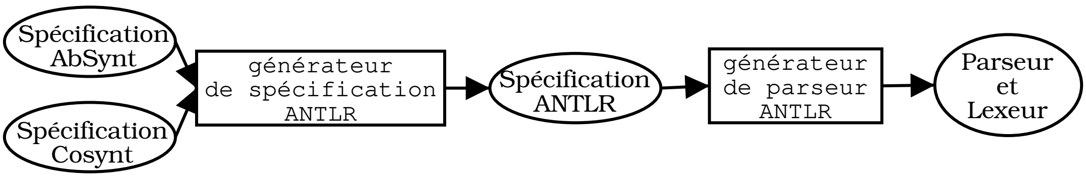


FIG. 3.9: Chaîne de génération d'analyseur syntaxique

Chapitre 4

Outils sémantiques : visiteurs et aspects

Le but de ce chapitre n'est pas d'étudier les différentes méthodes de description de sémantique ou les diverses analyses (pour cela, nous vous conseillons la lecture de la thèse de Valérie Gouranton [38]) mais de proposer des techniques d'analyse d'arbre adaptées à la programmation par objets en Java.

Rappels sur les analyses sémantiques

Avant de présenter le problème majeur des implémentations d'analyse d'arbre en programmation par objets et nos approches, nous donnons brièvement quelques rappels sur les analyses sémantiques dans d'autres langages de programmation, utiles pour mieux comprendre nos outils.

Il est souvent nécessaire d'extraire des données d'un programme (document) pour l'analyser, le transformer ou le réécrire. Tout d'abord, il faut organiser les données sous forme d'arbre de syntaxe abstraite en supprimant les sucres syntaxiques. Puis, à partir de cette structure d'arbre, il est possible de spécifier des analyses sur le sens des données (analyses sémantiques) ou de faire des preuves de programmes (nous ne nous intéressons pas à cet aspect).

Il existe diverses analyses sémantiques en fonction de :

– la classe de programmation du langage : impérative (Pascal, C), fonctionnelle (Lisp, Scheme, Caml), logique (Prolog) ou à objet (Smalltalk, Java, C++),
– la stratégie de calcul utilisée : séquentielle ou parallèle,
– le système de type du langage,
– le mode d'exécution (interprété, compilé ou semi-compilé),
– la modularité du langage.

On peut, toutefois, distinguer deux types d'analyses de programmes : les *analyses statiques* déterminant des propriétés à la compilation et les *analyses dynamiques* détectant des propriétés de comportement pour une exécution donnée. Les analyses statiques (vérificateur de type, analyse de flots de données dont les analyses de durée de vie et d'élagage ou *slicing*, etc) sont utilisées pour vérifier le programme source et améliorer l'efficacité du code produit (optimisation mémoire, etc). Les analyses dynamiques (évaluateur ou simulateur) servent à la mise au point des programmes. La plupart des analyses utilisent une *table des symboles* (pour stocker les noms, les types et parfois les valeurs des variables selon la visibilité ou *scope*, les signatures des méthodes, etc) ou laissent des informations dans les nœuds de l'arbre sous forme d'*attributs* (ou annotations). Les différences d'écriture des analyses proviennent aussi de la méthode de spécification sémantique utilisée :
– la *sémantique opérationnelle* qui décrit le comportement du programme à l'aide de règles de réécriture définissant une machine abstraite à états (ou système de transitions). L'exécution d'une instruction fait changer l'état de cette machine. Cette sémantique est la base de la sémantique opérationnelle structurée (SOS) ou *small-step* de Gordon Plotkin, et de la sémantique naturelle ou *big-step* de Gilles Kahn [45] ;
– la *sémantique axiomatique* qui utilise un système logique à base d'assertions (contenant des contraintes sur les variables) avant et après les instructions ;
– la *sémantique dénotationnelle* qui décrit le modèle abstrait associé à un langage. Elle s'écrit sous forme d'une traduction du langage vers ce modèle.

Présentation du problème

Le problème majeur de l'implémentation de ces analyses en programmation à objets est l'absence de la technique de filtrage ou *pattern matching*. Cette technique est très utilisée en programmation fonctionnelle ; par exemple pour filtrer le code à exécuter dans le corps d'une fonction selon la valeur de ses paramètres. Le code suivant montre comment un vérificateur de types pour notre langage TINY pourrait être écrit en fonctionnel (pseudo-ML) en utilisant cette technique de filtrage. La partie gauche contient le filtre sur le nœud, et la partie droite le code à exécuter (la sémantique) et les appels récursifs de la fonction sur les fils (parcours de l'arbre).

```
let rec typecheck tree = match tree with
  true              -> "boolean"
| false             -> "boolean"
| int(t)            -> "int"
| intDecl(t)        -> setType(t, "int");
                       ""
| booleanDecl(t)    -> setType(t, "boolean");
                       ""
|  plus (l, r)      -> (let ((type1 typecheck(l))
```

```
                            (type2 typecheck(r)))
                           if type1=type2 then
                              type1
                           else
                              "error"
                      )
| assign(v, e)      -> if getType(v) != typecheck(e) then
                          "error"
                       else
                          ""
| statments(sons) -> (map typecheck sons);
                          ""
| while(c, b)     -> typecheck(c);
                       typecheck(b);
                       ""
| ...;
```

L'intérêt de cette technique de filtrage est d'obtenir un code lisible et modulaire (rassemblé dans une seule fonction), ce qui facilite sa maintenance. Mais, en langage fonctionnel, le code n'est pas extensible.

En programmation à objets, la même analyse pourrait être écrite en distribuant le code des parties droites dans les classes des nœuds respectifs. L'analyse serait ainsi éclatée en méthodes dans différentes classes comme le montre le code ci-dessous. C'est le type dynamique de l'objet courant qui est utilisé pour déterminer quelle méthode de l'analyse doit être appelée.

```
public class TrueNode {
  public String typecheck() {
    return "boolean";
  }
  ...
}
```

```
public class PlusNode {
  public String typecheck() {
    String type1;
    return ((type1=getLeftNode().typecheck()) == getRightNode().typecheck() ?
                   type1 : "error");
  }
  ...
}
```

```
public class WhileNode {
  public String typecheck() {
    getCondNode().typecheck();
    getStatementsNode().typecheck();
    return "";
  }
  ...
}
```

Pour spécifier une nouvelle analyse, il faut ajouter une nouvelle méthode dans toutes les classes des nœuds et recompiler l'ensemble. Cette solution n'est pas satisfaisante car le code des analyses est dispersé donc difficile à lire et à maintenir. Pour suppléer le filtrage dans la programmation à objets, d'autres techniques d'analyse d'arbre doivent être élaborées, conservant la bonne caractéristique de modularité du fonctionnel et en rajoutant l'extensibilité qui lui faisait défaut.

La technique de base séparant le code sémantique et la structure afin obtenir des analyses modulaires est le patron de conception visiteur (*design visitor pattern*) [37], point de départ de nos travaux. Avant de présenter ces derniers dans les sections 4.2, 4.3 et 4.4, nous rappelons, pour le langage Java, les principes de cette technique de conception et expliquons nos approches tentant de pallier aux inconvénients de ce patron et du langage.

4.1 Contexte et présentation de notre approche

4.1.1 Principes du patron de conception visiteur pour le langage Java

Le patron de conception visiteur permet de définir de nouvelles opérations sur une structure d'objets en séparant le code (le visiteur) et la structure. Dans notre cas, ces opérations correspondent à des analyses sémantiques et la structure à un arbre de nœuds typés. L'avantage majeur de cette technique est de rendre possible la création de nouvelles opérations sans modification ou recompilation des classes des objets de la structure. De plus, elle propose une solution sans réflexivité, pour déterminer quelle méthode invoquer en fonction du type dynamique de l'objet courant.

Pour définir un visiteur, il suffit de créer une classe contenant autant de méthodes `visit` que de classes d'objets de la structure ; chaque méthode contenant le code de l'analyse et les appels récursifs sur les fils (structures imbriquées) qui sont, en fait, les instructions de parcours de la structure.

Comme chaque fils peut avoir des types différents et que l'interprète Java base sa recherche de méthodes sur les types statiques et non dynamiques des arguments, ces appels récursifs ne peuvent pas être directs. En effet, la méthode `visit` appelée serait sinon celle du type statique (type connu à la compilation) du fils. Une indirection dans la classe de l'objet doit donc être effectuée pour que la méthode `visit` correspondant au type dynamique soit invoquée. Cette indirection est réalisée par l'appel d'une méthode particulière, usuellement nommée `accept`, de l'objet à visiter qui appelle, à son tour, la méthode `visit` adéquate.

Exemple

Pour mieux comprendre, prenons, comme exemple, une structure (`StatementsNode`) d'objets de type statique commun `StatementType` (interface) mais de types dynamiques `AffectNode` ou `WhileNode`. Le code naïf que l'on aimerait écrire pour visiter une telle structure est le suivant :

```
public class Visitor1 {
   public void visit(StatementsNode node) {
      StatementType[] statL = node.getList();
      for (int i=0; i<statL.length; i++)
         visit(statL[i]);
   }
   public void visit(AffectNode node) {...}
   public void visit(WhileNode node) {...}
}
```

Le problème réside à la ligne 5 car on veut appeler la méthode `visit` de `AffectNode` ou de `WhileNode` selon le type dynamique de l'objet `statL[i]` et non celle de `StatementType` (son type statique). Une des solutions en Java pour appeler récursivement la méthode adéquate est l'utilisation d'une méthode d'indirection dans la classe de l'objet. Le code correct à écrire est le suivant :

```
public class Visitor1 implements Visitor {
   public void visit(StatementsNode node) {
      StatementType[] statL = node.getList();
      for (int i=0; i<statL.length; i++)
         statL[i].accept(this);
   }
   public void visit(AffectNode node) {...}
   public void visit(WhileNode node) {...}
}
```

Un tel code suppose la définition d'une méthode `accept` dans chaque classe et chaque interface des objets à visiter et la création d'une interface `Visitor` contenant toutes les signatures des méthodes `visit`. Dans notre exemple, cela correspond aux classes et interfaces suivantes :

```
public interface StatementType {
   public void accept(Visitor vis);
}
```

```
public class AffectNode implements StatementType {
   public void accept(Visitor vis) {
     vis.visit();
   }
}
```

```
public class WhileNode implements StatementType {
   public void accept(Visitor vis) {
     vis.visit();
   }
}
```

```
public class StatementsNode {
   public void accept(Visitor vis) {
     vis.visit();
   }
}
```

```
public interface Visitor {
   public void visit(StatementsNode node);
   public void visit(AffectNode node);
   public void visit(WhileNode node);
}
```

Les Figures 4.1 et 4.2 montrent les différentes indirections effectuées dans les classes des objets de la structure ; La Figure 4.2 suppose que le premier élément de la liste est une affectation ayant une variable en partie gauche et un entier en partie droite.

Il existe une importante littérature sur ce patron de conception, présentant différentes formes d'implémentation dans divers langages à objets. Il est important de préciser que

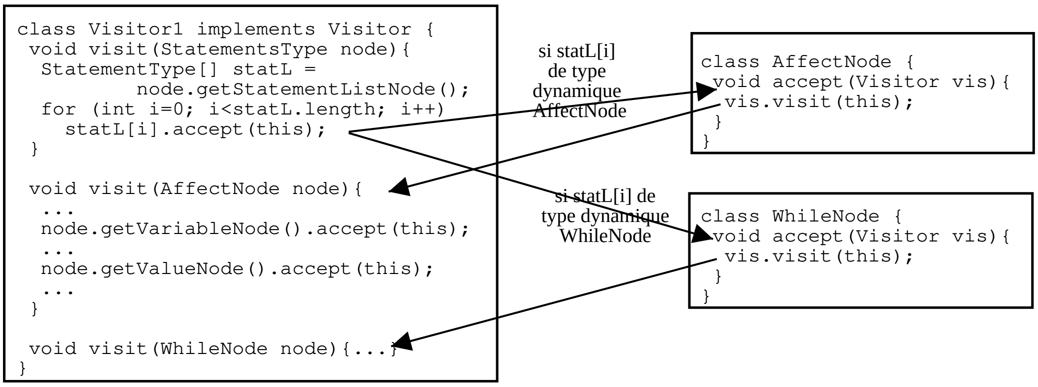


FIG. 4.1: Mécanisme d'indirection du patron de conception visiteur

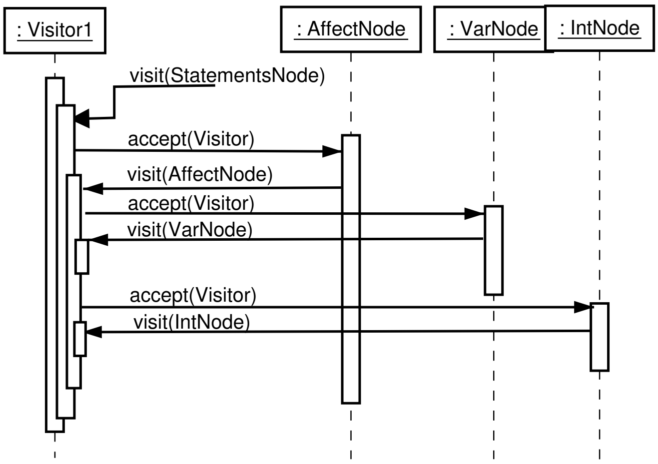


FIG. 4.2: Diagramme de séquence de la visite d'un nœud d'affectation

nos travaux, basés sur ce patron, se sont restreints à l'utilisation du langage Java[1]. Pour positionner nos travaux, nous ne citerons, de cette importante littérature, que des travaux relatifs à Java. L'article de Jens Palsberg et C. Barry Jay, the Essence of the Visitor Pattern [61], fut le point de départ de nos travaux. Il présente les évolutions d'implémentation de ce patron de conception.

[1]Nous avons choisi le langage Java car c'est un langage communément adopté aussi bien par les industriels que par les chercheurs. C'est aussi le langage avec lequel SMARTTOOLS a été entièrement développé. Il est clair que certains nos problèmes d'implémentation sont liés aux contraintes de ce langage ; par exemple, contrairement à Smalltalk, l'interpréteur Java effectue la résolution de méthodes en fonction des types statiques des arguments et non des types dynamiques. De plus, il ne possède pas encore la possibilité de définir des types paramétrés [24] qui pourraient changer l'implémentation des analyses.

4.1.2 Nos approches

Notre objectif principal est d'offrir des outils[2] d'analyses de programmes ou de documents simples et compréhensibles même par des personnes ayant peu de connaissances en techniques de programmation (par exemple, les concepteurs de langages métiers). Toute difficulté technique doit être cachée afin que le code à écrire soit le plus naturel et lisible possible. Il est aussi nécessaire que ce code soit extensible et réutilisable afin de faciliter toute extension de langage ou ajout de traitements particuliers lors des analyses. C'est l'une des propriétés intéressantes de ce patron de conception. Cependant, il est communément admis que ce patron de conception n'est pas intuitif [56].

Nous proposons deux approches complémentaires :

– la première met en place des techniques simplifiant le patron de conception visiteur afin de rendre le code des analyses naturel, aussi proche des méthodes classiques (avec des appels directs sur les méthodes) ;
– la deuxième donne de l'importance à la réutilisation afin de pouvoir, par composition, créer de nouvelles analyses. Mais, en contrepartie, le code est plus complexe à écrire.

La première approche : visiteur configurable

Pour cette première approche, afin d'obtenir un code plus naturel, nous avons essayé de cacher les inconvénients, usuellement connus, liés à l'indirection. En effet, les appels aux méthodes `accept` au lieu de `visit` sont, de prime abord, assez incongrus. De plus, l'utilisation de ces méthodes fige les signatures des méthodes du visiteur, rendant impossible la configuration des types de retour, du nombre du paramètres et de leurs types. Par exemple, JTB (*Java Tree Builder*) [12] propose des méthodes ayant un paramètre et une valeur de retour de type `Object`. Mais cela n'est pas suffisant car, pour passer plusieurs données, il faut soit créer une structure adéquate (un environnement) soit utiliser le paramètre comme un tableau d'objets et effectuer de nombreuses coercitions de type. Or, il nous semble important de manipuler des données correctement typées et d'avoir des signatures de méthodes explicites. Le code est plus lisible et cela évite les nombreuses coercitions de type et l'emploi abusif de variables d'instance. C'est pour ces raisons que les signatures des méthodes de nos visiteurs sont configurables. C'est l'une des différences de notre solution par rapport à JJTree [11], JTB et SableCC [36].

Pour réaliser un tel outil, nous avions, au début, utilisé une implémentation des multi-méthodes pour Java [35], développée à l'université de Marne-la-Vallée. Ce mécanisme permet de contrebalancer les inconvénients du langage Java. Il détermine quelle méthode invoquer en fonction du type de l'objet invoqué et des types dynamiques des paramètres. Dans notre contexte, le domaine des constructeurs et des types est fini (connu à l'avance). Nous nous sommes aperçus qu'une autre solution[3] était possible, simple, sans réflexivité et dédiée à nos besoins (donc non généraliste). Plus précisément, comme le domaine est fini, lors de la génération du visiteur de parcours par défaut, on engendre un pré-calcul qui effectue la recherche statique de méthodes. Le mécanisme de *dispatch* par les méthodes `accept` est

[2] Outils basés au-dessus de l'API DOM
[3] Nous avons pu tester ces deux implémentations pour nos visiteurs configurables.

ainsi évité et les méthodes `visit` peuvent être personnalisées. Plus précisement, avec ce pré-calcul, tout appel de méthode `visit` est remplacé, de manière cachée, par l'appel à une méthode générique centrale. Les auteurs de JastAdd [41] proposent une solution assez proche mais ils utilisent une transformation de programme source (nommée *class weaving*) et non pas la génération de code.

En nous intéressants à la programmation par aspects ([23, 49] ou chapitre 8 de [32]), nous nous sommes aperçus qu'il était très facile d'ajouter ce style de programmation restreint aux programmes sources construits à l'aide du patron de conception visiteur. En effet, en étudiant la transformation de programme source effectuée par l'outil AspectJ [3], nous nous sommes rendus compte que, dans notre cadre, il suffisait d'étendre notre méthode centrale avec un mécanisme similaire de branchement d'aspects. A l'époque de l'introduction des aspects dans nos visiteurs, AspectJ ne proposait que des aspects par transformation de code source et JAC (*Java Aspect Components*) [9] n'existait pas.

Les intérêts de cette approche de visiteurs configurés sont les suivants :

- une programmation naturelle d'analyses de programmes où les mécanismes complexes mis en œuvre sont cachés aux utilisateurs, grâce à la génération de code ;
- une programmation par aspects dédiée aux visiteurs sans MOP (*Meta-Object Protocol*) et sans utilisation de bibliothèques supplémentaires. La génération de code prépare le code des analyses à accepter des branchements d'aspects statiques ou dynamiques ;
- des analyses efficaces, n'utilisant pas la réflexivité, et extensibles par héritage ou par des aspects.

La deuxième approche : visiteur découplé

Pour cette approche, nous avons souhaité rendre le code le plus réutilisable possible afin de pouvoir créer, par composition, de nouvelles analyses. Pour obtenir un tel résultat, nous avons poussé plus loin le processus de séparation en partitionnant non seulement la structure mais aussi le parcours et les actions sémantiques. Ainsi une action ne contient que le code métier. Avec cette approche, le corps d'une méthode `visit` est décomposé en n morceaux (aspects) qui sont exécutés avant, après et entre les appels sur les fils. Une telle spécification déclarative peut être comparée à un aspect. Mais la contrepartie de la réutilisabilité du code est que le code est plus complexe à écrire, à cause de l'absence des instructions de parcours qui, d'une certaine manière, structurent le code.

Notre «visiteur» prend donc, en entrée, les actions sémantiques à composer et le parcours choisi, commun à toutes ces actions. Puis, en fonction du parcours, il avance sur la structure et tisse le code des actions. Nous avons ainsi réalisé une implémentation du patron de conception visiteur en aspect [39, 40].

Ainsi une analyse complexe peut être construite par composition d'analyses élémentaires (*separation of concerns*), simplifiant sa programmation. De plus, cette approche permet d'enrichir, avec de nouveaux traitements, une analyse existante sans la modifier ou la recompiler. Pour le langage TINY, nous avons, de cette manière, ajouté un module de vérification d'initialisation de variable à la vérification de type ; ces deux analyses se partageant une table de symboles.

Les intérêts de cette approche découplant le parcours et les actions sémantiques sont les suivants :
– des actions sémantiques réutilisables et dédiées à une seule préoccupation ;
– des analyses pouvant être construites par composition et faciles à enrichir.

Ce chapitre présente ces deux améliorations du patron de conception visiteur. La première section décrit les visiteurs à signatures et parcours configurables, et à aspects. La deuxième section présente les visiteurs ayant le parcours et les analyses découplés. Enfin la troisième section indique, brièvement, comment des visiteurs génériques (indépendants de tout langage) ont pu être créés[4].

4.2 Visiteur configurable et à aspects

La patron de conception visiteur est adapté à l'analyse de structures de données typées comme les arbres de syntaxe abstraite. Avec ce patron et en programmation par objets, les analyses, écrites de manière classique, présentent les bonnes caractéristiques de modularité et d'extensibilité. Mais le code de ces analyses est assez illisible à cause des appels récursifs sur les fils indirects (passage par la méthode `accept`) et des nombreuses coercitions de type des paramètres et des valeurs de retour des méthodes `visit` (voir, par exemple, la Figure 4.3). Ce code est assez compliqué à écrire, surtout pour des concepteurs de langages métiers qui n'ont pas forcément les compétences appropriées.

```
public Object visit(AffectNode node, Object env) throws VisitorException {
   String varName = node.getVariableNode().getValue();
   String typeLeft = ((TinyEnv)env).getType(varName);
   String typeRight = (String)node.getValueNode().accept(this, env);

   if (typeLeft == null) {
      errors.setError(node, "This variable " + varName + " was not declared");
   } else {
      if (!typeRight.equals(TinyEnv.ERROR) && (!typeLeft.equals(typeRight)))
         errors.setError(node, "Incompatible types: " + varName + " is a" +
                  (typeLeft.equals(TinyEnv.INT)?"int":"bool")  +" variable");
   }
   return null;
}
```

FIG. 4.3: Vérification de type du nœud *affect* avec une visite classique.

Nous proposons donc une amélioration de cette technique afin de :
– rendre possible la configuration des signatures des méthodes `visit` et du parcours ;
– d'avoir des appels récursifs sur les fils explicites (suppression des méthodes `accept`);
– et de permettre l'ajout d'aspects pendant la phase de programmation ou à l'exécution.

[4]Ils n'ont pas encore été portés en version 4.

Avec cette amélioration, la méthode `visit` de la Figure 4.3 peut être écrite bien plus simplement (Figure 4.4).

```
public Object check(AffectNode node, tiny.visitors.symtab.TinyEnv env)
              throws VisitorException {
   String varName = node.getVariableNode().getValue();
   String typeLeft = env.getType(varName);
   String typeRight =  check(node.getValueNode(), env); //visit the value node

   ... même code if que précédemment
   return null;
}
```

FIG. 4.4: Vérification de type du nœud *affect* avec notre première approche.

Une telle amélioration rendant le code plus lisible et extensible (aspects en plus de l'héritage) est obtenue grâce à notre approche générative.

4.2.1 Signatures des méthodes `visit` configurables

Nous offrons la possibilité de préciser les signatures (profils) des méthodes `visit` afin de générer les visiteurs par défaut avec des méthodes ayant les noms, les types Java de retour et les paramètres souhaités. La granularité de cette personnalisation des signatures, faite avec notre langage métier VIPROFILE, se situe au niveau des types (types ABSYNT) et a la syntaxe suivante :

```
TypeJavaRetour nomMethodeVisit "(" "%" NomTypeDeNoeud "," TypeJavaArg2 nomArg2
                                               ("," TypeJavaArg3 nomArg3)*)
```

A partir de la spécification ABSYNT d'un langage et d'une spécification VIPROFILE (voir Figure 4.5) associée, on génére un visiteur abstrait et un visiteur de parcours. Le visiteur abstrait définit une méthode par type et une par couple $(type, constructeur\ associé)$, et en déclare une abstraite par constructeur. Le visiteur de parcours en hérite et implémente ces méthodes abstraites afin d'effectuer un parcours en profondeur des arbres. Pour définir une nouvelle analyse (vérificateur de type, évaluateur, compilateur) ou une transformation, il suffit d'étendre par héritage ce visiteur de parcours et de surcharger certaines méthodes `visit`.

Le code ci-dessous présente les méthodes `visit` générées dans ces deux visiteurs pour le type `Statement` du vérificateur de type du langage TINY.

```
Spécification ABSYNT (voir Figure 3.3 page 51 pour la spécification complète)
Statement = affect(Var variable, Exp value),
            while(ConditionExp cond, Statements statements), ... ;

Spécification VIPROFILE (voir Figure 4.5 page 85 pour la spécification complète)
Object check(%Statement, tiny.visitors.symtab.TinyEnv env);

Méthodes générées que l'on peut surcharger
/*_______code généré du visiteur abstrait du vérificateur de type________*/
public java.lang.Object check(tiny.ast.StatementType node,
            tiny.visitors.symtab.TinyEnv env) throws VisitorException {
```

```
ViProfile TypeChecker;
Formalism tiny;
import tiny.visitors.symtab.TinyEnv;

Profiles
 Object check(%Top, tiny.visitors.symtab.TinyEnv env);
 Object check(%Decl, tiny.visitors.symtab.TinyEnv env);
 Object check(%Statements, tiny.visitors.symtab.TinyEnv env);
 Object check(%Statement, tiny.visitors.symtab.TinyEnv env);
 String check(%StringOrVar, tiny.visitors.symtab.TinyEnv env);
 String check(%Exp, tiny.visitors.symtab.TinyEnv env);
 String check(%ArithmeticOp, tiny.visitors.symtab.TinyEnv env);
 String check(%ConditionOp, tiny.visitors.symtab.TinyEnv env);
 String check(%ArithmeticExp, tiny.visitors.symtab.TinyEnv env);
 String check(%ConditionExp, tiny.visitors.symtab.TinyEnv env);
 String check(%Var, tiny.visitors.symtab.TinyEnv env);
```

FIG. 4.5: Fichier VIPROFILE associé au vérificateur de type du langage TINY

```
   return (java.lang.Object) invokeVisit(new Object[] {node, node, env});
}
public java.lang.Object check(tiny.ast.StatementType type,
            tiny.ast.AffectNode node, tiny.visitors.symtab.TinyEnv env)
                   throws VisitorException {
   return (java.lang.Object)check(node, env);
}
public java.lang.Object check(tiny.ast.StatementType type,
            tiny.ast.WhileNode node, tiny.visitors.symtab.TinyEnv env)
                   throws VisitorException {
   return (java.lang.Object)check( node, env);
}

/*____code généré du visiteur de parcours du vérificateur de type _______*/
public java.lang.Object check(tiny.ast.AffectNode node,
            tiny.visitors.symtab.TinyEnv env) throws VisitorException {
   visitAttributes(node.getAttributes(), new Object[] {env});
   check(node.getVariableNode(), env);
   check(node.getValueNode(), env);
   return null;
}
public java.lang.Object check(tiny.ast.WhileNode node,
            tiny.visitors.symtab.TinyEnv env) throws VisitorException {
   visitAttributes(node.getAttributes(), new Object[] env);
   check(node.getCondNode(), env);
   check(node.getStatementsNode(), env);
   return null;
}
```

Les méthodes générées de ces deux visiteurs peuvent être surchargées. La méthode du type `Statement` (la première de la liste) est invoquée lors d'appels récursifs de l'analyse sur des nœuds de type statique `Statement`. Grâce à la méthode `invokeVisit` et à des informations générées dans le visiteur abstrait, cet appel est envoyé sur la méthode `check` du couple $(\texttt{Statement}, \textit{type dynamique du noeud})$ puis enfin, sur celle du constructeur, comme le montrent la Figure 4.6 et le diagramme de séquence de la Figure 4.7.

Les méthodes $(type, constructeur)$ servent à différencier le traitement d'un constructeur en fonction de son type d'appartenance. Ainsi un constructeur appartenant à trois types

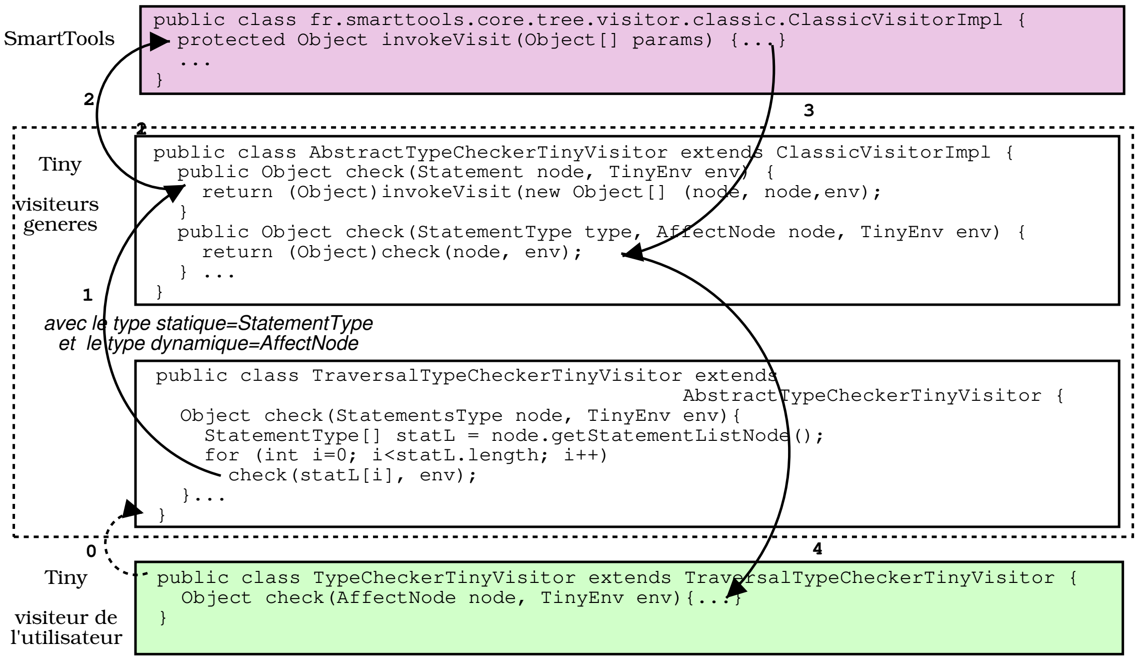


FIG. 4.6: Mécanisme d'appel des méthodes

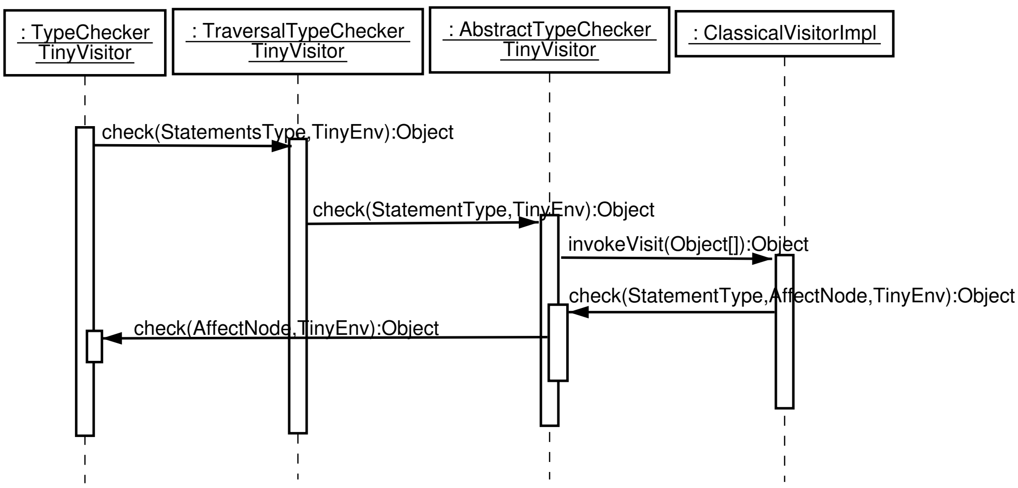


FIG. 4.7: Exemple de diagramme de séquence d'un visiteur configuré

peut avoir trois traitements différents. Cette possibilité permet la définition d'analyses plus précises.

Il est aussi possible d'indiquer un traitement particulier sur les attributs, commun à toutes les méthodes, avec la méthode `visitAttributes`. Ce traitement est effectué, par défaut, avant les appels récursifs sur les fils dans les méthodes du visiteur de parcours.

Contraintes sur les signatures

Les signatures doivent respecter certaines règles afin que les visiteurs générés puissent être compilés. Ces règles, dues à notre implémentation en Java, sont les suivantes :

Soient
`T1` et `T2` deux types ;
`op` un opérateur ;
`R1`, `R2`, $\mathtt{A}_0, \ldots, \mathtt{A}_m$, $\mathtt{B}_0, \ldots, \mathtt{B}_n$ des types Java ;
`R1 nomVisite1(%T1,` $\mathtt{A}_0$`, ...,` $\mathtt{A}_m$`)` et
`R2 nomVisite2(%T2,` $\mathtt{B}_0$`, ...,` $\mathtt{B}_n$`)` deux signatures.

si $\mathtt{T1} \cap \mathtt{T2} \neq \emptyset$
si $\mathtt{op} \subset \mathtt{T1}$ et $\mathtt{op} \subset \mathtt{T2}$
si `nomVisite1` = `nomVisite2`
si les nombres de types Java des signatures sont égaux (`m=n`), au moins un des types Java des signatures doit être différent (sous-classe autorisée).
sinon (c-à-d $\mathtt{T1} \subset \mathtt{T2}$)
`R1` = `R2` ou `R2` doit être un sous-type de `R1`. Le sous-typage des signatures est co-variant vis-à-vis du type de retour.
Les types A_0 à A_m doivent être inclus dans les types B_0 à B_n
si `nomVisite1` = `nomVisite2`, voir contrainte ci-dessus

Dans tous les autres cas, il n'y a pas de contraintes

4.2.2 Parcours configurable

Il est aussi possible de spécifier le parcours souhaité (du point de départ au(x) point(s) d'arrivée) du visiteur avec le langage VIPROFILE selon la syntaxe suivante :

```
"Traversal" NomDuParcours
   NomConstructeurOuTypeDepart   "->"    NomConstructeurOuTypeArrivee
                                    ("," NomConstructeurOuTypeArrivee)*
avec NomConstructeurOuTypeDepart = NomConstructeurOuTypeArrivee = "%" NomType | NomConstructeur
```

Ainsi, uniquement les visites des nœuds appartenant au chemin sont effectuées. Dans ce cas, l'exécution de l'analyse est plus rapide (caractéristique intéressante pour des arbres conséquents) et surtout, la taille du code des visiteurs générés est réduite. Une analyse de graphe de dépendance sur la définition ABSYNT détermine quelles sont les méthodes à générer et à appeller dans les visiteurs, abstrait et de parcours, correspondants. Par exemple, selon le parcours de la Figure 4.8, les méthodes générées et appelées sont seulement celles des constructeurs `while` et `affect` et celles des constructeurs contenus entre le type `Top` (la racine) et les constructeurs `while` et `affect` (selon la définition ABSYNT du langage : `program`, `statements` et `if`).

Par défaut, si aucun parcours n'est spécifié dans le fichier VIPROFILE, on génère un parcours complet en profondeur de l'arbre. Plusieurs parcours, donc plusieurs visiteurs avec les mêmes signatures de méthodes, peuvent être spécifiés, en même temps, dans un tel fichier. La génération de ces visiteurs peut aussi s'effectuer sans fichier VIPROFILE ; dans

```
Traversal Essai:
   %Top -> while, affect;
```

FIG. 4.8: Spécification d'un parcours en VIPROFILE pour aller de la racine vers les nœuds `while` et `affect`.

ce cas, le parcours en profondeur est complet, le type de retour des méthodes est `Object` et il y a un argument supplémentaire, en plus du nœud, de type `Object`, pouvant être utile pour stocker des données de l'analyse.

4.2.3 Ajout dynamique d'aspects

Avec notre technique, chaque appel récursif de la méthode `visit` avec le type statique d'un nœud est redirigé, de manière invisible pour l'utilisateur, vers la méthode générique `invokeVisit`. Il est ainsi très facile, dans cette méthode, de rajouter l'exécution de code supplémentaire avant et après l'appel effectif à la méthode `visit` souhaitée (selon le type dynamique du nœud). Cela permet d'introduire une notion d'aspect spécifique à nos visiteurs et sans transformation de programme.

Pour définir un aspect, il suffit de spécifier le code à exécuter avant et après les visites (voir Figure 4.9), puis de l'enregistrer, statiquement (voir ligne 13 de la Figure 4.10) ou dynamiquement, sur le visiteur choisi.

```
package fr.smarttools.core.tree.visitor.aspect;
import fr.smarttools.core.tree.visitor.VisitorException;
import fr.smarttools.core.tree.ast.Type;
import fr.smarttools.core.tree.UntypedNode;

public class TraceAspect implements Aspect {
   public void before(Type context, Object[] paramsVisit) throws VisitorException {
      UntypedNode node = (UntypedNode) paramsVisit[0];
      System.out.println ("node = " + node.toStringWithAttributes());
   }
   public void after(Type context, Object[] paramsVisit) throws VisitorException {}
}
```

FIG. 4.9: Code d'un aspect traçant les nœuds visités (*paramsVisit* contient tous les objets passés en paramètre des visites)

Afin d'offrir une meilleure granularité aux aspects, il existe trois sortes d'enregistrement :

- sur toutes les visites (de manière statique, avec la méthode *addAspect*)
- seulement sur les visites des nœuds appartenant à un certain type (*addAspectOnType*)
- seulement sur les visites des nœuds d'un certain constructeur (*addAspectOnOperator*).

Plusieurs aspects différents peuvent être enregistrés sur un même visiteur. Ils seront alors exécutés en série, selon l'ordre d'enregistrement. Ce branchement, comme le débranchement, peut se faire dynamiquement et à tout moment pendant l'exécution d'un visiteur.

```
package testing;
import tiny.visitors.MyVisitor;
import tiny.parsers.TinyParser;
import fr.smarttools.core.tree.UntypedNode;
import fr.smarttools.core.tree.visitor.aspect.TraceAspect;

public class MyVisitorLaunching {
   public static void main(String[] argv) {
      try {
         TinyParser parser = TinyParser(new java.io.FileReader("multiplication.tiny"));
         UntypedNode root = parser.parse().getDocumentElement();
         MyVisitor visitor = new MyVisitor();
         visitor.addAspectOnOperator("affect", new TraceAspect());
         Object params = null;
         visitor.start(root, params);
      } catch (java.io.FileNotFoundException e) {}
      } catch (fr.smarttools.core.tree.visitor.VisitorException e) {}
   }
}
```

FIG. 4.10: Exemple de lancement d'un visiteur avec branchement statique d'un aspect sur les nœuds `affect`

Le comportement d'un visiteur peut donc être modifié dynamiquement par ajout ou retrait d'aspects. Par exemple, le code d'un mode de *debug* graphique d'exécution pas-à-pas pour les visiteurs a été spécifié comme un aspect indépendamment du code des visiteurs et peut être ajouté à n'importe quel visiteur spécifié dans SMARTTOOLS.

REMARQUES
En cas d'appel d'une méthode `visit` avec un nœud correctement typé (type dynamique) ou de surcharge d'une méthode `visit` d'un type (par exemple, `Statement`), l'exécution du code de la méthode `invokeVisit` et donc des aspects n'est pas effectué. Pour résoudre ces problèmes très rares, l'utilisateur doit, soit effectuer une coercition de type (pour le premier cas), soit penser à appeler les méthodes de branchement d'aspects avant et après son code.

4.2.4 Détails d'implémentation

Pour obtenir des méthodes `visit` à signatures configurables, il est nécessaire de remplacer le mécanisme de résolution des types statiques en dynamiques des nœuds qui etait effectué par les méthodes `accept` des constructeurs. Il faudrait sinon regénérer l'ensemble des classes et interfaces des types et constructeurs à chaque nouvelle création d'analyse pour y ajouter les méthodes `accept` avec les nouvelles signatures. De plus, cette solution ne fonctionnerait pas si deux signatures de méthode `accept` du même constructeur avaient les mêmes paramètres et des types de retour incompatibles.

Une autre solution permettant, non seulement d'avoir des méthodes `visit` à signatures configurables, mais aussi des appels récursifs directs est l'emploi de la réflexivité et des multi-méthodes [35, 59]. Mais elle est peu performante en temps d'exécution. Pour l'améliorer, nous avions, dans la version précédente (version 3), généré une table d'indirection, stockant pour chaque couple $(type, \ constructeur \ associé)$ la référence Java de l'objet `java.lang.reflect.Method` à invoquer.

Dans la version actuelle, nous avons éliminé toute réflexivité. Il suffit de générer une table particulière et une méthode supplémentaire. La table stocke, de manière unique, tout couple $(type, constructeur)$ sous la forme d'une chaîne `"type@constructeur"`. La méthode (de nom `_switchTable`) gère, en fonction des indices de cette table (donc des couples), les appels aux méthodes `visit` avec les signatures adéquates. Ainsi la méthode centrale `invokeVisit` n'a plus qu'à invoquer cette méthode (avec les noms du type et du constructeur, et les paramètres) et à exécuter le code des aspects branchés avant et après l'appel (voir Figures 4.11 et 4.12). En apparence pour l'utilisateur, un appel récursif d'une méthode `visit` est direct mais, comme le montrent les Figures 4.6 et 4.11, il subit de nombreuses indirections cachées.

SmartTools

```
public class fr.smarttools.core.tree.visitor.classic.ClassicVisitorImpl {
  protected Object invokeVisit(Object[] params) {
     ...
     applyAspect(BEFORE, type, opName, params);
     _switchTable(typeName, opName, params);
     applyAspect(AFTER, type, opName, params);
     ...
  }
...                                                                      3'
}
```

Tiny - visiteur abstrait genere

```
public class AbstractTypeCheckerTinyVisitor extends ClassicVisitorImpl {
  protected final void _internalInit() {
     ...
     visitIndexTable.add("Statement@affect"); /* 22nd element of the table */
     visitIndexTable.add("Statement@while");
     ...
  }
  protected Object _switchTable(String typeName, String opName, Object[] params) {
     switch (visitIndexTable.indexOf(typeName + "@" + opName) {
     case 21 :
        return (java.lang.Object) check((tiny.ast.StatementType)param[0],    3''
                                        (tiny.ast.AffectNode)param[1],
                                        (tiny.visitors.symtab.TinyEnv)param[2]);
      ...
  }
  public Object check(StatementType type, AffectNode node, TinyEnv env) {...}
  ...
}
```

FIG. 4.11: Détails de l'étape 3 du mécanisme d'appel de la Figure 4.6

4.3 Visiteur découplé

Nous avons souhaité aller encore plus loin en séparant complètement le parcours de l'arbre (appels explicites aux méthodes `visit`) des traitements sémantiques. Ainsi, il est possible d'effectuer plusieurs traitements en un seul parcours et aussi d'enrichir une analyse avec d'autres modules pouvant même partager des données.

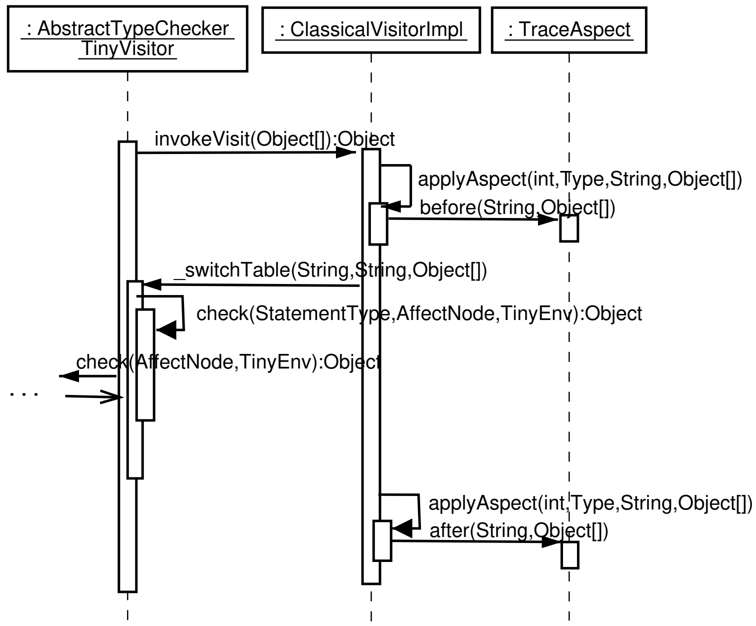


FIG. 4.12: Diagramme de séquence détaillé de la Figure 4.7

4.3.1 Séparation parcours/sémantique et composition de sémantiques

Supposons que le pseudo-code de la méthode `visit` du constructeur `op(T1 f1, T2 f2, T3[] f3)` soit de cette forme :

```
visit(OpNode node...) {
   codeBeforeOp
   visite du premier fils
   codeBetweenElem1_2
   visite du deuxième fils
   codeBetweenElem2_3
   codeBeforeArray3
   si tableau non vide alors
      visite du premier élément
      pour tous les autres éléments
          codeBetweenArray3Elem
          visite de l'élément courant
      finPour
   finSi
   codeAfterArray3
   codeAfterOp
}
```

On peut observer que la partie sémantique est décomposée en $(N\ fils + 1 + \sum_{i=1}^{T\ nb\ tableaux} (2 + P_i\ éléments))$ points. Ces points[5] - avant, entre et après les fils, avant et après un tableau, et entre les éléments des tableaux - peuvent être considérés comme des points de tissage et le code de la partie sémantique comme un aspect. A partir de cette observation, nous

[5] Les constructeurs atomiques ont seulement les points de tissage avant et après le constructeur

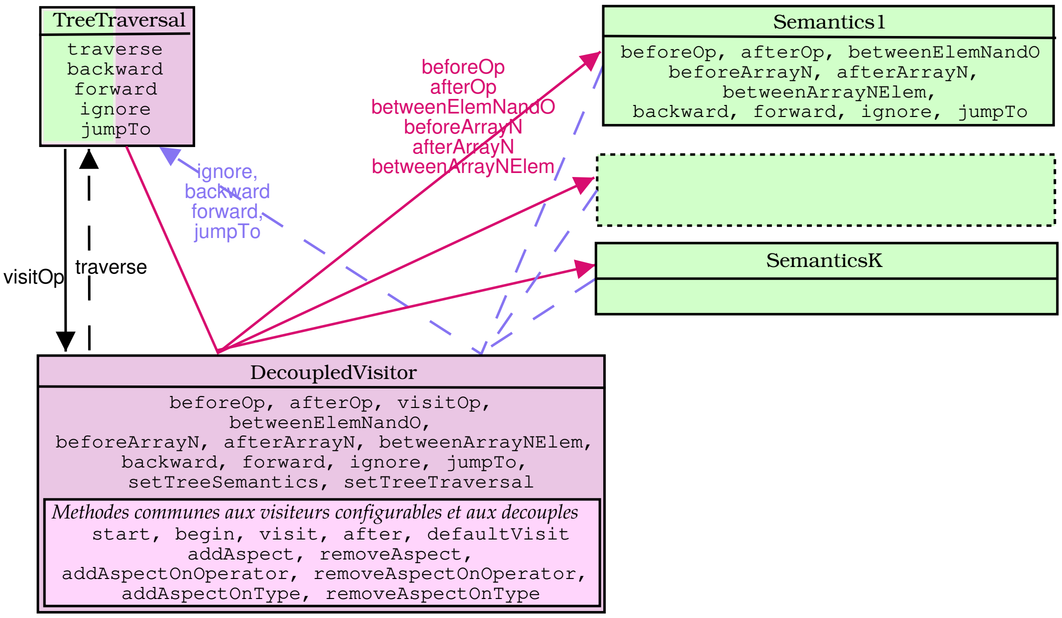


FIG. 4.13: Structure d'un visiteur découplé et principales méthodes

avons défini un visiteur prenant en paramètre un parcours d'arbre et une ou plusieurs sémantiques (ou aspects) comme le montre la Figure 4.13. Il est possible de l'étendre avec les aspects décrits dans le paragraphe 4.2.3. La Figure 4.16 présente la sémantique à écrire pour effectuer la vérification de type des nœuds de constructeur `affect`. Il n'y a plus d'appel récursif contrairement aux visiteurs configurables (voir Figure 4.4) mais il est nécessaire d'utiliser des piles pour transmettre les résultats des calculs des fils. Le diagramme de séquence de la Figure 4.15 montre quelles sont les méthodes appelées lors de la vérification de type (Figure 4.16 et de vérification d'initialisation de variables (Figure **??**) d'un nœud `affect` ; ces deux sémantiques composées sont respectivement nommées `Semantics` et `Semantics2` sur la Figure. Pour obtenir un schéma complet d'appels, un aspect (`TraceAspect`) a aussi été branché sur un visiteur découplé.

```
public Object beforeOp(AffectNode node, Object param) {return null;}
public Object betweenElem1and2(AffectNode node, Object param) {return null;}
public Object afterOp(AffectNode node, Object param) {
   String typeRight = (String)typeStack.pop();
   String typeLeft = (String)typeStack.pop();

   ... même code if que Figure 4.3
   return null;
}
```

FIG. 4.14: Sémantique de la vérification de type du constructeur *affect*

Dans certains cas (par exemple, l'écriture d'un évaluateur ou d'un interpréteur), il est nécessaire que la sémantique influence le parcours pour répéter (instructions itératives) ou

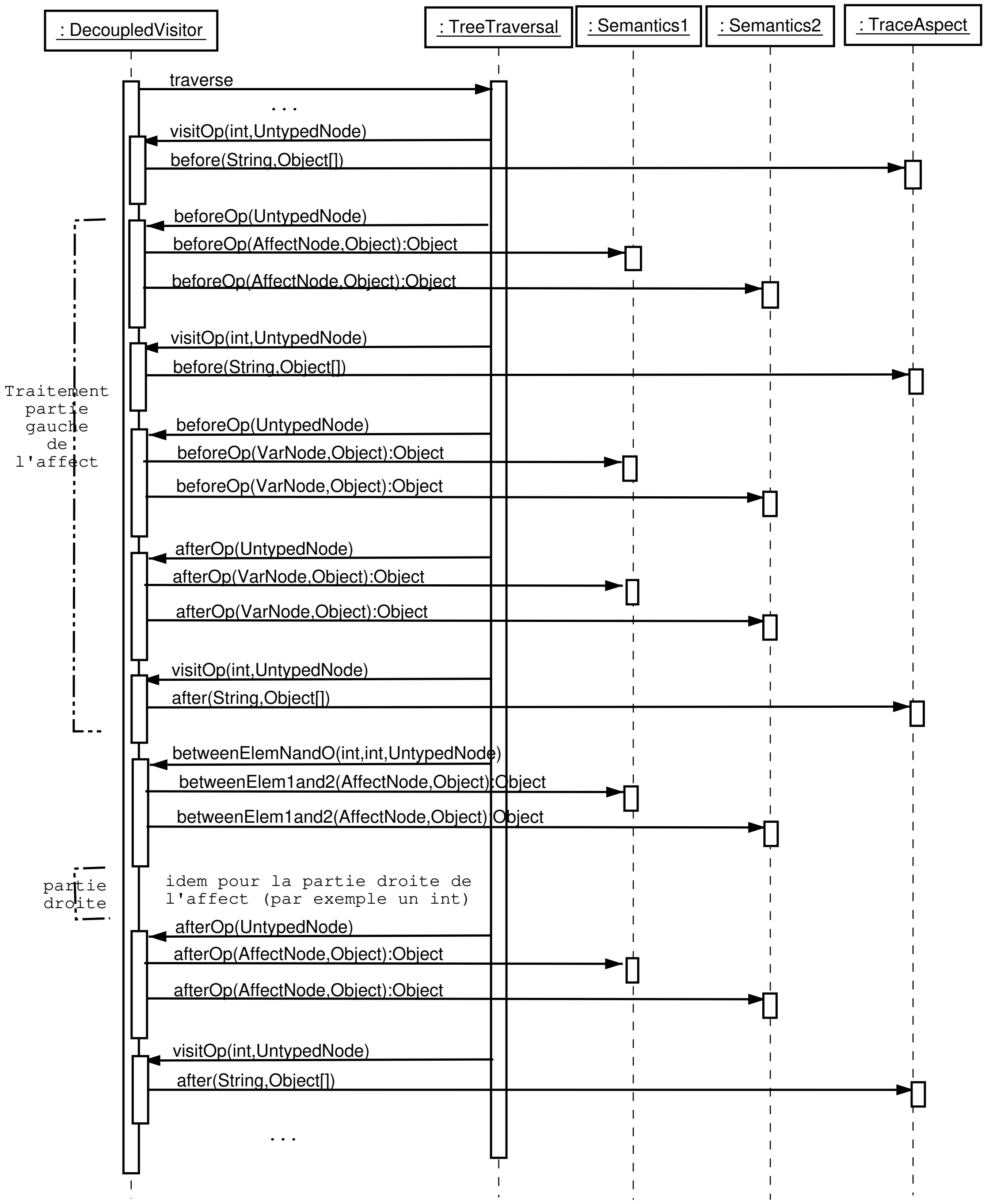


FIG. 4.15: Exemple de diagramme de séquence d'un visiteur découplé

```
public Object beforeOp(AffectNode node, Object param) {
   isTheAffectedVariable = true;
   return null;
}
public Object betweenElem1and2(AffectNode node, Object param) {
   isTheAffectedVariable = false;
   return null;
}
public Object afterOp(AffectNode node, Object param) {
   env.setInitialized(node.getVariableNode().getValue());
   return null;
}
```

FIG. 4.16: Sémantique de la vérification d'initialisation de variable du constructeur *affect*

pour sauter des nœuds (instructions conditionnelles). Elle doit pouvoir indiquer des changements de parcours au visiteur auquel elle est associée qui lui-même répercutera ces demandes au niveau de l'objet de parcours dynamique implémenté (notion comparable au `TreeWalker` de DOM). Ces instructions de modification de parcours sont les suivantes :

– *ignore* pour ignorer un nœud ou un ensemble de nœuds ;
– *jumpTo* pour aller au nœud indiqué ;
– *backward* pour reculer de n nœuds dans la liste des nœuds traités ;
– *forward* pour avancer de n nœuds si possible.

Il existe, actuellement, deux parcours d'arbre implémentés : total de gauche à droite et total de droite à gauche. L'utilisateur peut aussi s'en définir un spécifique pour son analyse. Avec ces instructions de modification de parcours, un évaluateur du langage TINY a pu être réalisé ; la Figure 4.17 montre le code du constructeur `while` où quand la condition est fausse, il faut ignorer le nœud de `statements` et quand elle est vraie, il faut à nouveau évaluer la condition après l'évaluation du nœud de `statements`.

```
public Object beforeOp(WhileNode node, Object param) {return null;}
public Object betweenElem1and2(WhileNode node, Object param) {
   if (((Boolean)conditionStack.peek()).booleanValue() == false)
      ignore(node.getStatements());
   return null;
}
public Object afterOp(WhileNode node, Object param) {
   if (((Boolean)conditionStack.pop()).booleanValue() == true)
      jumpTo(node.getCond());
   return null;
}
```

FIG. 4.17: Sémantique d'évaluation du constructeur *while*

L'intérêt principal de cette technique est de pouvoir enrichir, par composition et non pas par modification, une analyse existante avec de nouveaux modules. Une analyse complexe peut ainsi être décomposée en différents modules spécialisés sur une propriété et assemblés, dynamiquement, aux différents points de tissage par le visiteur. Cette possibilité de découpage modulaire améliore la réutilisabilité du code.

Par exemple, le vérificateur de type du langage TINY a ainsi été étoffé d'une vérification d'initialisation de variables, uniquement en composant les deux sémantiques (voir Figure 4.18). L'annexe B propose le code complet de ces deux sémantiques ainsi que le code de l'environnement. Cette annexe donne aussi, pour comparaison, la même vérification de type écrite avec un visiteur configuré (première approche).

```
TypeCheckerSem typeCheck = new TypeCheckerSem();
InitVarCheckerSem initVar = new InitVarCheckerSem(typeCheck.getEnv());
Visitor vis = new DecoupledVisitorImpl(new LeftToRightTreeTraversal(),
                             new Semantics[] {typeCheck, initVar});
vis.start(node, param);
System.out.println("--> " + typeCheck.getEnv().getVariables());
```

FIG. 4.18: Création d'un visiteur découplé à sémantiques composées

4.3.2 Détails d'implémentation et perspectives

Le rôle du parcours est de gérer le prochain nœud à visiter en fonction de la position courante, du parcours choisi et des instructions de modification de parcours envoyées par les sémantiques. Le visiteur, pour sa part, joue l'intermédiaire entre le parcours et les sémantiques (ils ne se connaissent pas) :

- il fait transiter les ordres de modifications de parcours envoyés par les sémantiques vers le parcours ;
- il tisse le code des sémantiques en appelant les méthodes des sémantiques en fonction des indications du parcours et en typant strictement, par réflexivité, les nœuds. Par exemple, si le parcours invoque `betweenElemNandO(1, 2, UntypedNode)` pour indiquer qu'il se trouve entre le fils 1 et 2 d'un nœud `affect`, le visiteur invoquera la méthode `betweenElem1and2(AffectNode, Object)` sur les sémantiques enregistrées ;
- il gère les aspects (décrits dans le paragraphe 4.2.3) mis sur un constructeur, un ensemble de constructeurs (type) ou sur tous.

Comme montré sur la Figure 4.19, les visiteurs configurables et les découplés héritent de la même classe, `AbstractVisitor`, et ont donc des méthodes communes (voir encadré de `DecoupledVisitor` de la Figure 4.13).

Cette technique semble très prometteuse, rendant les analyses plus modulaires et réutilisables, mais le code est encore assez compliqué à écrire à cause de la perte de la structure d'arbre. Dans un premier temps, il faudrait cacher l'utilisation des piles en générant, à partir de règles, un objet intermédiaire (environnement) de manipulation de données. Ainsi les empilages (`push`), dépilages (`pop`) et les inspections de l'élément au sommet (`peek`) seraient automatisés et invisibles pour l'utilisateur. Ces règles de composition [20] des sémantiques permettraient de vérifier que les assemblages sont cohérents vis-à-vis du parcours et des flux d'échanges de données.

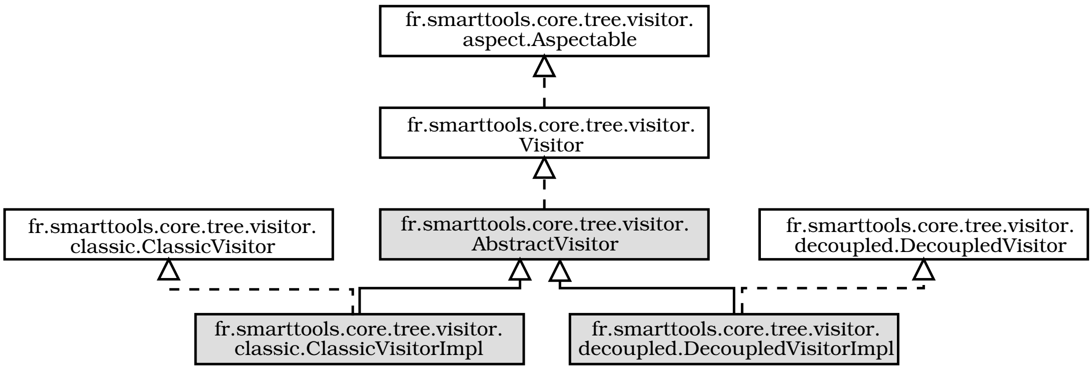


FIG. 4.19: Graphe d'héritage de nos visiteurs

4.4 Visiteur générique

Il est aussi possible de définir des visiteurs indépendants de tout langage (génériques). Les analyses, moins précises, peuvent ainsi être utilisées sur n'importe quel document quel que soit son langage. Dans la version précédente, nous avions ainsi défini une analyse permettant de représenter, sous forme graphique, n'importe quel arbre.

L'idée principale est de réaliser des méthodes `visit` au niveau des deux catégories des constructeurs et non au niveau, plus bas et dépendant d'un langage, des constructeurs. Il n'y a donc que deux méthodes `visit` : celle des constructeurs à arité de types fixe et à nœuds variable (`FixedNode`) et celle des constructeurs atomiques (`AtomicNode`).

Dans le visiteur générique par défaut, la méthode `visit` des constructeurs atomiques contient seulement les appels aux méthodes de visite des attributs et de la feuille (la valeur). L'autre méthode `visit` contient les instructions de parcours et les appels aux méthodes de visite suivants :
- sur attributs ;
- avant, après et entre les fils ;
- avant et après un tableau ;
- entre les éléments des tableaux ;

Par héritage et par surcharge, il est facile de se définir un nouveau visiteur générique. Dans la majorité des cas, toutes les méthodes sauf celles des catégories sont surchargées ; ce sont elles qui contiennent la sémantique.

Détails d'implémentation et perspectives

En fait la classe `GenericVisitor` contient une méthode supplémentaire (de type `UntypedNode` qui est la méthode centrale invoquée à chaque visite d'un nœud. C'est elle qui, avec un seul `instanceof`, détermine quelle méthode `visit` appeler entre celle de `AtomicNode` et celle de `FixedNode`. L'usage `instanceof`, bien que peu recommandé pour l'implémentation d'un visiteur, est acceptable dans notre cas où, il n'y a que deux éléments.

Dans le futur, il serait possible, à la manière du `DocumentTraversal` de DOM, de mettre des filtres sur certains nœuds. Il suffirait de tester, en fonction du nom du nœud, si celui-ci peut être ou non visiter.

Dans la méthode centrale, il serait aussi possible d'ajouter des aspects (ceux décrits dans le paragraphe 4.2.3).

En allant plus loin, nous pouvons aussi voir le code sémantique (toutes les méthodes sauf les visites sur les catégories des constructeurs) de ces visiteurs génériques comme des aspects. Des analyses génériques et composables pourraient être obtenues en changeant l'implémentation de la classe `GenericVisitor` et en rajoutant quelques tests. Ainsi, en un seul parcours d'arbre,plusieurs opérations pourraient être effectuées.

Mais de telles améliorations seraient incompatibles, à cause de l'augmentation du nombre de tests, avec des analyses à temps de réponse souhaité rapide ; comme, par exemple, celle construisant une représentation graphique des arbres pour l'interface utilisateur.

Conclusion

Nous avons proposé trois[6] complémentaires d'implémentation d'analyses d'arbres ayant chacune des qualités et des défauts :

– les analyses des visiteurs configurés sont simples, précises, extensibles mais peu réutilisables ;
– celles des visiteurs découplés sont réutilisables, précises, extensibles mais assez complexes ;
– celles des visiteurs génériques sont indépendantes de tout langage mais peu précises.

C'est l'utilisateur qui, en fonction de ses besoins et de ses connaissances, choisira quelle implémentation adopter pour programmer son analyse.

[6]Nous comptons, ici, en plus des deux approches présentées en introduction, le visiteur générique qui fut très brièvement décrit.

Chapitre 5

Architecture par composants

Introduction

Avec l'émergence d'Internet, la conception et le développement d'applications complexes doivent impérativement prendre en compte les aspects de répartition (application distribuée), déploiement et réutilisation de code. Cette évolution a pour conséquence un changement radical dans la programmation de telles applications. L'approche objet ne permet pas d'exprimer des interconnexions complexes entre les objets. Elle est trop orientée vers la programmation et se situe à un niveau trop fin. L'approche composant [25, 57] est apparue pour une meilleure séparation entre les aspects de communication et les parties fonctionnelles, pour s'abstraire des langages de programmation et pour construire des applications par assemblage de composants.

SMARTTOOLS est constitué d'outils de base et de composants spécifiques pour chaque langage traité. Ces outils et composants offrent et requièrent des services qu'il faut connecter et rendre accessibles à travers une interface utilisateur. Par exemple, chaque manipulation au niveau d'une vue graphique (sélection, remplacement, insertion ou appel d'un traitement) va se traduire par l'invocation des services au niveau du document associé. L'aspect générique de SMARTTOOLS, un méta-outil, impose que son interface utilisateur et son architecture soient les plus configurables et extensibles possibles pour pouvoir les adapter aux services spécifiques de chaque nouveau langage traité. Il est impératif que toutes les entités logicielles produites soient autonomes et puissent être exécutées sans nécessiter la présence de l'interface, par exemple en mode commande (*batch*) ou dans d'autres environnements. Il doit aussi être facile d'ajouter des composants externes en ne modifiant ni la plate-forme, ni ces composants.

Dans les premières versions de l'outil (version 3 comprise), l'absence d'une vraie approche par composants était un handicap pour son évolution. Ses composants étaient indissociables d'un bus logiciel [21] (voir paragraphe 4 page 20) qui assurait toutes les communications. Toute intégration et toute exportation de composants impliquaient la présence de ce bus. L'approche orientée composant s'est donc imposée pour la réalisation de la version 4 de notre outil. Aucun des modèles existants, tels que les *Web Services*, les composants CORBA (*CCM - CORBA Component Model*) [4, 58] ou les EJBs [7], ne semblait correspondre parfaitement à nos besoins spécifiques et faire le choix d'une technologie était assez difficile. De plus, cette dépendance vis-à-vis d'une technologie nous paraissait préjudiciable à l'évolution de l'outil. Nous souhaitions aussi avoir une implémentation «légère» et simple. Nous avons donc conçu notre propre modèle (indépendamment de ces technologies) adapté et restreint à nos besoins, mais surtout transposable vers ces technologies de composants. Cela nous a aussi permis de clairement identifier les besoins avant de commencer l'implémentation.

Il ressort, de nos premières expériences, que concevoir un modèle abstrait permet d'être en parfaite adéquation avec les besoins et contraintes du champ d'application. Nos langages de description de composants sont ainsi adaptés et offrent un moyen déclaratif pour décrire précisément les composants et l'assemblage qui forme une application. L'implantation effective du modèle (par génération de conteneurs à partir de ces descriptifs) prend en compte toutes les particularités de notre domaine sans pour autant nuire à la lisibilité de ces descriptifs. Cette génération de conteneurs automatise la programmation des parties non-fonctionnelles des composants. Elle facilite l'ajout de nouvelles fonctionnalités, non initialement prévues. Cette séparation entre le modèle et sa mise en œuvre rend possible l'utilisation de nos composants dans d'autres technologies. En effet, à partir de ces descriptifs, un outil de transformation génère les descriptions (ou interfaces) équivalentes pour les *Web Services*, les composants CORBA ou encore les EJBs. Cette exportation nous a permis d'identifier les particularités, avantages et inconvénients de chaque modèle. Cela a confirmé que l'utilisation d'une de ces technologies aurait été préjudiciable. En effet, certaines caractéristiques dynamiques de nos composants auraient été difficilement exprimables sans une spécialisation des composants en fonction des langages traités.

Ce chapitre se décompose en cinq sections. La première section décrit notre modèle de composants et les principaux composants de SMARTTOOLS. La deuxième section présente l'implémentation de ce modèle. La troisième section traite des particularités dynamiques de ce modèle et présente la projection des composants vers les technologies existantes. La quatrième section évalue notre modèle suivant des critères communément admis [57]. Enfin, la dernière section dresse le bilan de cette approche.

5.1 Description du modèle et des composants de SMARTTOOLS

Lors de la conception de notre modèle, nous nous sommes imposés deux contraintes : avoir une nette séparation entres les parties fonctionnelles et non-fonctionnelles, et avoir un modèle qui, bien que spécifique à nos besoins, soit «projetable» vers d'autres technologies existantes. En respectant ces contraintes, nous avons conçu un langage de description de composants en XML indépendant des technologies et créé des composants pour notre application.

5.1.1 Le langage de description des composants

Avec ce langage, on décrit le type d'un composant : les services (connecteurs) offerts ou demandés, les modes de communication et les attributs de configuration de l'état du composant. Pour chaque service, on indique son nom logique, la méthode à appeler dans l'interface de la partie métier (la façade du composant qui est l'unique point d'entrée ; cela correspond au patron de conception façade) et les paramètres (`parameter`) de la méthode. Les modes de communication sont asynchrones de catégorie `input`, `output` ou `inout`. La catégorie `input` (vs. `output`) indique que le service est offert (vs. demandé) par le composant. La catégorie `inout` indique que le service est offert par le composant, qu'il a un résultat et qu'il est exécuté de manière non interruptible par les autres services. Cette catégorie de services est plutôt employée dans la phase de connexion de deux composants qui sera détaillée plus loin.

Il est possible de factoriser les parties communes des descriptions dans des descriptions abstraites. Par exemple, la description abstraite `abstractContainer` (Figure 5.1) contient tous les connecteurs communs à nos composants avec leurs paramètres. Elle est donc importée par tous les autres descriptifs. La Figure 5.2 montre, sur ce descriptif abstrait, le formalisme adopté pour représenter graphiquement ces descriptifs. Ce formalisme est utilisé dans la sous-section suivante pour décrire rapidement nos composants (à la place du format XML).

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<component name="abstractContainer">
  <input method="quit" name="quit"/>
  <output name="connectTo" method="connectTo">
    <parameter name="ref_src" javatype="java.lang.String"/>
    <parameter name="type_dest" javatype="java.lang.String"/>
    <parameter name="id_dest" javatype="java.lang.String"/>
    <parameter name="actions" javatype="java.util.HashMap"/>
  </output>
  <output name="exit" method="exit"/>
  <output name="initData" method="initData">
    <parameter name="inits" javatype="java.util.HashMap"/>
  </output>
  <inout name="requestInitData" method="requestTree"
         output="initData" outputParameter="inits"/>
</component>
```

FIG. 5.1: Description abstraite `abstractContainer`

Rapides explications des services communs :

- `quit` qui avertit le composant qu'il doit se terminer. Ainsi, il peut mettre fin à ses connexions avec les autres ou sauvegarder des informations ;
- `requestInitData` qui demande les données publiques, exportables du composant (par exemple, l'AST du document) de manière non interruptible. Ces données

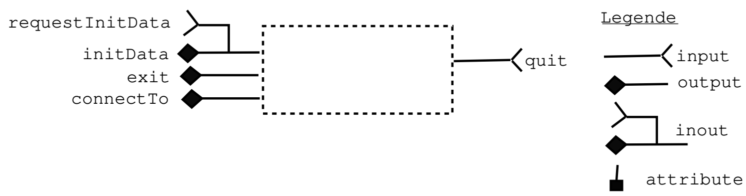


FIG. 5.2: Connecteurs communs à tous les composants

sont ensuite transmises sur le port `initData`. Ce service est utilisé pour transmettre des données lors de la connexion de deux composants, par exemple une vue et un document (ce mécanisme sera décrit en 5.2.2 page 114) ;
- `exit` qui demande la terminaison de l'application ;
- `connectTo` qui demande une connexion avec un autre composant en précisant son identification (une clé unique par instance), le nom du type ou l'identification du composant auquel il souhaite se connecter et les données d'initialisation (messages, attributs de configuration).

La description d'un composant concret contient, au minimum, les éléments suivants :
- le nom du composant ;
- son groupe d'appartenance[1] ;
- le nom de la description importée (optionnel) ;
- le nom du conteneur ;
- le nom de la façade de la partie métier ;
- les archives utilisées par le composant ;
- les attributs de configuration des instances du composant ;
- les noms et les catégories des services avec les noms des méthodes correspondantes de la façade et les noms et les types des paramètres.

Pour illustrer, nous allons donner la description de notre composant jouet (voir Figure 5.3), nommé `graph`.

Exemple de composant : le composant graph

Le composant graph est utilisé, par exemple, pour visualiser le graphe des composants chargés et des instances créées constituant l'application. Sa description indique qu'il s'appelle `graph` (`name=graph` ligne 2), qu'il appartient au groupe `graph` (`group=graph` ligne 2), qu'il étend la description abstraite `abstractContainer` (`extends=abstractContainer` ligne 2), que son conteneur s'appelle `GraphContainer` (ligne 3) et sa façade `GraphApp` (ligne 4), qu'il utilise l'archive `koala-graphics.jar` (ligne 5), qu'il n'a pas d'attribut de configuration et qu'il fournit, en plus des services communs à tous les composants, quatre services (de la ligne 6 à la ligne 21). Ces services sont les suivants :
- `addNode` qui ajoute un nœud au graphe en précisant son nom, sa forme (ovale, rectangle, etc) et sa couleur ;

[1] Cette notion est expliquée dans la sous-section suivante.

– `addEdge` qui ajoute un arc entre deux nœuds en précisant le nom du nœud d'origine et celui de destination ;
– `removeNode` qui enlève le nœud nommé ;
– `removeEdge` qui enlève l'arc existant entre deux nœuds en précisant le nom du nœud d'origine et celui de destination.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<component name="graph" group="graph" extends="abstractContainer">
  <containerclass name="GraphContainer"/>
  <facadeclass name="GraphFacade" userclassname="GraphApp"/>
  <dependance name="koala-graphics" jar="koala-graphics.jar"/>
  <input name="addNode" method="addNode">
    <parameter name="nodeName" javatype="java.lang.String"/>
    <parameter name="nodeType" javatype="java.lang.String"/>
    <parameter name="nodeColor" javatype="java.lang.String"/>
  </input>
  <input name="addEdge" method="addEdge">
    <parameter name="srcNodeName" javatype="java.lang.String"/>
    <parameter name="destNodeName" javatype="java.lang.String"/>
  </input>
  <input name="removeNode" method="removeNode">
    <parameter name="nodeName" javatype="java.lang.String"/>
  </input>
  <input name="removeEdge" method="removeEdge">
    <parameter name="srcNodeName" javatype="java.lang.String"/>
    <parameter name="destNodeName" javatype="java.lang.String"/>
  </input>
</component>
```

FIG. 5.3: Description du composant graphe

La Figure 5.4 montre sa représentation graphique.

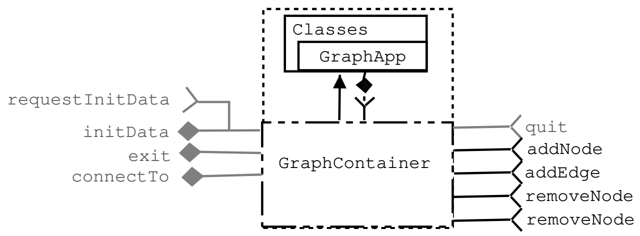


FIG. 5.4: Le composant graphe

5.1.2 Les principaux composants de SmartTools

Tous les composants de SmartTools possèdent, par défaut, les services décrits dans la description abstraite `abstractContainer` (Figure 5.1). Ils peuvent être classés en deux groupes selon leurs fonctionnalités : celui relatif aux vues et celui relatif aux documents. Le groupe graphe avec son unique composant est juste utile pour nos tests, non pour le fonctionnement principal de l'application. Cette distinction en groupes facilite la factorisation de connecteurs et du code.

Cette sous-section présente les différents composants de SMARTTOOLS. Seules les Figures peuvent être lues pour comprendre ces composants. Les descriptions des connecteurs sont aussi fournies dans le but de mieux appréhender le fonctionnement de SMARTTOOLS mais peuvent être ignorées dans une première lecture ; elles ne servent pas à comprendre le modèle de composants sous-jacent.

Le groupe Vue

Ce groupe regroupe tous les composants de visualisation qui ont, par nature, des comportements communs. Par exemple lors de leur construction, le composant vue et le composant interface utilisateur procédent de la même manière : ils appliquent une transformation XSLT sur l'AST envoyé par le composant document (connecteur `initData`), puis, sur le résultat obtenu, une feuille CSS et enfin l'interpréteur de BML qui crée les objets graphiques de la vue ; les objets graphiques sont crées du côté de la vue pour éviter la monopolisation du composant document. Toute vue dépend d'un document et tout document peut avoir plusieurs vues graphiques associées.

Certains comportements communs peuvent donc être factorisés ainsi que certains connecteurs (Figure 5.5 :

- `modification` qui avertit que l'AST dont dépend la vue a été modifié pour que cette modification soit répercutée sur la forme concrète. Le chemin XPath indiquant où la modification doit être effectuée ainsi que le nouveau sous-arbre (de syntaxe abstraite) à insérer ou à remplacer et le type de la modification (remplacement, insertion avant ou après, suppression) sont précisés ;
- `initData` qui reçoit les données publiques d'un autre composant en réponse à la demande `RequestInitData` ;
- `selection` qui avertit qu'une ou plusieurs sélections de sous-arbres ont été effectuées sur le document pour que ces sélections apparaissent sur la forme concrète. Pour chaque sélection, le chemin XPath indiquant où doit être mise la sélection ainsi que le type de sélection (courante, visiteur, etc) sont précisés ;
- `addBehavior` qui indique à la vue qu'il faudrait rajouter de nouveaux objets graphiques (items dans un menu, boutons) avec des comportements particuliers. C'est un moyen pour spécialiser la vue en fonction du type du document. De nouveaux connecteurs en sortie (`output`) peuvent ainsi être ajoutés dynamiquement à l'instance, par exemple pour pouvoir exécuter un traitement (visiteur) sur le document ;
- `select` qui demande une nouvelle sélection en précisant le chemin XPath du nœud à sélectionner et le type de sélection voulue ;
- `loadComponent` qui demande le chargement d'un composant (non son instanciation). Son nom, le nom et le chemin de l'archive (une *jar*) qui le contient ainsi que ses dépendances sont précisés ;
- `requestFile` qui demande l'ouverture du fichier nommé. Pour que cette demande puisse réussir, il faut que le composant du langage dont dépend le document soit chargé. Une telle requête provoquera la création d'une instance pour le document et une autre pour la vue ;
- `requestInitData` qui demande à un composant ses données publiques ;

– et les services communs à tous les composants (`requestInitData` et `initData`, `exit`, `connectTo` et `quit`).

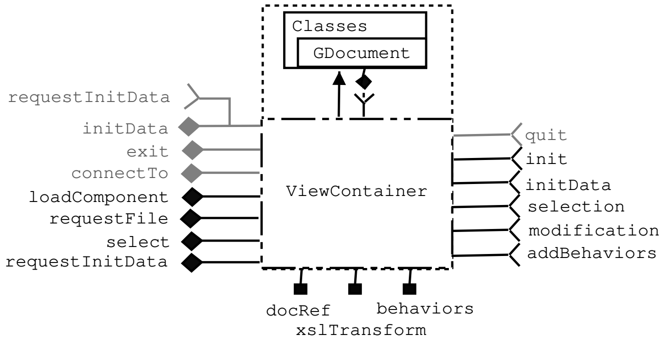


FIG. 5.5: Connecteurs communs à tous les composants de visualisation.

Ces composants possèdent aussi trois attributs communs de configuration : `docRef` qui indique l'identification du document auquel est rattachée la vue, `xslTranform` qui fournit le nom de la feuille de style XSLT à appliquer au document pour obtenir cette vue et enfin `behaviors` qui donne le nom du fichier à utiliser pour connaître les objets graphiques (et les comportements) à rajouter à la vue. Ces attributs sont initialisés lors de l'instanciation du composant. Ces composants dépendent d'outils liés à CSS.

Le composant vue - gview

Le type de composant vue «hérite» des services communs décrits ci-dessus, attributs et dépendances sans en ajouter de nouveau. Mais sa description n'est plus abstraite : elle indique que le nom du conteneur est `GViewContainer` et le nom de la façade `GdocView`. Ce type de composant est indépendant des langages traités pour pouvoir être réutilisé pour tous les langages traités. Il peut être spécialisé dynamiquement, en fonction du langage du document dont il dépend, par le service `addBehaviors`. De cette manière, il est possible d'ajouter un menu avec des actions spécifiques au langage du document (voir Figure 5.6).

`component/parts/diagrams/picGview.eps`

FIG. 5.6: Une vue d'un programme du langage TINY.

Le composant vue structurée - editionStructView

Cette vue (voir Figure 5.7) indique toutes les opérations d'édition possibles (insertion ou remplacement dirigés par le type de nœud attendu, suppression, édition libre, etc) sur le sous-arbre sélectionné et interagit avec le document pour les effectuer. Elle possède donc des connecteurs spécifiques (Figure 5.8) pour ces opérations :

– `editionStructData` qui récupére les informations utiles pour l'édition du nœud courant (opérateurs et types possibles, définitions de son opérateur et de celui de son nœud parent, numéro de rang de ce nœud) ;
– `copy` qui demande que le nœud courant (les attributs sont considérés comme des nœuds pour DOM) soit copié dans le bloc-notes (*clipboard*) ;
– `cut` qui demande que le nœud courant soit mis dans le bloc-notes et supprimé de l'AST ;
– `paste` qui demande que le nœud courant soit remplacé par le nœud présent dans le bloc-notes ;
– `insert` qui demande que la chaîne donnée soit insérée avant ou après le nœud courant si elle est syntaxiquement correcte et du type adéquat ;
– `replace` qui demande le remplacement du nœud courant par la chaîne donnée si elle est syntaxiquement correcte ;

component/parts/diagrams/picEditStruct.eps

FIG. 5.7: La vue d'édition structurée du langage TINY.

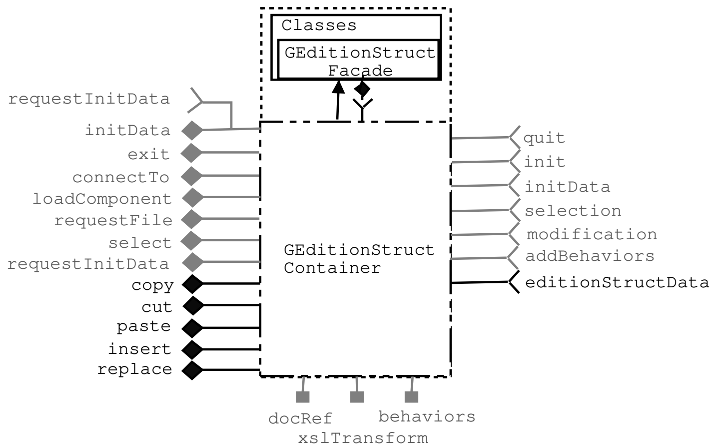


FIG. 5.8: Composant de la vue de l'édition structurée.

Le composant de debug des visiteurs - debugView

Cette vue (voir Figure 5.9) permet d'interagir avec un traitement sémantique (un visiteur) pour l'exécuter en mode pas-à-pas ou avec un temps de temporisation pour le ralentir et indique quelle est la méthode courante traitée. Cette vue n'a que deux connecteurs sup-

plémentaires (Figure 5.10) :
– `debugVisitorOutput` qui indique les noms de la méthode visitée et du nœud traité ;
– `debugVisitorInput` qui demande le changement du temps de temporisation, le lancement ou l'arrêt du visiteur.

component/parts/diagrams/picDebugView.eps

FIG. 5.9: La vue de *debug*.

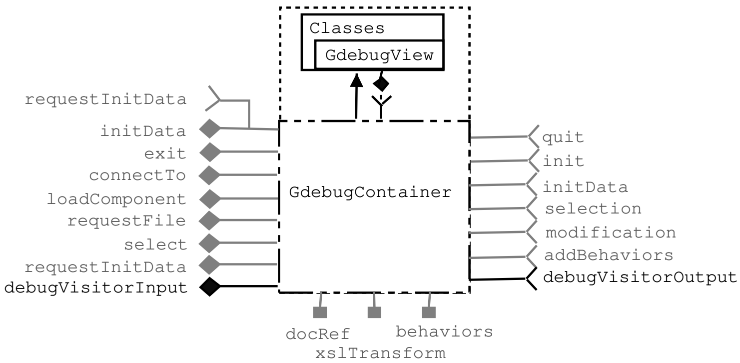


FIG. 5.10: Composant de la vue de *debug*.

Le composant interface utilisateur - glayout

Cette vue est l'interface utilisateur (voir Figure 5.11). Elle est composée d'onglets, de fenêtres, de *panels* contenant d'autres vues et de séparations verticales et horizontales. Son contenu et l'agencement de ses objets graphiques sont décrits par un arbre et donc cette vue n'est qu'une forme concrète de cet document. Ce composant est très similaire au composant `gview`. Il possède en plus trois services (Figure 5.12) :
– `statusInfo` qui affiche le texte donné dans la ligne d'état de la fenêtre courante ;
– `statusWarning` qui affiche le message d'erreur dans la ligne d'état de la fenêtre courante ;
– `update` qui indique que l'utilisateur souhaite ajouter un onglet, enlever l'onglet courant, ajouter une séparation horizontale ou verticale dans le *panel* courant, ou bien le supprimer.

component/parts/diagrams/picGUI.eps

FIG. 5.11: Exemple d'interface graphique.

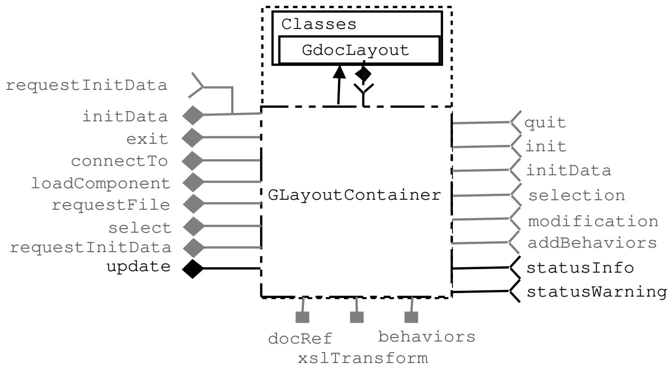


FIG. 5.12: Composant de l'interface utilisateur.

Le groupe Document

Ce groupe regroupe tous les composants relatifs aux langages de programmation ou métiers. Chaque langage traité constitue un composant spécifique regroupant tous ses outils (analyseurs syntaxiques, traitements, etc). Le descriptif de tels composants peut être complété d'informations supplémentaires pour indiquer :

- pour indiquer le nom du formalisme, de son fichier ABSYNT et de sa DTD (avec comme hypothèse que tous les répertoires des langages sont organisés de la même façon) ;
- pour mettre en correspondance les analyseurs syntaxiques disponibles et les extensions de fichiers ;
- pour préciser les comportements à ajouter ;
- pour décrire quelle(s) vue(s) par défaut et quels objets graphiques créer dans l'interface graphique lors de l'ouverture d'un document de ce langage.

Par défaut, tous les composants de ce groupe possèdent un attribut de configuration, `urlToSource` (qui identifie de manière unique où se trouve le document ; c'est une URI - *Uniform Resource Identifier*) et les connecteurs suivants :

- `selection` qui indique toutes les sélections présentes sur l'arbre (liste de chemins XPath et de types de sélection) quand un changement a lieu ;
- `modification` qui indique les modifications effectuées dans l'arbre (chemin XPath de la modification, nouveau sous-nœud, type d'opération de modification : insertion avant ou après, remplacement, suppression) pour qu'elles soient répercutées ;

– `editionStructData` qui donne les informations relatives à l'édition structurée du nouveau nœud courant ;
– `statusInfo` qui demande l'affichage d'un texte dans la ligne d'état de la fenêtre courante de l'interface graphique ;
– `statusWarning` qui demande l'affichage d'un message d'erreur dans la ligne d'état de l'interface graphique ;
– `copy` qui copie le nœud courant dans le bloc-notes ;
– `cut` qui met le nœud courant dans le bloc-notes et puis le supprime de l'arbre ;
– `paste` qui remplace le nœud courant par celui contenu dans le bloc-notes s'il est du type adéquat ;
– `insert` qui insére avant ou après le nœud courant la chaîne donnée si elle est syntaxiquement correcte ;
– `replace` qui remplace le nœud courant par la chaîne donnée si elle est syntaxiquement correcte et du type adéquat ;
– `select` qui change l'emplacement de la sélection courante pour le nœud donné (chemin XPath) ;
– `launchVisitor` qui lance le visiteur dont le nom est donné sur l'arbre ;
– `launchXSL` qui applique la transformation XSLT dont le nom est donné sur l'arbre ;
– `save` qui sauvegarde sur le disque dur le document (le nom du fichier ainsi que le nom de la syntaxe concrète à utiliser sont précisés) ;
– `startEditionStruct` qui avertit qu'une vue d'édition structurée vient être ajoutée. Cette information est nécessaire pour, lors d'un changement de sélection (connecteurs `select` et `selection`), donner des informations sur le nouveau nœud selectionné par le connecteur `editionStructData`.

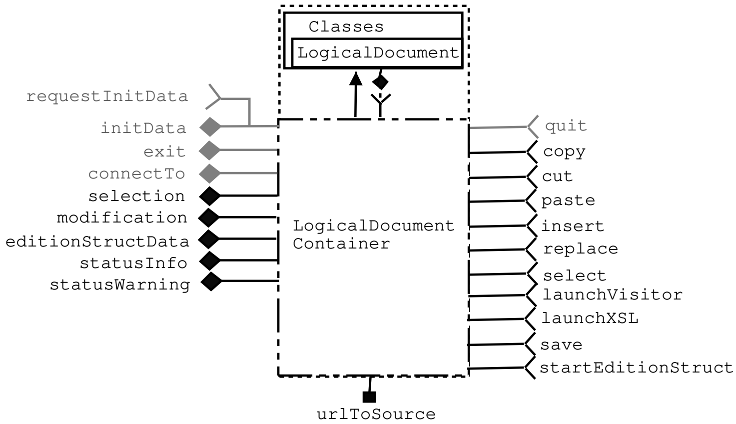


FIG. 5.13: Connecteurs communs à tous les composants de type Document.

Ces quatre derniers connecteurs (`launchVisitor`, `launchXSL`, `save` et `startEditionStruct`) n'apparaissent pas dans les descriptifs statiques des autres composants. En fait, ces connecteurs sont rajoutés dynamiquement à certains composants de visualisation lors de leur connexion avec un composant document. Nous détaillerons ce phénomène d'extensibilité dans la prochaine sous-section.

Pour l'instant, les erreurs relatives à des opérations impossibles sur l'arbre ne sont pas reportées aux vues. Mais ce manque devrait rapidement être comblé.

Exemple de composant d'un langage - le composant TINY

Ce composant «hérite» des connecteurs et de l'attribut de configuration communs à tous les composants documents. Il n'a pas de nouveaux services spécifiques. Sa façade s'appelle `TinyFacade` et son conteneur `TinyContainer`. Son descriptif (Figure 5.14) contient aussi les noms de ses deux analyseurs syntaxiques (un pour le XML et l'autre spécifié avec ANTLR pour une forme concrète plus lisible), du fichier de configuration de l'interface utilisateur et du fichier décrivant les comportements spécifiques au langage à rajouter aux vues.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<component name="tiny" group="document" extends="logicaldocument">
   <formalism name="tiny" file="tiny.absynt" dtd="tiny.dtd"/>
   <containerclass name="TinyContainer"/>
   <facadeclass name="TinyFacade"/>
   <parser type="xml" classname="tiny.parsers.TinyXMLParser">
      <extention name="tinyxml"/>
   </parser>
   <parser type="plain-text" generator="ANTLR" file="tiny.g"
                             classname="tiny.parsers.TinyParser">
      <extention name="tiny"/>
   </parser>
   <lml  name="DEFAULT" file="resources:tiny-default.lml"/>
   <behavior file="resources:tiny-behaviors.xml"/>
</component>
```

FIG. 5.14: Descriptif du composant TINY

REMARQUES

Cette sous-section montre la complexité des connexions entre composants qui étaient difficilement gérables avec un bus logiciel (version 3). En effet, à chaque nouvelle connexion, il fallait rajouter du code dans le «composant» demandeur du service (méthode `receive` de l'événement approprié) et dans le «composant» fournisseur du service (méthode `send` de l'évenement). Ainsi, nos «composants» étaient dépendants de ce bus et donc non exportables sans lui. Ils ne pouvaient pas établir une connexion point-à-point ou `broadcast` entre eux sans lui. Intégrer un nouveau composant impliquait aussi de le modifier pour qu'il ait la connaissance de ce bus. De plus, le déploiement de l'application était enfoui dans le code et donc non configurable facilement. La composition d'une application était statique. Le bus freinait l'évolution de l'outil.

Tous ces problèmes de code à rajouter pour les connexions, de «composants» difficilement intégrables ou exportables, de déploiement statique et d'évolution ont disparu avec notre modèle.

5.2 Mise en œuvre du modèle

En fonction des contraintes de la plate-forme, un gestionnaire de composants et un générateur de conteneurs ont été conçus pour l'implantation du modèle.

Pour chaque application construite avec SMARTTOOLS, un fichier de lancement précise les composants à charger et à instancier au démarrage.

5.2.1 Le générateur

Le générateur produit le conteneur du composant à partir de son descriptif. Tout conteneur hérite de `AbstractContainer` qui factorise les mécanismes et les structures de données liés au cycle de vie du composant et à la communication entre composants.

Pour avoir un outil interactif, il était nécessaire que l'interface graphique ne soit pas bloquée lors de traitements sur les documents. Chaque composant devait donc avoir son propre fil d'exécution (*thread*). Une application est donc composée de plusieurs *threads*. Par exemple, le lancement de l'application et l'affichage d'un programme impliquent la création de quatre *Threads* : un pour le gestionnaire de composants (qui est le premier composant créé et qui crée les autres comme expliqué ci-dessous), un pour l'interface graphique, un pour le document et un pour la vue. Etre dans un environnement *multi-threads* pose des problèmes d'accès concurrents aux ressources partagées. C'est dans cette classe, `AbstractContainer`, que sont gérés tous les mécanismes de synchronisation pour avoir des sections de code atomiques ou des verrous sur les données. Cette classe possède les données suivantes :

– une file d'attente FIFO (boîte aux lettres) pour stocker les demandes de services des autres composants (messages asynchrones) ;
– une table stockant, par service offert (catégories `input` ou `inout`), les références sur les conteneurs connectés demandeurs du service (même nom de connecteur mais en demande). Les services offerts non demandés ne sont pas référencés dans cette table ;
– une table stockant, par service offert, la référence sur le code à exécuter dans la façade ;
– une table stockant, par service demandé (catégories `output` ou `inout`), les références sur les conteneurs connectés offrant ce service ;
– l'état du composant : inactif, actif ou suspendu ;
– et la description du composant.

Le conteneur généré ne contient que :

– la création de la façade ;
– l'ajout des services offerts dans la table de mise en correspondance nom de service et code à exécuter dans la façade ;
– et l'implantation des interfaces des *listeners* (utiles pour que la façade communique avec le conteneur) spécifiques aux services demandés.

Le but du conteneur, s'il est dans un état actif, est de traiter les demandes de services mises dans sa file d'attente et de demander des services aux autres composants.

La Figure 5.15 montre le code généré à partir du descriptif de la Figure 5.3 pour le conteneur du *graph*. Les conteneurs des langages traités (documents) héritent de `LogicalDocumentContainer` qui factorise les définitions de services offerts et attributs, et l'implantation des interfaces des *listeners* communes à tous les conteneurs des langages.

```
package graph;
import fr.smarttools.component.*;
import fr.smarttools.util.*;
import java.util.HashMap;
import graph.*;

public class GraphContainer extends AbstractContainer
                            implements fr.smarttools.component.Container {
  public graph.GraphFacade facade;
  public void setFacade(graph.GraphFacade v) {this.facade = v;}
  public graph.GraphFacade getFacade() {return facade;}

  {
    calls.put("addEdge", new MethodCall() {
      public Object call(ContainerProxy expeditor, HashMap parameters) {
        ((GraphFacade) facade).addEdge(
                        (java.lang.String)parameters.get("destNodeName"),
                        (java.lang.String) parameters.get("srcNodeName"));
        return null;
      }});
    calls.put("removeNode", new MethodCall() {
      public Object call(ContainerProxy expeditor, HashMap parameters) {
        ((GraphFacade) facade).removeNode(
                        (java.lang.String) parameters.get("nodeName"));
        return null;
      }});
    ...
  }
  public   GraphContainer() {
    try {
      facade = new graph.GraphFacade();
      idName = "GraphContainer";
      resourceFilename = "/resources/graph.cdml";
    } catch (Exception e){
      e.printStackTrace();
    }
  }
  public  void serialize(){}
  public  String toString(){
    String res = "GraphContainer:[" + getIdName() + "]";
    return res;
  }
}
```

FIG. 5.15: Le conteneur `GraphContainer` généré

Le conteneur connait la façade car il la crée alors que la façade ne le connait pas et ne doit pas le connaître à priori. La communication conteneur vers façade (`input`) est faite naturellement par invocation de méthodes alors qu'il faut ajouter de nouveaux mécanismes pour rendre possible la communication façade vers conteneur (`output`). Il est donc nécessaire d'étendre la façade s'il y a des connecteurs en `output` et si elle ne possède pas ces mécanismes utiles à la communication ; leur présence est vérifiée par introspection. La façade doit pouvoir communiquer à l'extérieur de la partie métier de manière uniforme quel que soit le service demandé et le conteneur auquel elle est rattachée. La solution[2] est d'utiliser le mécanisme de *listeners*, correspondant au patron de conception *Observer*. Quand la façade souhaite communiquer avec le conteneur pour demander l'exécution d'un service en `output`, elle produit un événement et exécute le code associé à cet événement (le *handler*) de chaque conteneur enregistré comme *listener*. Chaque service en `output` correspond donc à une interface avec une méthode (le *handler*) qui est implémentée par le conteneur.

La façade étendue possède aussi la méthode `main` pour faciliter les tests unitaires et l'utilisation des traitements sémantiques en mode de commande (`batch`).

Une archive (ou paquetage de déploiement) peut aussi être générée lors de la compilation avec le système Ant, contenant l'ensemble des classes dont le conteneur et les descriptifs du composant. Chaque langage traité correspond à une archive. L'avantage est que seules les archives utiles à l'application sont chargées au démarrage pour l'accélérer et qu'elles peuvent être chargées dynamiquement à la demande lors de l'exécution. De plus, les *Web-Services* et les EJBs ont besoin d'une *jar* pour se déployer.

Cette phase de génération rend transparents les mécanismes de communication mis en œuvre dans le conteneur et automatise les tâches de création de paquetage pour chaque nouveau langage traité.

5.2.2 Le gestionnaire de composants

Le gestionnaire de composants est aussi un composant auquel tout composant est connecté pour pouvoir profiter de ses services dont le service `connectTo` nécessaire pour mettre directement en relation deux composants. Il charge aussi les paquetages des composants et crée les instances en fonction d'une description de lancement de l'application (Figures 5.20 et 5.19) et des demandes interactives des utilisateurs (par exemple, l'ouverture du premier document associé à un langage donné - message de type `requestFile`). Il mémorise l'ensemble des paquetages chargés et des instances créées. Il a aussi des services spécifiques (Figure 5.16) :

– `exit` qui termine l'application en envoyant un message `quit` à toutes les instances pour qu'elles se déconnectent proprement ;
– `connectTo` qui établit la connexion entre deux composants et peut même créer une instance d'un composant si nécessaire. Ce service est très important. Son fonctionnement est illustré ci-dessous avec une connexion d'une vue et d'un document ;
– `loadComponent` qui charge l'archive du composant donné et ses dépendances ;
– `availableComponent` qui indique toutes les instances créées ;
– `sleepComponent`, `startComponent`, `wakeUpComponent` `stopComponent` qui sont des services agissant sur le cycle de vie des instances pour les faire changer d'état (actif, suspendu, etc). Pour l'instant, seul le service `startCompon-`

[2] Stocker la référence du conteneur dans la façade étendue est une solution à éviter car elle implique des réferences mutuelles entre ces deux objets.

ent est utilisé pour le fonctionnement normal de l'application ; ces services sont réservés pour un futur usage ;
- requestFile qui envoie la demande d'ouverture du fichier au document Lml de l'interface graphique par le biais du connecteur processLmlSkeleton ;
- addNode, addEdge, removeNode, removeEdge qui indiquent le chargement d'un nouveau composant, la création ou la destruction d'une instance, et l'établissement des connexions entre deux instances.

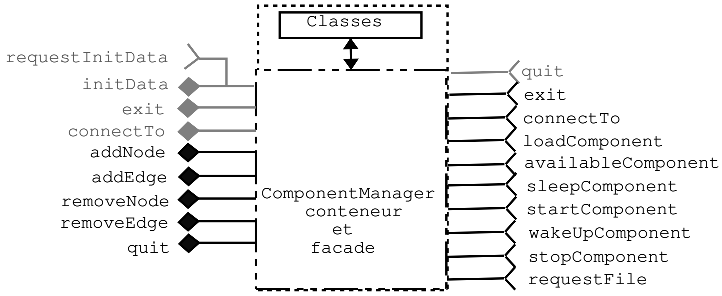


FIG. 5.16: Le gestionnaire de composants.

Le service connectTo

La Figure 5.17 décrit l'établissement de la connexion entre une vue et le document de langage tiny dont elle dépend. Cette connexion a lieu lors de l'initialisation de la vue. Les phases de cette connexion sont les suivantes :

1. la vue poste un message au gestionnaire de composants en lui indiquant son numéro d'identification, l'identification du document auquel elle veut se connecter et le message à envoyer (de type requestInitData) au document quand les connexions sont établies ;
2. le gestionnaire lit la demande de connectTo. Si le document indiqué n'a pas été chargé, il crée, tout d'abord, l'instance le représentant puis il la connecte avec lui-même . Puis, il établit des connexions entre les deux instances en appelant la méthode connect des instances. En fonction des noms, les connecteurs de sortie (*vs.* d'entrée) sont mis en relation avec les connecteurs d'entrée (*vs.* de sortie) de l'autre instance, s'ils existent. Les tables des services offerts et demandés sont mises à jour dans chaque conteneur. Après ce processus, les deux instances peuvent dialoguer directement entre elles sans passer par le gestionnaire. La Figure 5.19 montre la complexité des connexions établies entre ces trois composants. Ce schéma démontre la nécéssité d'utiliser un autre medium de communication qu'un bus logiciel ;
3. le gestionnaire poste le message requestInitData qui était encapsulé dans le message connectTo au document de la part de la vue. Ce message sert à initialiser les communications ;
4. Le document traite la requête de manière atomique et poste la réponse à la vue par le biais du connecteur initData.

5. la vue lit ce message pour finir de s'initialiser. Ce message contient l'AST utile à la création des objets graphiques de la vue, le formalisme et les comportements à rajouter à la vue. Ces comportements adaptent l'instance de la vue au document pour qu'elle puisse proposer aux utilisateurs les services spécifiques du langage traité (ici `launchVisitor` et `startEditionStruct`).

Seules les phases 1 et 2 sont relatives au processus de connexion. Les autres phases sont utiles à l'initialisation de la vue.

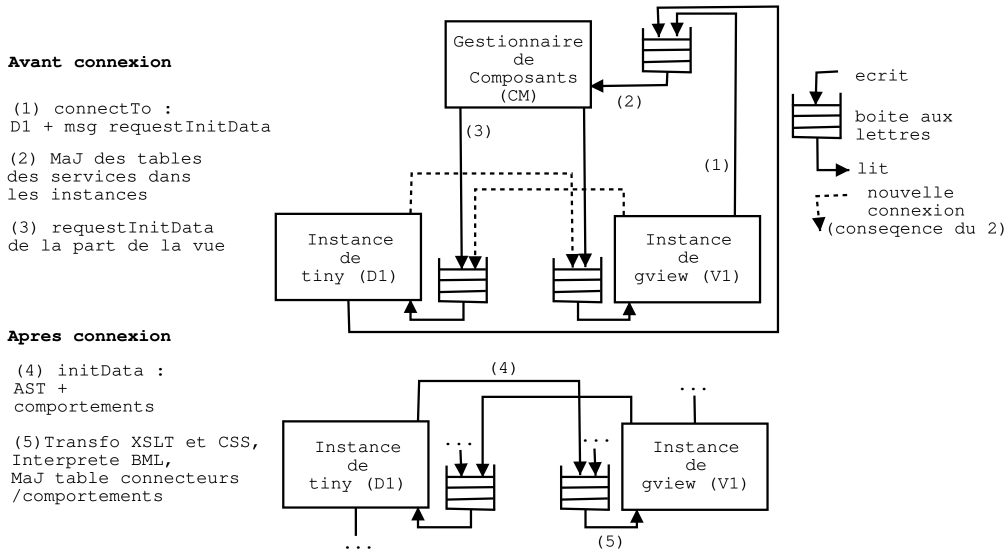


FIG. 5.17: Vue physique du processus de connexion

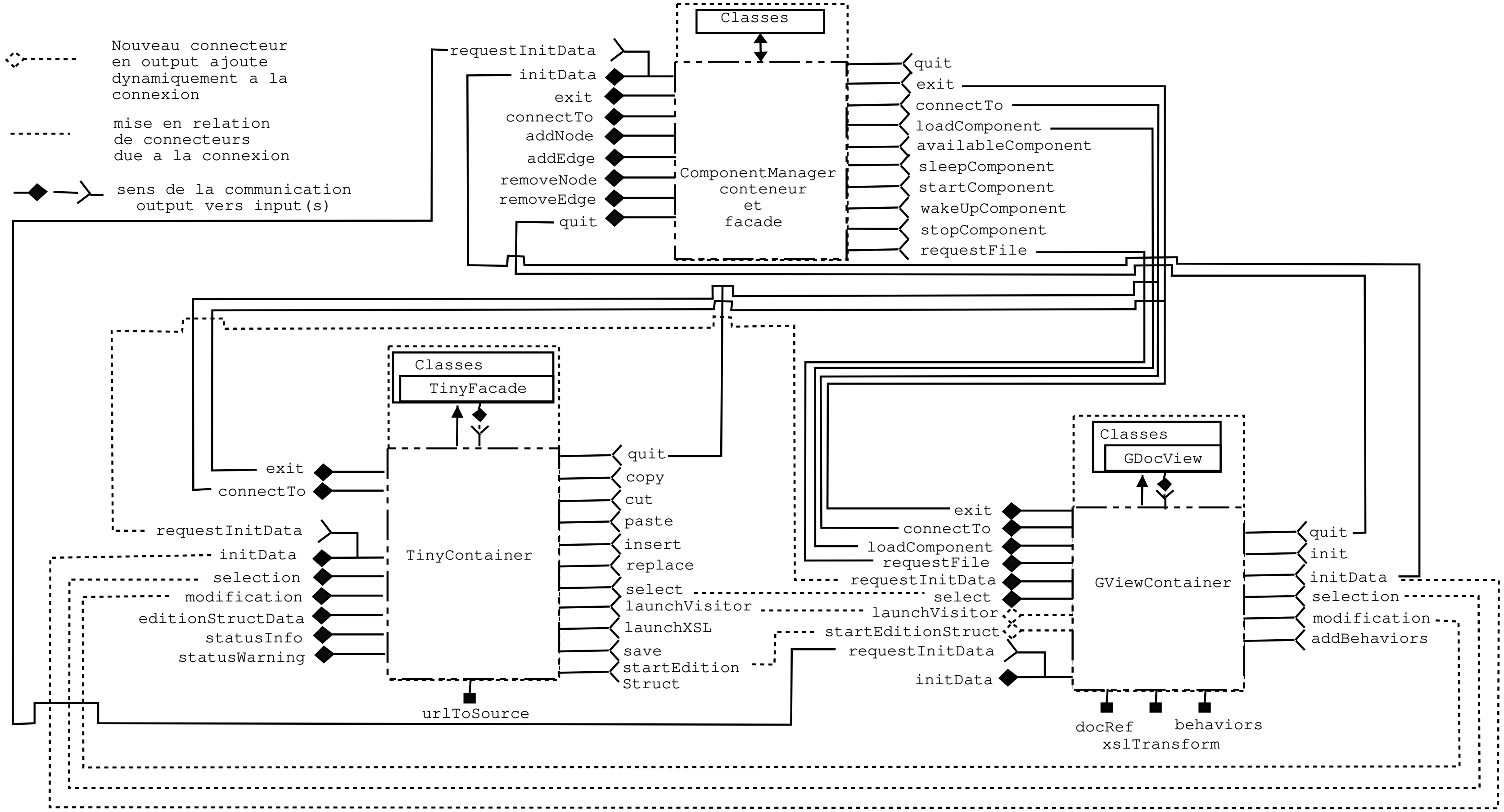


FIG. 5.18: Vue logique (ou électrique) après le processus de connexion

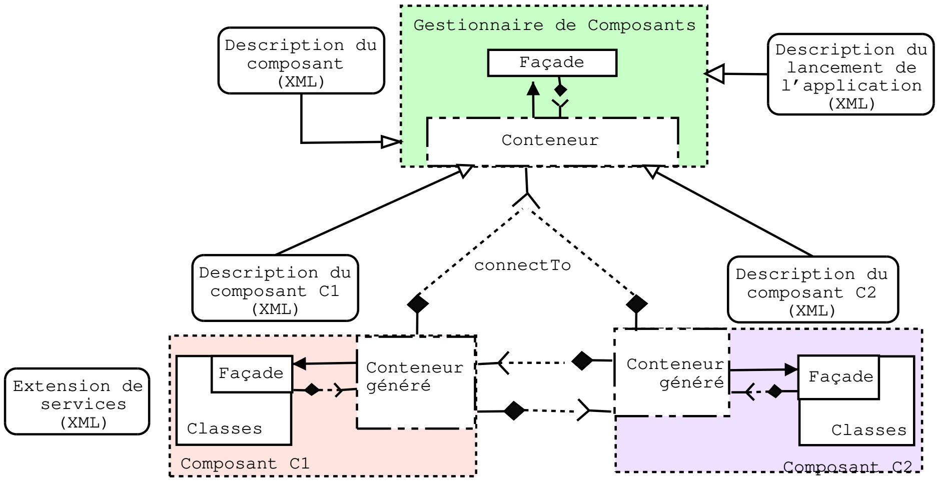


FIG. 5.19: Schéma de notre modèle de composant

Exemple de lancement d'une application

Au démarrage de notre outil, le gestionnaire de composants est chargé et créé automatiquement. Puis en fonction du fichier de lancement (Figure 5.20), il charge trois types des composants (`glayout`, `lml` et `tiny`) et établit une connexion entre lui-même et une instance du composant interface utilisateur (`glayout`) qu'il créera. Les attributs du service `connectTo` sont en fait les attributs d'initialisation cette instance. Le premier correspond au fichier de configuration *boot.lml* (Figure 5.21) qui décrit les documents et les vues à ouvrir dans cette instance (voir explication ci-dessous), le deuxième à la feuille de style à utiliser pour obtenir les objets graphiques de l'interface et le dernier aux comportements à rajouter (services et objets graphiques). Le message `initData` est envoyé par le gestionnaire pour étendre dynamiquement l'instance avec les comportements offerts par le gestionnaire ; ce message n'est lu que quand l'instance est dans un état actif. Lors du changement d'état de l'instance (de créé à active par la méthode `switchOn`), une demande de connexion entre le document `boot.lml` et elle est effectuée.

5.2.3 Interface utilisateur

L'interface utilisateur est réalisée suivant le même schéma que pour un programme : c'est une vue sur un arbre qui décrit l'ensemble des vues et des documents ouverts à un instant donné. L'état de l'interface peut ainsi être sauvegardé (*serialisable*). Les actions sur l'interface (ajout d'un nouvel onglet, fermeture d'une vue, etc) correspondent à des actions d'édition sur cet arbre. Le fichier de configuration `boot.lml` de l'interface utilisateur (voir la Figure 5.21) donne l'arbre dans un format XML. Pour chaque vue, on indique le chemin vers le document à visualiser, le type de vue, la transformation à effectuer sur le document pour obtenir les objets graphiques de cette vue et la feuille de style à appliquer sur ces objets.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<application repository="file:stlib/"  library="file:lib/">
  <load_component jar="view.jar"  name="glayout"/>
  <load_component jar="lml.jar" name="lml"/>
  <load_component jar="tiny.jar" url="file:extralib/tiny.jar" name="tiny"/>
  <connectTo id_src="ComponentManager" type_dest="glayout">
    <attribute name="docRef" value="file:resources/lml/boot.lml"/>
    <attribute name="xslTransform" value="file:resources/xsl/lml2bml.xsl"/>
    <attribute name="behaviors" value="file:resources/behaviors/bootbehav.xml"/>
    <message name="initData">
      <attribute name="inits">
        <collection>
          <item name="behavior" value="resources:cmbehaviors.xml"/>
        </collection>
      </attribute>
    </message>
  </connectTo>
</application>
```

FIG. 5.20: Exemple de descriptif de lancement

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE layout SYSTEM "file:resources/lml.dtd">
<layout>
  <frame title="Smarttools V4.Alpha" statusBar="on">
    <set title="LML Layout">
      <view title="Layout" behavior="" viewType="fr.smarttools.view.GdocView"
            docRef="file:resources/lml/boot.lml"
            styleSheet="file:resources/css/lmlstyles.css"
            transform="file:resources/xsl/genericBoxes.xsl" />
    </set>
    <set title="Tiny Workbench">
      <view title="samples/tiny/ex1.tiny" behavior=""
            viewType="fr.smarttools.view.GdocView"
            docRef="file:samples/tiny/ex1.tiny"
            styleSheet="file:resources/css/tiny-styles.css"
            transform="file:resources/xsl/genericBoxes.xsl" />
    </set>
  </frame>
</layout>
```

FIG. 5.21: Exemple d'un arbre de l'interface graphique (`boot.lml`)

La Figure 5.22 montre les types de composant chargés (les rectangles) et les instances créées (les ovales) pour cette configuration (les fichiers des Figures 5.20 et 5.21).

5.3 Avantages du modèle

5.3.1 Composants extensibles

Dans notre cadre, l'interface des composants doit être extensible dynamiquement pour l'ajout de nouveaux services, par exemple, aux composants de visualisation. En effet, ceux-ci ne connaissent pas les services spécifiques des composants documents auxquels ils sont

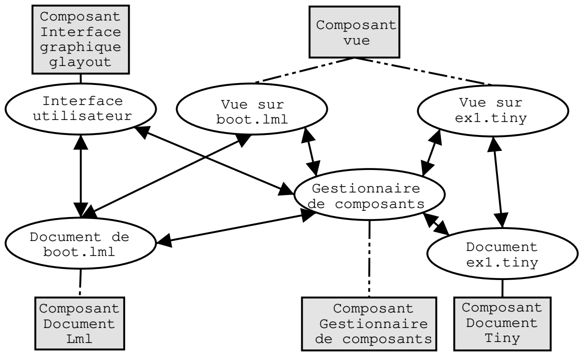


FIG. 5.22: Composants chargés et instances créées avec leurs connexions

rattachés. Ils doivent donc être enrichis dynamiquement par ces services décrits dans un fichier d'extension du composant document. Ce fichier précise les éléments graphiques (menus et barre d'outils) à ajouter aux composants de visualisation et les nouvelles connections à établir entre ces composants et le document.

Une autre caractéristique du modèle est la possibilité d'ajouter des services qui agissent à l'intérieur même de nos composants. En effet, notre plate-forme fournit une technique de programmation par aspect [23, 64] pour mieux spécifier (par *separation of concerns*) les traitements sur les arbres. Notre technique a l'avantage d'être totalement dynamique (sans transformation de programmes). L'effet de bord de cette possibilité est que les interfaces de nos composants documents doivent impérativement être extensibles. L'exemple type, que nous traitons déjà, est l'ajout d'un mode d'exécution pas-à-pas aux divers traitements, totalement réalisé à l'aide d'un aspect. Cet aspect utilise une vue graphique particulière et cela demande d'introduire dynamiquement de nouveaux services (dans les deux sens) entre le composant document et cette vue. Il nous reste encore à généraliser cette possibilité sur d'autres exemples plus complexes.

5.3.2 Exportation des composants vers d'autres technologies

Pour valider notre approche, nous avons conçu un outil de transformation de nos descriptifs de composant vers les descriptions (ou interfaces) équivalentes pour les *Web Services*, les composants CORBA et les composants EJB. La Figure 5.23 donne un aperçu[3] des fichiers générés automatiquement par cet outil pour chaque technologie.

Mais surtout, il ressort de cette expérience que l'utilisation d'une de ces technologies correspond à faire un choix de conception dont on ne peut s'abstraire par la suite. Utiliser EJB implique un modèle client/serveur, CORBA une dépendance vis-à-vis de son protocole de communication (ORB - *Object Request Broker*) et les *Web-Services* une spécialisation Internet des composants. En d'autres termes, concevoir un modèle spécifique à ses besoins évite de choisir une solution trop liée à une technologie existante. De plus, certains de nos besoins (extensions de services) sont difficilement exprimables dans ces différents modèles.

[3]Donner tous les détails techniques et particularités nuirait à la lisibilité de cette Figure.

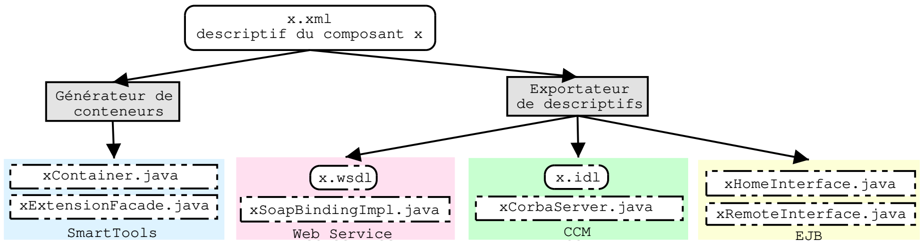


FIG. 5.23: Exemple de descriptif de composant

Par exemple, il faudrait spécialiser les composants vues pour chaque langage en fonction des services proposés par le composant document associé.

Les intérêts de définir un modèle abstrait sont les suivants :

- spécifications appropriées aux fonctionnalités des composants sans transposition des besoins et sans les contraintes liées à une technologie ;
- implantation efficace et en parfaite adéquation avec les contraintes de l'application ;
- projection plus simple vers les modèles existants.

5.4 Évaluation du modèle

Comme il existe un nombre important de modèles et de concepts, il est assez difficile de faire une comparaison précise et exhaustive de notre approche. Nous utiliserons plutôt le patron d'évaluation, issu des principaux modèles de composants, proposé par [57] qui présente un état de l'art des différents modèles et une taxonomie des rôles dans le processus de production d'applications à base de composants. Nous allons regarder l'adéquation de notre démarche avec cette taxonomie pour préciser les avantages ou inconvénients de notre modèle.

Analyse des besoins en terme d'applicatif et de composant

Notre modèle est issu d'une analyse des besoins spécifiques à notre plate-forme et est donc adapté à ce domaine d'application.

Conception des types de composant

Notre langage de description de composants permet de nous abstraire d'une technologie de composant particulière. Ce langage contient les trois aspects d'un composant : ses interfaces, ses propriétés, ainsi que la manière dont il coopère avec les autres types de composants.

Implantation des types de composant

La partie non-fonctionnelle des composants est assurée par la génération automatique des conteneurs. Une partie des contraintes techniques (noms des conteneurs, des façades et des archives générés) est décrite dans les descriptifs des composants.

Diffusion des implantations de composants

Notre générateur de conteneur prend en charge la constitution d'une archive (paquetage) pour chaque composant.

Assemblage des types de composant

Les langages de déploiement et de description de l'interface graphique permettent de spécifier et de configurer l'assemblage des composants d'une application. Par contre, il n'est pas encore possible de considérer le résultat de cet assemblage comme un composant.

Déploiement des implantations de composant et des applicatifs

Notre gestionnaire de composants joue ce rôle de déploiement de composants. Il prend en charge la création de nouvelles instances et la recherche des instances existantes. Dans notre modèle, le déploiement n'est pas encore configurable en terme de structure d'accueil (par exemple pour une version répartie sur différentes machines).

Utilisation des instances de composant et des applicatifs

La découverte des services offerts par un composant s'effectue dans notre modèle exclusivement par l'intermédiaire des descriptifs des composants (format XML). Nous utilisons une introspection sur ce format neutre, indépendant d'un langage de programmation particulier.

Bien que notre modèle soit dédié à notre outil, il possède de nombreuses bonnes propriétés d'un modèle de composants en regard de cette taxonomie. Il contient tout de même quelques particularités et restrictions.

La topologie d'une application est dynamique et configurable. Ce changement de topologie s'effectue en cours d'exécution et la prise de décision de connexion est locale à un composant. Par exemple, si un nouveau composant vue est créé, c'est lui qui va prendre en charge dynamiquement sa connection avec le composant document associé.

Les possibilités d'extension de nos composants pour la qualité de service (comme les besoins de sécurité, de persistance et de transaction) sont actuellement peu traitées dans notre modèle. Ces besoins n'étaient pas vitaux pour un premier fonctionnement de notre plate-forme. De plus, l'exportation de nos composants vers d'autres modèles nous affranchit de fournir de tels services couramment proposés par les autres modèles. En les exportant, ils sont ainsi utilisables dans des applications ou environnements plus complexes qui requièrent ces types de services. Par contre, notre outil demande des services particuliers comme par exemple, la persistance des sessions d'édition. Pour cela, le choix de représenter le contenu de l'interface utilisateur sous la forme d'un arbre a permis de réaliser facilement ce service.

Mais l'intérêt d'un modèle par composant est de laisser ces parties non-fonctionnelles à la charge du gestionnaire de conteneurs. Ainsi, pour intégrer ces types de services afin de faire évoluer notre plate-forme, il suffit de modifier ce générateur. Par exemple, la réalisation d'une version répartie devrait être facilitée par ce générateur. De même, notre outil de transformation pourrait être étendu pour d'autres modèles de composants (par exemple ceux proposés par la plate-forme ObjectWeb [13]).

Conclusion

La principale motivation de ce travail était de définir un modèle de composants en adéquation avec nos besoins et non d'en faire un modèle général. Ce modèle est un moyen déclaratif de décrire l'architecture de SMARTTOOLS, indépendamment d'une technologie de composants et d'un langage de programmation. Les technologies existantes, comme les composants CORBA ou EJB, nous ont semblé être assez éloignées de nos préoccupations ou ne répondaient pas complètement à nos attentes. Pour autant, notre modèle s'appuie fortement sur les mêmes concepts de base mais sa mise en œuvre est adaptée à notre mode de fonctionnement. Nous avons préféré définir notre propre modèle, indépendamment des technologies existantes, tout en permettant une transformation vers celles-ci. Cette démarche correspond à la nouvelle stratégie défendue par l'OMG, MDA (*Model-Driven Architecture*) [26], qui est basée sur une notion de transformation de modèles. Un des résultats importants de notre démarche est de montrer, sur un exemple particulier, l'intérêt de définir un modèle indépendant de l'implantation (PIM - *Plateform Independant Model*) et ensuite de proposer différentes transformations vers des modèles spécifiques (PSM - *Plateform Specific Model*).

Les perspectives intéressantes de cette approche sont de pouvoir étendre les composants en modifiant leurs comportements internes à l'aide de la technique de programmation par aspect. L'une des principales caractéristiques de SMARTTOOLS est d'être basée sur une approche par génération de code à partir de spécifications, non seulement pour les parties non-fonctionnelles mais aussi pour les parties fonctionnelles. Pour les composants documents, cette approche est utilisée pour offrir une technique dynamique de programmation par aspect au-dessus des visiteurs. Pour répondre aux besoins d'extensions ou de *separation of concerns* il existe essentiellement trois types d'approche [23] : par transformation de programme, par mécanisme de réflexivité [54] (MOP - *Meta-Object Protocol*) et enfin par génération de code. L'intérêt de l'approche par génération (conteneur ou visiteur) est de produire un code incluant directement les mécanismes appropriés d'extension. Cette approche, bien que non généraliste, rend ces mécanismes transparents à l'utilisateur et reste simple et efficace dans son implantation (vu nos premiers résultats).

Tous ces travaux cherchent à résoudre les problèmes rencontrés lors de la conception et réalisation d'un tel méta-outil. Par son auto-utilisation dans sa propre construction, SMARTTOOLS offre un cadre pour tester concrètement les solutions proposées. C'est avant tout une plate-forme d'expérimentation pour nos futurs travaux de recherche, en particulier sur la composition d'aspects et de services pour les composants.

Conclusion

Avant de synthétiser les points forts de SMARTTOOLS et de l'approche générative à partir de modéles, nous dressons la liste des champs possibles d'application d'un tel méta-outil, montrant ainsi son intérêt.

Perspectives d'applications

Dans cette partie, nous présentons diverses possibilités d'utilisation de SMARTTOOLS dans des domaines très variés. Pour étayer/justifier nos propos, nous indiquons aussi les contrats ou les propositions de collaboration avec des équipes extérieures. Cela montre que nos perspectives d'utilisation sont concrètes et donne une meilleure vision des atouts de l'outil. Cette présentation a un caractère très publicitaire et est organisée en six parties selon les catégories des perspectives.

Les langages de programmation ou métiers

Le champ d'application «naturel» est la conception d'environnements de développement pour des langages de programmation ou métiers. De plus, la conversion DTD (bientôt XML Schema) vers ABSYNT facilite l'introduction de nouveaux langages métiers puisque ces derniers sont de plus en plus définis à l'aide de ces langages de définition de structure. L'exemple type est l'environnement pour le langage *ant* (*makefile* écrit en Java) créé, sans réel effort, juste à partir de la DTD du langage.

Nous avons aussi conçu un environnement[1] pour Java Card, un langage de programmation des cartes à puce, ainsi que pour son *byte-code*, dans le cadre d'un contrat industriel avec Bull CP8.

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 DTDs et les XML Schemas

[1]Cet environnement n'a pas été porté en version 4.

ont été développés. D'autres environnements pour XSLT, SVG, CSS et XML générique ont aussi été créés.

L'environnement XML générique a été réalisé pour permettre de manipuler n'importe quel document XML, mais avec une granularité moins fine (arbres moins typés) qu'avec un environnement dédié. Ce langage est uniquement composé d'éléments et d'attributs[2].

Pour créer l'environnement XSLT, nous avons utilisé la même technique pour gérer les balises inconnues du langage cible (BML, XHTML, etc). En effet, les parties droites des règles de transformation comportent non seulement des balises du langage XSLT mais aussi du texte et/ou des balises dépendant du langage cible choisi. L'enchevêtrement de balises rendait impossible la création d'un environnement par composition du langage XSLT et du langage cible ; les constructeurs de ce dernier n'ont pas été conçus pour cela. Le langage XSLT est donc un langage assez particulier, rendant la vérification de validité impossible sur les balises du langage cible.

Tout cela illustre que les formalismes du W3C sont potentiellement des champs d'application pour SMARTTOOLS qui, grâce à son approche générique, rend la création d'environnements de programmation rapide. Pour preuve, SMARTTOOLS a été proposé comme démonstrateur des technologies XML, dans le cadre du projet Européen QUESTION-HOW[3] du W3C. Il montre comment une application complexe peut être construite en intégrant, de manière homogène et à différents niveaux, les technologies du W3C.

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 - 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) réalisés à l'aide de nos techniques.

Les langages de méta-modélisation

Depuis l'émergence d'outils de méta-modélisation autour d'UML (*Unified Modeling Language*), des analogies entre leurs méta-langages, tels que le MOF (*Meta-Object Facility*) et l'OCL (*Object Constraint Language*), et les langages de programmation commencent à être identifiées. Nos techniques de programmation par aspect pourraient être utilisées pour la description de la sémantique de ces modèles (*Action Semantic* d'UML).

Une proposition[4] d'ARC (*Action de Recherche Coopérative*) avait été soumise l'année passée, avec les équipes GOAL de Lille, Hector de Toulouse, Triskell de Rennes et du pôle de modélisation de Nantes pour comprendre les points de synergie de différentes familles technologiques telles que les grammaires formelles, les architectures de méta-modélisation, les langages de description de données basés sur XML, et les systèmes d'ingénierie ontologique.

[2] Toutes les autres informations (*processing*, commentaires, DTD interne, etc) ne sont pas traitées. Pour ce «langage» particulier, il n'y a pas de correspondance entre les noms des balises et les noms des nœuds.

[3] Quality Engineering Solutions via Tools, Information and Outreach for the New Highly-enriched Offerings from W3C : Evolving the Web in Europe http://www.w3.org/2001/qh/

[4] http://www-sop.inria.fr/oasis/personnel/Didier.Parigot/SAM

Une proposition de projet RNTL (*Réseau National de recherche et d'innovation en Technologies Logicielles*), de nom XLUC (*XML Languages UML Contrats*), avait aussi été soumise l'année passée, avec les équipes UNSA-CNRS-I3S, UBS/Valoria, SOFTEAM - l'une des entreprises spécialistes d'UML en France - et le crédit agricole breton afin de développer un environnement pour générer, à partir d'un modèle UML, vers plusieurs langages cibles et ayant une extension d'OCL comme langage de contrats.

Un projet RNTL plate-forme est à l'étude, en collaboration avec SOFTEAM et Thalès. La mise en commun des travaux d'*Objecteering*™, l'AGL de SOFTEAM, et de SMARTTOOLS devrait être le point de départ de futures collaborations.

Les systèmes d'ingénierie ontologique (RDF) du Web sémantique
Le thème de recherche du Web sémantique regroupe tout un ensemble de notions très variées dont le lien avec SMARTTOOLS semble être l'analogie entre les systèmes d'ingénierie ontologique [31] (RDFS - *Resource Description Framework Schema*) et les notions de syntaxe abstraite (système de type). Il nous semble raisonnable de penser que nos techniques (visiteur ou programmation par aspects) pourront être utilisées pour certaines applications de ce domaine.

SMARTTOOLS est impliqué dans l'action nationale de recherche de développement SYNTAX (qui vient de débuter), regroupant plusieurs projets INRIA et aussi des industriels dont France Télécom, EADS, Thalès, CEA, EDF et Dassault Aviation. Le but de cette action est de fournir une plate-forme d'intégration et de gestion de documents spécialisés. Dans ce cadre, nous espérons approfondir le rapprochement avec le domaine des ontologies.

Les langages métiers de SMARTTOOLS
L'outil utilise intensivement les technologies objets et XML pour :

- la description de ses langages métiers (définition d'AST avec ABSYNT, de forme concrète avec COSYNT, de profils des méthodes des visiteurs avec VIPROFILE et de l'interface avec LML) et composants,
- ses fichiers de configuration (scripts de lancement),
- la personnalisation des vues graphiques (CSS).

Cette utilisation intensive, à différents niveaux, valide nos choix et est susceptible d'engendrer de nouveaux champs d'application. Les outils de SMARTTOOLS présentent la caractéristique d'être directement utilisés pour définir ses langages métiers donc ils sont très pragmatiques.

L'intérêt de l'approche

Cette thèse illustre une nouvelle approche de développement d'applications complexes ayant comme clé de voûte la génération de code à partir de modèles. Cette approche orientée modèle peut, dans une certaine manière, être comparée[5] à l'approche MDA [26, 67, 69] de l'OMG.

[5] Nous avions adopté une approche orientée modèle pour la construction de SMARTTOOLS avant la mise en avant par l'OMG de l'approche MDA.

Nous exposons, en quelques mots, le principe de l'approche MDA afin de mieux faire comprendre le parallèle avec notre démarche, très pragmatique basée sur un principe proche et réutilisé à différents niveaux. Cette approche, focalisée sur les architectures à base de composants, traite séparément la logique métier et les contraintes techniques. La logique métier, habituellement cachée dans le code, est ainsi modélisée de manière abstraite et indépendante de toute implémentation (PIM). Puis ce modèle, par des règles de transformation, est automatiquement traduit vers la ou les plates-formes souhaitées (PSM). Les intérêts principaux d'une telle approche sont une meilleure qualité logicielle due à la mise en exergue du code intelligent et à la génération de code, une facilité de portabilité vers d'autres plates-formes et une prise en compte rapide des évolutions technologiques (par création ou modification des règles de transformation).

Lors de la construction de SMARTTOOLS, cette approche, selon notre vision, a été adoptée non seulement pour l'architecture, mais aussi pour la définition de structure des langages, pour la création d'afficheurs, d'analyseurs syntaxiques, de vues et de l'interface graphique, et pour la configuration des visiteurs. Nous montrons, ci-dessous, comment elle a été appliquée à ces différentes parties. Précisément sur chaque partie, on explicitera le modèle PIM, le modèle PSM et le type de la transformation employée (parfois de manière implicite). Le tableau de la Figure Cl.1 résume nos différents modèles et transformations. Nous noterons $\{n\}$ pour référer l'entrée n de ce tableau dans les explications ci-dessous.

Définition de structure des langages

Le langage ABSYNT (PIM), indépendant de tout langage de programmation et adapté à nos besoins (fils optionnels ou tableaux), a été créé afin d'obtenir une définition, de haut niveau, des syntaxes abstraites des langages. A partir d'une telle définition de langage, on génère des classes Java (la cible ou PSM) {1}, construites au-dessus de l'API DOM ; ces classes serviront pour représenter les nœuds de ses arbres. Ainsi les concepteurs de langages se concentrent uniquement sur cette définition, non sur l'implémentation du langage. De plus, il est facile, par simple modification du générateur, d'adopter une autre API d'arbres. La transformation du PIM en PSM est de type génération de code et le PSM correspond à un raffinement du PSM API DOM. Dans le futur, ce dernier pourrait être remplacé par d'autres PSMs.

Pour faciliter la création de langages métiers dédiés au Web, toute spécification ABSYNT (PIM) peut être traduite en DTD ou en XML Schema (PSMs)[6], par génération de code {2}. Les concepteurs de tels langages, en utilisant SMARTTOOLS, peuvent ainsi profiter d'une syntaxe de description de documents beaucoup plus concise donc plus facile à appréhender et de ses outils pour la visualisation ou les traitements sémantiques.

Afin que l'outil soit ouvert et ait d'autres champs d'applications, les passerelles inverses ont aussi été (ou vont être) établies. Cela correspond à une approche «*reverse*-MDA» car la transformation part d'une technologie (DTD ou XML Schema) pour abstraire vers notre modèle (ABSYNT) {3}. Cette abstraction ou conversion est assez complexe à spécifier car le modèle de départ est plus riche et peut donc comporter quelques pertes d'information (les contraintes de structure sont plus strictes pour les langages de programmation).

[6]Ces PSMs sont liés aux technologies XML.

	Logique métier (∼PIM)	Cible générée (∼PSM)	Règles de transformation
1] Nœuds	ABSYNT	classes Java au-dessus de DOM	Java
2] Traduction ABSYNT vers DTD ou XML Schema	ABSYNT	DTD ou XML Schema	Java
3] Conversion DTD ou XML Schema vers ABSYNT [7]	DTD ou XML Schema	ABSYNT	Java
4] Afficheur	ABSYNT + COSYNT	XSLT	Java
5] Analyseur syntaxique	ABSYNT + COSYNT	fichier ANTLR	Java
6] Analyse syntaxique[7]	programme + analyseur syntaxique	arbre de syntaxe abstraite	Java
7] Vue graphique	XSLT + arbre de syntaxe abstraite	arbre de syntaxe concrète (en BML)	XSLT
8] Interface graphique	XSLT + arbre de syntaxe abstraite (en LML)	arbre de syntaxe concrète (en BML)	XSLT
9] Personnalisation d'une vue graphique ou d'une interface graphique	arbre de syntaxe concrète (*beans*) + feuille CSS	arbre enrichi d'attributs de style	CSS
10] Visiteurs par défaut	VIPROFILE + ABSYNT	visiteurs en Java	Java
11] Glue composant SMARTTOOLS	descriptif de composant	la classe du conteneur + la classe d'extension de la façade	Java
12] Glue *Web-Service*	descriptif de composant	fichier WSDL + la classe de mise en correspondance avec SOAP	Java
13] Glue *CCM*	descriptif de composant	fichier IDL + la classe du serveur CORBA	Java
14] Glue *EJB*	descriptif de composant	les interfaces *remote* et *home*	Java

FIG. Cl.1: Les différents modèles de SMARTTOOLS et leurs transformations

Visualisation des arbres

Le langage COSYNT, indépendant de toute implémentation, sépare les différentes préoc-

[7] Approche «*reverse*-MDA» : on part d'une technologie et on tente d'abstraire le modèle.

cupations relatives à la syntaxe concrète d'un langage et à l'affichage ; la partie syntaxe concrète se compose de la BNF, de la valeur du *lookahead* et des expressions régulières du lexeur, et la partie affichage de noms de style, de règles de transformation de l'arbre de syntaxe concrète en arbre de boîte, et d'objets graphiques à utiliser selon la sortie souhaitée. Grâce à la spécification ABSYNT d'un langage et à une de ses spécifications COSYNT (l'ensemble formant le PIM), l'outil peut générer soit une spécification d'analyseur syntaxique ANTLR {5}, soit un ou plusieurs afficheurs écrits à l'aide de transformation XSLT {4} (PSMs). L'analyseur syntaxique ANTLR est obtenu en raffinant le PSM analyseur de technique LL(k). De la même manière, il serait possible d'obtenir un analyseur syntaxique JavaCC mais pas CUP (technique LALR).

L'analyse syntaxique d'un programme peut être comparée à une approche «*reverse*-MDA» puisque, à partir d'une technique d'analyse et d'un programme, on extrait l'arbre de syntaxe abstraite {6}.

Avec la feuille de style générée de l'afficheur, le processeur XSLT transforme un arbre de syntaxe abstraite (la feuille de style et l'arbre formant le PIM) en un arbre de boîtes (*beans*) ou un fichier texte (PSMs) {7,8}. L'arbre de boîtes peut ensuite être personnalisé (raffiné) par une feuille de style CSS. Les vues graphiques et l'interface graphique sont obtenues selon le même procédé. De manière conceptuelle, l'arbre de syntaxe abstraite est transformé en un arbre de syntaxe concrète puis de boîtes et enfin de boîtes enrichies d'attributs de style, alors que l'implémentation génére directement ce dernier[8] ; l'arbre de syntaxe abstraite étant le PIM et l'arbre de boîtes enrichies d'attributs de style le PSM, obtenu par raffinements successifs de l'arbre de syntaxe concrète et de l'arbre de boîtes.

Configuration des visiteurs

ABSYNT et le langage VIPROFILE forment un modèle (PIM) utile pour définir des visiteurs à signatures de méthodes et à parcours configurables, par simple génération de classes Java (PSM) {10}. Comme Java effectue une résolution de méthodes sur les types statiques des arguments (et non dynamiques comme Smalltalk) et pour éviter l'emploi de la réflexivité, il est nécessaire de générer du code supplémentaire pour effectuer l'indirection, de manière transparente pour les utilisateurs. Nous profitons de cette phase de génération de code pour enrichir les visiteurs d'une technique de branchement dynamique d'aspects sur les méthodes `visit`. Contrairement aux autres systèmes par aspects, nous n'utilisons ni la transformation de code ni la réflexivité ; la transformation de programme ne permet pas l'ajout d'aspects à l'exécution (technique statique) et pose des problèmes lors de références aux méthodes de la classe parente (cas du `super`), et la réflexivité des problèmes d'efficacité et de passage au niveau méta. Ceci est possible car nous proposons une spécialisation de la programmation par aspects, adaptées à nos besoins.

Les visiteurs écrits par les utilisateurs ont un code simple, indépendant de toute technique d'implémentation et peuvent être enrichis, à l'exécution, de code supplémentaire sans être modifiés.

[8] C'est de la déforestation, selon Didier.

Composants
Nous avons choisi de concevoir notre propre modèle abstrait de composants (PIM) répondant à nos besoins spécifiques et de l'implémenter {11}. Afin de rendre nos composants exportables, ce modèle est projetable vers des modèles spécifiques tels que EJB, CCM ou les *Web-Services* (PSMs) {12,13,14}. Avec cette caractéristique, il serait possible, par exemple, de créer des adaptateurs à nos composants pour qu'ils puissent être utilisés dans l'environnement *Eclipse* [6] (nouveau PSM).

L'innovation de SMARTTOOLS provient de la mise en commun de techniques de différents domaines, due à l'utilisation intensive de l'approche générative à partir de modèles. Tous les autres systèmes utilisent cette approche uniquement pour une facette c'est-à-dire soit pour les composants [81], les visiteurs [36, 41, 51], les aspects[49], ou les transformations XSLT [34]. De nombreux ponts entre domaines [52] ont ou vont être établis comme le schématise la Figure Cl.2 ; les ponts vers les bases de données ou vers UML restant à confirmer.

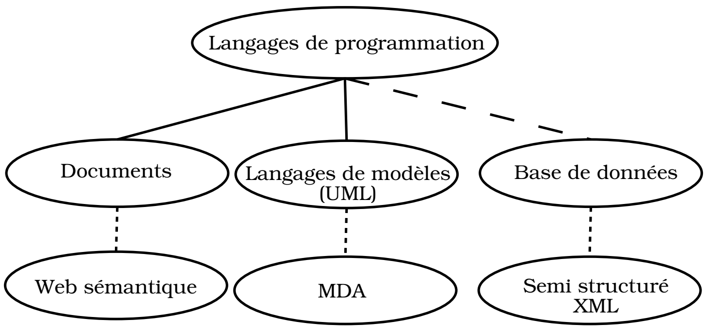


FIG. Cl.2: Relations des différents domaines

Annexe A

SmartTools mini-HowTo for designers

This tutorial is available at http://www-sop.inria.fr/oasis/SmartTools/st4up.

A.1 Overview

A.1.1 SmartTools project

SmartTools is a semantic framework generator, based on XML and object technologies. Thanks to a process of automatic generation from specifications, SmartTools makes it possible to quickly develop environments dedicated to domain-specific and programming languages. Some of these specifications (XML, DTD, Schemas, XSLT) are issued from the W3C which is an important source of varied emerging domain-specific languages. SmartTools uses object technologies such as visitor patterns and aspect-oriented programming. It provides code generation adapted to the usage of those technologies to support the development of semantic analyses. In this way, we obtain at minimal cost the design and implementation of a modular development platform which is open, interactive, uniform, and most important prone to evolution.

SmartTools is not:

- an integrated development environment (IDE). But SmartTools can be used to develop such kind of tool.
- another version of Centaur, even though the two first versions of SmartTools did inherit from some of the work done around the Centaur project.

What you can do with SMARTTOOLS :

- Design domain specific languages.
- Write semantic analysis tools for your language.
- Design a concrete syntax for an XML language and generate an associated parser.

A.1.2 SMARTTOOLS platform

The SMARTTOOLS project produced several different tools adressing different purposes. Some tools can help you define the abstract and concrete syntax of a language, others can help you write a parser for you language, or others can help you write your own analysis programs for that language for example. Those tools are independent software components, and most of them can be used directly from a java command line. The SMARTTOOLS platform connect some of those software components to each other and to a graphical user interface to propose a user friendly approach of SMARTTOOLS features. It also demonstrates how it is possible to build a component model to use SMARTTOOLS components all together and how a graphic layout abstract model can be used to build a user interface with XML based technologies.

Let's have a look at a screenshot of the SMARTTOOLS platform : Figure 1 - SMARTTOOLS platform screenshot

What we see on this screenshot is a set of three different views of one single document (an abstract syntax tree). That document is a small program written in our toy/demo language named Tiny. The top left view shows a graphic tree structure of the document. The bottom left view is a colored syntax textual output of the same document. The right view is an XML print of the same document. It is clearly visible here that SMARTTOOLS manipulates abstract syntax tree structures internally and has the capacity to build any kind of graphic or text output of such structures. Every view displays the content of the tree and its status. For example we see that one node of the tree is selected (an affect statement) and that selection status is highlighted in a different way in every view.

To obtain the above screenshot, the SMARTTOOLS platform activated a set of software components and connected them to each other. Here is a figure illustrating that principle :

FIGURE

Figure 2 - A few components in SMARTTOOLS platform

The components manager creates instances of the other components listed in a start-up configuration file and connect them to each other. After direct connections are established between two components, they exchange messages directly without the need of the components manager. The figure 2 above shows that the component manager created one instance of a document component and three different instances of view component. Each view component has been connected directly to the document.

It is important to note that we have an abstract model which is the abstract syntax tree. And from that model SMARTTOOLS can provide different concrete text or graphic representations. Every view contains a different concrete form that SMARTTOOLS created by applying a transformation to the abstract model. You can find more detailed information about how views are built in the view component overview, the CoSynt language reference, and CSS in SMARTTOOLS.

The user interface design
To be precise, the components shown in the figure 2 are only a part of what is needed to obtain the screenshot in figure 1. The three views are displayed inside a frame, with a specific layout, and also with menus. The description of the whole user interface is based on the document / view principle that we have seen in the above example: a document component contains an abstract model of the user interface's layout and a transformation is applied to that model to produce a concrete view. That concrete view is the frame that contains all the other views. The figure 3 illustrates the user interface model and view.

Figure 3 - User interface layout's document and view

We designed a small language named Layout Markup Language (LML) to describe the user interface's layout. LML files are used in two different cases:

1. At boot up, a LML file describes which views will be displayed and the user interface layout.

2. When the user opens a new file, there can be an LML description associated to the language of the file to open. That LML description contains the list of views to open every time a document of that language is created and the layout for those views.

User interface actions and menus
Every view component can receive a description of actions and menus associated to that view. That description is a XML file usually named a "behaviors" file. You can fnd a detailed description of such behaviors files in How to Customize the user interface. When using the platform, the menu that is displayed on top of the window is a combination of different menu definitions. It is made of : the menu provided by the LML document (which is considered as being the general menu) + the menu provided by the selected view's document + the menu defined for the selected view itself. It means that every time the user clicks in a view, that view becomes the selected one (if it was not already selected), and the menu bar changes.

A.1.3 Component framework

A.1.4 Document component

A.1.5 ABSYNT component

A.1.6 View component

A.1.7 Component manager

A.1.8 Graph component

A.1.9 Generated components

A SMARTTOOLS component is formely a jar file, with special data on it (in jar MetaData and cdml file), see Component Framework part for basic concept. This part explain what is the content of a generated component, if you want to know how to generate a component, see the HowTo part.

Common concept for all component

Content of the Jar file
The produced jar (for a component L, under namespace "ns") file have this organization:

```
META-INF/MANIFEST.MF
resources/L.cdml
ns/*.class
ns/*.java
doc/*.html
```

Metainfo part

See jar file documentation for common tag. We add a special 'Components' tag in the Manifest file, this tag contains a list of reference to cdml files (at least ONE reference must be present). We use this in the component manager when a new component is loaded. The Main-Class attribute tell what call use in case of command-line use.

Example (extrat from tiny component) :

```
Manifest-Version: Version 1
Class-Path: . ../stlib/st-core.jar ../lib/xerces.jar ../lib/xmlParserAPIs.jar ../lib/antlr.jar ../lib/B
Components: resources/tiny.cdml
Main-Class: tiny.Launcher
```

Resources

This directory contain the component description files (see CDML). At least one cdml file must be present, and each of them must be reference in the Manifest file.

Source and class files

Contain all user-defined and generated java class.

What is generated We generate all needed code to use this component with the SMARTTOOLS Platfforme, but also to be use stand-alone. In order to be use with as a component, we produce Facade and Container classes (See ...) and eventually Event/Listener couple (for Facade -> Container communication). We also produce a stand-alone main class and finally a Launcher (to deal with classloader). Produced classes content depends of cdml specification and in particular Attributes, Inputs/Outputs data.

Fo each attributes in the cdml file, we have:

- a new entry in the setter table in the Container class (to send attribute data to the Facade)
- a new entry in the command line attribute setting in the main method of the Facade class.

For each inputs in the cdml file, we have:

- the corresponding method in the Facade class (where the code is)
- a new entry in the call table in the Container class (a link between service name and method invocation in the Facade)
- a new entry on the command line action in the main method of the Facade class

For each O output in the cdml file we have:

- a couple OEvent/OListener java class (communication between Facade and Listener)
- a method that send a new event on each O listener in the Facade class
- an implementation of the O Listener in the Container class (the corresponding method send a new "o" message)
- a registration of the Container as Listener of the Facade class on O Event.

If user as specified that is already have a Facade class, the generated one extends the user-one in order to add missing behaviors. But this user classes must respect some createria :

- for each output define, having the corresponding method with correct signature
- for each input define, having the corresponding method with correct signature in order to implement the desired behavior.
- do not be declare as final!

Example: Look at this cdml file:

```
<component name="cmp1"
           type="cmp"
           extends="abstractContainer"
           ns="cmp1">
   <containerclass name="Cmp1Container"/>
   <facadeclass name="Cmp1AppFacade" userclassname="Cmp1App"/>

   <!--        -->
   <!-- OUTPUT -->
   <!--        -->
   <output name="updateLabel"
           method="updateLabelMthd"
           doc="update Label content">
      <arg name="message" doc="label content" javatype="java.lang.String"/>
   </output>
</component>
```

It define an ouput call "updateLabel", link to the method "updateLabelMthd" with a String call "message" as arugment. This output declaration produce some java code or classes as see on this figure :

FIGURE

What happened in case of Document?

Document component is a special case, we produce more java file. This new produced file are used to deal with a language.

Additionally produced files are:

- a DocType (java version of an absynt) for the language
- a Document (encapsulation of an AST)
- a xml-based parser
- some classes that represent the nodes
- and eventually user-defined parsers, pretty-printers, gui layouts, menu/button definitions, ...

A.2 Howtos

A.2.1 SmartTools platform

A.2.2 Run the SmartTools platform

What you need to run SmartTools

- SmartTools is developped in Java language. You will need Â at least a Java virtual machine to run SmartTools. The version 1.4 or a later version of Sun Java SDK is recommended. You must copy the corresponding version of tools.jar (located in the JDK distribution under lib/ directory) to the lib directory of the SmartTools distribution.

- The SMARTTOOLS platform is provided in a single jar file named st4.jar
- At least one startup script file (named worldfile) such as world-boot-demo.xml which is provided with the platform.
- The jar files related to the language components you wish to load in SmartTools and eventually it's dependance (jar files).
- Some extrernal tools (provided in the distribution package)
  - From Apache Software Foundation[1]: Ant, Common-Collections, Xalan, Xerces (optionaly : bcel, batik)
  - From IBM AlphaWorks[2]: bmlplayer[3]
  - From MageLang Institute: antlr[4]
  - From Koala Team[5]: koala-graphics[6]

Running SMARTTOOLS The typical way to run SmartTools is to type a command line from the St4/ current directory with the following format:

```
java -jar st4.jar -worldfile <startup script file>
```

Exemple:

```
java -jar st4.jar -worldfile file:scripts/world-boot-demo.xml
```

To avoid typing too long command lines, a few shell/batch scripts are provided like:

st4-demo.sh (for Unix or Linux systems);

st4-demo.bat (for Windows systems).

We suggest that you make a first try with the world-boot-demo.xml wordfile and then you can look in the scripts/ directory and test a few.

[1] http://www.apache.org/
[2] http://www.alphaworks.ibm.com/
[3] http://www.alphaworks.ibm.com/tech/bml
[4] http:///www.antlr.org/
[5] http://koala.ilog.fr/
[6] http://www.koalagraphics.com/

Annexe B

Exemples de visiteurs

L'ENVIRONNEMENT UTILE POUR LA VÉRIFICATION DE TYPE DU LANGAGE TINY

```
package tiny.visitors.symtab;
import java.util.HashMap;
import java.util.ArrayList;
import java.util.Stack;

public class TinyEnv {
    public static final String INT = "i";
    public static final String BOOLEAN = "b";
    public static final String ERROR = "e";
    public static final int INIT = 0;
    public static final int NONE_INIT = 1;
    public static final int UNDETERMINED = 2;
    public static final int UNKNOWN = 3;
    private HashMap variables;
    private int currentVarNum = 0;
    private int currentState = -1;
    private ArrayList statesVar;
    private Stack ifStack;
    private Stack elseStack;
    private Stack whileStack;

    public TinyEnv() {
        variables = new HashMap();
        statesVar = new ArrayList();
        ifStack = new Stack();
        elseStack = new Stack();
        whileStack = new Stack();
    }
    public void createTheFirstState() {
        int[] sts = new int[currentVarNum];
        for (int i=0; i<currentVarNum; i++)
            sts[i] = NONE_INIT;
        statesVar.add(sts);
```

```
        currentState++;
    }
    public void setVariable(String name, String type) {
        variables.put(name, new Variable(type, currentVarNum++));
    }
    public void setInitialized(String varName) {
        // create a new state with the INIT value for the given variable
        Variable var = (Variable)variables.get(varName);
        if (var != null) {
            int[] sts = new int[currentVarNum];
            int[] oldSts = (int[])statesVar.get(currentState);

            for (int i=0; i<currentVarNum; i++) {
                if (i == var.pos) // the given variable
                    sts[i] = INIT;
                else
                    sts[i] = oldSts[i];
            }
            statesVar.add(sts);
            currentState++;
        }
    }
    public void beginIf() {
        // put the current state number in the if stack and copy the state
        ifStack.push(new Integer(currentState));
        int[] sts = new int[currentVarNum];
        int[] oldSts = (int[])statesVar.get(currentState);

        for (int i=0; i<currentVarNum; i++)
            sts[i] = oldSts[i];
        statesVar.add(sts);
        currentState++;
    }
    public void beginElse() {
        // copy the state of the before if
        elseStack.push(new Integer(currentState));
        int[] sts = new int[currentVarNum];
        int[] beforeIfSts = (int[])statesVar.get(((Integer)ifStack.pop()).intValue());

        for (int i=0; i<currentVarNum; i++)
            sts[i] = beforeIfSts[i];
        statesVar.add(sts);
        currentState++;
    }
    public void endIf() {
        // compare the after then state and the after else state
        // if 2 values differ, set as UNDETERMINED
        int[] sts = new int[currentVarNum];
        int[] afterThenSts = (int[])statesVar.get(((Integer)elseStack.pop()).intValue());
        int[] afterElseSts = (int[]) statesVar.get(currentState);

        for (int i=0; i<currentVarNum; i++) {
            if (afterElseSts[i] == afterThenSts[i])
                sts[i] = afterThenSts[i];
            else
                sts[i] = UNDETERMINED;
        }
        statesVar.add(sts);
        currentState++;
    }
    public void beginWhile() {
        whileStack.push(new Integer(currentState));
```

```
    }
    public void endWhile() {
        int[] sts = new int[currentVarNum];
        int[] beforeWhileSts = (int[])statesVar.get(((Integer)whileStack.pop()).intValue());
        int[] afterLoopSts = (int[])statesVar.get(currentState);

        for (int i=0; i<currentVarNum; i++) {
            if (beforeWhileSts[i] != afterLoopSts[i])
                sts[i] = UNDETERMINED;
            else
                sts[i] = beforeWhileSts[i];
        }

        statesVar.add(sts);
        currentState++;
    }
    public int getStatusVar(String varName) {
        Variable var = (Variable)variables.get(varName);
        if (var == null)
            return UNKNOWN;
        else
            return ((int[])statesVar.get(currentState))[var.pos];
    }
    public String getType(String varName) {
        Variable var = (Variable)variables.get(varName);
        return (var==null? null:var.type);
    }
    public HashMap getVariables() {
        return variables;
    }
    private class Variable {
        String type;
        boolean isUsed;
        int value;
        int pos;

        Variable(String type, int pos) {
            this.type = type;
            isUsed = false;
            this.pos = pos;
        }
    }
}
```

Visiteur configurable pour la vérification de type

```
package tiny.visitors;

import fr.smarttools.core.tree.visitor.VisitorException;
import tiny.ast.*;
import tiny.visitors.symtab.TinyEnv;
import fr.smarttools.core.tree.UntypedNode;

public class TypeCheckerProfiledVisitor extends TraversalTypeCheckerTinyVisitor {
    public Object start(UntypedNode node, Object params) throws VisitorException {
        TinyEnv env =  new TinyEnv();
        Object o = visit(node, env);
        System.out.println("--> Variables : " + env.getVariables());
        return o;
    }
    public Object check(IntDeclNode node, TinyEnv env) throws VisitorException {
        String varName = node.getVarNameAttr();
```

```
        env.setVariable(varName, TinyEnv.INT);
        return null;
    }
    public Object check(BooleanDeclNode node, TinyEnv env) throws VisitorException {
        String varName = node.getVarNameAttr();
        env.setVariable(varName, TinyEnv.BOOLEAN);
        return null;
    }
    public Object check(AffectNode node, TinyEnv env) throws VisitorException {
        String varName = node.getVariableNode().getValue();
        String typeLeft = env.getType(varName);
        String typeRight = check(node.getValueNode(), env);
        if (typeLeft == null)
            System.err.println("This variable " + varName + " was not declared");
        else {
            if (!typeRight.equals(TinyEnv.ERROR) && (!typeLeft.equals(typeRight)))
                System.err.println("Incompatible types: " + varName + " is a " + typeLeft +" variable");
        }
        return null;
    }
    public String check(TrueNode node, TinyEnv env) throws VisitorException {
        return TinyEnv.BOOLEAN;
    }
    public String check(FalseNode node, TinyEnv env) throws VisitorException {
        return TinyEnv.BOOLEAN;
    }
    public String check(EqualNode node, TinyEnv env) throws VisitorException {
        return checkConditionalOp(node, env);
    }
    public String check(NotEqualNode node, TinyEnv env) throws VisitorException {
        return checkConditionalOp(node, env);
    }
    public String check(PlusNode node, TinyEnv env) throws VisitorException {
        return checkArithmeticOp(node, env);
    }
    public String check(MinusNode node, TinyEnv env) throws VisitorException {
        return checkArithmeticOp(node, env);
    }
    public String check(DivNode node, TinyEnv env) throws VisitorException {
        return checkArithmeticOp(node, env);
    }
    public String check(MultNode node, TinyEnv env) throws VisitorException {
        return checkArithmeticOp(node, env);
    }
    public String check(VarNode node, TinyEnv env)throws VisitorException {
        String name = node.getValue();
        String type = env.getType(name);
        if (type == null) {
            System.err.println("This variable " + name + " was not declared");
            return TinyEnv.ERROR;
        } else
            return type;
    }
    public String check(IntNode node, TinyEnv env)throws VisitorException {
        return TinyEnv.INT;
    }
    private String checkConditionalOp(ConditionOpType ph, TinyEnv env) throws VisitorException {
        String type1 = check(ph.getLeftNode(), env);
        String type2 = check(ph.getRightNode(), env);
        if (!type2.equals(TinyEnv.ERROR) && (!type2.equals(TinyEnv.INT)))
            System.err.println("The type is incorrect (int expected)");
        if (!type1.equals(TinyEnv.ERROR) && (!type1.equals(TinyEnv.INT)))
```

```
            System.err.println("The type is incorrect (int expected)");
        return TinyEnv.BOOLEAN;
    }
    private String checkArithmeticOp(ArithmeticOpType ph, TinyEnv env) throws VisitorException {
        String type1 = check(ph.getLeftNode(), env);
        String type2 = check(ph.getRightNode(), env);
        if (!type2.equals(TinyEnv.ERROR) && (!type2.equals(TinyEnv.INT)))
            System.err.println("The type is incorrect (int expected)");
        if (!type1.equals(TinyEnv.ERROR) && (!type1.equals(TinyEnv.INT)))
            System.err.println("The type is incorrect (int expected)");
        return TinyEnv.INT;
    }
}
```

VISITEUR DÉCOUPLÉ POUR LA VÉRIFICATION DE TYPE

```
package tiny.visitors;
import java.util.Stack;
import tiny.ast.*;
import tiny.visitors.symtab.TinyEnv;
import fr.smarttools.core.tree.visitor.decoupled.semantics.SemanticsImpl;

public class TypeCheckerSem extends SemanticsImpl {
    private TinyEnv env;
    private Stack typeStack;

    public TypeCheckerSem() {
        env = new TinyEnv();
        typeStack = new Stack();
    }
    public TinyEnv getEnv() {return env;}
    // IntDeclNode
    public Object beforeOp(IntDeclNode node, Object param) {
        String varName = node.getVarNameAttr();
        env.setVariable(varName, TinyEnv.INT);
        return null;
    }
    public Object afterOp(IntDeclNode node, Object param) {return null;}
    // BooleanDeclNode
    public Object beforeOp(BooleanDeclNode node, Object param) {
        String varName = node.getVarNameAttr();
        env.setVariable(varName, TinyEnv.BOOLEAN);
        return null;
    }
    public Object afterOp(BooleanDeclNode node, Object param) {return null;}
    // AffectNode
    public Object beforeOp(AffectNode node, Object param) {return null;}
    public Object betweenElem1and2(AffectNode node, Object param) {return null;}
    public Object afterOp(AffectNode node, Object param) {
        String typeRight = (String)typeStack.pop();
        String typeLeft = (String)typeStack.pop();

        if (typeLeft == null)
            System.err.println(node + ":: This variable " +
                               node.getVariableNode().getValue() + " was not declared");
        else {
            if (!typeRight.equals(TinyEnv.ERROR) && (!typeLeft.equals(typeRight)))
                System.err.println(node + ":: Incompatible types: " +
                                   node.getVariableNode().getValue() + " is a " +
                                   (typeLeft.equals(TinyEnv.INT)?"int":"bool") + " variable");
        }
        return null;
```

```
}
// NotEqualNode
public Object beforeOp(NotEqualNode node, Object param) {return null;}
public Object betweenElem1and2(NotEqualNode node, Object param) {return null;}
public Object afterOp(NotEqualNode node, Object param) {
    typeStack.push(checkConditionalOp(node));
    return null;
}
// TrueNode
public Object beforeOp(TrueNode node, Object param) {
    typeStack.push(TinyEnv.BOOLEAN);
    return null;
}
public Object afterOp(TrueNode node, Object param) {return null;}
// FalseNode
public Object beforeOp(FalseNode node, Object param) {
    typeStack.push(TinyEnv.BOOLEAN);
    return null;
}
public Object afterOp(FalseNode node, Object param) {return null;}
// VarNode
public Object beforeOp(VarNode node, Object param) {
    String name = node.getValue();
    String type = env.getType(name);
    if (type == null) {
        System.err.println(node + ":: This variable " + name + " was not declared");
        typeStack.push(TinyEnv.ERROR);
    } else
        typeStack.push(type);;
    return null;
}
public Object afterOp(VarNode node, Object param) {return null;}
// PlusNode
public Object beforeOp(PlusNode node, Object param) {return null;}
public Object betweenElem1and2(PlusNode node, Object param) {return null;}
public Object afterOp(PlusNode node, Object param) {
    typeStack.push(checkArithmeticOp(node));
    return null;
}
// MinusNode
public Object beforeOp(MinusNode node, Object param) {return null;}
public Object betweenElem1and2(MinusNode node, Object param) {return null;}
public Object afterOp(MinusNode node, Object param) {
    typeStack.push(checkArithmeticOp(node));
    return null;
}
// MultNode
public Object beforeOp(MultNode node, Object param) {return null;}
public Object betweenElem1and2(MultNode node, Object param) {return null;}
public Object afterOp(MultNode node, Object param) {
    typeStack.push(checkArithmeticOp(node));
    return null;
}
// DivNode
public Object beforeOp(DivNode node, Object param) {return null;}
public Object betweenElem1and2(DivNode node, Object param) {return null;}
public Object afterOp(DivNode node, Object param) {
    typeStack.push(checkArithmeticOp(node));
    return null;
}
// IntNode
public Object beforeOp(IntNode node, Object param) {
```

```
        typeStack.push(TinyEnv.INT);
        return null;
    }
    public Object afterOp(IntNode node, Object param) {return null;}
    // ProgramNode
    public Object beforeOp(ProgramNode node, Object param) {return null;}
    public Object beforeArray1(ProgramNode node, Object param) {return null;}
    public Object betweenArray1Elem(ProgramNode node, Object param) {return null;}
    public Object betweenElem1and2(ProgramNode node, Object param) {return null;}
    public Object afterArray1(ProgramNode node, Object param) {return null;}
    public Object afterOp(ProgramNode node, Object param) {return null;}

    ... Traitements vides aussi pour StatementNode, PrintlnNode, WhileNode, IfNode,
           EqualNode et StringNode

    private String checkConditionalOp(ConditionOpType ph) {
        String type1 = (String)typeStack.pop();
        String type2 = (String)typeStack.pop();
        if (!type2.equals(TinyEnv.ERROR) && (!type2.equals(TinyEnv.INT)))
            System.err.println(ph.getRightNode() +" :: The type is incorrect (int expected)");
        if (!type1.equals(TinyEnv.ERROR) && (!type1.equals(TinyEnv.INT)))
            System.err.println(ph.getLeftNode() + " :: The type is incorrect (int expected)");
        return TinyEnv.BOOLEAN;
    }
    private String checkArithmeticOp(ArithmeticOpType ph) {
        String type1 = (String)typeStack.pop();
        String type2 = (String)typeStack.pop();
        if (!type2.equals(TinyEnv.ERROR) && (!type2.equals(TinyEnv.INT)))
            System.err.println(ph.getRightNode() + ":: The type is incorrect (int expected)");
        if (!type1.equals(TinyEnv.ERROR) && (!type1.equals(TinyEnv.INT)))
            System.err.println(ph.getLeftNode() + ":: The type is incorrect (int expected)");
        return TinyEnv.INT;
    }
}
```

VISITEUR DÉCOUPLÉ POUR LA VÉRIFICATION DE L'INITIALISATION DES VARIABLES

Ce visiteur est composé avec le visiteur précédent pour enrichir la vérification de type.

```
package tiny.visitors;
import java.util.Stack;
import tiny.ast.*;
import tiny.visitors.symtab.*;
import fr.smarttools.core.tree.visitor.decoupled.semantics.*;

public class InitVarCheckerSem extends SemanticsImpl {
    private TinyEnv env;
    private boolean isTheAffectedVariable = false;
    private boolean isDeclarations = false;

    public InitVarCheckerSem(TinyEnv env) {
        super();
        this.env = env;
    }
    // ProgramNode
    public Object beforeOp(ProgramNode node, Object param) {return null;}
    public Object beforeArray1(ProgramNode node, Object param) {return null;}
    public Object betweenArray1Elem(ProgramNode node, Object param) {return null;}
    public Object betweenElem1and2(ProgramNode node, Object param) {
        env.createTheFirstState();
        System.err.println("ProgramNode visit1 -> cree le premier etat");
        return null;
   }
```

```
    public Object afterArray1(ProgramNode node, Object param) {
        isDeclarations = false;
        return null;
    }
    public Object afterOp(ProgramNode node, Object param) {return null;}
    // AffectNode
    public Object beforeOp(AffectNode node, Object param) {
        isTheAffectedVariable = true;
        return null;
    }
    public Object betweenElem1and2(AffectNode node, Object param) {
        isTheAffectedVariable = false;
        return null;
    }
    public Object afterOp(AffectNode node, Object param) {
        env.setInitialized(node.getVariableNode().getValue());
        return null;
    }
    // WhileNode
    public Object beforeOp(WhileNode node, Object param) {
        env.beginWhile();
        return null;
    }
    public Object betweenElem1and2(WhileNode node, Object param) {return null;}
    public Object afterOp(WhileNode node, Object param) {
        env.endWhile();
        return null;
    }
    // IfNode
    public Object beforeOp(IfNode node, Object param) {
        env.beginIf();
        return null;
    }
    public Object betweenElem1and2(IfNode node, Object param) {return null;}
    public Object betweenElem2and3(IfNode node, Object param) {
        env.beginElse();
        return null;
    }
    public Object afterOp(IfNode node, Object param) {
        env.endIf();
        return null;
    }
    // VarNode
    public Object beforeOp(VarNode node, Object param) {
        if ((!isTheAffectedVariable) && (!isDeclarations)) {
            String name = node.getValue();
            int res = env.getStatusVar(name);
            if (res == TinyEnv.NONE_INIT)
                System.err.println(node +  ":: This variable " + name +
                                   " was not initialized before its use");
            else if (res == TinyEnv.UNDETERMINED)
                System.err.println(node + ":: This variable " + name +
                                   " may have been not initialized before its use");
        }
        return null;
    }
    public Object afterOp(VarNode node, Object param) {return null;}

    ... Traitements vides pour IntDeclNode,BooleanDeclNode, StatementsNode,
        PrintlnNode,EqualNode, NotEqualNode,TrueNode, FalseNode,
        PlusNode, MinusCode, MultNode, DivNode, IntNode, StringNode
}
```

Bibliographie

[1] ANTLR - ANother Tool for Language Recognition. http://www.antlr.org.

[2] Apache Ant - The Jakarka Project (Apache). http://jakarta.apache.org/ant.

[3] Aspectj - aspect-oriented programming (aop) for java. http://aspectj.org.

[4] Corba 3.0 ccm. http://www.omg.org/.

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

[6] Eclipse. http://www.eclipse.org/.

[7] Entreprise java beans. http://java.sun.com/products/ejb.

[8] http://xml.apache.org/batik/. The Apache Batik Project.

[9] JAC - Java Aspect Component. http://jac.aopsys.com/.

[10] Java Compiler Compiler (JavaCC) - The Java Parser Generator (Sun). http://www.webgain.com/products/java\protect\T1\textunderscorecc.

[11] JJTree. http://www.webgain.com/products/java_cc/.

[12] JTB - Java Tree Builder. http://www.cs.purdue.edu/jtb/index.html.

[13] Objectweb. http://www.objectweb.org/.

[14] UML - Unified Modeling Language. http://www.uml.org.

[15] The World Wide Web Consortium. http://www.w3.org/.

[16] Richard Andersen, Mark Birbeck, Michael Kay, Steven Livingstone, Brian Loesgen, Didier Martin, Stephen Mohr, Nikola Ozu, Bruce Peat, Jonathan Pinnock, Peter Stark, and Kevin Williams. *Professional XML*. Wrox Press Ltd, August 2000. ISBN 1-861003-11-0.

[17] Isabelle Attali, Denis Caromel, Carine Courbis, Ludovic Henrio, and Henrik Nilsson. An Integrated Development Environment For Java Card. *Computer Networks, Elsevier Science Publishers*, 36(4) :291–405, July 2001. special issue on Smart Cards, ftp://ftp-sop.inria.fr/oasis/personnel/Carine.Courbis/jcn2000.pdf.

[18] Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, Joël Fillon, Didier Parigot, Claude Pasquier, and Claudio Sacerdoti Coen. SmartTools : a development environment generator based on XML technologies. In *XML Technologies and Software Engineering*, Toronto, Canada. ICSE'2001, ICSE workshop proceedings. http://www-sop.inria.fr/oasis/Didier.Parigot/publications/Parigot01b.ps.gz.

[19] Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, and Didier Parigot. SmartTools : a Generator of Interactive Environments Tools. In *Compiler Construction CC'2001*, volume 2027 of *Lecture Notes in Computer Science*, Genova, Italy, April 2001. Springer-Verlag. Tool demonstration.

[20] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS : A Tool Suite for Building GenVoca Generators. In *5th International Conference in Software Reuse*, June 1998.

[21] J.A. Bergstra and P. Klint. The discrete time ToolBus – A software coordination architecture. *Science of Computer Programming*, 31(2-3) :205–229, July 1998.

[22] Patrick Borras, Dominique Clément, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and Valérie Pascual. CENTAUR : the System. *SIGSOFT Software Eng. Notes*, 13(5) :14–24, November 1988. ftp://ftp-sophia.inria.fr/pub/centaur/papers/sde3.ps.

[23] Noury M. N. Bouraqadi-Saâdani and Thomas Ledoux. Le point sur la programmation par aspects. In *Technique et Sciences Informatiques*, volume 20, page 505 à 528. Hermès, 2001. http://www.emn.fr/dept_info/perso/ledoux/Publis/tsi01.pdf.

[24] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past : Adding Genericity to the Java Programming Language. In *Proc. OPPSLA'98*, Vancouver, Canada, October 1998. http://www.research.avayalabs.com/user/wadler/gj/Documents/gj-oopsla.pdf.

[25] Eric Bruneton and Michel Riveill. Experiments with JavaPod, a platform designed for the adaptation of non-functional properties. In *Metalevel Architectures and Separation of Crosscutting Concerns, REFLECTION 2001*, volume 2192 of *LNCS*, pages 52–72, Kyoto, Japan, sept 2001.

[26] Jean Bézivin. From Object Composition to Model Transformation with MDA. In *TOOLS USA*, Santa-Barbara, August 2001. IEEE TOOLS-39. http://www.sciences.univ-nantes.fr/info/lrsg/Recherche/mda/TOOLS.USA.pdf.

[27] Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. Schéma générique de développement par composition. In *Approches Formelles dans l'Assistance au Développement de Logiciel AFADL'98*, Poitiers - Futuroscope, 1998. ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/Correnson98b.ps.gz.

[28] Carine Courbis, Pascal Degenne, Alexandre Fau, and Didier Parigot. Un modèle de composants pour l'atelier de développement SMARTTOOLS. In *Journée systèmes à composants adaptables et extensibles*, Grenoble, France, oct 2002. ftp://ftp-sop.inria.fr/oasis/personnel/Carine.Courbis/smartcompo02.pdf.

[29] Carine Courbis, Pascal Degenne, Alexandre Fau, and Didier Parigot. L'apport des technologies XML et Objets pour un générateur d'environnements : SmartTools. *revue L'Objet, numéro spécial XML et les Objets*, 2003. à paraître, ftp://ftp-sop.inria.fr/oasis/personnel/Carine.Courbis/smartobjet02.pdf.

[30] Carine Courbis, Alexandre Fau, and Didier Parigot. Programmation par visiteurs et par aspects dynamiques. ftp://ftp-sop.inria.fr/oasis/personnel/Carine.Courbis/smartlmo02.pdf, 2001.

[31] S. Cranefield. UML and the Semantic Web. In *Proceedings of the International Semantic Web Working Symposium (SWWS)*, 2001. http://www.semanticweb.org/SWWS/program/full/paper1.pdf.

[32] Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming : Methods, Techniques, and Applications*. Addison-Wesley, June 2000. ISBN 0201309777 chapter Aspect-Oriented Decomposition and Composition http://www-ia.tu-ilmenau.de/~czarn/aop/.

[33] Simon Dobson, Paddy Nixon, Vincent Wade, Sotirios Terzis, and John Fuller. Vanilla : an open language framework. In Krzysztof Czarnecki and Ulrich W. Eisenecker, editors, *Generative and component-based software engineering (GCSE'99)*, number 1799 in LNCS, pages 91–104, Erfurt (Germany), September 1999. http://www.cs.tcd.ie/publications/tech-reports/reports.00/TCD-CS-2000-12.ps.

[34] Jérôme Euzenat and Laurent Tardif. XML transformation flow processing. In *Extreme markup languages*, pages 61–72, Montréal (Canada), 2001. http://transmorpher.inrialpes.fr/wpaper/.

[35] Rémi Forax, Etienne Duris, and Gilles Roussel. Java Multi-Method Framework. In *International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'00)*, November 2000. http://igm.univ-mlv.fr/~duris/PUBLICATIONS/forax00.ps.gz.

[36] Etienne Gagnon and Laurie J. Hendren. SableCC : An Object-Oriented Compiler Framework. In *In Proceedings of TOOLS 1998 - 26th International Conference and Exhibition*, pages 140–154, August 1998. http://www.sablecc.org/tools-98.pdf.

[37] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, Reading, MA, 1995. ISBN 0-201-63361-2-(3).

[38] Valérie Gouranton. *Dérivation d'analyseurs dynamiques et statiques à partir de spécifications opérationnelles*. PhD thesis, Université de Rennes I, IFSIC, IRISA, septembre 1997. numéro d'ordre 1845, ftp://ftp.irisa.fr/local/lande/vgthese.ps.gz.

[39] Ouafa Hachani and Daniel Bardou. Les aspects pour la réalisation de patrons de conception. In *Journée Systèmes à composants adaptables et extensibles*, Grenoble, Oct 2002. http://arcad.essi.fr/2002-10-composants/papiers/03-long-hachani.pdf.

[40] Jan Hannemann and Gregor Kiczales. Design Pattern Implementation in Java AspectJ. In *Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Seattle, USA, Nov. 2002. http://www.cs.ubc.ca/labs/spl/papers/2002/oopsla02-patterns.pdf.

[41] Görel Hedin and Eva Magnusson. JastAdd—a Java-based system for implementing front ends. In Mark van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science, LDTA'01 First Workshop on Language Descriptions, Tools and Application, ETAPS'2001*, volume 44, Genova, Italy, April 2001. Elsevier Science Publishers. http://www.cs.lth.se/home/Gorel_Hedin/LDTA/RefJava.Submitted.pdf.

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

[43] J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming*, volume 1129 of *Lect. Notes in Comp. Sci.*, pages 68–114. Springer-Verlag, 1996. http://www.cs.chalmers.se/~patrikj/poly/afp96/notes.ps.gz.

[44] Martin Jourdan, Didier Parigot, Catherine Julié, Olivier Durin, and Carole Le Bellec. Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System. In *Conf. on Programming Languages Design and Implementation*, pages 209–222, White Plains, NY, June 1990. Published as *ACM SIGPLAN Notices*, 25(6) ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/sgpln90-t.ps.gz.

[45] G. Kahn. Natural semantics. In *Proceedings of Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lect. Notes in Comp. Sci.*, Passau, Germany, 1987.

[46] Joseph Kesselman and Matthew J. Duftler. *Bean Markup Language (version 2.3) Tutorial*. IBM TJ Watson Research Center, september 1999. http://www.alphaworks.ibm.com/formula/bml.

[47] Gregor Kiczales. Aspect-Oriented Programming : A Position Paper From the Xerox PARC Aspect-Oriented Programming Project. In Max Muehlhauser, editor, *Special Issues in Object-Oriented Programming*. ?, 1996. http://www.parc.xerox.com/spl/projects/aop/position.html.

[48] Gregor Kiczales and Jim des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[49] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, NY, June 1997. http://aspectj.org/documentation/papersAndSlides/ECOOP1997-AOP.pdf.

[50] Paul Klint. A Meta-Environment for Generating Programming Environments. *ACM Transactions on Software Engineering Methodology*, 2(2) :176–201, 1993. http://www.cwi.nl/~paulk/publications/TOSEM93.pdf.

[51] Tobias Kuipers and Joost Visser. Object-Oriented Tree Traversal with JJForester. In Mark van den Brand and Didier Parigot, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001. http://www.cwi.nl/~jvisser/papers/JJForester.pdf.

[52] Ivan Kurtev, Jean Bézivin, and Mehmet Aksit. Technological Spaces : an Initial Appraisal. In *International federated conferences (DOA, ODBASE, CoopIS) Industry program*, Irvine, USA, Oct 2002. http://wwwhome.cs.utwente.nl/~kurtev/TechnologicalSpaces.doc.

[53] Karl J. Lieberherr and Doug Orleans. Preventive Program Maintenance in Demeter/Java. In *Proceedings of the 19th International Conference on Software Engineering*, pages 604–605. ACM Press, May 1997. http://www.ccs.neu.edu/research/demeter/biblio/demjava.html.

[54] Jacques Malenfant and Pierre Cointe. Aspect-Oriented Programming versus Reflection : a first draft. In *Position Statement for the OOPLSA '96 AOP meeting*, 1996. http://citeseer.nj.nec.com/malenfant96aspectoriented.html.

[55] Jacques Malenfant, Christophe Dony, and Pierre Cointe. A Semantics of Introspection in a Reflective Prototype-Based Language. *Lisp and Symbolic Computation*, 9(2/3) :153–180, May/June 1996. http://www.emn.fr/cs/object/biblio/publications/lsc96.pdf.gz.

[56] Jean Marc Jézéquel, Michel Train, and Christine Mingins. *Design Patterns and Contracts*. Addison Wesley Longman, 1999. ISBN 0-201-30959-9.

[57] R. Marvie and M.-C. Pellegrini. Modèles de composants, un état de l'art. *Numéro spécial de L'Objet*, 8(3), 2002. http://www.lifl.fr/~marvie/Research/docs/D2.pdf.

[58] Raphäel Marvie, Philippe Merle, Jean-Marc Geib, and Mathieu Vadet. OpenCCM : une plate-forme ouverte pour composants CORBA. In *Actes de la seconde Conférence Française sur les Systèmes d'Exploitation (CFSE'2)*, Paris, France, Avril 2001. http://www.lifl.fr/~marvie/Research/docs/openccm.ps.gz.

[59] Todd Millstein and Craig Chambers. Modular statically typed multimethods. In R. Guerraoui, editor, *Proceedings ECOOP'99*, LCNS 1628, pages 279–303, Lisbon, Portugal, June 1999. Springer-Verlag. ftp://ftp.cs.washington.edu/homes/chambers/ecoop99.ps.gz.

[60] Oscar Nierstrasz and Dennis Tsichritzis, editors. *Object-Oriented Software Composition*. Prentice-Hall, 1995. http://www.iam.unibe.ch/~oscar/OOSC/download.html.

[61] Jens Palsberg and C. Barry Jay. The Essence of the Visitor Pattern. In *COMPSAC'98, 22nd IEEE International Computer Software and Applications Conference*, pages 9–15, Vienna, Austria, August 1998. http://www.cs.purdue.edu/homes/palsberg/paper/compsac98.ps.gz.

[62] Jens Palsberg, Boaz Patt-Shamir, and Karl Lieberherr. A New Approach to Compiling Adaptive Programs. In Hanne Riis Nielson, editor, *European Symposium on Programming*, pages 280–295, Linkoping, Sweden, 1996. Springer Verlag. ftp://ftp.ccs.neu.edu/pub/people/lieber/compile-adapt-2.ps.

[63] Jens Palsberg, Cun Xiao, and Karl Lieberherr. Efficient Implementation of Adaptive Software. *ACM Transactions on Programming Languages and System*, 17(2) :264–292, March 1995. http://www.cs.purdue.edu/homes/palsberg/paper/toplas95-pxl.ps.gz.

[64] Didier Parigot, Carine Courbis, Pascal Degenne, Alexandre Fau, Claude Pasquier, Joël Fillon, Christophe Help, and Isabelle Attali. Aspect and XML-oriented Semantic Framework Generator : SmartTools. In *ETAPS'2002, LDTA workshop*, Grenoble, France, April 2002. Electronic Notes in Theoretical Computer Science (ENTCS). ftp://ftp-sop.inria.fr/oasis/personnel/Carine.Courbis/smartldta02.pdf.

[65] Thomas Reps and Tim Teitelbaum. The Synthesizer Generator. In *ACM SIGSOFT/SIGPLAN Symp. on Practical Software Development Environments*, pages 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.

[66] Gilles Roussel, Didier Parigot, and Martin Jourdan. Coupling Evaluators for Attribute Coupled Grammars. In Peter A. Fritzson, editor, *5th Int. Conf. on Compiler Construction (CC' 94)*, volume 786 of *Lect. Notes in Comp. Sci.*, pages 52–67, Edinburgh, April 1994. Springer-Verlag. ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/couplingevaluatorAG.ps.gz.

[67] Richard Soley and the OMG Staff Strategy Group. Model-Driven Architecture. Technical report, OMG, November 2000. ftp://ftp.omg.org/pub/docs/omg/00-11-05.pdf.

[68] Croap team. *A Centaur Tutorial*. INRIA Sophia-Antipolis, July 1994. http://www-sop.inria.fr/croap/centaur/tutorial/main/main.html.

[69] OMG Architecture Board MDA Drafting Team. Model Driven Architecture - A Technical Perspective. Technical report, OMG, July 2001. http://www.omg.org/cgi-bin/doc?ormsc/01-07-01.pdf.

[70] Laurent Thery. Ppml - aioli, dec 1998. INRIA Sophia-Antipolis, Projet CROAP, http://www-sop.inria.fr/croap/aioli.

[71] Laurent Thery, Bruno Conductier, and Yves Bertot. Figue, dec 1998. INRIA Sophia-Antipolis, Projet CROAP, http://www-sop.inria.fr/croap/figue.

[72] A. van Deursen and P. Klint. Little languages : Little maintenance ? *Journal of Software Maintenance*, 1998. http://www.cwi.nl/~arie/papers/domain.pdf.

[73] Joseph George Variamparambil. Getting SMARTTOOLS and VisualStudio .NET to talk to each other using SOAP and web-services. Technical report, INRIA, 2001. ftp://ftp-sop.inria.fr/oasis/Didier.Parigot/publications/Rapport/Variamparambil.pdf.

[74] Joseph George Variamparambil. Enabling SMARTTOOLS components with component technologies : WEB-SERVICES, CORBA and EJBs. Technical report, INRIA, July 2002. ftp://ftp-sop.inria.fr/oasis/publications/2002/josephVariamparambilStage2002.pdf.

[75] W3C. XML Schema Requirements, february 1999. W3C note http://www.w3c.org/TR/NOTE-xml-schema-req.

[76] W3C. eXtensible Markup Language (XML) 1.0 (second edition), october 2000. W3C recommendation http://www.w3c.org/TR/2000/REC-xml-20001006.

[77] W3C. XML Schema Part 0 : Primer, may 2001. W3C recommendation http://www.w3c.org/TR/xmlschema-0/.

[78] W3C. XML Schema Part 1 : Structures, may 2001. W3C recommendation http://www.w3c.org/TR/xmlschema-1/.

[79] W3C. XML Schema Part 2 : Datatypes, may 2001. W3C recommendation http://www.w3c.org/TR/xmlschema-2/.

[80] Sanjiva Weerawaeana and Matthew J. Duftler. *Bean Markup Language (version 2.3) User Guide*. IBM TJ Watson Research Center, september 1999. http://www.alphaworks.ibm.com/formula/bml.

[81] T Ziadi, B Traverson, and Jean-Marc Jézéquel. From a UML Platform Independent Component Model to Platform Specific Component Models. In *International workshop in Software Model Engineering (WiSME02) at UML2002*, Dresden (Germany), September 2002. http://www.metamodel.com/wisme-2002/papers/ziadi.pdf.

CONTRIBUTION À LA PROGRAMMATION GÉNÉRATIVE.
APPLICATION DANS LE GÉNÉRATEUR SMARTTOOLS : TECHNOLOGIES XML, PROGRAMMATION PAR ASPECTS ET COMPOSANTS.

RÉSUMÉ. Avec l'émergence d'Internet et la prolifération de nouvelles technologies, la conception et le développement d'applications complexes doivent impérativement prendre en compte les standards et les aspects de répartition, déploiement et réutilisation de code. C'est la source d'une nouvelle problématique liée à la programmation. Un changement radical des méthodologies est nécessaire pour aisément prendre en compte ces différentes facettes.
Cette thèse jette les bases d'une nouvelle manière de programmer où ces facettes ou intentions sont automatiquement intégrées aux spécifications ou modèles abstraits de l'application lors de phases de génération de code source. Cette nouvelle programmation est dit générative. Cette idée a été appliquée, à différents niveaux, lors de la réalisation d'une application, aussi bien pour la représentation de données ou d'environnements interactifs que pour les traitements sémantiques ou l'architecture. Ainsi le code final obtenu après génération s'appuie sur les technologies XML pour la représentation des données, les technologies objets et beans pour les vues et l'interface utilisateur, la programmation par aspect et le patron de conception visiteur pour les traitements sémantiques et la programmation par composants pour une architecture ouverte et une application répartie et déployable.
Les principaux gains d'une telle programmation sont une meilleure qualité du logiciel due à la séparation de la logique métier et des spécificités technologiques, une simplification du code métier à écrire, l'ajout rapide de nouvelles facettes et un portage vers d'autres plates-formes ou technologies facilité.

MOTS-CLÉS. programmation générative, programmation par aspects, composant, technologies XML, MDA (Model-Driven Architecture) de l'OMG, patron de conception, langages métiers, atelier logiciel.

CONTRIBUTION TO GENERATIVE PROGRAMMING.
APPLICATION INTO THE SMARTTOOLS GENERATOR : XML TECHNOLOGIES, ASPECT-ORIENTED PROGRAMMING AND COMPONENTS.

ABSTRACT. With the emergence of the Internet and proliferation of new technologies, the design and programming of complex applications need to take into account standards and notions of code distribution, deployment and reuse. There is a need to change the programming methodologies to take into account these different facets.
This thesis lays the foundations for a new way of programming based on generative programming that automatically integrates specific technologies and user specifications (abstract models). This idea was successfully used, at different levels - data representation, interactive environments, semantic treatments and the architecture, in the design and realisation of SMARTTOOLS, a software framework for domain-specific languages. In this way, the generated source code makes use of XML technologies for the data representation, object and bean technologies for the views and GUI, aspect-oriented programming and visitor design pattern for semantic treatments and components to obtain an open architecture and a distributed and deployable application. This idea is very close to the MDA (Model-Driven Architecture) proposal of the OMG consortium that advocates a platform-independent model that can be transformed into one or more platform-specific models.
The main results of this thesis is better software quality due to business logic and technology separation, more straightforward code, a rapid addition of new facets and a means that facilitates the portability of applications towards new technologies or platforms.

KEYWORDS. generative programming, aspect-oriented programming, component, XML technologies, MDA (Model-Driven Architecture) of the OMG, design pattern, domain-specific language, software framework.

Carine Courbis
INRIA Sophia-Antipolis Projet OASIS 2004, route des Lucioles BP93 06902 Sophia-Antipolis
Carine.Courbis@insalien.org
http://www-sop.inria.fr/oasis/personnel/Carine.Courbis/these
Décembre 2002