2M11.2974.2

Université de Montréal

Alimentation d'un dépôt de code source pour l'analyse détaillée de systèmes de taille industrielle

par

Jean-François Bédard

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de
Maître ès sciences (M.Sc.) en informatique

Avril 2002

© Jean-François Bédard, 2002

QA
76
U54
2002
V.030

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :

Alimentation d'un dépôt de code source pour
l'analyse détaillée de systèmes de taille industrielle

présenté par :

Jean-François Bédard

a été évalué par un jury composé des personnes suivantes :

Président-rapporteur :	Houari Abdelkri Sahraoui
Directeur de recherche :	Rudolf K. Keller
Membre du jury :	Peter G. Kropf

Mémoire accepté le 12 août 2002

Résumé

Parmi les activités supportant la compréhension des systèmes logiciels, l'analyse statique du code source permet d'effectuer un parallèle entre les intentions des concepteurs et leur implantation technique. À cet effet, l'examen exhaustif des arbres syntaxiques abstraits fait ressortir les relations entre les composantes d'un système tel qu'il est implanté. Les activités d'investigation sont facilitées lorsque ces composantes sont stockées dans un dépôt, accessible par les applications d'un atelier de génie logiciel.

Dans cette optique, ce mémoire présente la façon par laquelle le dépôt de l'atelier SPOOL, basé sur le métamodèle UML, a été étendu par l'intégration d'un mécanisme d'extension, le profil RCR, permettant la modélisation des éléments constitutifs des méthodes. L'ouvrage expose les approches mises en œuvre pour générer les fichiers d'échange XMI contenant les informations systémiques tirées des arbres syntaxiques abstraits; de plus, il fournit une description de l'importateur permettant le traitement des informations contenues dans les fichiers d'échange.

Mots clés : rétro-ingénierie, compréhension, réingénierie, analyse statique de code source, métamodèle UML, profil RCR, format d'échange XMI, dépôt conceptuel.

Abstract

Among the activities supporting software systems comprehension, static analysis of the source code allows a comparison between the designers' intentions and their technical manifestation. To this end, exhaustive examination of abstract syntax trees highlights the relationships between the constituents of a software system. The investigation activities are facilitated when these components are stored in a repository, accessible by the utilities that make up a software reverse engineering environment.

This thesis presents the way the SPOOL environment repository, based on the UML metamodel, was extended by integrating an extension mechanism, the RCR profile, allowing the modeling of the internals of the methods. It exposes the approaches put in place for the generation of XMI interchange files containing the systemic information recovered from the abstract syntax trees; furthermore, it supplies a description of the importer that enables the processing of the information contained in the interchange files.

Keywords: reverse engineering, system comprehension, reengineering, static analysis of source code, UML metamodel, RCR profile, XMI exchange format, design repository.

Table des matières

Résumé iii

Abstract iv

Table des matières v

Liste des tableaux ix

Liste des figures x

Liste des sigles et abréviations xi

Remerciements xii

Chapitre 1 : Introduction 1

1.1 Motivation 1

1.2 Contexte de la recherche 3

1.2.1 Projet SPOOL 4

1.2.2 Environnement SPOOL 4

1.2.3 Passerelles d'importation 6

1.3 Objectifs de la recherche 6

1.4 Contribution principale 9

1.5 Publication 9

1.6 Structure du mémoire 10

Chapitre 2 : Rappels 11

2.1 lex et yacc 11

2.2 Arbres syntaxiques abstraits (AST) 12

2.3 Graphes sémantiques abstraits (ASG) 13

2.4 UML 15

2.5 XMI 16

2.6 Profil RCR 18

2.7 Conclusion 18

Chapitre 3 : État de l'art 20

3.1 GEN++ 20

3.2 Datrix 21

3.3 CPPX 22

3.4 Columbus 23

3.5 DISCOVER 24

3.6 Passerelle Datrix-SPOOL 25

3.7 Comparaison des différents projets et produits 26

3.8 Conclusion 28

Chapitre 4 : Énoncé du problème 29

4.1 Énoncé général du problème 29

4.2 Éléments constitutifs d'un système considérés 30

4.3 Éléments constitutifs du code source exclus 31

4.3.1 Directives de précompilation 32

4.3.2 Raccourcis syntaxiques 32

4.3.3 Commentaires et mise en page du code source 32

4.4 Langages de programmation considérés 33

4.5 Attentes techniques 33

4.5.1 Rapidité des processus 33

4.5.2 Format des fichiers d'échange 34

4.5.3 Production de fichiers d'échange compacts 34

4.5.4 Lisibilité des fichiers d'échange 34

4.5.5 Séparation des fichiers d'échange en deux groupes distincts 34

4.5.6 Utilisation d'identifiants uniques 35

4.6 Ébauche de la solution 35

4.7 Conclusion 36

Chapitre 5 : Génération des fichiers XMI 38

5.1 Génération des fichiers XMI de base 38

5.2 Génération des fichiers XMI/RCR 40

5.3 Conclusion 42

Chapitre 6 : Dépôt de données SPOOL/RCR 43

6.1 Description de l'approche 43

6.2 Architecture du dépôt SPOOL/RCR 45

6.2.1 Classes fondamentales du dépôt 45

6.2.2 Classes issues de l'intégration du profil RCR 51

6.3 Stratégies de conception de l'importateur XMI/RCR 61

6.4 Conclusion 62

Chapitre 7 : Discussions 64

7.1 Ajustements au profil RCR 64

7.1.1 Dépendances 65

7.1.2 Énoncés 65

7.1.3 Expressions 67

7.1.4 Pas d'évaluation 68

7.2 Utilisation littérale des balises RCR 69

7.3 Tests de génération XMI et XMI/RCR 71

7.4 Tests d'importation dans SPOOL des fichiers XMI de base 73

7.5 Impacts de l'utilisation d'une particule d'indentation 75

7.6 Comparaison de la taille des fichiers XMI générés par deux approches différentes 77

7.7 Objectifs atteints 79

7.7.1 Contenu des fichiers XMI de base 79

7.7.2 Contenu des fichiers XMI/RCR 80

7.7.3 Performance du processus d'exportation des fichiers 81

7.7.4 Performance du processus d'importation des fichiers XMI de base 81

7.7.5 Validation des hypothèses émises 82

7.8 Leçons retenues 83

7.9 Conclusion 85

Chapitre 8 : Conclusion.. 86

8.1 Synthèse 86

8.2 Travaux futurs 91

8.2.1 Implantation du dépôt SPOOL/RCR 91

8.2.2 Implantation de l'importateur XMI/RCR 91

8.2.3 Ajustements aux scripts d'exportation 91

8.2.4 Développement de nouveaux outils d'analyse et de visualisation pour l'atelier SPOOL 92

8.2.5 Application des algorithmes de génération des fichiers XMI et XMI/RCR à d'autres processeurs frontaux 92

8.3 Réflexion finale 93

Bibliographie i-1

Liste des tableaux

Tableau I : Comparaison des différents projets et produits 27

Tableau II : Statistiques de génération des fichiers XMI de base et XMI/RCR 72

Tableau III : Statistiques d'importation des fichiers XMI de base 74

Tableau IV : Statistiques concernant l'utilisation d'une particule d'indentation
pour la génération des fichiers XMI de base et XMI/RCR 76

Tableau V : Génération des fichiers XMI : comparaison de deux approches 78

Liste des figures

Figure 1 : Aperçu de l'environnement SPOOL 5

Figure 2 : Extrait de code source et son arbre syntaxique abstrait (AST) correspondant 13

Figure 3 : Graphe sémantique abstrait (ASG) correspondant à l'AST de la figure 2 14

Figure 4 : Exemple de diagramme de classes UML (adapté de [UML]) 15

Figure 5 : Exemple de modèle UML simple et son encodage XMI 17

Figure 6 : Hiérarchie des pas d'évaluation du profil RCR (tirée de [StDenis_2001]) 19

Figure 7 : Composantes de la solution retenue 36

Figure 8 : Classes de haut niveau du dépôt SPOOL/RCR 46

Figure 9 : Classes systémiques du dépôt SPOOL/RCR 47

Figure 10 : Classes structurelles du dépôt SPOOL/RCR 48

Figure 11 : Classes de particularités du dépôt SPOOL/RCR 49

Figure 12 : Classes relationnelles du dépôt SPOOL/RCR 50

Figure 13 : Classes de blocs de code du dépôt SPOOL/RCR 51

Figure 14 : Classes d'énoncés généraux du dépôt SPOOL/RCR 53

Figure 15 : Classes d'énoncés de branchement du dépôt SPOOL/RCR 54

Figure 16 : Classes d'énoncés d'itération du dépôt SPOOL/RCR 55

Figure 17 : Classes d'énoncés de saut du dépôt SPOOL/RCR 56

Figure 18 : Classes d'expressions du dépôt SPOOL/RCR 57

Figure 19 : Classes de pas d'évaluation du dépôt SPOOL/RCR 59

Figure 20 : Classe de modélisation de variable du dépôt SPOOL/RCR 61

Liste des sigles et abréviations

ASG	Graphe sémantique abstrait *(Abstract Semantics Graph)*
AST	Arbre syntaxique abstrait *(Abstract Syntax Tree)*
CNRC	Conseil national de recherches Canada
CPPML	*C++ Markup Language*
CRSNG	Conseil de recherches en sciences naturelles et en génie du Canada
CSER	*Consortium for Software Engineering Research* (Consortium de recherche en génie logiciel)
DTD	*Document Type Definition*
GCC	*GNU Compiler Collection*
GNU	*GNU's Not Unix*
GXL	*Graph eXchange Language*
HTML	*HyperText Markup Language*
MOF	*Meta-Object Facility*
OMG	*Object Management Group*
RCR (profil)	Rétroconception, compréhension, réingénierie
RSF	*Rigi Standard Format*
SPOOL	*Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems*
TA	*Tuple-Attribute (Language)*
UML	*Unified Modeling Language*
VCG	*Visualization of Compiler Graphs*
W3C	*World Wide Web Consortium*
XMI	*XML Metadata Interchange*
XML	*eXtensible Markup Language*

Remerciements

Alors que j'étais encore un étudiant au premier cycle, j'éprouvais le désir d'élargir mes connaissances en participant à l'avancement de la science dans le domaine du développement orienté objet. C'est par le biais de navigations dans l'Internet que j'ai découvert un projet de recherche prometteur, en ligne avec mes intérêts personnels, au sein duquel je croyais être en mesure de réaliser mes ambitions. Aujourd'hui, ceci est chose faite, et je ne saurais passer sous silence le rôle qu'a joué un professeur réputé dans l'atteinte de mes objectifs. À cet effet, j'aimerais remercier chaleureusement mon directeur, monsieur Rudolf K. Keller, pour m'avoir intégré à son équipe dans le cadre du projet SPOOL et m'avoir donné tout le support requis pour conduire mes activités de recherche vers la réussite.

Le dynamisme de la recherche universitaire ne saurait exister sans l'apport financier d'organismes qui investissent dans les gens, en offrant un appui à de nombreux étudiants et chercheurs, et en encourageant les entreprises à participer à l'innovation technologique. Pour cette raison, je désire remercier les différents organismes subventionnaires qui ont contribué au financement de cette étude et du projet SPOOL dans son ensemble : le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) par l'octroi d'une bourse d'études supérieures, et le *Consortium for Software Engineering Research* (CSER — Consortium de recherche en génie logiciel), co-financé par le Conseil national de recherches Canada (CNRC), le CRSNG et plusieurs entreprises dont notre partenaire industriel, Bell Canada.

L'environnement dans lequel évolue un étudiant a un impact déterminant sur son niveau de motivation envers les études qu'il entreprend. À ce chapitre, le Département d'informatique et de recherche opérationnelle de l'Université de Montréal constitue un endroit par excellence pour les études supérieures. Dans cette optique, j'aimerais remercier les professeurs du Département et mes collègues du laboratoire GÉLO pour leur professionnalisme, leur compétence et leur détermination, et tout particulièrement mes collègues travaillant sur le projet SPOOL, pour leur contribution au maintien d'un climat favorisant le respect et la collaboration.

Finalement, la poursuite d'exigeantes études ne pourrait se faire en l'absence de quelques moments de détente, qui permettent ensuite de mieux s'attaquer aux fastidieuses activités de recherche. À cet effet, j'aimerais offrir mes cordiaux remerciements à tous les étudiants de l'Université que j'ai connus pendant la durée de mes études, en pensant aux moments de joyeuse camaraderie et d'euphorie que j'ai partagés avec eux, dans la belle ville de Montréal.

1

Introduction

Le chapitre 1, *« Introduction »*, présente le contexte au sein duquel les activités de recherche exposées dans ce mémoire furent conduites, à savoir le projet SPOOL et son environnement technologique et applicatif sous-jacent. Il précise les objectifs visés par la recherche, la contribution principale de l'auteur, et expose la structure générale du mémoire.

Ce chapitre contient les sections suivantes :

Section	*Page*
1.1 Motivation	1
1.2 Contexte de la recherche	3
1.3 Objectifs de la recherche	6
1.4 Contribution principale	9
1.5 Publication	9
1.6 Structure du mémoire	10

1.1 Motivation

La rétro-ingénierie constitue une activité fructueuse du génie logiciel, en ce sens qu'elle permet de récupérer les composantes conceptuelles d'un système à partir d'une représentation de niveau inférieur, par exemple, le code source. Dans cette optique, Chikofsky et Cross [Chikofsky_1990] définissent la rétro-ingénierie comme suit :

> *"Reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction. "*

Cependant, l'analyse statique du code source peut s'avérer une tâche fastidieuse, puisque les séquences de caractères composant à proprement parler le code sont éloignées des intentions conceptuelles initiales des ingénieurs en logiciel.

Afin d'identifier les composantes de haut niveau présentes dans le code source, les compilateurs peuvent apporter une aide précieuse. En effet, lors de la traduction du code source en code exécutable, les compilateurs construisent en mémoire une représentation intermédiaire, les arbres syntaxiques abstraits, qui recréent en quelque sorte les structures de haut niveau élaborées lors de la conception d'un système. L'exploration approfondie de ces représentations internes permet de mettre en relief les relations d'utilisation et de dépendance entre les composantes pour valider, par exemple, la qualité d'une conception logicielle.

Afin d'examiner librement les relations entre les entités conceptuelles d'un système, les informations obtenues par l'analyse statique du code source doivent être stockées de façon persistante dans un dépôt conçu à cet effet. L'élaboration d'un tel dépôt peut être facilitée par l'utilisation de standards de modélisation applicables en ce domaine, tels UML et MOF [OMG_Catalog]. Plus particulièrement, un soin spécifique doit être apporté lors de la définition des structures représentant les microcomposantes d'un système, tels les éléments constitutifs du détail des méthodes, et dans la confection des fichiers d'échange permettant l'interopérabilité avec les différents ateliers de génie logiciel développés dans les milieux académique et industriel. En cette matière, Demeyer et al. [Demeyer_1999] dressent une critique austère des capacités de UML et de MOF :

> *"UML is currently embraced as 'the' standard in object-oriented modeling languages, the recent work of OMG on the Meta Object Facility (MOF) being the most noteworthy example. We welcome these standardization efforts, yet warn against the tendency to use UML as the panacea for all exchange standards. In particular, we argue that UML is not sufficient to serve as a tool-interoperability standard for integrating round-trip engineering tools, because one is forced to rely on UML's built-in extension mechanisms to adequately model the reality in source code."*

La modélisation des microcomposantes systémiques, dans une optique d'échanges informationnels entre ateliers de génie logiciel, constitue certes un exercice ardu, mais sûrement pas insurmontable. Les mécanismes d'extension de UML peuvent être soigneusement utilisés à cet effet, comme le soulignent Schauer et al. [Schauer_2002] en guise de réponse à Demeyer et al. :

> *" We wholeheartedly agree that there is a lack of complete and precise mappings of programming languages to the UML. However, we consider this as a challenge for researchers, rather than a reason for abandoning the UML. With its Stereotype extension mechanism, the UML does provide constructs to capture the many details of source code written in different programming languages. The issue at hand is to define unambiguously how to map the various UML constructs to source code constructs and to provide tool support for the traceability in both directions. "*

Ce mémoire démontre que le métamodèle UML peut être étendu de façon structurée par un mécanisme d'extension intégré, le profil RCR [StDenis_2001], effectuant la modélisation des microcomposantes des méthodes des systèmes logiciels. L'adjonction d'informations détaillées permet de conduire des analyses systémiques minutieuses à l'aide d'outils conçus à cet effet; de plus, le partage de ces informations est rendu possible par l'extension parallèle d'un format d'échange relié à UML, en l'occurrence XMI [XMI].

1.2 Contexte de la recherche

Cette section décrit le contexte dans lequel les activités de recherche s'inscrivent : le projet SPOOL, l'environnement technologique et applicatif de l'atelier de génie logiciel[1] SPOOL, ainsi que les travaux relatifs au *parsing* de code source et à l'importation des informations recueillies par l'intermédiaire d'une passerelle.

---

[1] Tout au long de ce document, le terme « atelier de génie logiciel » sera utilisé pour caractériser l'environnement de rétro-ingénierie SPOOL, même si cette appellation s'applique généralement aux environnements axés sur le *forward engineering*.

1.2.1 Projet SPOOL

Le travail présenté dans ce mémoire a été réalisé dans le cadre du projet SPOOL *(Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems)*, une collaboration étroite entre l'équipe d'évaluation de la qualité du logiciel de Bell Canada et le groupe de recherche en génie logiciel GÉLO de l'Université de Montréal. Le projet s'intègre dans les activités du Consortium de recherche en génie logiciel *(Consortium for Software Engineering Research — CSER)*, un regroupement constitué d'intervenants industriels, académiques et gouvernementaux qui a pour mission de perfectionner les pratiques de génie logiciel au Canada, de favoriser le développement de main-d'œuvre hautement qualifiée et d'améliorer le système éducatif qui forme les futurs professionnels du logiciel.

Parmi les objectifs principaux du projet SPOOL, on retrouve l'identification et la promotion des meilleures pratiques de conception et de développement des systèmes logiciels orientés objet. Quatre axes complémentaires définissent l'étendue de la recherche : les métriques de conception orientée objet, l'analyse de l'impact des changements, la traçabilité des transformations et le génie logiciel basé sur les patrons de conception.

1.2.2 Environnement SPOOL

L'environnement de rétroconception SPOOL, présenté à la figure 1, utilise une architecture à trois niveaux afin de départager clairement les outils de visualisation et d'analyse, le schéma et les objets des modèles récupérés par rétro-ingénierie, ainsi que l'unité de stockage. Le niveau inférieur consiste en un système de gestion de bases de données orientées objet, qui assure la sauvegarde des informations conceptuelles et des modèles récupérés à partir du code source. Le niveau intermédiaire constitue le schéma du dépôt de données, comprenant la structure (les classes, les attributs et les relations entre les composantes), les comportements (les fonctionnalités de base telles la création, l'accession et la manipulation des objets) et les mécanismes complexes (tels le parcours ordonné des objets, la notification des changements et l'accumulation des dépendances). Le dépôt SPOOL est composé des deux niveaux précédemment décrits. Le niveau

supérieur comprend les outils qui implantent des fonctionnalités spécifiques à leur domaine d'application, tels les utilitaires de *parsing* et d'importation, ainsi que les outils de visualisation et d'analyse des modèles récupérés.

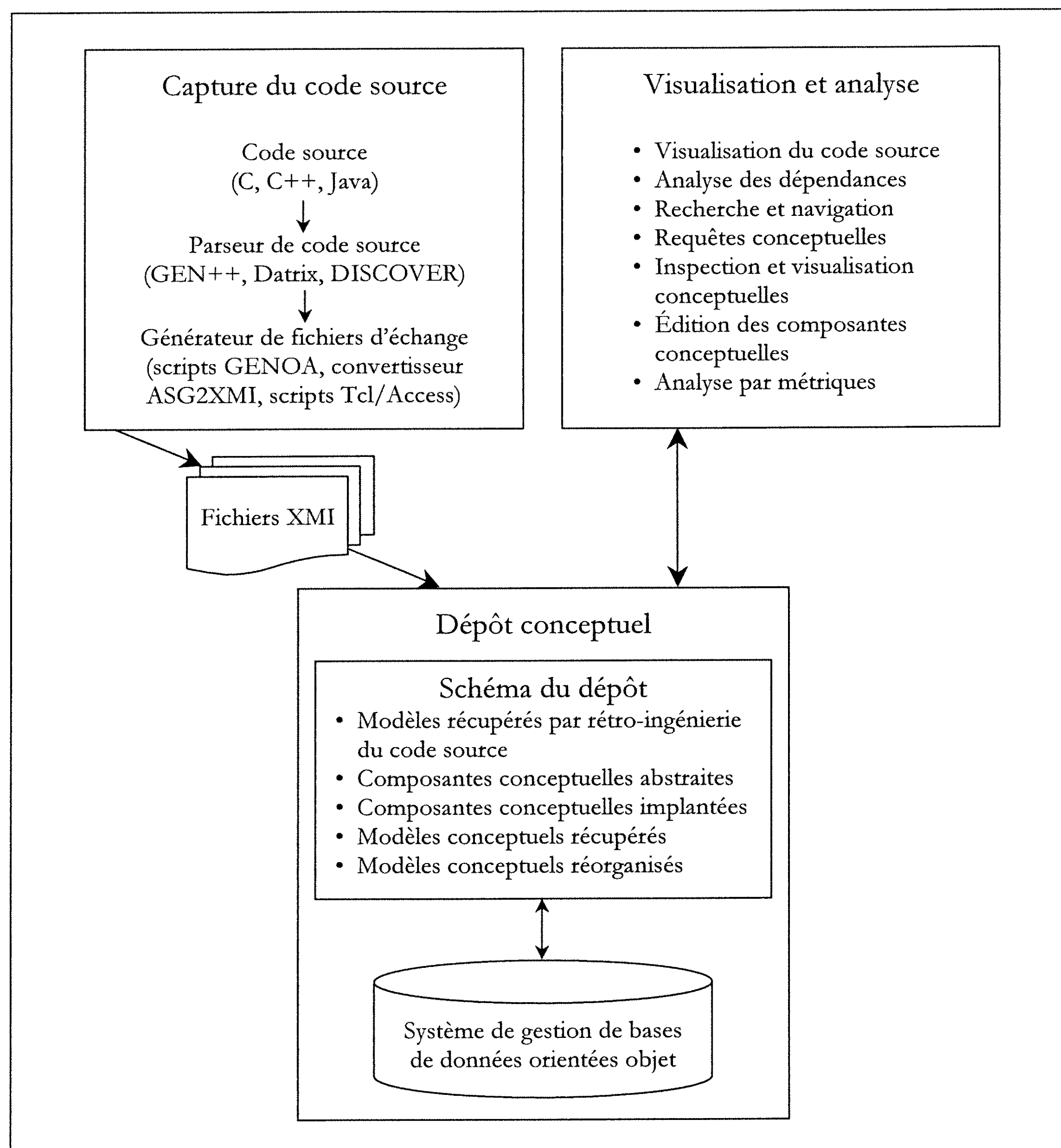

Figure 1. Aperçu de l'environnement SPOOL

Le schéma du dépôt SPOOL est composé d'une hiérarchie de classes orientée objet qui définit la structure et le comportement des objets des modèles récupérés par rétro-ingénierie du code source, les composantes conceptuelles implantées et celles identifiées par inspection du code, ainsi que les modèles conceptuels récupérés et réorganisés. Ce

schéma est basé sur une version étendue du métamodèle UML [UML]. Le dépôt est physiquement implanté à l'aide d'une base de données orientée objet POET 6.1 [POET], qui se conforme au standard ODMG 3.0 [ODMG].

Des descriptions étendues des facettes de l'environnement SPOOL et des recherches conduites à l'aide de ses outils d'analyse se retrouvent dans [Keller_2002], [Robitaille_2000], [Schauer_1999] et [Schauer_2002].

1.2.3 Passerelles d'importation

Lors du développement de l'atelier de génie logiciel SPOOL, deux passerelles ont été précédemment déployées afin d'effectuer le *parsing* du code source et d'importer dans le dépôt de l'atelier les éléments conceptuels recueillis.

La première de ces passerelles est construite à l'aide de GEN++ [Devanbu_1998], un générateur d'analyseurs de code source C++. La seconde utilise les parseurs de la suite d'outils Datrix [Datrix_2000], et comprend une application de conversion des fichiers `.asg` générés par les parseurs en fichiers XMI, qui sont traités par la suite à l'aide d'un importateur conçu à cette intention.

Les deux passerelles existantes possèdent des lacunes importantes, soit au niveau du traitement des systèmes rédigés en Java, soit au niveau de la prise en charge du détail des corps des méthodes. Une nouvelle passerelle est donc requise pour l'atelier SPOOL, afin de fournir un support adéquat pour le langage Java et de permettre l'analyse détaillée de systèmes par l'adjonction d'informations supplémentaires relatives aux micro-éléments (énoncés, expressions, etc.) constituant les corps des méthodes.

1.3 Objectifs de la recherche

La recherche présentée dans ce mémoire vise principalement à doter l'atelier de génie logiciel SPOOL d'une infrastructure idéale, lui permettant d'offrir un support approprié pour les systèmes écrits en Java, en plus de permettre le stockage, dans son dépôt de

code source, d'informations détaillées extraites des méthodes des systèmes. La capacité de traiter les systèmes rédigés en Java est devenue indispensable, compte tenu du nombre grandissant de systèmes écrits à l'aide de ce langage dans les milieux académique et industriel. L'apport d'informations détaillées extraites des méthodes, quant à lui, permet la conduite d'activités fructueuses dans les domaines de l'évaluation de la qualité du logiciel et de la réingénierie, telles le découpage orienté objet, la détection de clones et la récupération de microconceptions.

Afin d'extraire du code source les composantes de haut niveau, celles présentes dans les corps des méthodes ainsi que les relations de dépendance entre toutes ces composantes, nous formulons comme hypothèse que le parcours dirigé des arbres syntaxiques abstraits (AST) permet d'obtenir toutes les informations requises pour l'identification des composantes et relations ci-haut nommées. Les AST constituent une représentation intermédiaire générée par un processus de compilation depuis le code source, à partir de laquelle l'implantation physique d'un système peut être dérivée. Puisque le code exécutable constitue la manifestation tangible d'une conception logicielle et qu'elle est générée à partir des AST, il appert que les composantes conceptuelles sont présentes dans les AST de façon hautement structurée; par conséquent, l'examen approfondi des AST devrait permettre de récupérer les modèles conceptuels à partir de la compilation du code source.

À cet effet, un processeur frontal de qualité industrielle, DISCOVER [MKS], sera utilisé pour effectuer le *parsing* du code source. À titre de caractéristique principale, les parseurs de la suite DISCOVER traitent les systèmes rédigés en C, C++ et Java; mais surtout, ils stockent de façon persistante les AST générés lors du processus de compilation dans un dépôt interne, lequel est par la suite accessible par programmation à l'aide d'un langage de scriptage conçu à cette intention, Tcl/Access. L'utilisation d'un langage de scriptage permet la mise au point rapide des programmes qui parcourent les structures complexes que constituent les AST. Les algorithmes destinés à la récupération des composantes conceptuelles seront conçus de façon générique afin d'être applicables à d'autres processeurs frontaux qui donnent accès à des AST similaires.

L'architecture idéale de l'atelier SPOOL requiert que des fichiers d'échange soient générés par le processus d'extraction des composantes logicielles, et que le contenu de ces fichiers soit par la suite chargé dans le dépôt à l'aide d'un importateur conçu à cet effet. Cette architecture facilite grandement l'interopérabilité avec les autres ateliers de génie logiciel, pour autant que le format d'échange utilisé soit répandu et supporté par l'industrie; à ce moment, les fichiers d'échange de l'atelier SPOOL peuvent être fournis à d'autres environnements d'analyse, tandis que les fichiers provenant d'autres ateliers peuvent être traités par l'importateur de SPOOL. Dans cette optique, l'emploi du format d'échange XMI s'avère hautement bénéfique, étant donné les nombreux efforts consacrés à sa standardisation et sa diffusion. De plus, le dépôt conceptuel de SPOOL est basé sur UML, duquel XMI dérive; ce lien de descendance directe facilite l'établissement des correspondances entre les structures conceptuelles présentes dans le code source, celles stockées dans le dépôt et leur manifestation commune encodée dans les fichiers d'échange.

Cependant, le traitement des informations détaillées tirées des méthodes des systèmes nécessite des modifications importantes au dépôt de SPOOL, en plus de soulever des défis de taille en ce qui concerne le format d'échange utilisé. Nous formulons comme hypothèse qu'un mécanisme d'extension du métamodèle UML, en l'occurrence le profil RCR, pourra être mis à profit pour l'adjonction de nouvelles classes de stockage au sein du dépôt de SPOOL et, parallèlement, pour la définition de structures d'encodage additionnelles intégrées au format d'échange XMI. Bien que le profil RCR ait été conçu expressément à cet effet, son adéquation sera mise à l'épreuve lors de sa mise en œuvre technique, au cours du processus d'identification des microcomposantes conceptuelles présentes dans les AST; le profil devra alors fournir des constructions adéquates pour modéliser ces microcomposantes afin d'en dériver leur encodage au sein des fichiers d'échange et, de façon ultime, leur représentation dans le dépôt de l'atelier SPOOL.

1.4 Contribution principale

La contribution principale de l'auteur dans le cadre du projet de recherche s'articule autour des trois volets suivants :

- la conception d'algorithmes génériques effectuant l'extraction de l'information détaillée d'un système à partir des arbres syntaxiques abstraits, générés par la conduite d'un processus de compilation et stockés dans un dépôt intermédiaire accessible par programmation, celui de l'atelier de génie logiciel DISCOVER;

- l'exportation de l'information conceptuelle recueillie sous la forme de fichiers d'échange utilisant un format largement répandu et supporté par l'industrie, XMI;

- la modélisation d'un dépôt de code source étendu pour le stockage de l'information systémique reliée au détail des corps des méthodes; à cette fin, l'intégration et la validation d'un ensemble de mécanismes d'extension, le profil RCR, à un métamodèle abondamment utilisé par l'industrie, UML.

1.5 Publication

Dans le cadre des activités de recherche du projet SPOOL, l'auteur a participé à la rédaction d'une publication [Keller_2001] portant sur le dépôt de code source de l'atelier SPOOL. L'article présente le dépôt en tant qu'implantation dérivée du métamodèle UML; il décrit ses mécanismes internes et ses outils de visualisation qui sont utilisés pour observer les dépendances entre les composantes des systèmes étudiés; enfin, il expose les expériences conduites avec des systèmes de taille importante et formule les travaux futurs envisagés, reliés à la technologie des dépôts conceptuels, notamment en ce qui concerne l'adjonction de données supplémentaires modélisant le comportement dynamique des systèmes.

1.6 Structure du mémoire

Ce mémoire est structuré en chapitres, de la façon suivante :

Le chapitre 2, *« Rappels »*, expose les concepts et les technologies dont la connaissance est requise pour la compréhension de l'ouvrage.

Le chapitre 3, *« État de l'art »*, présente certains projets de recherche et produits commerciaux qui s'apparentent à l'étude courante par les buts visés et les technologies utilisées, en plus d'effectuer une comparaison entre ces projets et produits.

Le chapitre 4, *« Énoncé du problème »*, décrit la problématique constituant le sujet principal de ce mémoire : l'alimentation d'un dépôt de code source permettant l'analyse détaillée de systèmes logiciels de taille industrielle, et présente une ébauche de la solution retenue.

Le chapitre 5, *« Génération des fichiers XMI »*, expose l'approche retenue pour la génération des fichiers XMI de base, qui contiennent les composantes principales d'un système, et la confection des fichiers XMI/RCR, qui contiennent essentiellement le détail des corps des fonctions d'un système.

Le chapitre 6, *« Dépôt de données SPOOL/RCR »*, présente l'architecture du dépôt SPOOL/RCR, version étendue du dépôt actuel de l'atelier SPOOL, en plus de proposer des stratégies de conception pour l'importateur XMI/RCR, pièce maîtresse pour la conduite du processus d'alimentation du dépôt.

Le chapitre 7, *« Discussions »*, expose les résultats des expérimentations effectuées avec les scripts d'exportation XMI de base et XMI/RCR, en termes de génération de fichiers d'échange et d'importation de données dans le dépôt de l'atelier SPOOL. Les objectifs atteints et les leçons retenues du travail d'implantation technique sont aussi présentés.

Le chapitre 8, *« Conclusion »*, effectue la synthèse de l'ouvrage, propose des travaux futurs reliés à l'étude courante et offre une réflexion finale.

2

Rappels

Le chapitre 2, *« Rappels »*, examine les concepts et technologies reliés de près à l'étude courante. Une connaissance approfondie de ces éléments est souhaitable pour la compréhension de cet ouvrage.

Ce chapitre contient les sections suivantes :

Section	***Page***
2.1 lex et yacc	11
2.2 Arbres syntaxiques abstraits (AST)	12
2.3 Graphes sémantiques abstraits (ASG)	13
2.4 UML	15
2.5 XMI	16
2.6 Profil RCR	18
2.7 Conclusion	18

2.1 lex et yacc

lex et yacc sont des utilitaires Unix principalement utilisés pour construire des compilateurs, même s'ils peuvent être employés dans d'autres applications ayant aussi pour but de transformer des données structurées. Les applications de cette nature font face à deux types de tâches répétitives : diviser les données d'entrée en composantes significatives, puis découvrir les relations entre ces composantes.

L'analyse lexicale constitue la tâche de division des données d'entrée en unités significatives, appelées jetons. La conduite de cette tâche est facilitée par l'utilitaire lex

qui génère, à partir d'une spécification décrivant les différents jetons à reconnaître, un programme C effectuant l'analyse lexicale.

Une fois que les données d'entrée sont décomposées en jetons, une autre application doit souvent établir les relations entre ces particules. Par exemple, un compilateur C++ doit déterminer si une séquence de jetons représente une définition de classe, une expression arithmétique ou un appel à une fonction. Cette étape s'appelle le *parsing*; la liste des règles qui définissent les relations valides entre les jetons constitue une grammaire. L'utilitaire yacc prend en entrée une telle grammaire et produit un programme C qui effectue le *parsing* correspondant. Le parseur obtient les jetons nécessaires à son travail par le biais d'un lexeur; ce dernier peut être généré avec lex.

Dans le cas précis d'un compilateur, la grammaire yacc contient des règles qui construisent en mémoire un arbre syntaxique abstrait (AST) en fonction des structures rencontrées. L'AST, une fois complété, est parcouru par un autre module qui produit le code objet correspondant.

L'ouvrage [Levine_1995] se consacre entièrement à la description des utilitaires lex et yacc.

2.2 Arbres syntaxiques abstraits (AST)

Les arbres syntaxiques abstraits (*Abstract Syntax Trees*, AST) sont des graphes dirigés qui servent à encoder une représentation intermédiaire entre le code source et le code objet lors du processus de compilation. Chaque nœud d'un AST représente un élément structurel du langage du code source sous-jacent; les nœuds non terminaux contiennent des opérateurs, tandis que les nœuds terminaux renferment des littéraux et des noms de variables.

Les AST contiennent toute l'information pertinente tirée du code source des systèmes, à l'exception des commentaires, des macrocommandes qui y sont étendues et des directives de précompilation. Ils peuvent donc être stockés dans un dépôt spécialisé

pour la conduite d'analyses statiques de code source, telles le découpage, l'étude du flot de contrôle et l'analyse des dépendances.

La figure 2 présente un extrait de code source sous la forme d'un AST.

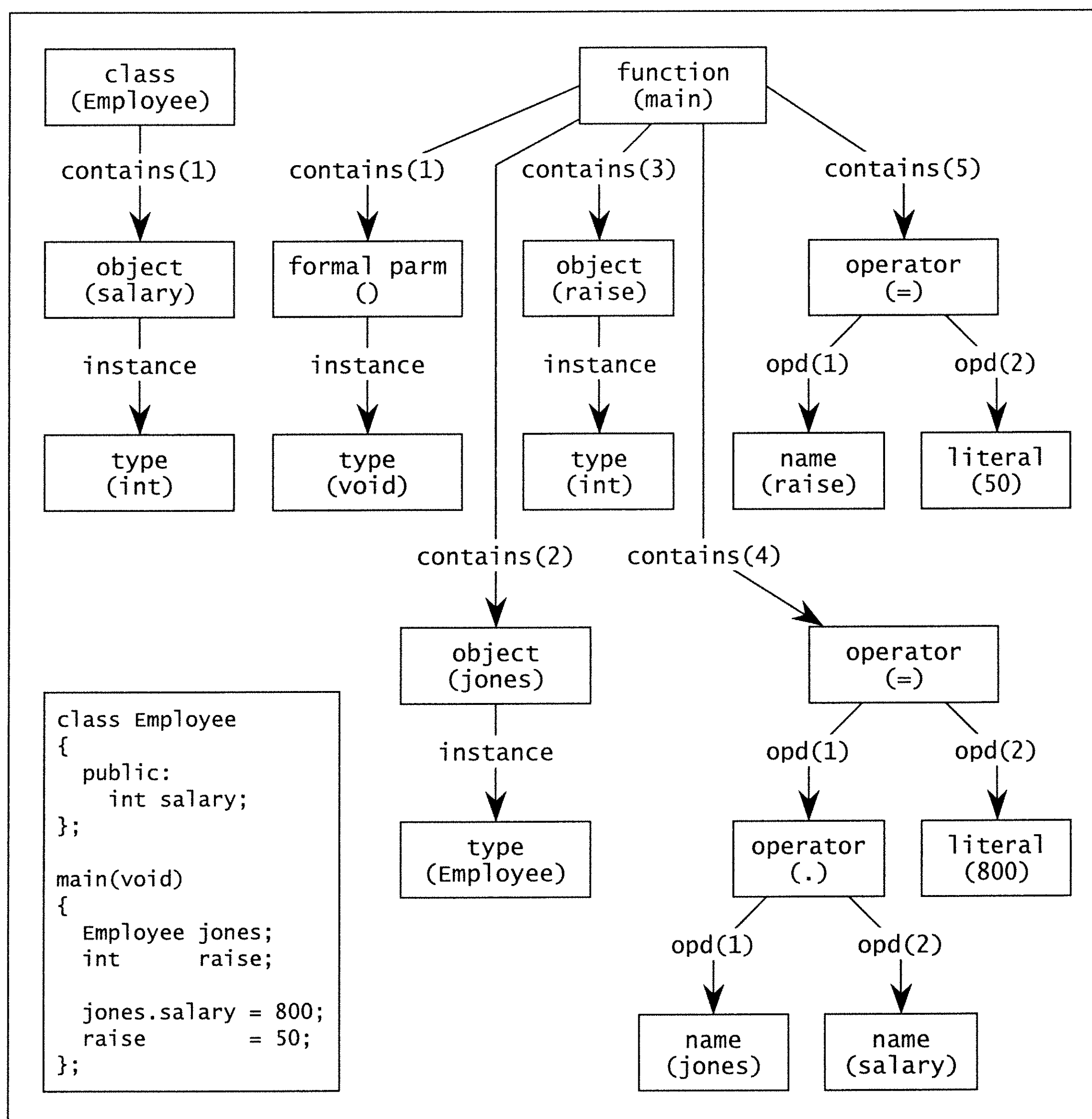

Figure 2. Extrait de code source et son arbre syntaxique abstrait (AST) correspondant

2.3 Graphes sémantiques abstraits (ASG)

Les graphes sémantiques abstraits (*Abstract Semantics Graphs*, ASG) constituent essentiellement des AST auxquels des informations sémantiques ont été ajoutées.

Par exemple, dans un AST, une référence à une entité est modélisée par un arc pointant vers un nœud terminal qui contient le nom de l'entité. Dans un ASG, ce type de référence est représenté par un arc dirigé vers la racine du sous-arbre associé à la déclaration de l'entité référée. Un ASG capture donc l'information sémantique en connectant, à l'aide d'arcs dirigés, les utilisations des entités et leurs déclarations respectives.

Un langage de spécification des ASG, Reprise, est décrit dans [Rosenblum_1991].

La figure 3 présente le même extrait de code source que la figure 2, mais sous la forme d'un ASG.

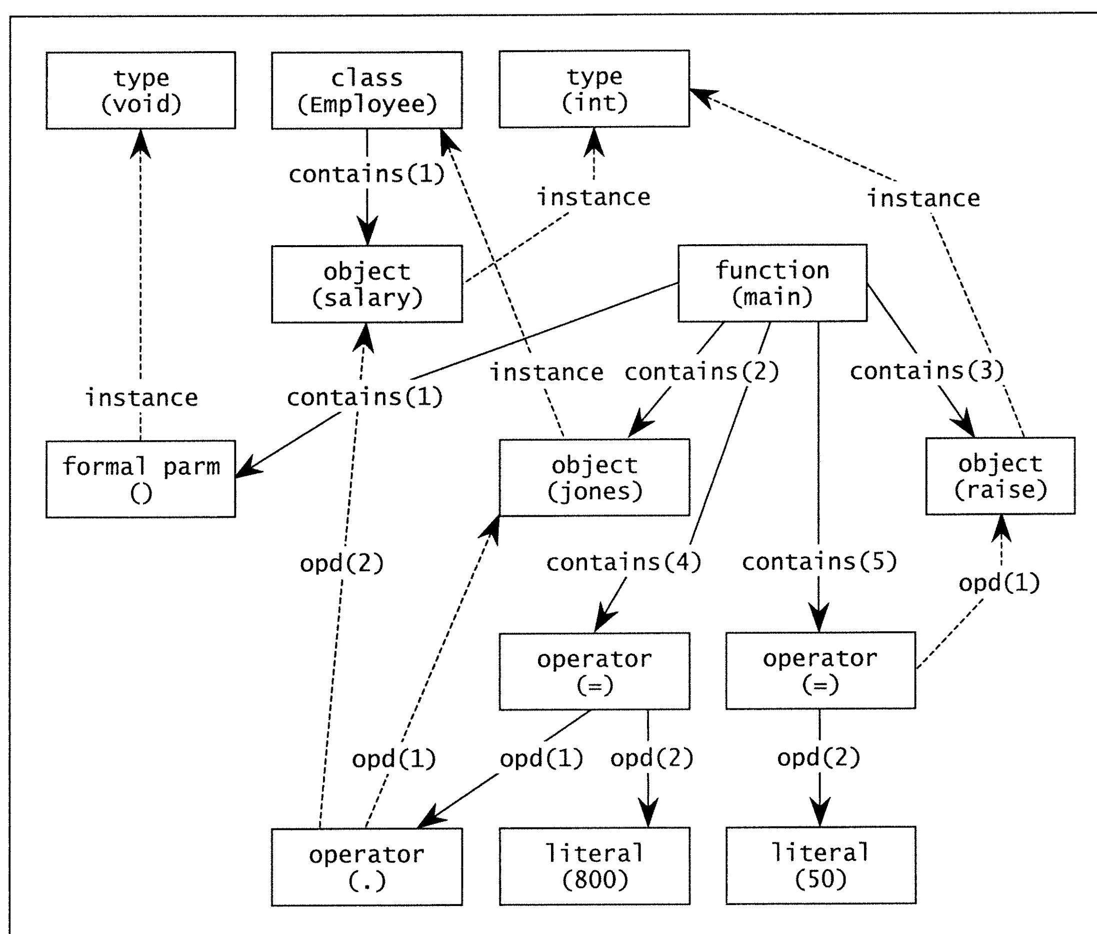

Figure 3. Graphe sémantique abstrait (ASG) correspondant à l'AST de la figure 2

2.4 UML

UML *(Unified Modeling Language)* est un langage graphique pour la visualisation, la construction, la spécification et la documentation des composantes des systèmes logiciels. UML offre des méthodes standard pour élaborer des modèles, tels les processus d'affaires, les fonctions d'un système, les schémas de bases de données et les composantes logicielles réutilisables. En bref, UML représente une collection des meilleures pratiques qui ont fait leurs preuves pour concevoir des systèmes complexes et de grande taille, particulièrement dans le domaine de l'orienté objet.

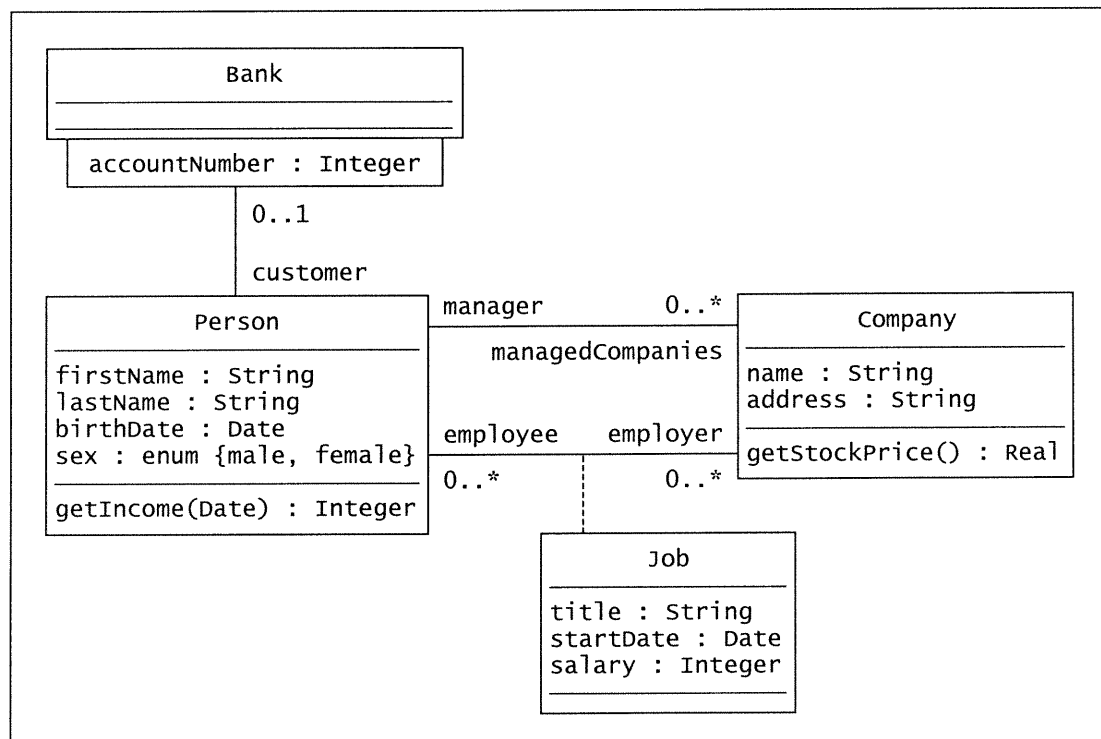

Figure 4. Exemple de diagramme de classes UML (adapté de [UML])

UML offre différents types de diagrammes qui servent à composer un ensemble de perspectives d'un système. Les diagrammes sont regroupés en quatre familles distinctes : les diagrammes de cas d'utilisation *(use case diagrams)*; les diagrammes de classes *(class diagrams)*; les diagrammes comportementaux *(behavior diagrams)* comprenant les diagrammes d'état *(statechart diagrams)*, d'activité *(activity diagrams)*, de séquence *(sequence diagrams)* et de collaboration *(collaboration diagrams)*; et les diagrammes d'implantation

(implementation diagrams) comprenant les diagrammes de composantes *(component diagrams)* et de déploiement *(deployment diagrams).* Ces diagrammes facilitent la communication des idées entre les différents intervenants lors de l'analyse et du développement d'un système et contribuent à le doter d'une architecture solide.

Un exemple de diagramme de classes UML est présenté à la figure 4.

La spécification complète de UML peut être téléchargée à partir du site Web de l'Object Management Group [OMG_Catalog].

2.5 XMI

XMI *(XML Metadata Interchange)* est un format d'échange intégrant trois standards industriels : XML, UML et MOF. Il vise principalement à faciliter l'échange de métamodèles entre les outils de modélisation basés sur UML et les dépôts conceptuels basés sur MOF.

La publication des spécifications UML et MOF en 1997 a constitué une étape majeure vers l'émergence d'un consensus impliquant les technologies de modélisation de systèmes et les dépôts conceptuels. L'avènement de XMI vise à réduire la panoplie de formats d'échange propriétaires existants en s'appuyant sur l'adoption répandue des standards du World Wide Web Consortium [W3C] (XML) et de l'Object Management Group [OMG] (UML et MOF).

La spécification XMI, telle que publiée par l'Object Management Group, expose la méthode d'encodage des métamodèles dans un document XML qui respecte une définition de document (*Document Type Definition*, DTD) donnée. Dans un premier temps, la spécification établit les règles de production pour transformer un métamodèle basé sur MOF en son DTD associé; par la suite, elle précise les règles d'encodage d'une instance de modèle en son document XMI correspondant. Des DTD concrets pour UML et MOF sont fournis en annexe de cette spécification.

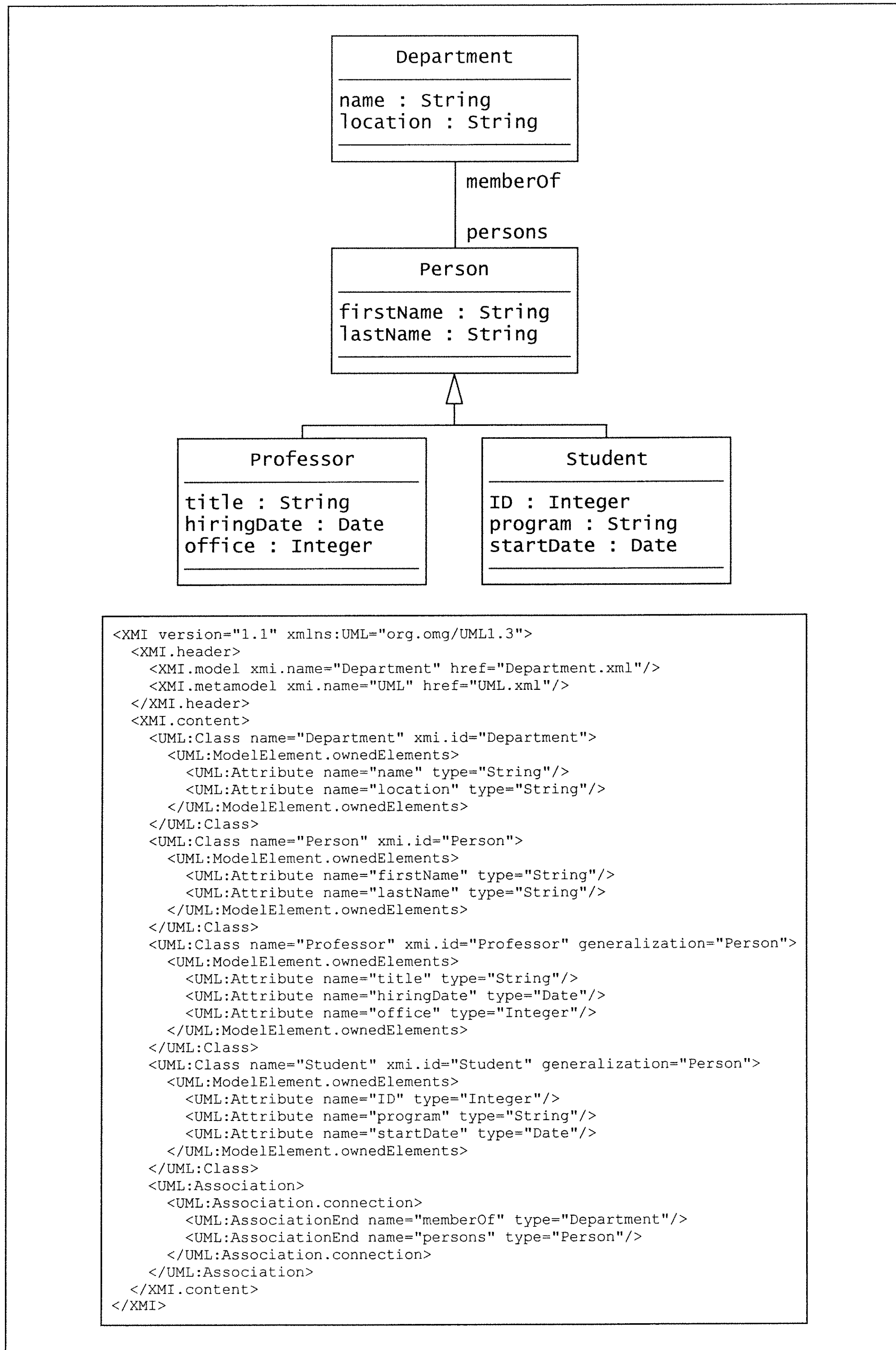

Figure 5. Exemple de modèle UML simple et son encodage XMI

La figure 5 présente un modèle UML simple et son encodage dans le format d'échange XMI.

Les spécifications complètes de XMI et MOF sont disponibles par l'entremise du site Web de l'Object Management Group [OMG_Catalog]; celle de XML est présente sur le site du World Wide Web Consortium [XML].

2.6 Profil RCR

Le profil RCR (rétroconception, compréhension, réingénierie) offre une solution compacte et efficace, en relation avec UML, pour la modélisation détaillée des corps des méthodes d'un système. Il permet de combler une lacune d'UML, en ce sens que ce dernier ne fournit pas de métaéléments standard pour la modélisation détaillée des corps des méthodes.

Le profil RCR propose des extensions au métamodèle UML sous la forme de métaéléments stéréotypés. Ces nouveaux métaéléments sont regroupés en quatre familles : les blocs, les énoncés, les expressions et les pas d'évaluation. À la tête de chaque famille se trouve un métaélément UML duquel dérive l'ensemble des métaéléments RCR qui le stéréotypent. À titre d'exemple, la figure 6 présente la hiérarchie des métaéléments RCR modélisant les pas d'évaluation.

Une description exhaustive du profil RCR ainsi que certaines discussions concernant sa mise en œuvre et son positionnement par rapport à d'autres métamodèles sont exposées dans [StDenis_2001].

2.7 Conclusion

Ce chapitre a décrit quelques concepts et technologies sur lesquels cette étude est basée. Le chapitre suivant aborde l'état de l'art en ce domaine, en présentant des produits commerciaux et projets de recherche apparentés.

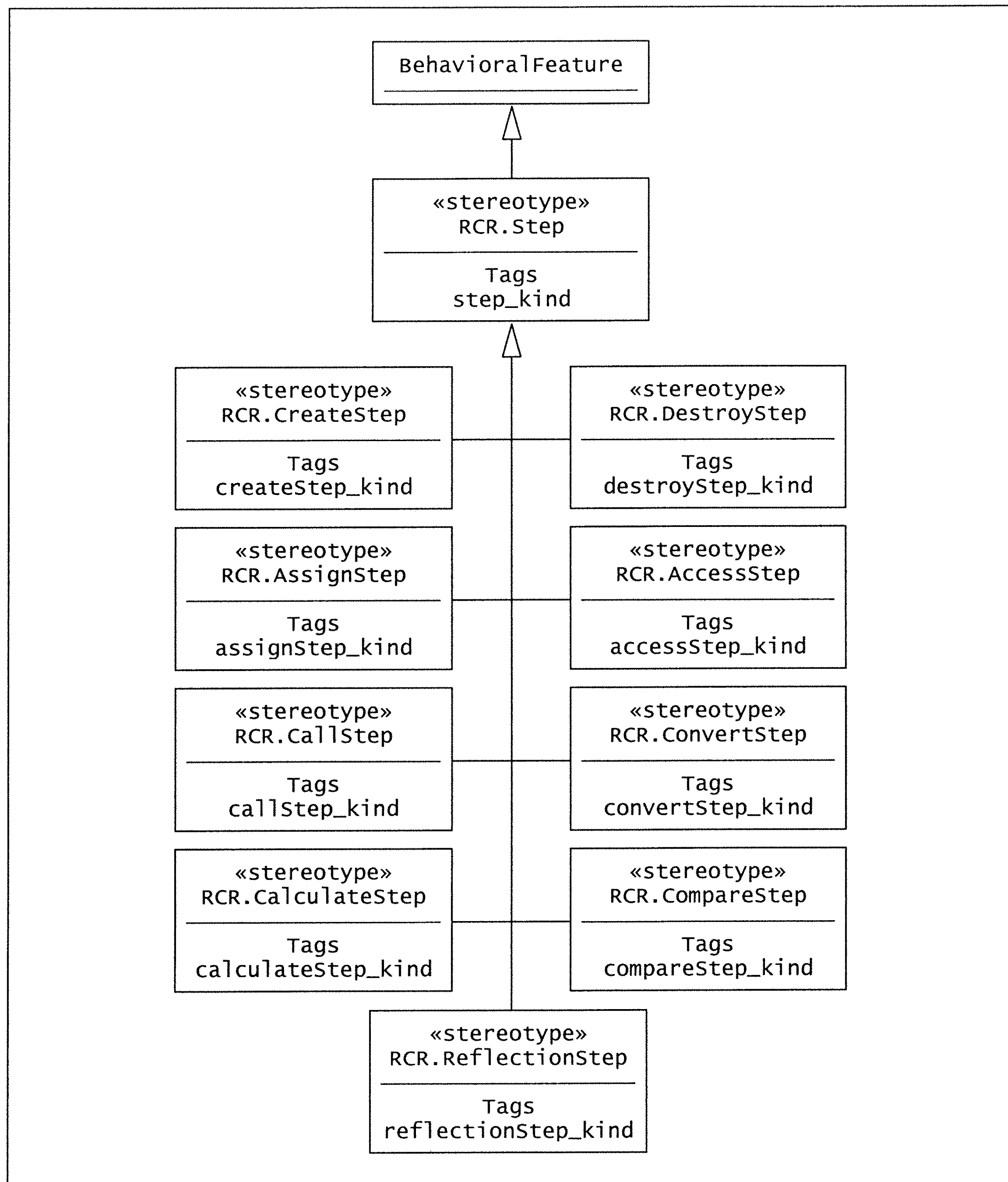

Figure 6. Hiérarchie des pas d'évaluation du profil RCR (tirée de [StDenis_2001])

3
État de l'art

Le chapitre 3, *« État de l'art »*, présente succinctement certains projets de recherche et produits commerciaux apparentés à l'étude courante par les objectifs visés et les technologies employées.

Ce chapitre contient les sections suivantes :

Section	*Page*
3.1 GEN++	20
3.2 Datrix	21
3.3 CPPX	22
3.4 Columbus	23
3.5 DISCOVER	24
3.6 Passerelle Datrix-SPOOL	25
3.7 Comparaison des différents projets et produits	26
3.8 Conclusion	28

3.1 GEN++

GEN++ est un générateur d'analyseurs de code source C++. Il est basé sur le processeur frontal Cfront et sur un langage abstrait de spécification de requêtes, GENOA.

GENOA visualise le code source du point de vue des graphes sémantiques abstraits (ASG). Il offre un éventail de constructions pour la conception de requêtes appliquées aux ASG : navigation dans les arbres, application de tests, exécution d'énoncés impératifs et construction d'expressions complexes.

Un analyseur de code source bâti avec GEN++ interagit avec son processeur frontal de la façon suivante : il invoque tout d'abord le processeur pour effectuer le *parsing* d'un fichier source. Lorsque le processeur est rendu au point où la génération du code objet peut prendre place, un pointeur visant le nœud racine de l'ASG construit en mémoire est retourné à l'analyseur. Ce dernier exécute par la suite différentes explorations de l'ASG : par exemple, rechercher des nœuds de types spécifiques, inspecter les structures des sous-arbres, valider les relations entre les structures, afin de générer un rapport en sortie.

Une spécification GEN++ est généralement constituée d'un ensemble de procédures et de fonctions. Les procédures peuvent être utilisées à titre de commandes, tandis que les fonctions, retournant toujours un nœud d'ASG, servent à la construction d'expressions complexes. Chaque procédure est assignée à un type de nœud d'ASG particulier et déclenche un traitement avec le nœud qu'elle reçoit implicitement en paramètre. Les constructions GENOA présentes dans les procédures agissent sur ce même nœud, et génèrent les informations requises dans le rapport externe.

GEN++ a été délibérément conçu comme un langage simple, avec un nombre limité de types de données, d'expressions prédéfinies et de structures de contrôle. Cependant, il est possible d'intégrer aux spécifications GEN++ des extraits de code rédigés en C++ et de faire appel à des fonctions externes.

Des informations supplémentaires sont disponibles sur le site Web de GEN++ [Devanbu_1998].

3.2 Datrix

La suite d'outils Datrix, de Bell Canada, comporte plusieurs applications destinées à évaluer la qualité du logiciel, en termes de facilité de maintenance et d'évolution.

Les parseurs de la suite Datrix traitent les systèmes rédigés en C et en C++. Après avoir effectué le prétraitement des fichiers source à l'aide du préprocesseur intégré à la suite, les parseurs construisent en mémoire les ASG correspondants et émettent leurs graphes selon le format TA-like ou VCG. Le format TA-like est dérivé du format TA

(Tuple-Attribute) [Holt_2002], à l'exception qu'il ne possède pas de section de schéma et que les déclarations de tous les nœuds précèdent celles des arcs les reliant. Le format VCG *(Visualization of Compiler Graphs)* [Sander_1995] est un format d'encodage de graphes, utilisé par un outil de visualisation du même nom.

Les fichiers TA-like générés sont conçus pour être facilement traités par d'autres applications. Par exemple, ils peuvent être récupérés par un analyseur faisant partie de la suite; celui-ci extrait, à partir de leur contenu, des métriques au niveau des méthodes, des classes et des fichiers source du système. Ces fichiers TA-like sont aussi utilisés dans l'atelier de génie logiciel SPOOL [SPOOL] par le convertisseur ASG2XMI, qui assemble en mémoire les nœuds et les arcs présents dans les fichiers d'échange et émet les composantes répertoriées sous la forme de fichiers XMI.

Les ASG sont représentés au sein des fichiers TA-like à l'aide d'un modèle propre aux outils Datrix. Le modèle poursuit deux objectifs principaux, à savoir la complétude, permettant de nombreuses analyses de rétro-ingénierie sans avoir à recourir au code source, et l'indépendance vis-à-vis du langage de programmation utilisé. Un manuel de référence, décrivant en détail le modèle des ASG utilisé [Datrix_ASG_2000], est disponible sur le site Web des outils Datrix [Datrix_2000].

3.3 CPPX

CPPX est un compilateur C++ qui produit en sortie une base d'informations au lieu de l'habituel code exécutable.

CPPX utilise le compilateur GCC de GNU [GCC] pour l'analyse syntaxique et sémantique du code source C++. Une fois l'analyse terminée, CPPX convertit la structure de données interne modélisant le code source compilé en une représentation basée sur une version légèrement modifiée du modèle des ASG de Datrix [Datrix_ASG_2000].

La base d'informations générée en sortie est une collection de faits à propos d'un programme et de ce qu'il contient, définit, utilise, modifie, requiert, importe et exporte.

Cette base peut par la suite être fournie en tant qu'intrant à de nombreux outils de génie logiciel pour analyser et visualiser le code source, récupérer les architectures, analyser le flot de données, restructurer un système, et ainsi de suite. En quelque sorte, CPPX constitue un processeur frontal universel pour le langage C++.

La base d'informations est un graphe entités-relations; ses nœuds correspondent aux classes, méthodes, énoncés, expressions, variables et constantes présents dans le système; ses arêtes représentent les relations syntaxiques entre les composantes (faisant ainsi apparaître le *parse tree* des programmes) et les relations sémantiques telles les liens entre les identifiants et leurs déclarations, les utilisations des méthodes et leurs définitions, les différents objets et leurs types respectifs.

Le graphe exporté est émis dans le format d'échange GXL [GXL]. CPPX peut aussi émettre le graphe à l'aide des formats TA [Holt_2002] et VCG [Sander_1995].

Des informations complémentaires sont disponibles sur le site Web du processeur CPPX [CPPX].

3.4 Columbus

Columbus est un cadre d'application de rétro-ingénierie logicielle pour l'analyse, la représentation interne, le filtrage et l'exportation de code source, rédigé dans des langages de programmation arbitraires, en des formats d'échange variés. L'indépendance vis-à-vis des langages de programmation, de la structure de la représentation interne et des formats d'exportation est obtenue par l'utilisation de modules *(plug-ins)* spécifiques qui se raccordent au cadre d'application générique.

Dans sa version actuelle, Columbus contient un module d'extraction pour le langage C++ (CAN), un éditeur de liens pour ce module d'extraction (CANLink) et différents modules d'exportation pour les formats CPPML (*C++ Markup Language* — produisant des documents XML conformes au DTD du schéma Columbus), GXL [GXL], HTML [HTML] et RSF *(Rigi Standard Format)* [Rigi].

Le processus d'extraction prend en entrée les fichiers source d'un système et produit, pour chacun d'entre-eux, un fichier objet correspondant. Par la suite, le processus de liaison charge l'ensemble des fichiers objets générés et produit un seul fichier où chaque entité incorporée dans les fichiers objets est représentée une seule fois. Lors du processus d'exportation final, l'application filtre les entités présentes dans le modèle global selon les besoins de l'utilisateur et produit un fichier d'échange qui peut être visualisé ou fourni en tant qu'intrant à des ateliers de génie logiciel.

Les entités colligées par l'extracteur CAN sont les structures (`class`, `struct`, `union`) et leurs composantes (attributs et fonctions), les énumérations, les *templates* et les objets globaux. Le processus de filtrage permet de discriminer les entités à exporter selon leur appartenance à des structures ou à des espaces de nomenclature *(namespaces)* spécifiques. L'extracteur CAN ne traite pas les corps des fonctions (énoncés et expressions) dans sa version actuelle.

Columbus possède une version modifiée de son extracteur C++, CANCG, qui permet de produire le graphe des appels de fonctions, exportable dans le format RSF.

Des informations supplémentaires concernant Columbus, son schéma et ses modules associés sont présentes dans son guide de l'utilisateur, fourni lors du téléchargement du logiciel à partir du site Web de la compagnie FrontEndART Ltd. [FrontEndART].

3.5 DISCOVER

DISCOVER, de la compagnie MKS (auparavant Upspring Software), est un atelier de génie logiciel consistant en une suite d'outils aidant à la compréhension des systèmes.

Au cœur de l'atelier DISCOVER se trouve un dépôt alimenté par le *parsing* détaillé du code source. L'information présente dans les tables de symboles et les arbres syntaxiques abstraits (AST) des parseurs est capturée et stockée dans ce dépôt. On y retrouve les macrocommandes étendues ainsi que les instructions ajoutées implicitement

par le processus de compilation. Au nombre des langages supportés par DISCOVER, on compte C, C++, Java et SQL.

DISCOVER donne accès aux AST ainsi emmagasinés par l'entremise d'un langage de scriptage dérivé de Tcl, Tcl/Access. Il est possible d'émettre des commandes en mode interactif afin de consulter le contenu du dépôt, mais aussi de rédiger des scripts complexes qui repèrent des constructions syntaxiques nuisibles, implantent des changements de façon automatisée, exécutent des tâches répétitives, ou produisent des rapports élaborés d'après le contenu des AST.

La suite DISCOVER offre aussi des outils de visualisation des composantes présentes dans le dépôt, des scripts de validation de pratiques de programmation, des générateurs de rapports d'assurance-qualité et des applications détectant les sections de code mort et supprimant les directives d'inclusion superflues.

Le site de la compagnie MKS renferme des informations complémentaires sur la suite d'outils DISCOVER [MKS].

3.6 Passerelle Datrix-SPOOL

L'atelier de génie logiciel SPOOL [SPOOL] possède une passerelle pour alimenter son dépôt orienté objet de code source. Il s'agit de la passerelle Datrix-SPOOL, qui recueille les fichiers `.asg` produits par les parseurs de la suite Datrix et importe leur contenu dans le dépôt.

La passerelle est composée de deux applications. La première, le convertisseur ASG2XMI, lit les fichiers `.asg` générés par les parseurs Datrix en créant en mémoire les graphes sémantiques abstraits correspondants, puis exporte les éléments contenus dans les graphes sous la forme de fichiers XMI. La deuxième application, l'importateur XMI, traite les fichiers générés précédemment par le convertisseur ASG2XMI et crée dans le dépôt les objets que les fichiers contiennent. Les outils de visualisation de SPOOL peuvent ainsi consulter, par l'entremise du dépôt, les constructions initialement présentes dans le code source des systèmes logiciels.

L'importateur XMI, dans sa version actuelle, ne reconnaît qu'un sous-ensemble des métaéléments XMI tout en traitant certaines extensions implantées pour les besoins de l'atelier SPOOL.

3.7 Comparaison des différents projets et produits

La section courante offre une comparaison entre les différents projets et produits présentés dans le cadre de ce chapitre. À cet effet, chaque approche est évaluée selon sept critères distincts présentés dans le tableau I. La passerelle Datrix-SPOOL ne figure pas au nombre des projets examinés, puisqu'elle s'approprie les caractéristiques des parseurs de la suite Datrix.

Au niveau du traitement du code source C, seuls les parseurs des suites Datrix et DISCOVER en possèdent la capacité. Les autres produits traitent le code C pour autant qu'il soit compilable en tant que code C++ valide; par exemple, le code C fourni en entrée ne doit jamais utiliser un mot-clé réservé du C++, tel `this`, pour déclarer une variable locale.

Tous les produits examinés sont capables d'effectuer l'analyse du code source C++, selon des dialectes et des niveaux de tolérance aux erreurs de *parsing* variables. Cependant, seulement les parseurs de la suite DISCOVER traitent le code source rédigé en Java. Le traitement du langage Java figure à titre de projet de développement pour les parseurs de la suite Datrix, ainsi que pour le cadre d'application Columbus : de nouveaux modules *(plug-ins)* seront éventuellement conçus à cette intention.

La production d'ASG à partir d'AST, soit l'ajout de liens reliant les utilisations des variables, des types et des fonctions à leurs déclarations respectives, est effectuée dans le cadre des applications GEN++ et CPPX. Dans sa version actuelle, Datrix crée ces liens de façon partielle; Columbus n'offre pas de modules permettant d'extraire les informations requises pour la composition de telles relations; DISCOVER, même s'il n'effectue pas ces liens de façon explicite, donne cependant accès à toute l'information requise pour la création exhaustive des liens, et ce pour l'intégralité d'un système traité.

Propriété	*GEN++*	*Datrix*	*CPPX*	*Columbus*	*DISCOVER*
Traitement du code source C	+ –	+	+ –	+ –	+
Traitement du code source C++	+	+	+	+	+
Traitement du code source Java	–	–	–	–	+
Production d'ASG à partir d'AST	+	+ –	+	–	–
Accession aux AST/ASG par programmation	+	–	–	–	+
Traitement des composantes systémiques de haut niveau	+	+	+	+	+
Traitement du détail des corps des méthodes	+	+ –	+ –	–	+

Tableau I. Comparaison des différents projets et produits

Au titre de l'accession par programmation aux AST ou aux ASG, seuls GEN++ et DISCOVER permettent à un usager de rédiger des scripts permettant l'exploration des structures internes d'un programme. Les autres produits génèrent des rapports ou des fichiers d'échange dont les éléments constituants sont prédéterminés. La programmation de scripts Tcl/Access au sein de l'environnement DISCOVER est cependant beaucoup plus conviviale que la rédaction de spécifications GENOA dans GEN++.

Tous les produits et projets répertoriés traitent les composantes systémiques de haut niveau telles les structures et leurs composantes, les fonctions, les énumérations, etc. Par contre, le détail des corps des fonctions est pris en charge à des niveaux divers. Dans le cas de Columbus, cette fonctionnalité demeure à l'état de projet. Pour Datrix et CPPX, certaines informations sont colligées au niveau des corps des fonctions, telles les variables utilisées, les fonctions sollicitées et les instanciations d'objets. Seuls GEN++ et DISCOVER donnent accès par programmation aux AST ou ASG sous leur forme intégrale, ce qui permet d'inspecter la structure des énoncés et des expressions constituant les méthodes, afin d'en extraire toute l'information requise pour la conduite d'analyses détaillées.

Le tableau I effectue la synthèse des comparaisons précédemment effectuées. Une inspection approfondie de ce tableau fait ressortir que seuls les parseurs de l'atelier de

génie logiciel DISCOVER permettent d'apporter les nouvelles fonctionnalités requises pour l'atelier SPOOL (consulter les sections 1.2.3, *« Passerelles d'importation »* et 1.3, *« Objectifs de la recherche »*), c'est-à-dire traiter le code source C, C++ et Java tout en fournissant de l'information hautement détaillée concernant les corps des méthodes. Bien que les parseurs de DISCOVER n'établissent pas explicitement les liens référentiels propres aux ASG, ils rendent accessible toute l'information requise pour leur construction éventuelle. DISCOVER constituera donc une composante clé au sein de la solution retenue, présentée à la section 4.6, *« Ébauche de la solution »*.

3.8 Conclusion

Ce chapitre a présenté quelques projets de recherche et produits commerciaux reliés au sujet de ce mémoire. Ces projets et produits partagent entre eux des affinités au niveau des buts visés et des technologies employées. De plus, il a dressé une comparaison de ces projets et produits, axée sur sept critères reliés aux fonctionnalités souhaitées de l'atelier SPOOL, soit la prise en charge des systèmes rédigés en C, C++ et Java, le traitement des composantes logicielles de haut niveau et du détail des corps des méthodes, ainsi que l'établissement et l'exploitation de liens entre l'utilisation et la déclaration des variables, des types et des fonctions.

Le chapitre suivant expose l'énoncé du problème traité dans cet ouvrage : l'alimentation d'un dépôt de code source permettant l'analyse détaillée de systèmes logiciels de taille industrielle.

4

Énoncé du problème

Le chapitre 4, *« Énoncé du problème »*, présente la problématique étudiée dans le cadre de ce mémoire, à savoir l'alimentation d'un dépôt de code source permettant l'analyse détaillée de systèmes logiciels de taille industrielle.

Ce chapitre contient les sections suivantes :

Section	***Page***
4.1 Énoncé général du problème	29
4.2 Éléments constitutifs d'un système considérés	30
4.3 Éléments constitutifs du code source exclus	31
4.4 Langages de programmation considérés	33
4.5 Attentes techniques	33
4.6 Ébauche de la solution	35
4.7 Conclusion	36

4.1 Énoncé général du problème

Le problème constitue à alimenter un dépôt de code source, faisant partie de l'atelier de génie logiciel SPOOL, permettant l'analyse détaillée de systèmes de taille industrielle. Les informations conservées dans le dépôt sont présentées à la section suivante : toutes ces informations doivent être extraites du code source des systèmes étudiés. Des outils de visualisation, propres à l'environnement SPOOL, récupèrent les objets stockés dans le dépôt et permettent d'effectuer l'analyse des systèmes traités.

4.2 Éléments constitutifs d'un système considérés

Les informations conceptuelles suivantes doivent être colligées pour les systèmes étudiés. Ces éléments, une fois recueillis et entreposés dans un dépôt, permettent la conduite d'analyses systémiques complètes et détaillées :

- les répertoires dans lesquels sont conservés les fichiers source du système;
- les liens d'inclusion entre ces mêmes répertoires;
- pour chaque répertoire, les fichiers source qu'il contient;
- pour chaque fichier source, les fichiers inclus à l'aide d'une directive `#include`;
- les différentes structures de données (`class`, `struct`, `union`, `interface`);
- pour chaque structure, un lien d'inclusion entre la structure et le fichier à l'intérieur duquel sa déclaration se trouve;
- pour chaque classe, l'ensemble de ses superclasses et de ses sous-classes;
- pour chaque classe, l'ensemble des interfaces qu'elle implante;
- pour chaque structure, l'ensemble des structures englobées *(inner classes)*;
- pour chaque structure, l'ensemble des structures et fonctions amies (`friend`);
- pour chaque structure, l'ensemble de ses attributs et leurs types respectifs;
- les différentes énumérations et leurs littéraux respectifs;
- les alias (`typedef`) et les types qu'ils représentent;
- les fonctions, les noms et types de leurs paramètres, leurs types de retour et les exceptions qu'elles lancent;
- pour chaque énumération, alias et fonction appartenant à une structure, un lien d'inclusion entre cet élément et la structure englobante;

- pour chaque énumération, alias et fonction n'appartenant à aucune structure, un lien d'inclusion entre cet élément et une classe utilitaire propre au fichier du système à l'intérieur duquel sa déclaration est rencontrée;

- les types complexes utilisés (types constants, pointeurs, références, tableaux) et leurs sous-types représentés; un lien d'inclusion pour chaque type complexe pointant vers une classe utilitaire correspondante, selon le type de type complexe;

- les types fonctions utilisés avec les types de leurs paramètres et leurs types de retour; un lien d'inclusion pour chaque type fonction pointant vers la classe utilitaire correspondante;

- les variables globales de tout le système, ainsi que les valeurs d'initialisation qui leur sont associées;

- les expressions d'initialisation associées aux variables statiques des structures;

- les expressions d'initialisation associées aux littéraux des énumérations;

- le détail des corps des fonctions, incluant les blocs d'initialisation statiques des structures, sous la forme de constructions représentant fidèlement les énoncés et expressions présents dans le code source;

- pour chaque bloc d'initialisation statique, un lien d'inclusion entre cet élément et la structure associée;

- pour chaque utilisation d'un objet local à une fonction ou global au système, un lien pointant vers la déclaration de l'objet correspondant.

4.3 Éléments constitutifs du code source exclus

Les éléments d'information inhérents au code source présentés dans cette section ne sont pas considérés dans le cadre de cette étude. Une motivation des choix effectués est présentée pour chaque élément.

4.3.1 Directives de précompilation

Le prétraitement des fichiers source effectue, pour les langages C et C++, la compilation conditionnelle de sections de code (directives `#ifdef` et `#ifndef`), l'inclusion de fichiers (directive `#include`) ainsi que l'expansion de macrocommandes (précédemment définies à l'aide de directives `#define`). Puisque les éléments constitutifs d'un système, énumérés dans la section précédente, sont en partie déterminés à partir de l'application des directives de précompilation, ces dernières ne sont pas modélisées en tant que telles dans le dépôt, mais selon le résultat de leur application.

4.3.2 Raccourcis syntaxiques

Les raccourcis syntaxiques permettent d'alléger le code source en substituant une syntaxe légère à des expressions encombrantes. Un bon exemple est offert par la concaténation de chaînes de caractères en Java, laquelle peut aussi être représentée à l'aide de l'application répétée de la méthode `String.concat()`. Les opérateurs surchargés en C++ constituent un second exemple approprié. Les raccourcis syntaxiques seront modélisés à l'aide de leur représentation étendue, puisque cette dernière fait ressortir de façon explicite les méthodes sollicitées à l'intérieur des expressions.

4.3.3 Commentaires et mise en page du code source

Les commentaires et les éléments qui effectuent la mise en page du code (espacements, tabulations, etc.) ne sont pas recueillis lors du *parsing* ni stockés ultérieurement dans le dépôt de l'atelier SPOOL. Même si les commentaires peuvent s'avérer judicieux pour la compréhension du code source, leur non-considération reflète les choix initialement effectués pour la conception du dépôt et des outils de visualisation de SPOOL.

4.4 Langages de programmation considérés

Les langages C, C++ et Java sont choisis à titre de langages considérés. Il est nécessaire de restreindre la gamme des langages retenus afin de ne pas complexifier à outrance les algorithmes qui recueilleront les éléments constitutifs d'un système par inspection de son code source. De plus, ces trois langages sont ceux qui sont le plus utilisés dans l'industrie, surtout dans le domaine des télécommunications au sein duquel œuvre notre partenaire industriel. Grâce à ce choix, il est possible de couvrir une large part des systèmes logiciels présents dans les milieux universitaire et industriel. Les aptitudes techniques du profil RCR et de l'atelier SPOOL serviront à modéliser et stocker les éléments conceptuels définis par ces langages; les parseurs de la suite DISCOVER permettront le traitement précis et exhaustif du code source rédigé dans ces mêmes langages.

4.5 Attentes techniques

Cette section spécifie les attentes techniques relatives aux différentes étapes conduisant à l'alimentation du dépôt de code source de SPOOL, en s'attardant spécifiquement aux caractéristiques des fichiers d'échange générés.

4.5.1 Rapidité des processus

La rapidité des différents processus mis en œuvre pour l'alimentation du dépôt est primordiale. Plus particulièrement, les applications effectuant la génération des fichiers d'échange et les important par la suite dans le dépôt devront être conçues à l'aide de technologies et d'algorithmes permettant une performance optimale. Les fichiers d'échange devront être générés de façon telle qu'ils pourront être importés rapidement.

4.5.2 Format des fichiers d'échange

Les fichiers utilisés pour effectuer l'importation des systèmes dans l'atelier SPOOL devront être produits à l'aide d'un format d'échange répandu et largement supporté, ce qui permettra leur utilisation éventuelle dans plusieurs autres ateliers de génie logiciel.

4.5.3 Production de fichiers d'échange compacts

En plus de permettre une importation rapide, les fichiers d'échange générés devront être les plus compacts possible afin de réduire la consommation d'espace disque. Une technologie de compression des fichiers dans des archives pourra être utilisée si elle s'avère utile.

4.5.4 Lisibilité des fichiers d'échange

La possibilité de générer des fichiers d'échange qui puissent être consultés aisément par un être humain comporte des avantages indéniables. Ceci facilite grandement la mise au point des applications produisant ces fichiers, tout en permettant à leurs usagers éventuels d'inspecter leur contenu avant leur importation. Cependant, la lisibilité désirée ne doit pas être obtenue au détriment de fichiers trop volumineux.

4.5.5 Séparation des fichiers d'échange en deux groupes distincts

Les fichiers d'échange, comportant les informations modélisant un système donné, seront produits en deux groupes. Le premier groupe contiendra principalement les définitions des structures et des éléments qu'elles englobent, tandis que le deuxième groupe comportera presque exclusivement le détail des corps des fonctions. Le premier groupe de fichiers pourra être importé sans le deuxième. Cette séparation est effectuée puisqu'il est anticipé que le premier groupe de fichiers sera beaucoup moins volumineux que le deuxième, et que plusieurs analyses pourront être conduites dans l'atelier SPOOL en utilisant strictement les informations présentes dans le premier groupe de fichiers.

4.5.6 Utilisation d'identifiants uniques

Chacune des composantes d'un système, exportées à l'intérieur des fichiers d'échange, doit se voir attribuer un identifiant unique. Ces identifiants sont principalement utilisés pour la modélisation des liens de typage et des références entre l'utilisation et la déclaration des variables et des fonctions. L'utilisation d'identifiants uniques est rendue nécessaire puisque des éléments de même nom peuvent être déclarés dans des portées différentes : ces identifiants permettent alors d'effectuer les liens précis entre les utilisations des éléments et leurs déclarations correspondantes.

4.6 Ébauche de la solution

Cette section présente une ébauche de la solution retenue, intégrable à l'atelier SPOOL. Les composantes conçues dans le cadre du projet SPOOL sont exposées dans les chapitres qui suivent celui-ci.

Afin d'amorcer le traitement des systèmes à étudier, le logiciel DISCOVER est utilisé. Celui-ci effectue le *parsing* des fichiers source et construit, dans des fichiers de stockage indépendants, les AST associés aux unités de compilation du système.

Par la suite, des scripts d'exportation rédigés dans le langage Tcl/Access sont exécutés dans l'environnement DISCOVER et produisent les deux groupes de fichiers d'échange décrits dans la section précédente.

Le format d'échange retenu est XMI. Afin de produire le deuxième groupe de fichiers d'échange comportant principalement le détail des corps des fonctions, le profil RCR est utilisé pour obtenir un format d'échange dérivé de XMI, nommé XMI/RCR dans le reste de ce document.

Les fichiers d'échange sont traités à l'aide d'un importateur XMI standard. Cet importateur comporte un mécanisme de gestion de profil permettant d'interpréter les balises RCR que les fichiers renferment. Le premier groupe de fichiers est traitable par l'importateur couramment implanté dans l'atelier SPOOL.

Le dépôt alimenté par l'importateur est une version étendue du dépôt actuel de l'atelier SPOOL, lui-même basé sur UML. Le dépôt est principalement modifié pour permettre l'importation des éléments modélisant le détail des corps des fonctions.

La figure 7 schématise les différentes composantes de la solution retenue et leurs interrelations.

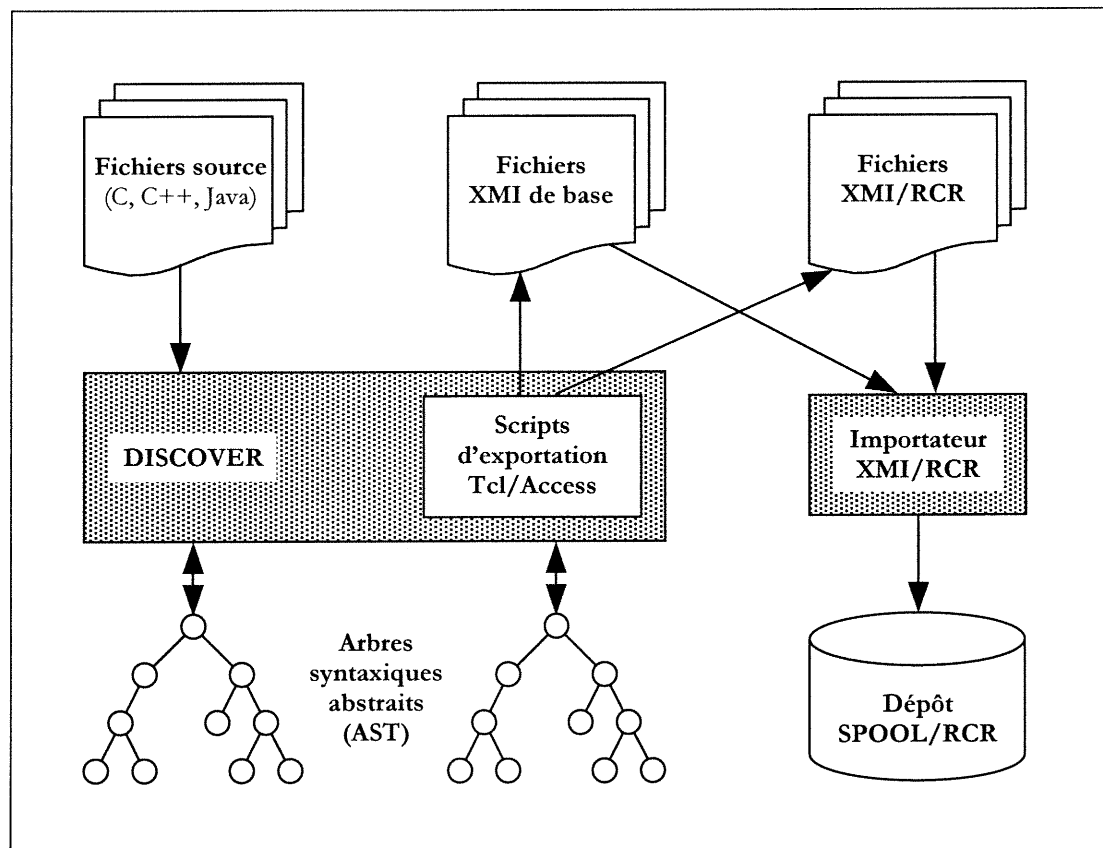

Figure 7. Composantes de la solution retenue

4.7 Conclusion

Ce chapitre a présenté le problème étudié dans le cadre de ce mémoire : l'alimentation d'un dépôt de code source permettant l'analyse détaillée de systèmes logiciels de taille industrielle.

Après avoir énoncé le problème, les éléments d'un système logiciel considérés pour l'importation dans un dépôt ont été énumérés, ainsi que les constructions présentes dans le code source qui ne sont pas retenues.

Par la suite, les langages de programmation devant être supportés par les applications implantant le processus d'importation ont été présentés, et les attentes techniques reliées aux processus et aux fichiers d'échange générés ont été précisées.

Pour clore le chapitre, une esquisse de la solution retenue a été présentée.

Les chapitres suivants exposent les composantes clés de la solution. Le chapitre 5 présente les approches retenues pour la génération des fichiers XMI, tandis que le chapitre 6 expose l'architecture du dépôt de données SPOOL/RCR et de l'importateur associé.

5

Génération des fichiers XMI

Le chapitre 5, *« Génération des fichiers XMI »*, présente la façon dont les fichiers XMI de base, décrivant les éléments d'un système, et les fichiers XMI/RCR, modélisant principalement les corps des fonctions d'un système, sont générés à partir des informations contenues dans les AST de DISCOVER.

Ce chapitre contient les sections suivantes :

Section	***Page***
5.1 Génération des fichiers XMI de base	38
5.2 Génération des fichiers XMI/RCR	40
5.3 Conclusion	42

5.1 Génération des fichiers XMI de base

L'approche générale retenue pour la génération des fichiers XMI s'articule autour de deux facettes principales :

1. la construction préalable de tables de symboles globales;

2. la génération proprement dite des fichiers XMI.

La construction de tables de symboles globales permet de répertorier, pour un système entier, les structures (`class`, `struct`, `union`, `interface`) et leurs attributs, les énumérations et leurs littéraux, les alias (`typedef`) ainsi que les déclarations et définitions de fonctions.

Cette opération s'accompagne de la confection d'autres tables globales qui contiennent des informations primordiales pour la génération ultérieure des fichiers

XMI : la table de hiérarchie des répertoires et fichiers, la table d'inclusion (`#include`) entre fichiers, la table de hiérarchie des classes (superclasses et sous-classes) et la table d'implantation des interfaces.

Une fois les tables de symboles globales construites, la génération des fichiers XMI peut prendre place. Les informations préalablement stockées dans les tables de symboles globales sont alors utilisées pour identifier les éléments exportés dans les fichiers XMI. De plus, d'autres tables globales sont remplies au fur et à mesure que le processus d'exportation se déroule, par exemple lors de la rencontre de nouveaux types au niveau des attributs des structures ou des paramètres des fonctions : la table des types complexes (types constants, pointeurs, références et tableaux), la table des types fonctions et la table des types non résolus.

Des structures de données spécialisées ont été conçues pour l'élaboration des tables de symboles globales et la modélisation des transferts d'information avec les fonctions qui participent à leur construction.

Les constantes et les objets globaux utilisés par les diverses procédures et fonctions implantant la mécanique d'exportation XMI ont été encapsulés dans deux packages distincts dans le but d'offrir une architecture modulaire appropriée et de faciliter la maintenance des scripts.

Les procédures et fonctions qui implantent les algorithmes de génération ont été classifiées en trois grandes catégories : les procédures et fonctions de support, les procédures et fonctions affectées à la construction des tables de symboles globales et les procédures affectées à la génération des fichiers XMI. Ces procédures et fonctions sont décrites en détail dans un rapport technique [Bédard_2002] en étant accompagnées de descriptions informelles en langage courant et de transcriptions en pseudo-code des algorithmes mis en œuvre. Ces transcriptions en pseudo-code constituent un reflet fidèle de l'implantation fonctionnelle réalisée à l'aide du langage Tcl/Access de DISCOVER. À la différence du code Tcl/Access, le pseudo-code possède une notation orientée objet, ce qui facilite la compréhension des algorithmes.

Aux fins de discrimination des éléments d'un système au sein des fichiers XMI générés, les identifiants uniques (`etag`) de DISCOVER sont retenus. À chaque nœud d'AST de DISCOVER est associé un identifiant unique à l'ensemble du système. L'utilisation des mêmes identifiants au cœur des fichiers XMI permet d'effectuer des comparaisons rapides entre les fichiers générés et les structures présentes dans les AST de DISCOVER, ce qui rend la mise au point des scripts plus aisée, en plus de réduire considérablement la taille des tables globales utilisées pour répertorier les différents objets d'un système et leurs identifiants uniques.

5.2 Génération des fichiers XMI/RCR

L'approche générale pour la génération des fichiers XMI/RCR s'apparente à celle retenue pour la génération des fichiers XMI de base. Elle s'articule autour de trois facettes principales :

1. la construction préalable de tables de symboles globales propres au processus d'exportation XMI/RCR;

2. la génération de fichiers XMI/RCR contenant les objets globaux d'un système;

3. la génération de fichiers XMI/RCR contenant principalement le détail des corps des fonctions, à l'aide de procédures affectées au traitement des nœuds d'AST de DISCOVER.

Dans le cadre du processus d'exportation XMI/RCR, une seule table de symboles résolus est construite : il s'agit de la table des objets globaux qui répertorie l'ensemble des objets globaux d'un système entier. Il est important de noter que le langage C++ possède la notion d'objet global, contrairement au langage Java.

Une fois la table des objets globaux construite, la génération des fichiers XMI/RCR peut s'amorcer. Les déclarations des objets globaux et leurs valeurs d'initialisation sont émises au sein d'un premier ensemble de fichiers XMI/RCR. Par la suite, un second ensemble de fichiers XMI/RCR est produit; ce dernier renferme les expressions

d'initialisation associées aux variables statiques des structures (`class`, `struct`, `union` et `interface`) et aux littéraux des énumérations, ainsi que le détail des corps des fonctions et des blocs d'initialisation statiques des structures.

D'autres tables globales sont remplies au fur et à mesure que le processus d'exportation se déroule : il s'agit de la table des objets globaux non résolus et de la table des fonctions non résolues.

Afin d'exporter les structures versatiles présentes dans les AST, qui servent à décrire le détail des corps des fonctions et les autres éléments exportés dans le second ensemble de fichiers XMI/RCR, chaque nœud d'AST se voit associé une procédure spécifique d'exportation XMI/RCR. L'implantation actuelle du script d'exportation XMI/RCR contient 104 procédures spécifiques; 68 d'entre-elles sont regroupées à l'intérieur de sept familles génériques; les 36 autres procédures fournissent un traitement singulier pour autant de nœuds. Une procédure maîtresse, appelée par la plupart des procédures de traitement de nœuds, reçoit en paramètre un nœud quelconque et dirige l'exécution vers la procédure spécifique associée au traitement de ce type de nœud. Le balayage complet des AST est assuré par le respect intégral de la règle suivante : chaque nœud est responsable du déclenchement du traitement de chacun de ses nœuds enfants de premier niveau.

Des structures de données spécialisées sont utilisées pour construire les tables de symboles globales et effectuer des transferts d'information entre les procédures et fonctions.

Les constantes et les objets globaux utilisés par les diverses procédures et fonctions effectuant l'exportation XMI/RCR ont été encapsulés dans deux packages distincts afin d'offrir une architecture modulaire appropriée et de faciliter la maintenance des scripts.

Certaines procédures et fonctions supportant la génération des fichiers XMI sont utilisées dans le cadre de la génération des fichiers XMI/RCR et font l'objet de redirections; ces dernières fournissent des raccourcis syntaxiques pour la rédaction et la compréhension des scripts d'exportation XMI/RCR.

Les procédures et fonctions qui implantent les algorithmes de génération ont été classifiées en deux grandes catégories : celles qui dirigent la mécanique générale d'exportation XMI/RCR et celles qui effectuent l'émission XMI/RCR des nœuds d'AST. Dans la première catégorie se retrouvent les fonctions de support, les procédures et fonctions affectées à la construction des tables de symboles globales et les procédures affectées à la génération des fichiers XMI/RCR. La deuxième catégorie renferme les procédures et fonctions de support pour le traitement des nœuds d'AST, les procédures génériques de traitement des nœuds d'AST et les procédures spécifiques de traitement des nœuds d'AST. Toutes ces procédures et fonctions sont décrites en détail dans un rapport technique [Bédard_2002] en étant accompagnées de descriptions informelles en langage courant et de transcriptions en pseudo-code des algorithmes mis en œuvre. Ces transcriptions en pseudo-code constituent un reflet fidèle de l'implantation fonctionnelle réalisée à l'aide du langage Tcl/Access de DISCOVER. Le langage de pseudo-code employé est le même que celui utilisé pour la description des procédures et fonctions implantant la génération des fichiers XMI de base.

5.3 Conclusion

Ce chapitre a présenté sommairement la façon dont les fichiers d'échange XMI sont générés à partir des informations contenues dans les AST de DISCOVER. Les approches générales retenues ont été décrites en mettant en relief les différentes facettes autour desquelles elles s'articulent, tant pour la génération des fichiers XMI de base que pour celle des fichiers XMI/RCR.

Toutes les procédures et fonctions qui implantent les algorithmes de génération sont décrites en détail dans un rapport technique [Bédard_2002] et sont accompagnées de descriptions informelles en langage courant et de transcriptions en pseudo-code des algorithmes mis en œuvre.

Le chapitre suivant présente l'architecture du dépôt SPOOL/RCR, destiné à recevoir le contenu des fichiers XMI de base et XMI/RCR par la conduite d'un processus d'importation.

6
Dépôt de données SPOOL/RCR

Le chapitre 6, *« Dépôt de données SPOOL/RCR »*, présente l'architecture du dépôt SPOOL/RCR, destiné à recevoir le contenu des fichiers XMI de base et XMI/RCR, en plus de proposer des stratégies de conception pour l'importateur XMI/RCR associé.

Ce chapitre contient les sections suivantes :

Section	*Page*
6.1 Description de l'approche	43
6.2 Architecture du dépôt SPOOL/RCR	45
6.3 Stratégies de conception de l'importateur XMI/RCR	61
6.4 Conclusion	62

6.1 Description de l'approche

L'architecture du dépôt étendu SPOOL/RCR, présentée dans le cadre de ce chapitre, a été élaborée à partir de l'architecture actuelle du dépôt SPOOL, elle-même basée sur UML. Afin que les outils de visualisation et d'analyse de l'atelier SPOOL puissent interagir avec le dépôt étendu avec peu ou pas de modifications, il est primordial de conserver la structure de base du dépôt dans sa version présente.

Lors de la génération des fichiers d'échange XMI/RCR, les métaéléments du profil RCR, un mécanisme d'extension du métamodèle UML, sont mis à profit pour modéliser les microcomposantes logicielles présentes dans les corps des méthodes des systèmes. Afin de pouvoir charger dans le dépôt SPOOL ces composantes encodées dans les fichiers d'échange, il importe d'élaborer de nouvelles classes qui serviront à leur stockage spécifique. Puisque le format d'échange XMI, basé sur UML, a été étendu avec succès à

l'aide du profil RCR, la solution la plus appropriée pour la création des nouvelles classes dans le dépôt SPOOL, lui aussi basé sur UML, consiste à utiliser la hiérarchie des métaéléments du profil pour générer une hiérarchie de classes similaire dans le dépôt SPOOL/RCR. Lors de cette opération, chaque valeur étiquetée, associée à un métaélément donné, est représentée par un attribut rattaché à sa classe correspondante. Chaque dépendance dont un métaélément est le client est modélisée par un attribut dans la classe associée; le type de l'attribut est déterminé par le type de fournisseur spécifié par la dépendance.

Afin d'effectuer le lien entre une méthode donnée et son bloc de code principal, dont l'encodage des énoncés et expressions est contenu dans les fichiers d'échange XMI/RCR, un attribut est ajouté à la classe modélisant les méthodes dans le dépôt étendu.

Puisque les actions de création et de destruction d'objets, d'appel et de retour de fonctions sont modélisées de façon plus expressive avec les nouvelles classes de stockage apportées par l'intégration du profil RCR qu'avec les anciennes classes modélisant les actions dans le dépôt SPOOL, ces dernières sont abandonnées et retirées du modèle du dépôt SPOOL/RCR. Une refonte partielle des outils d'analyse et de visualisation en découle nécessairement afin que ceux-ci puissent tirer leurs informations à partir des nouvelles classes.

L'architecture du dépôt SPOOL/RCR, résultant de l'application de l'approche choisie, est plus complexe que celle du dépôt SPOOL. Le nombre important de classes lui étant ajoutées rend plus difficile sa maintenance éventuelle. Ceci aurait pu être évité en stockant les instances des métaéléments du profil RCR comme des instances des classes dont ils dérivent en tant que stéréotypes, un ensemble de valeurs étiquetées déterminant l'identité précise de chaque instance. Cependant, l'approche retenue produit un dépôt plus performant, en séparant les instances des métaéléments RCR des composantes fondamentales d'un système. De nombreuses requêtes étant exclusivement associées à ces composantes, leur exécution s'effectue plus rapidement lorsque le nombre d'instances qu'elles doivent balayer pour fournir un résultat est limité. Cette

remarque est aussi applicable au sujet des requêtes éventuellement rédigées pour inspecter les propriétés relatives aux instructions contenues dans les corps des fonctions.

6.2 Architecture du dépôt SPOOL/RCR

La section courante expose en détail l'architecture du dépôt SPOOL/RCR, destiné à entreposer les informations contenues dans les fichiers XMI de base et XMI/RCR.

Afin d'alléger le texte, les classes implantées pour optimiser les fonctionnalités du dépôt et pour supporter les outils de visualisation ne sont pas présentées dans cette section. Elles devront néanmoins faire partie intégrante d'une réalisation technique du dépôt SPOOL/RCR.

Les figures 8 à 12 présentent les classes fondamentales du dépôt, qui sont principalement utilisées pour stocker les informations conceptuelles de base pour un système donné. Ces informations sont incluses dans les fichiers XMI de base.

Les figures 13 à 20, quant à elles, exposent les classes issues de l'intégration du profil RCR dans le dépôt. Ces classes sont alimentées par le contenu des fichiers XMI/RCR.

6.2.1 Classes fondamentales du dépôt

La figure 8 présente les classes de haut niveau du dépôt SPOOL/RCR.

La classe `MModelElement` constitue la superclasse de toutes les classes du dépôt présentées ci-après. L'attribut `uuid` renferme l'identifiant unique associé à un élément. L'attribut `name` représente le nom de l'élément, unique dans la portée identifiée par l'attribut `namespace`. L'attribut `taggedValues` renferme les valeurs étiquetées rattachées à l'élément; les attributs `clientDependencies` et `supplierDependencies` recensent les dépendances auxquelles participe l'élément.

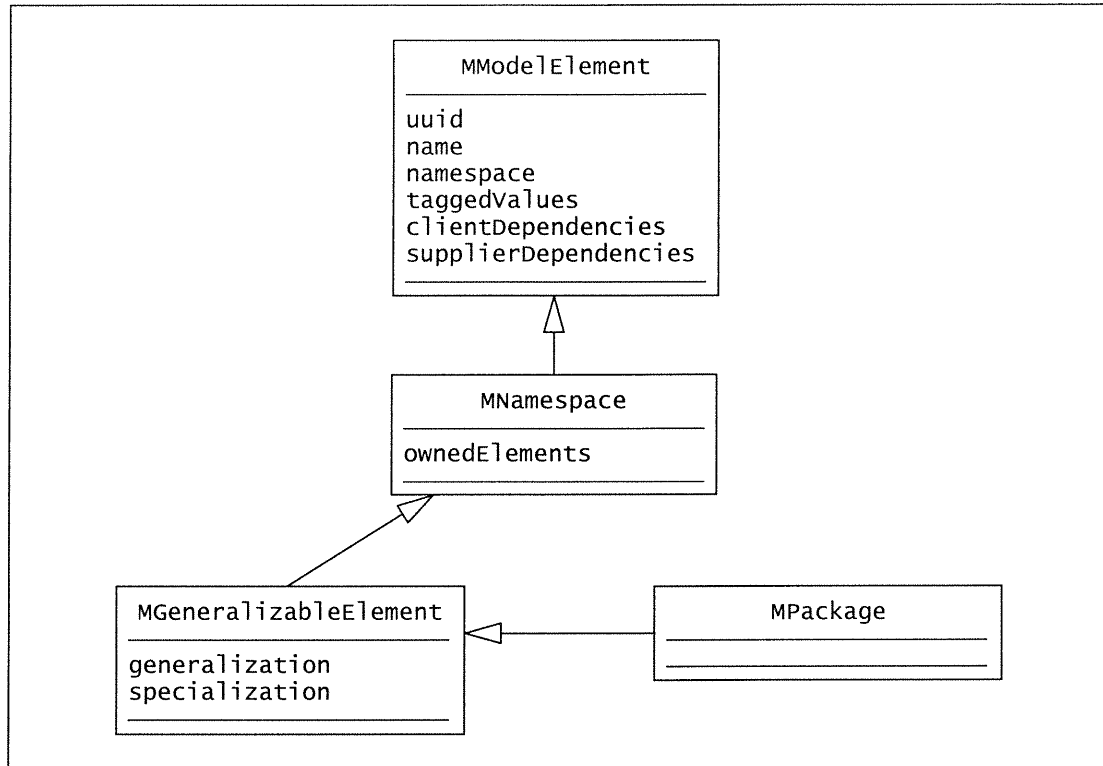

Figure 8. Classes de haut niveau du dépôt SPOOL/RCR

La classe `MNamespace` agit à titre de conteneur pour les éléments modélisés par la classe `MModelElement`. Son attribut `ownedElements` regroupe les différents éléments appartenant à une de ses instances.

La classe `MGeneralizableElement` modélise les éléments pouvant participer à une relation de généralisation. Les ancêtres immédiats d'un élément sont représentés par l'attribut `generalization`; les descendants immédiats, par l'attribut `specialization`.

La classe `MPackage` est utilisée pour modéliser le regroupement d'éléments dans un modèle, à l'intérieur d'un système, d'un répertoire ou d'un fichier, par exemple.

Les classes systémiques du dépôt sont présentées à la figure 9. La classe `MPackage`, présentée précédemment, constitue la superclasse de cette hiérarchie.

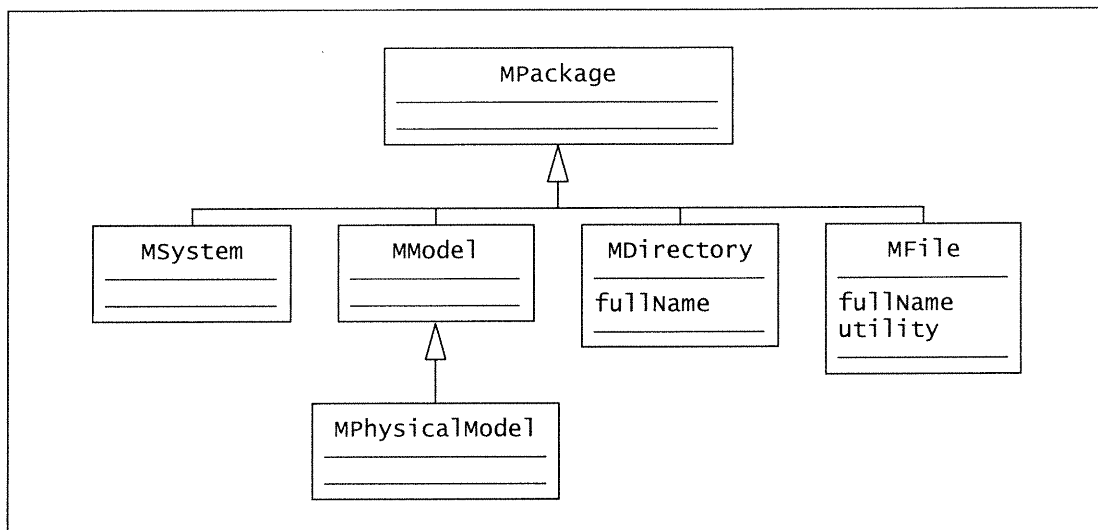

Figure 9. Classes systémiques du dépôt SPOOL/RCR

Les classes `MSystem`, `MModel` et `MPhysicalModel` représentent les systèmes, modèles et modèles physiques, respectivement. Un modèle physique consiste en un ensemble de répertoires et de fichiers, modélisés par les classes `MDirectory` et `MFile`. La classe `MDirectory` peut contenir d'autres répertoires et fichiers en son attribut hérité `ownedElements`. L'attribut `fullName` des classes `MDirectory` et `MFile` contient le nom complet de l'élément, à partir du répertoire racine. L'attribut `utility` de la classe `MFile` effectue le lien entre le fichier et sa classe utilitaire répertoriant les éléments non associés à une structure spécifique.

La figure 10 recense les classes structurelles faisant partie du modèle du dépôt. La classe abstraite `MClassifier` constitue la pierre angulaire de cette hiérarchie; elle modélise les entités auxquelles il est généralement possible d'associer des composantes (attributs) et des comportements (méthodes).

Les sous-classes immédiates `MAnonUnion`, `MClass`, `MInterface`, `MStruct` et `MUnion` sont respectivement utilisées pour représenter les unions anonymes, les classes, les interfaces, les structures `struct` et les structures `union`. La classe `MInterface` constitue un ajout par rapport au modèle précédent du dépôt de SPOOL, afin de supporter adéquatement le langage Java.

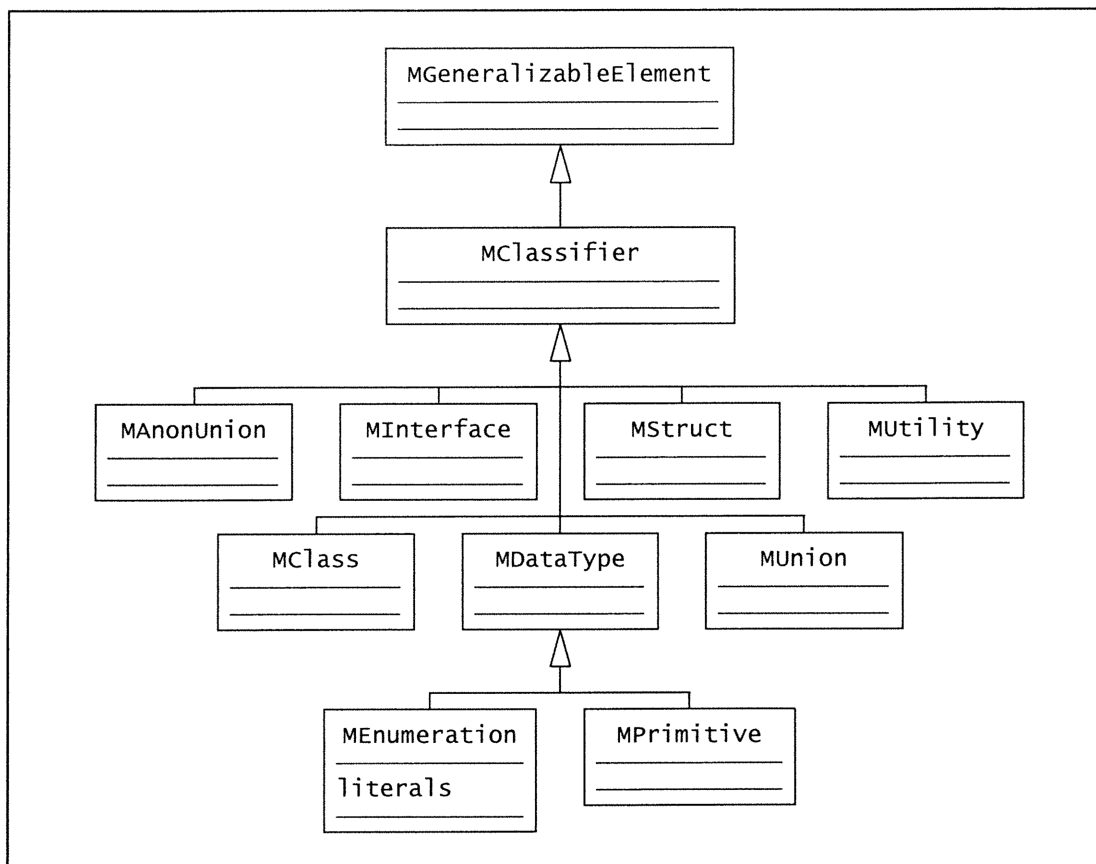

Figure 10. Classes structurelles du dépôt SPOOL/RCR

La classe `MUtility` sert à répertorier, par le contenu de son attribut hérité `ownedElements`, les éléments non associés à une structure, présents dans un fichier du système.

La classe abstraite `MDataType` représente les types de données possédant des valeurs pures, sans identité propre. Ses sous-classes, `MEnumeration` et `MPrimitive`, modélisent les énumérations et les types primitifs. L'attribut `literals` de la classe `MEnumeration` contient la liste des littéraux d'une énumération donnée.

Les diverses classes de particularités sont présentées à la figure 11.

La classe abstraite `MFeature` regroupe les attributs `visibility` et `ownerScope`, représentant respectivement la visibilité (publique, protégée ou privée) et la portée (instance ou structure) d'une particularité.

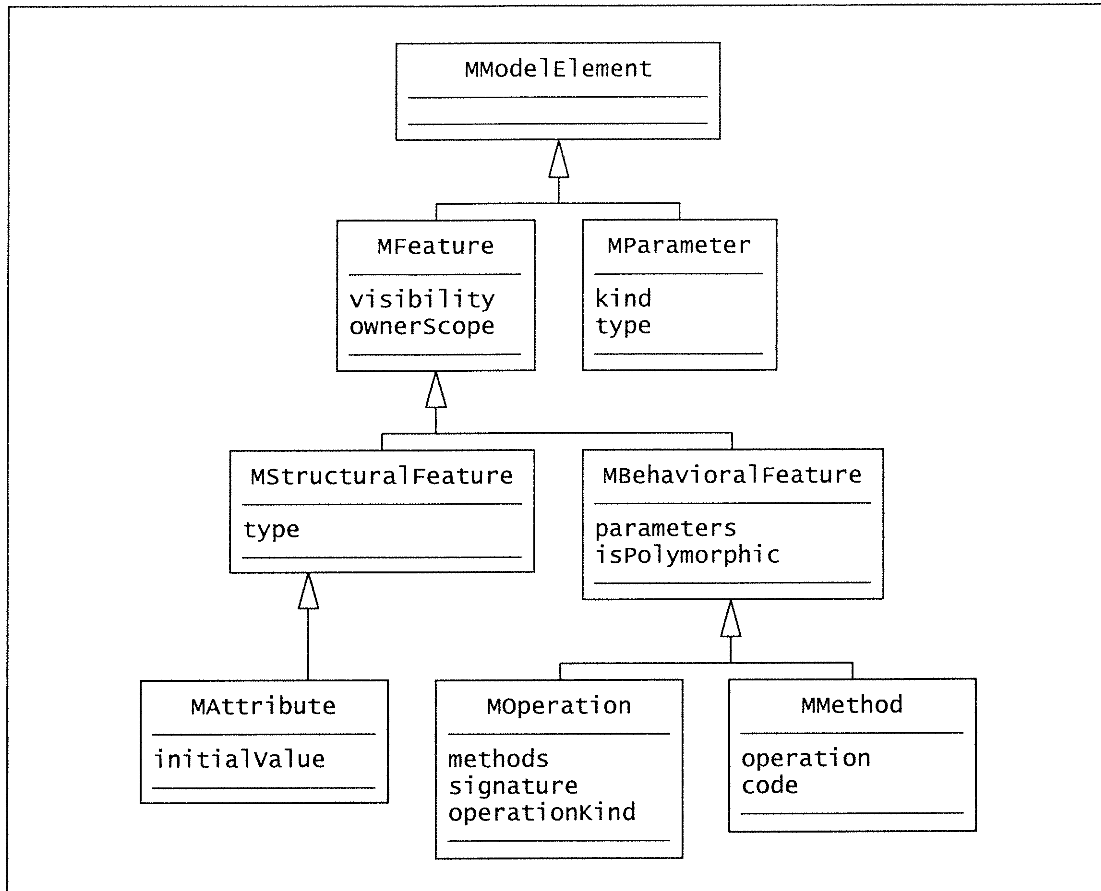

Figure 11. Classes de particularités du dépôt SPOOL/RCR

Les paramètres d'une opération et d'une méthode sont représentés à l'aide d'instances de la classe `MParameter`. L'attribut `kind` spécifie s'il s'agit d'un paramètre d'entrée ou d'entrée/sortie. L'attribut `type` pointe vers la structure déterminant le type du paramètre.

La classe abstraite `MStructuralFeature` modélise les composantes statiques des structures, c'est-à-dire les attributs. Son attribut `type` pointe vers la structure déterminant le type de la composante.

La classe `MAttribute`, utilisée pour représenter les attributs des structures, possède un seul attribut qui lui est propre, `initialValue` : celui-ci renferme la valeur d'initialisation de l'attribut concerné.

Les composantes dynamiques des structures sont représentées par la classe abstraite `MBehavioralFeature`. L'attribut `parameters` renferme la liste des paramètres associés à une composante; l'attribut booléen `isPolymorphic` détermine son caractère polymorphique.

La classe `MOperation` modélise les opérations, soit les services fournis par les objets. L'attribut `methods` renferme la liste des méthodes, implantations de l'opération courante. L'attribut `signature` représente le nom de l'opération avec ses paramètres. La catégorie de l'opération, parmi les valeurs standard, opérateur, constructeur et destructeur, est assignée à l'attribut `operationKind`.

Les méthodes, implantations des opérations, sont représentées par la classe `MMethod`. L'attribut `operation` permet de faire le lien vers l'opération implantée. Le bloc de code principal de la méthode est rattaché à l'instance courante par le biais de l'attribut `code`.

Deux classes du dépôt servent à modéliser les relations entre éléments structurels d'un système : il s'agit de `MGeneralization` et `MDependency`. Ces classes sont présentées à la figure 12.

La classe `MGeneralization` représente une relation entre un élément générique et un élément plus spécifique. Les éléments générique et spécifique faisant partie de la relation sont identifiés par les attributs `supertype` et `subtype`; la visibilité associée à la relation (publique, protégée ou privée) est affectée à l'attribut `visibility`.

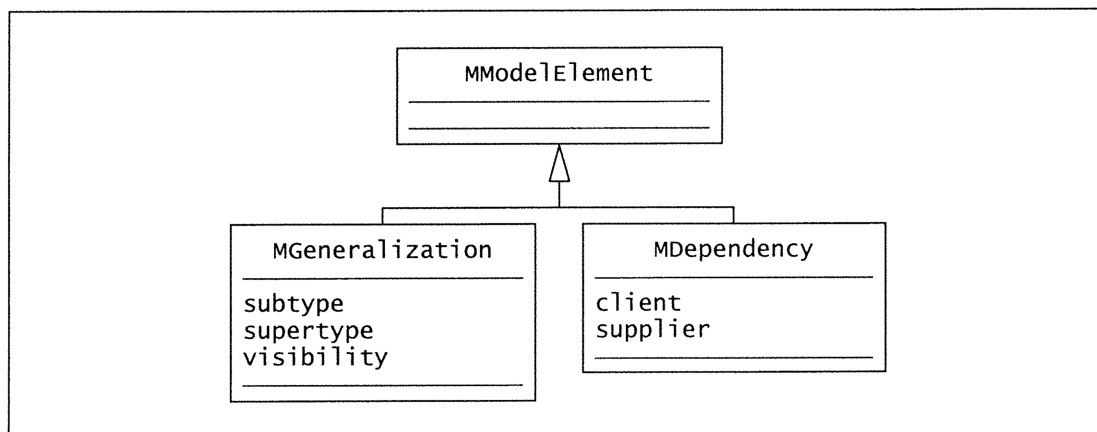

Figure 12. Classes relationnelles du dépôt SPOOL/RCR

Les dépendances sémantiques entre deux éléments d'un système sont modélisées à l'aide d'instances de la classe `MDependency`. Les éléments constituants de la relation sont assignés aux attributs `client` (le client) et `supplier` (le fournisseur), le client étant dépendant du fournisseur. La catégorie de la dépendance peut être précisée à l'aide d'une valeur étiquetée.

6.2.2 Classes issues de l'intégration du profil RCR

La figure 13 présente les classes modélisant les blocs de code dans le dépôt.

La classe générique `MRCRBlock` recueille les blocs de code ordinaires en plus de constituer la superclasse des classes `MRCRTryBlock` et `MRCRSynchronizedBlock`. L'attribut `code_language` permet de spécifier, au besoin, le langage utilisé pour la rédaction du bloc de code. L'attribut hérité `ownedElements` renferme la liste des énoncés constituant le bloc.

La classe `MRCRTryBlock` modélise les blocs `try` des langages C++ et Java. Les attributs `catch` et `finally` pointent vers des instances des classes `MRCRCatchStatement` et `MRCRFinallyStatement`, respectivement.

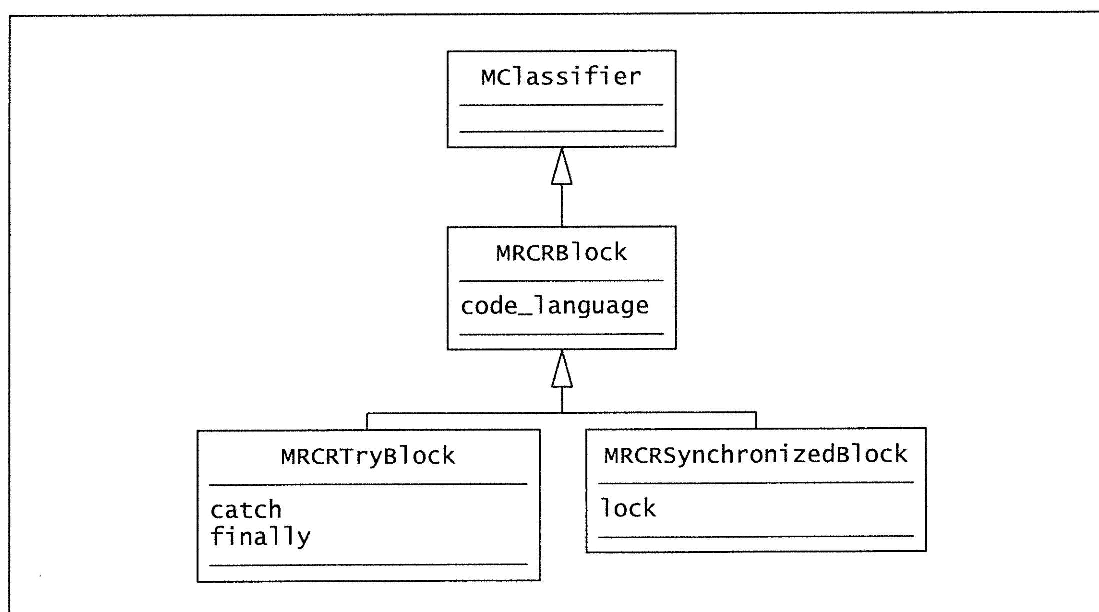

Figure 13. Classes de blocs de code du dépôt SPOOL/RCR

La classe MRCRSynchronizedBlock représente les blocs synchronized du langage Java. L'attribut lock modélise le verrou utilisé par le bloc sous la forme d'une instance de la classe MRCRImperativeExpression.

Les classes d'énoncés généraux du dépôt sont représentées dans la figure 14. Toutes les classes dérivent de la superclasse abstraite MRCRStatement; au sein de celle-ci, l'attribut exit représente, sous la forme d'une liste d'instances de la classe MRCRStatement, les instructions implicites de destruction d'objets locaux ajoutées par le processus de compilation.

La classe abstraite MRCRBranchStatement sert d'assise pour les sous-classes MRCRIfStatement et MRCRSwitchStatement. L'attribut value, de type MRCRImperativeExpression, constitue la valeur selon laquelle le branchement s'effectue; l'attribut branch, une liste d'instances de la classe MRCRStatement, représente les branches d'exécution appartenant à l'instruction.

La classe MRCRCaseStatement modélise les étiquettes case et leurs instructions associées, à l'intérieur des structures switch des langages C++ et Java. L'attribut value, de type MRCRImperativeExpression, représente la valeur associée à l'étiquette, tandis que l'attribut statement, une liste d'instances de la classe MRCRStatement, contient les énoncés rattachés à l'étiquette.

Les clauses catch des langages C++ et Java sont modélisées à l'aide d'instances de la classe MRCRCatchStatement. L'attribut exception modélise l'exception reçue sous la forme d'une instance de la classe MRCRDeclarationStatement. Les instructions exécutées dans le cadre de la clause sont représentées par l'attribut statement sous la forme d'une liste d'instances de la classe MRCRStatement.

La classe MRCRCompoundStatement est utilisée pour représenter l'inclusion d'un bloc de code à l'intérieur d'un autre. L'attribut block, de type MRCRBlock, constitue le bloc interne.

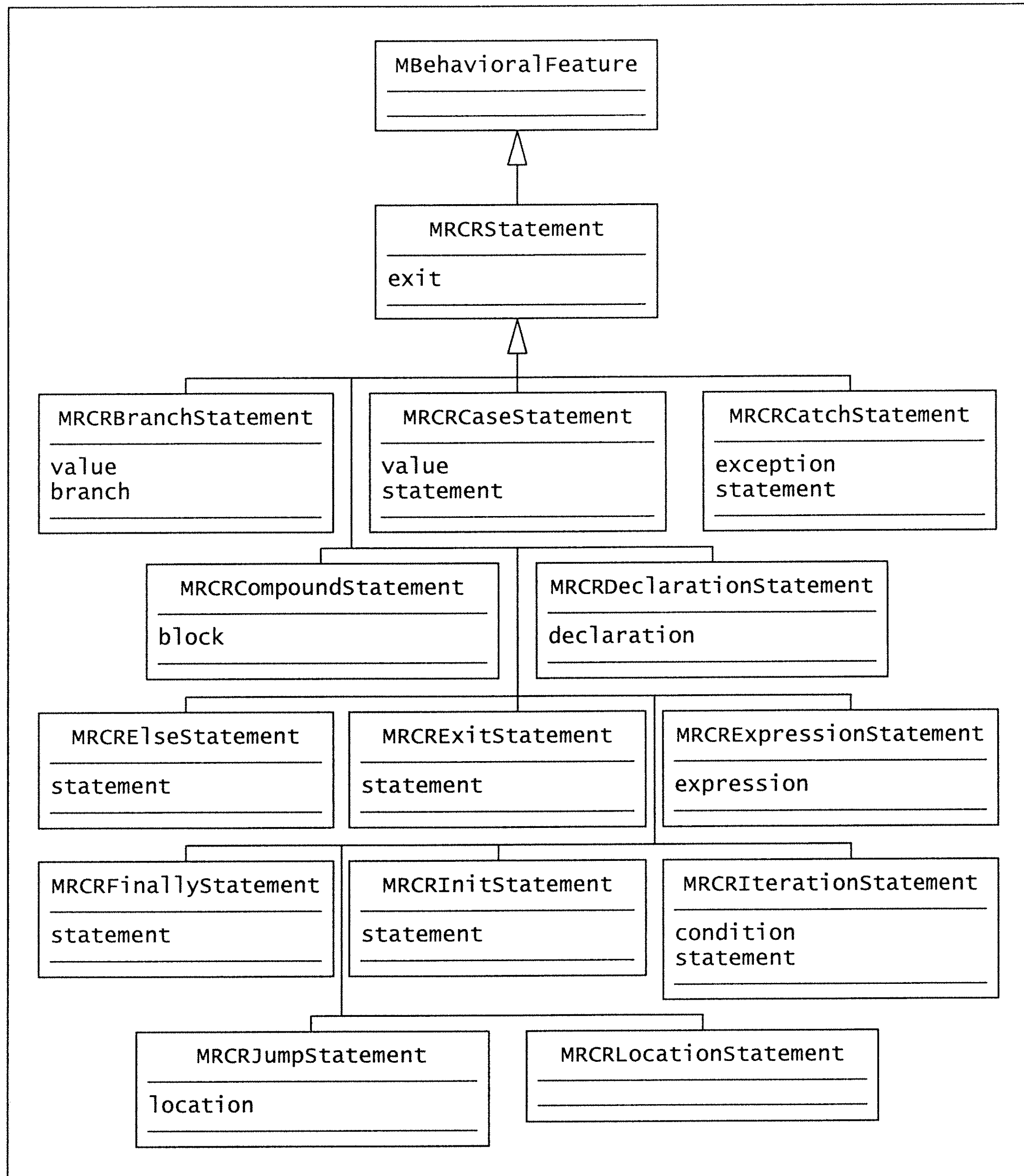

Figure 14. Classes d'énoncés généraux du dépôt SPOOL/RCR

La déclaration d'une variable est modélisée par une instance de la classe `MRCRDeclarationStatement`. L'attribut `declaration`, de type `MRCRVariable`, renferme la variable introduite.

La classe `MRCRElseStatement` sert à représenter les branches `else` des énoncés `if` dans les langages C++ et Java. Son attribut `statement`, une liste d'instances de la classe `MRCRStatement`, modélise les énoncés constituant la branche `else`.

Les instances des classes `MRCRExitStatement` introduisent des énoncés ayant été ajoutés implicitement par le processus de compilation et effectuant généralement la destruction d'objets locaux à la fin d'une instruction. Le lien vers ces énoncés est effectué par l'attribut `statement`, une liste d'instances de la classe `MRCRStatement`.

La classe `MRCRExpressionStatement` modélise des énoncés qui évaluent un ensemble d'expressions à l'aide de pas de calcul, pour l'obtention d'un résultat éventuellement assigné à une variable. L'expression sous-jacente est stockée dans l'attribut `expression` de type `MRCRImperativeExpression`.

Les clauses `finally` du langage Java sont représentées à l'aide d'instances de la classe `MRCRFinallyStatement`. L'attribut `statement` représente les instructions exécutées dans le cadre de la clause, sous la forme d'une liste d'instances de la classe `MRCRStatement`.

La classe `MRCRInitStatement` modélise les énoncés implicitement insérés par le processus de compilation avant le premier énoncé d'un bloc de code. L'attribut `statement` contient ces énoncés dans une liste d'instances de la classe `MRCRStatement`.

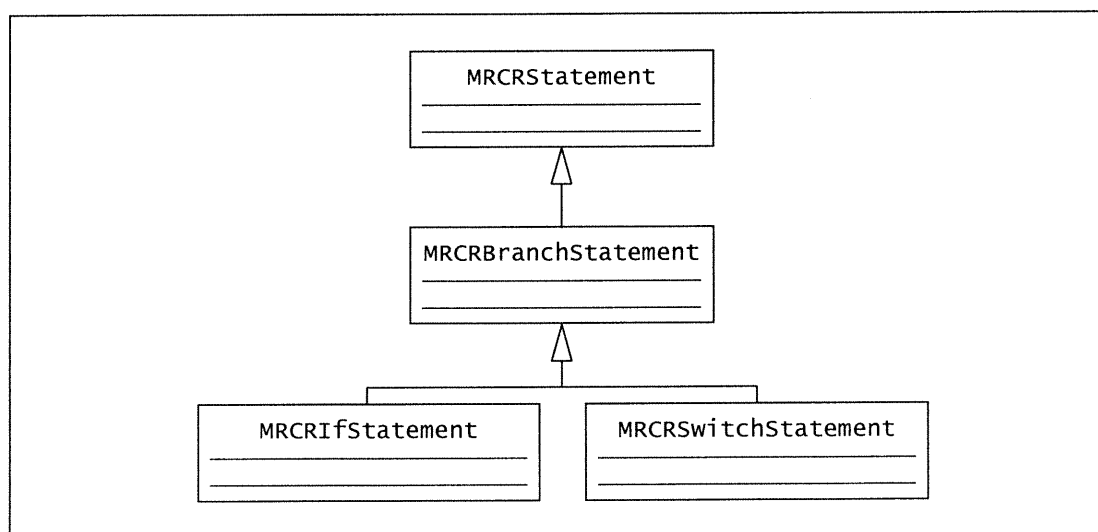

Figure 15. Classes d'énoncés de branchement du dépôt SPOOL/RCR

La classe abstraite `MRCRIterationStatement` sert d'assise pour les sous-classes `MRCRDoStatement`, `MRCRForStatement` et `MRCRWhileStatement`. Ses attributs `condition` et `statement` modélisent les composantes communes aux trois énoncés d'itération. L'attribut `condition`, de type `MRCRImperativeExpression`, représente la condition d'arrêt des énoncés; l'attribut `statement`, une liste d'instances de la classe `MRCRStatement`, renferme le bloc de code soumis au processus d'itération.

La classe abstraite `MRCRJumpStatement`, représentant la famille des énoncés de saut, engendre les sous-classes `MRCRBreakStatement`, `MRCRContinueStatement`, `MRCRGotoStatement`, `MRCRReturnStatement` et `MRCRThrowStatement`. Son attribut `location`, de type `MRCRLocationStatement`, permet de spécifier la destination d'un saut.

Les instances de la classe `MRCRLocationStatement` représentent les étiquettes de saut présentes dans le code. La classe ne possède pas d'attribut particulier.

La figure 15 présente les classes spécifiques du dépôt modélisant les énoncés de branchement. La classe `MRCRIfStatement` représente les énoncés `if` des langages C++ et Java, tandis que la classe `MRCRSwitchStatement` représente les énoncés `switch`. Ces deux classes obtiennent leurs attributs par héritage de leurs superclasses.

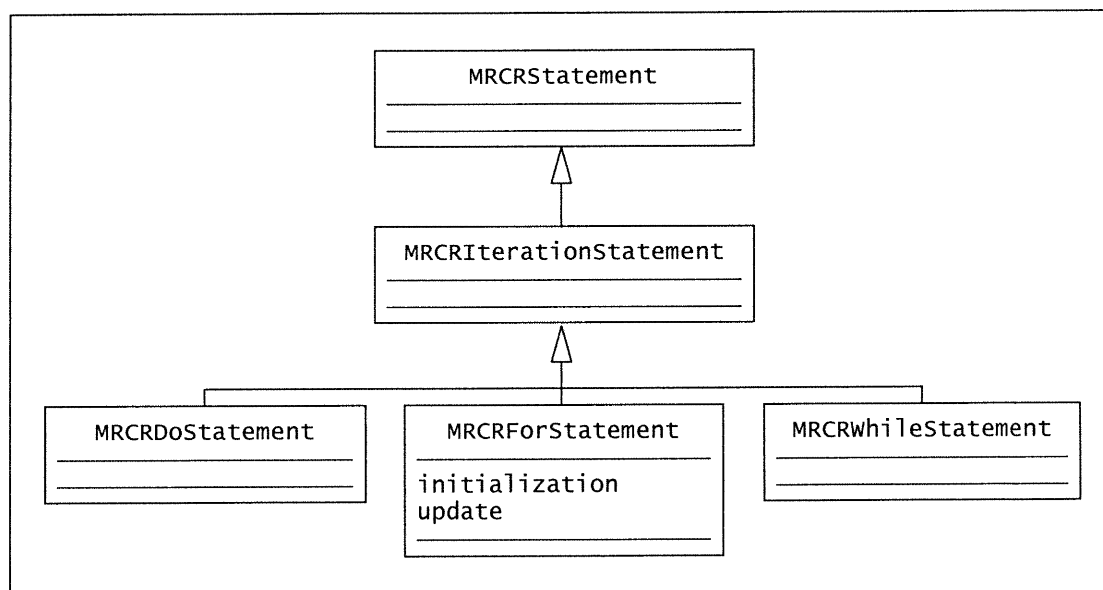

Figure 16. Classes d'énoncés d'itération du dépôt SPOOL/RCR

Les classes spécifiques représentant les énoncés d'itération sont présentées à la figure 16. Les classes `MRCRDoStatement`, `MRCRForStatement` et `MRCRWhileStatement` modélisent respectivement les énoncés `do ... while`, `for` et `while` des langages C++ et Java. La classe `MRCRForStatement` possède deux attributs spécifiques : `initialization`, une liste d'instances de la classe `MRCRStatement`, représente les instructions exécutées avant la première itération de la boucle; `update`, une liste d'instances de la classe `MRCRImperativeExpression`, capte les instructions exécutées après chaque itération.

La figure 17 regroupe les classes spécifiques représentant les énoncés de saut. Les classes `MRCRBreakStatement`, `MRCRContinueStatement`, `MRCRGotoStatement`, `MRCRReturnStatement` et `MRCRThrowStatement` sont respectivement associées aux énoncés `break`, `continue`, `goto`, `return` et `throw` des langages C++ et Java. L'attribut facultatif `expression` de la classe `MRCRReturnStatement` est utilisé pour représenter la valeur retournée par l'instruction. L'attribut `exception` de la classe `MRCRThrowStatement` modélise l'objet lancé à titre d'exception.

Les classes du dépôt modélisant les expressions sont présentées à la figure 18.

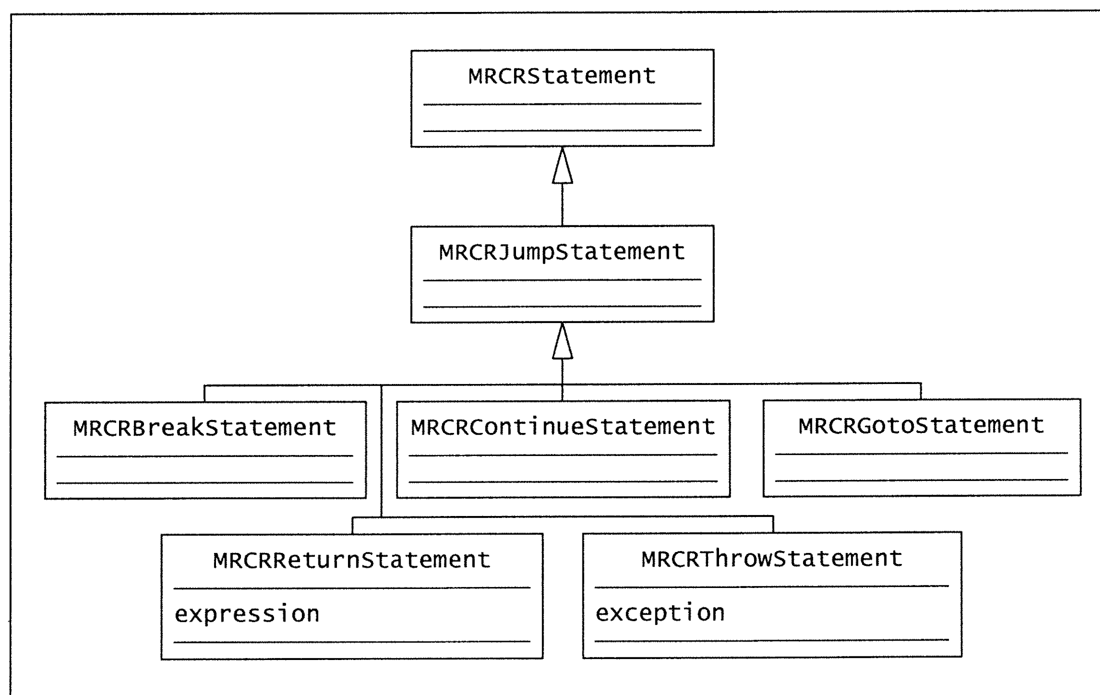

Figure 17. Classes d'énoncés de saut du dépôt SPOOL/RCR

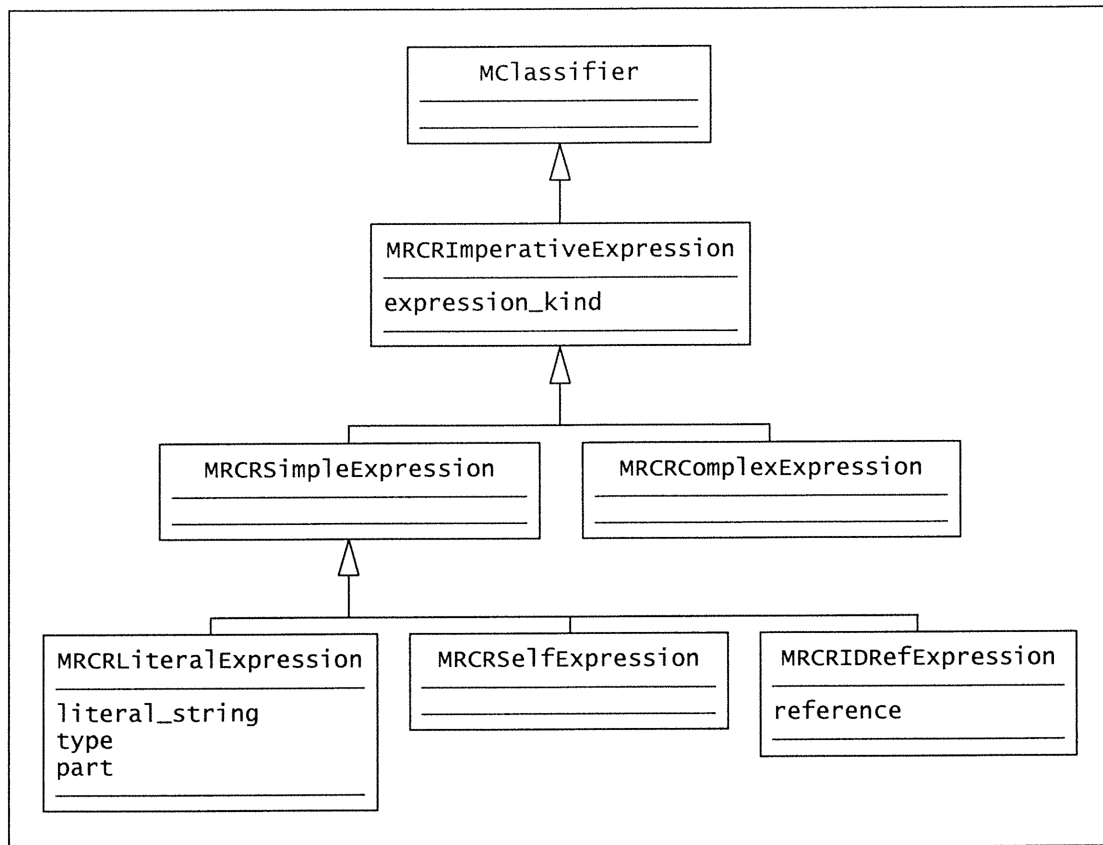

Figure 18. Classes d'expressions du dépôt SPOOL/RCR

La classe abstraite `MRCRImperativeExpression` forme la base de cette hiérarchie. Son attribut `expression_kind` permet de raffiner, au sein des sous-classes, les expressions modélisées.

La classe abstraite `MRCRSimpleExpression` sert d'assise pour les sous-classes `MRCRLiteralExpression`, `MRCRSelfExpression` et `MRCRIDRefExpression`. Elle ne possède aucun attribut particulier.

La classe `MRCRComplexExpression` modélise les expressions complexes composées d'opérateurs et d'appels à des fonctions. L'attribut hérité `ownedElements` contient la hiérarchie des pas d'évaluation constituant l'expression complexe. Les pas d'évaluation sont représentés par des instances de la classe `MRCRStep`.

Les littéraux sont représentés par des instances de la classe `MRCRLiteralExpression`. L'attribut `literal_string` renferme la chaîne de caractères représentant le littéral tandis

que l'attribut type, de type MClassifier, précise le type du littéral. Pour la représentation d'un littéral complexe utilisé pour l'initialisation d'un tableau, l'attribut part, une liste d'instances de la classe MRCRLiteralExpression, renferme les composantes d'un niveau déterminé du littéral complexe.

La classe MRCRSelfExpression sert à modéliser une référence à l'objet courant, tel qu'effectué par le mot-clé this dans les langages C++ et Java, ou une référence à la superclasse de l'objet courant, comme le fait le mot-clé super de Java. L'attribut hérité expression_kind permet de discriminer les deux expressions.

La classe MRCRIDRefExpression permet d'effectuer un lien entre l'utilisation d'une variable ou d'une fonction et sa définition au sein du système. L'attribut reference constitue la référence vers la définition de la composante utilisée.

La figure 19 présente la hiérarchie complète des classes représentant les pas d'évaluation dans le dépôt.

La classe abstraite MRCRStep, située au sommet de la hiérarchie, transmet à l'ensemble de ses sous-classes l'attribut step_kind. Ce dernier permet de préciser, lorsque cela est requis, le type de pas d'évaluation.

Les pas d'accès à une composante d'une entité sont modélisés à l'aide des instances de la classe MRCRAccessStep. L'attribut accessStep_kind précise le type d'accès (par exemple, ".", "->" ou "[]"). Les attributs composite et part, tous deux de type MRCRImperativeExpression, représentent respectivement l'entité et la composante accédées.

La classe MRCRAssignStep est utilisée pour représenter les pas d'assignation d'une valeur à une entité. L'attribut assignStep_kind détermine le type de pas d'assignation (par exemple, "=", "+=", "*=", etc.). Les attributs target et value, de type MRCRImperativeExpression, représentent l'entité assignée et la valeur affectée.

Les pas de calcul primitifs sont modélisés grâce à la classe MRCRCalculateStep. Le type de pas de calcul est représenté par l'attribut calculateStep_kind (par exemple, "+",

"&&", "post++", etc.). L'attribut operand renferme les opérandes requis par le pas de calcul, sous la forme d'une liste d'instances de la classe MRCRImperativeExpression.

La classe MRCRCallStep représente un appel à une fonction. L'attribut facultatif callStep_kind permet de représenter une particularité relative à l'invocation. L'attribut operation, de type MRCRImperativeExpression, effectue le lien vers la fonction sollicitée. Les arguments fournis lors de l'appel de la fonction sont contenus dans l'attribut argument, une liste d'instances de la classe MRCRImperativeExpression.

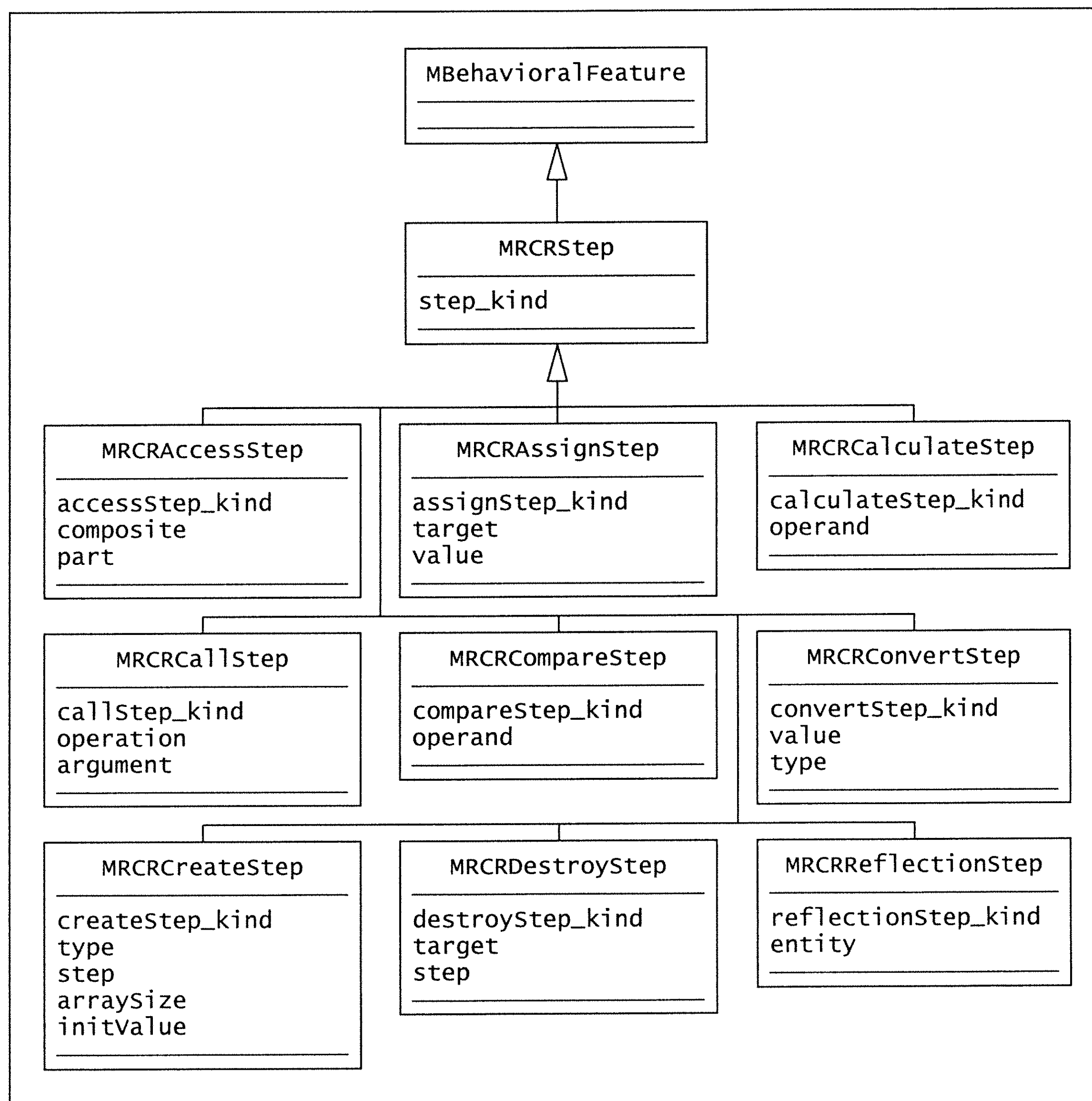

Figure 19. Classes de pas d'évaluation du dépôt SPOOL/RCR

Les pas de comparaison sont modélisés à l'aide d'instances de la classe MRCRCompareStep. L'attribut compareStep_kind identifie le type de comparaison effectuée (par exemple, "==", "<", "!=", etc.). L'attribut operand, une liste d'instances de la classe MRCRImperativeExpression, renferme les deux opérandes requis pour le pas de comparaison.

La classe MRCRConvertStep est utilisée pour modéliser les pas de conversion de type, qu'ils soient implicites ou explicites. L'attribut convertStep_kind peut être utilisé pour apporter une précision concernant le pas de conversion. L'attribut value, de type MRCRImperativeExpression, renferme la valeur à convertir; l'attribut type, de type MClassifier, précise le type de l'expression résultant de la conversion.

La classe MRCRCreateStep représente les pas de création d'instance. L'attribut createStep_kind précise l'énoncé utilisé dans le code pour la création (par exemple, "new", "new[]", etc.). L'attribut type, de type MClassifier, spécifie le type de l'instance créée. L'attribut step, de type MRCRImperativeExpression, sert à préciser le constructeur utilisé pour l'instanciation et l'initialisation de l'objet. Lors de la création d'un tableau, l'attribut arraySize, de type MRCRImperativeExpression, renferme l'expression spécifiant les dimensions du tableau instancié; l'attribut initValue, de type MRCRLiteralExpression, permet d'effectuer un lien vers le littéral complexe assigné au tableau nouvellement créé.

Les pas de destruction d'objet sont modélisés grâce à la classe MRCRDestroyStep. L'attribut optionnel destroyStep_kind spécifie l'instruction du code utilisée pour la destruction. L'objet détruit lors de l'opération est représenté par l'attribut target de type MRCRImperativeExpression. Le destructeur sollicité au cours de l'opération est identifié par l'attribut step de type MRCRImperativeExpression.

La classe MRCRReflectionStep représente les pas de réflexion, c'est-à-dire les opérateurs qui retournent une caractéristique de l'implantation physique d'une donnée. L'attribut reflectionStep_kind permet de spécifier l'opérateur invoqué (par exemple, "address_of", "dereference", etc.). L'entité visée par l'opérateur est représentée par l'attribut entity de type MRCRImperativeExpression.

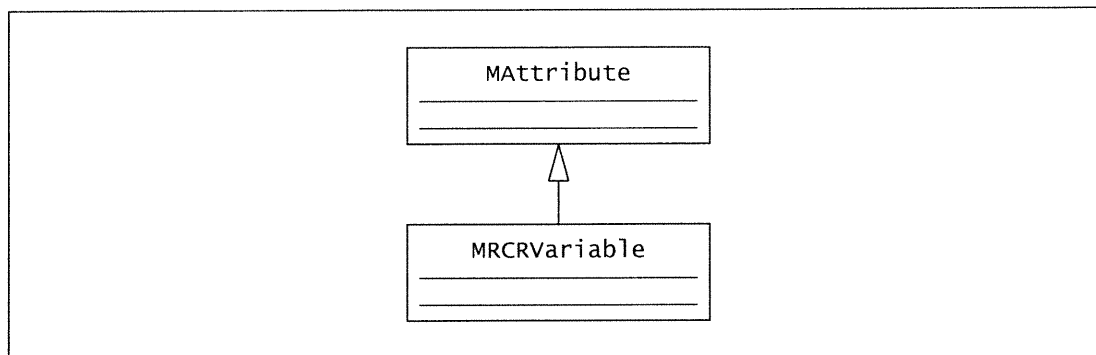

Figure 20. Classe de modélisation de variable du dépôt SPOOL/RCR

Le dépôt possède une seule classe pour la modélisation des variables : il s'agit de la classe `MRCRVariable`, présentée à la figure 20. Cette classe hérite de l'attribut `initialValue` par l'entremise de sa superclasse `MAttribute`.

6.3 Stratégies de conception de l'importateur XMI/RCR

Cette section présente sommairement les stratégies de conception de l'importateur XMI/RCR, une application conçue pour lire les fichiers XMI de base et XMI/RCR et pour créer les objets correspondants dans le dépôt de données SPOOL/RCR. L'architecture d'une telle application étant relativement simple, cette section exposera brièvement les lignes directrices conduisant à sa réalisation technique.

L'importateur XMI/RCR est supporté par un parseur XML tel *XML Parser for Java* [XML4J]. Le parseur effectue la lecture séquentielle des fichiers XMI de base et XMI/RCR et invoque, pour chaque balise reconnue, une sous-routine de traitement correspondante. Le parseur fournit à la sous-routine les attributs affectés à la balise sous la forme d'une liste de paires nom d'attribut – valeur de l'attribut. Il existe une sous-routine de traitement pour chaque balise ouvrante et chaque balise fermante pouvant être rencontrée dans les fichiers à traiter.

L'importateur possède une pile interne qui contient à tout moment le contexte dans lequel chaque balise est rencontrée. Après avoir été traitée, chaque balise ouvrante est empilée avec ses attributs dans la pile interne. Chaque balise fermante supprime

l'élément placé au-dessus de la pile; l'élément retiré doit toujours être la balise ouvrante assortie à la balise fermante couramment repérée.

Le traitement d'une balise ouvrante consiste à créer l'objet correspondant dans le dépôt de données, en prenant soin d'initialiser ses attributs avec les valeurs fournies par le parseur. L'objet est créé en fonction du contexte dans lequel la balise est rencontrée : à cet effet, l'importateur consulte le contenu de la pile interne jusqu'à la profondeur requise. C'est ainsi qu'un fichier est lié au répertoire qui le contient, qu'un attribut est associé à sa structure propriétaire, qu'un énoncé est enregistré à l'intérieur du bloc de code englobant, et qu'un pas de calcul est placé dans la hiérarchie d'étapes d'évaluation d'une expression donnée.

L'importateur possède idéalement un mécanisme de journalisation qui permet d'enregistrer les erreurs survenues lors du processus d'importation : par exemple, une balise fermante non associée à la balise ouvrante présente au sommet de la pile interne, un attribut à caractère obligatoire omis pour une balise, une balise invalide dans le contexte représenté par le contenu de la pile, et ainsi de suite.

L'importateur possède également un mécanisme général d'extraction des fichiers à traiter de l'archive `.zip` dans laquelle ils sont emmagasinés; il doit être en mesure de déterminer dans quel ordre les fichiers initialement compressés doivent être examinés, puis coordonner l'ouverture, le traitement et la fermeture de ces mêmes fichiers.

6.4 Conclusion

Ce chapitre a présenté l'architecture du dépôt SPOOL/RCR, destiné à recevoir le contenu des fichiers XMI de base et XMI/RCR par l'entremise d'un processus d'importation.

Dans un premier temps, une description de l'approche retenue pour la conception du dépôt SPOOL/RCR a été introduite. Par la suite, les classes composant le dépôt ont été présentées. Celles-ci se regroupent en deux catégories distinctes : les classes fondamentales et les classes issues de l'intégration du profil RCR.

En dernier lieu, des stratégies pour la conception de l'importateur XMI/RCR ont été exposées de façon succincte.

Le chapitre suivant présente les résultats des expérimentations conduites avec les fichiers XMI de base et XMI/RCR, générés à l'aide des scripts d'exportation.

7

Discussions

Le chapitre 7, *« Discussions »*, présente les résultats des expérimentations conduites avec les scripts de génération des fichiers XMI de base et XMI/RCR, en termes d'exportation de systèmes et d'importation de fichiers dans le dépôt de l'atelier SPOOL. Les considérations relatives à la mise en œuvre du profil RCR sont abordées; le chapitre clôt en décrivant les objectifs atteints et les leçons retenues de la conception des scripts d'exportation.

Ce chapitre contient les sections suivantes :

Section	*Page*
7.1 Ajustements au profil RCR	64
7.2 Utilisation littérale des balises RCR	69
7.3 Tests de génération XMI et XMI/RCR	71
7.4 Tests d'importation dans SPOOL des fichiers XMI de base	73
7.5 Impacts de l'utilisation d'une particule d'indentation	75
7.6 Comparaison de la taille des fichiers XMI générés par deux approches différentes	77
7.7 Objectifs atteints	79
7.8 Leçons retenues	83
7.9 Conclusion	85

7.1 Ajustements au profil RCR

Cette section présente les ajustements effectués au profil RCR afin de faciliter sa mise en œuvre. Les modifications apportées permettent une modélisation appropriée des

informations pertinentes conservées dans les AST de DISCOVER et stockées dans le dépôt de données SPOOL/RCR.

7.1.1 Dépendances

Deux nouvelles dépendances sont créées afin de modéliser les types de liens requis par les modifications apportées au profil.

La dépendance `RCR.Dep.Location` permet d'effectuer un lien vers un métaélément `RCR.LocationStatement` dans le cadre du métaélément `RCR.JumpStatement` et de ses métaéléments sous-classés.

La dépendance `RCR.Dep.Literal` est implantée au sein du métaélément `RCR.LiteralExpression` pour participer à la modélisation des littéraux complexes utilisés pour l'initialisation des tableaux. Cette dépendance permet aussi d'effectuer le lien vers le littéral complexe d'initialisation dans le cadre du métaélément `RCR.CreateStep`, lorsque ce dernier est utilisé pour modéliser l'instanciation d'un tableau.

Stéréotype	**Type de *supplier* exigé**
`RCR.Dep.Location`	`RCR.LocationStatement`
`RCR.Dep.Literal`	`RCR.LiteralExpression`

7.1.2 Énoncés

Métaélément RCR.Statement

La valeur étiquetée `statement_label` du métaélément `RCR.Statement` n'est pas utilisée, puisque les étiquettes de saut sont modélisées à l'aide du nouveau métaélément `RCR.LocationStatement`.

De plus, une dépendance `exit` est ajoutée au métaélément pour représenter les instructions implicites de destruction d'objets locaux, ajoutées par le processus de compilation.

Stéréotype	**RCR.Statement**
Superclasse	UML.Foundation.Core.BehavioralFeature
Valeurs étiquetées	
Dépendances	1. exit : RCR.Dep.Statement
Contraintes	
Notes	

Métaélément RCR.LocationStatement

Le nouveau métaélément `RCR.LocationStatement` modélise les étiquettes de saut présentes dans le code comme des instructions à part entière. Les étiquettes sont d'ailleurs représentées en tant qu'instructions dans les AST de DISCOVER. Le métaélément nécessite la création d'une nouvelle dépendance, `RCR.Dep.Location`.

Stéréotype	**RCR.LocationStatement**
Superclasse	RCR.Statement
Valeurs étiquetées	
Dépendances	
Contraintes	
Notes	

Métaélément RCR.JumpStatement

La dépendance `location` à laquelle participe le métaélément `RCR.JumpStatement` est de type `RCR.Dep.Location`, qui pointe vers le nouveau métaélément `RCR.LocationStatement`.

Stéréotype	**RCR.JumpStatement**
Superclasse	RCR.Statement
Valeurs étiquetées	
Dépendances	1. location : RCR.Dep.Location
Contraintes	
Notes	

Métaélément RCR.ForStatement

Le métaélément `RCR.ForStatement` voit sa dépendance `declaration` renommée `initialization`; cette dernière est de type `RCR.Dep.Statement`.

Stéréotype	**RCR.ForStatement**
Superclasse	RCR.IterationStatement
Valeurs étiquetées	
Dépendances	1. initialization : RCR.Dep.Statement 2. update : RCR.Dep.Expression
Contraintes	
Notes	

Métaélément RCR.CatchStatement

Le type de la dépendance `exception` du métaélément `RCR.CatchStatement` est modifié pour `RCR.Dep.DeclarationStatement`.

Stéréotype	**RCR.CatchStatement**
Superclasse	RCR.Statement
Valeurs étiquetées	
Dépendances	1. exception : RCR.Dep.DeclarationStatement 2. statement : RCR.Dep.Statement
Contraintes	
Notes	

7.1.3 Expressions

Métaélément RCR.LiteralExpression

Une dépendance `part` est ajoutée au métaélément `RCR.LiteralExpression` afin de permettre la modélisation des littéraux complexes utilisés pour l'initialisation des tableaux.

Stéréotype	**RCR.LiteralExpression**
Superclasse	RCR.SimpleExpression
Valeurs étiquetées	1. literal_string
Dépendances	1. type : RCR.Dep.Type 2. part : RCR.Dep.Literal
Contraintes	
Notes	

7.1.4 Pas d'évaluation

Métaélément RCR.CreateStep

Le métaélément `RCR.CreateStep` subit un changement en profondeur. Il possède maintenant quatre dépendances : `type`, `step`, `arraySize` et `initValue`.

La dépendance `type`, inchangée, spécifie le type de l'instance créée.

La dépendance optionnelle `step` spécifie le constructeur utilisé pour l'instanciation et l'initialisation de l'objet. L'expression sous-jacente est généralement modélisée par un unique métaélément `RCR.CallStep`. Cette dépendance n'est pas utilisée pour la création de tableaux.

La dépendance optionnelle `arraySize`, utilisée pour l'instanciation de tableaux, spécifie l'expression représentant les dimensions du tableau à créer.

La dépendance optionnelle `initValue` pointe vers le littéral complexe utilisé pour l'initialisation du tableau nouvellement créé.

Stéréotype	**RCR.CreateStep**
Superclasse	RCR.Step
Valeurs étiquetées	1. createStep_kind
Dépendances	1. type : RCR.Dep.Type 2. step : RCR.Dep.Expression 3. arraySize : RCR.Dep.Expression 4. initValue : RCR.Dep.Literal
Contraintes	
Notes	

Métaélément RCR.DestroyStep

Une nouvelle dépendance, `step`, ajoutée au métaélément `RCR.DestroyStep`, permet d'effectuer un lien vers le destructeur sollicité lors de l'étape. Ce dernier est généralement modélisé par un unique métaélément `RCR.CallStep`.

Stéréotype	**RCR.DestroyStep**
Superclasse	RCR.Step
Valeurs étiquetées	1. destroyStep_kind
Dépendances	1. target : RCR.Dep.Expression 2. step : RCR.Dep.Expression
Contraintes	
Notes	

7.2 Utilisation littérale des balises RCR

Lors de la génération des fichiers XMI/RCR, les différents métaéléments RCR sont littéralement émis à l'aide de balises RCR, et non pas à l'aide de balises XMI affublées des identifiants des stéréotypes correspondants. La section présente motive ce choix initialement effectué pour le développement des scripts d'exportation.

Une structure XMI/RCR représentant un appel à une fonction objet recevant deux arguments est présentée ci-dessous :

```
<RCR.ExpressionStatement>
  <RCR.ExpressionStatement.expression>
    <RCR.ComplexExpression>
      <UML:Foundation.Core.Classifier.feature>
        <RCR.CallStep>
          <RCR.CallStep.operation>
            <RCR.IDRefExpression>
              <RCR.IDRefExpression.reference>
                <UML:Foundation.Core.Operation
                  idref="identifiant_unique_de_la_fonction"/>
              </RCR.IDRefExpression.reference>
            </RCR.IDRefExpression>
          </RCR.CallStep.operation>
          <RCR.CallStep.argument>
            <RCR.SelfExpression expression_kind="super">
            </RCR.SelfExpression>
            <RCR.IDRefExpression>.
              <RCR.IDRefExpression.reference>
                <RCR.Variable
                  idref="identifiant_unique_de_la_variable"/>
              </RCR.IDRefExpression.reference>
```

```
            </RCR.IDRefExpression>
          </RCR.CallStep.argument>
        </RCR.CallStep>
      </UML:Foundation.Core.Classifier.feature>
    </RCR.ComplexExpression>
  </RCR.ExpressionStatement.expression>
</RCR.ExpressionStatement>
```

Étant donné les métaéléments RCR suivants et leurs identifiants de stéréotypes d'éléments XMI :

Métaélément RCR	Identifiant de stéréotype d'élément XMI
RCR.ComplexExpression	MM.5010
RCR.IDRefExpression	MM.5020
RCR.SelfExpression	MM.5030
RCR.Variable	MM.5500
RCR.ExpressionStatement	MM.6010
RCR.CallStep	MM.6020
RCR.ExpressionStatement.expression	MM.7010
RCR.CallStep.operation	MM.7020
RCR.CallStep.argument	MM.7030
RCR.IDRefExpression.reference	MM.7040

La même structure XMI/RCR s'exporte de la façon suivante, en inscrivant les éléments RCR sous la forme d'éléments XMI stéréotypés :

```
<UML:Foundation.Core.BehavioralFeature stereotype="MM.6010">
  <UML:Foundation.Core.Dependency stereotype="MM.7010">
    <UML:Foundation.Core.Classifier stereotype="MM.5010">
      <UML:Foundation.Core.Classifier.feature>
        <UML:Foundation.Core.BehavioralFeature
          stereotype="MM.6020">
          <UML:Foundation.Core.Dependency stereotype="MM.7020">
            <UML:Foundation.Core.Classifier stereotype="MM.5020">
              <UML:Foundation.Core.Dependency
                stereotype="MM.7040">
                <UML:Foundation.Core.Operation
                  idref="identifiant_unique_de_la_fonction"/>
              </UML:Foundation.Core.Dependency>
            </UML:Foundation.Core.Classifier>
          </UML:Foundation.Core.Dependency>
          <UML:Foundation.Core.Dependency stereotype="MM.7030">
            <UML:Foundation.Core.Classifier stereotype="MM.5030">
              <UML:ModelElement.stereotype>
                <UML:Stereotype>
                  <UML:Stereotype.requiredTag>
                    <UML:TaggedValue
                      tag="expression_kind"
                      value="super"/>
                  </UML:Stereotype.requiredTag>
                </UML:Stereotype>
              </UML:ModelElement.stereotype>
```

```
                </UML:Foundation.Core.Classifier>
                <UML:Foundation.Core.Classifier stereotype="MM.5020">
                  <UML:Foundation.Core.Dependency
                    stereotype="MM.7040">
                    <UML:Foundation.Core.Attribute
                      idref="identifiant_unique_de_la_variable"
                      stereotype="MM.5500"/>
                  </UML:Foundation.Core.Dependency>
                </UML:Foundation.Core.Classifier>
              </UML:Foundation.Core.Dependency>
            </UML:Foundation.Core.BehavioralFeature>
          </UML:Foundation.Core.Classifier.feature>
        </UML:Foundation.Core.Classifier>
      </UML:Foundation.Core.Dependency>
    </UML:Foundation.Core.BehavioralFeature>
```

Il apparaît *de visu* que la structure utilisant directement les balises RCR est légèrement plus compacte : en effet, elle représente les valeurs étiquetées des éléments stéréotypés d'une façon plus succincte. Mais surtout, elle offre une lisibilité supérieure, le lecteur n'ayant pas à effectuer mentalement la correspondance entre les identifiants des stéréotypes et les métaéléments RCR.

Les fichiers XMI/RCR produits doivent cependant être traités à l'aide d'un importateur conçu pour lire les éléments XMI standard et offrir un mécanisme de gestion de profil afin d'effectuer la correspondance entre les balises RCR et les éléments XMI stéréotypés, ou encore avec un importateur conçu sur mesure pour traiter tous les types de balises que les fichiers XMI/RCR peuvent renfermer. Ceci constitue un désavantage mineur, compte tenu du gain considérable obtenu par la production de fichiers XMI/RCR qui peuvent être facilement examinés par un être humain.

7.3 Tests de génération XMI et XMI/RCR

Afin de valider les extrants produits par l'exécution des scripts d'exportation, quatre systèmes furent choisis et traités à l'aide de DISCOVER : ET++ [ET++_1993] (rédigé en C++), Apache [Apache] (C), SPOOL [SPOOL] (Java) et jKitGo [jKitGo] (Java). Les scripts d'exportation ont été exécutés à partir de l'environnement DISCOVER 7.3, sur une station de travail Sun dotée de quatre microprocesseurs UltraSPARC II 450 MHz, de 4 Go de mémoire vive et du système d'exploitation SunOS 5.8. Les scripts ont produit des fichiers XMI de base et XMI/RCR valides. Les statistiques concernant les fichiers générés sont présentées dans le tableau II.

	ET++ *(C++)*	Apache *(C)*	SPOOL *(Java)*	jKitGo *(Java)*
Fichiers source				
Nombre de fichiers	500	114	689	380
Taille des fichiers (Ko)	1 961	1 793	2 276	2 071
Nombre de classes	619	84	725	345
Nombre d'attributs	1 618	218	461	372
Nombre de fonctions	7 809	805	4 638	4 604
Fichiers XMI de base générés				
Taille des fichiers (Ko)	11 060	1 467	6 993	6 315
Temps de génération (s)	505	28	261	199
Débit (Ko/s)	21,9	52,4	26,8	31,7
Ratio taille des fichiers XMI / taille des fichiers source	5,6	0,8	3,1	3,0
Taille des fichiers compressés (Ko)	510	71	286	251
Ratio de compression	4,6 %	4,8 %	4,1 %	4,0 %
Fichiers XMI/RCR générés				
Taille des fichiers (Ko)	195 666	57 610	63 336	41 187
Temps de génération (s)	1 050	371	379	260
Débit (Ko/s)	186,3	155,3	167,1	158,4
Ratio taille des fichiers XMI/RCR / taille des fichiers source	99,8	32,1	27,8	19,9
Taille des fichiers compressés (Ko)	6 004	1 650	2 117	1 373
Ratio de compression	3,1 %	2,9 %	3,3 %	3,3 %
Tous fichiers générés				
Taille des fichiers (Ko)	206 726	59 077	70 329	47 502
Temps de génération (s)	1 555	399	640	459
Débit (Ko/s)	132,9	148,1	109,9	103,5
Ratio taille de tous les fichiers / taille des fichiers source	105,4	32,9	30,9	22,9
Taille des fichiers compressés (Ko)	6 514	1 721	2 403	1 624
Ratio de compression	3,2 %	2,9 %	3,4 %	3,4 %

Tableau II. Statistiques de génération des fichiers XMI de base et XMI/RCR

À la lecture des résultats obtenus, il est possible d'établir les constats suivants :

- les ratios taille des fichiers générés / taille des fichiers source sont variables et dépendent principalement du style de programmation utilisé pour la confection du code source. L'inspection visuelle des fichiers source du système Apache fait ressortir que de nombreux fichiers comprennent essentiellement des déclarations destinées au préprocesseur, lesquelles ne génèrent pas d'informations exportables dans les AST de DISCOVER, contrairement au code source du système ET++. De plus, les fichiers source du système SPOOL sont généreusement commentés, par opposition à ceux du système ET++ : les commentaires n'étant pas exportés dans les fichiers XMI de base et XMI/RCR, leur présence en grand nombre fait nécessairement chuter ces ratios;

- un système comprenant peu de classes, d'attributs et de fonctions, tel Apache, possédera des fichiers XMI de base peu volumineux par rapport à ses fichiers XMI/RCR puisque la taille des fichiers XMI de base est proportionnelle au nombre de classes, d'attributs et de fonctions présents dans le système;

- les fichiers générés comportent beaucoup de redondance et présentent un facteur de compression élevé lorsqu'ils sont stockés dans des archives `.zip` standard : en effet, l'algorithme de compression réussit à réduire la taille des fichiers par un facteur variant entre 20 et 35. Les fichiers XMI/RCR, présentant un plus haut niveau de redondance que les fichiers XMI de base, offrent un facteur de compression supérieur.

7.4 Tests d'importation dans SPOOL des fichiers XMI de base

À la suite de la génération des fichiers XMI de base et XMI/RCR, des bases de données SPOOL vierges ont été créées, une pour chacun des systèmes exportés, à l'aide du système de gestion de bases de données orientées objet POET 6.1. L'importateur actuel de l'atelier SPOOL, conçu à l'aide des classes Java 1.3, a été utilisé pour charger le

contenu des fichiers XMI de base dans les bases de données vierges. La station de travail utilisée était un PC doté d'un microprocesseur Pentium II 350 MHz, de 256 Mo de mémoire vive et du système d'exploitation Windows NT 4.0. L'opération d'importation s'est déroulée avec succès dans tous les cas; le tableau III présente les statistiques colligées.

	ET++ *(C++)*	**Apache** *(C)*	**SPOOL** *(Java)*	**jKitGo** *(Java)*
Importation des fichiers XMI de base Taille des fichiers (Ko)	11 060	1 467	6 993	6 315
Temps d'importation (s)	24,3	4,6	16,7	13,0
Débit (Ko/s)	455,1	318,9	418,7	485,8
Taille des bases de données résultantes (Ko)	10 240	1 536	5 888	4 608

Tableau III. Statistiques d'importation des fichiers XMI de base

À la lecture des statistiques obtenues, il est possible de dégager les observations suivantes :

- l'étape d'importation est un processus très rapide, les fichiers XMI de base étant traités à un débit moyen de 420 Ko/s;
- le langage utilisé pour la rédaction du code source ne semble pas influencer le débit d'importation, ce qui est normal puisque les fichiers XMI de base contiennent exclusivement des éléments conceptuels possédant un niveau d'abstraction supérieur aux constructions syntaxiques propres aux langages de programmation des systèmes importés;
- les bases de données résultantes possèdent une taille du même ordre que celle des fichiers XMI de base correspondants utilisés pour l'importation.

7.5 Impacts de l'utilisation d'une particule d'indentation

Les scripts d'exportation renferment une constante qui définit la particule d'indentation utilisée pour décaler les lignes XMI générées vers la droite, proportionnellement à leur niveau d'imbrication.

L'inspection visuelle des fichiers générés permet de constater que ces derniers peuvent renfermer des niveaux d'indentation très élevés, surtout dans le cas des fichiers XMI/RCR où l'on peut retrouver des niveaux d'imbrication supérieurs à 50. Puisque les particules d'indentation ajoutées consomment inévitablement beaucoup d'espace disque et pourraient éventuellement dégrader la performance des processus d'exportation et d'importation, de nouveaux fichiers XMI de base et XMI/RCR ont été générés pour les systèmes considérés, avec la même configuration matérielle et logicielle que celle décrite à la section 7.3, mais sans utiliser de particule d'indentation. La section courante présente les différences constatées entre les fichiers générés avec une particule d'indentation de deux espaces, et ceux générés sans indentation.

À l'examen des statistiques recueillies dans le tableau IV, il est possible de tirer les conclusions suivantes :

- les fichiers générés sans particule d'indentation utilisent effectivement beaucoup moins d'espace disque : pour les fichiers XMI de base, on peut constater une économie d'espace moyenne de 25 %; pour les fichiers XMI/RCR, il s'agit d'une économie moyenne de 60 %, ce qui permet de conclure que les fichiers XMI/RCR présentent des niveaux d'indentation plus élevés que les fichiers XMI de base;

- les fichiers XMI de base, générés sans particule d'indentation, offrent un taux de compression inférieur à ceux générés avec une particule d'indentation, lorsqu'ils sont stockés dans des archives `.zip` standard. Il en est déduit que le retrait de l'indentation réduit la redondance au sein de ces fichiers, du point de vue d'un algorithme de compression. Une telle relation n'a toutefois pu être établie pour les fichiers XMI/RCR;

- le temps de génération des fichiers XMI de base et XMI/RCR n'est pas affecté de façon significative par l'utilisation d'une particule d'indentation;

	ET++ *(C++)*	Apache *(C)*	SPOOL *(Java)*	jKitGo *(Java)*
Fichiers XMI de base générés				
Taille, avec particule (Ko)	11 060	1 467	6 993	6 315
Taille, sans particule (Ko)	8 187	1 080	5 277	4 797
Ratio sans / avec particule	74,0 %	73,6 %	75,5 %	76,0 %
Taille, avec particule, compressés (Ko)	510	71	286	251
Taille, sans particule, compressés (Ko)	456	64	256	224
Ratio sans / avec particule	89,4 %	90,1 %	89,5 %	89,2 %
Temps de génération, avec particule (s)	505	28	261	199
Temps de génération, sans particule (s)	493	29	260	198
Ratio sans / avec particule	97,6 %	103,6 %	99,6 %	99,5 %
Fichiers XMI/RCR générés				
Taille, avec particule (Ko)	195 666	57 610	63 336	41 187
Taille, sans particule (Ko)	72 591	20 743	26 210	18 505
Ratio sans / avec particule	37,1 %	36,0 %	41,4 %	44,9 %
Taille, avec particule, compressés (Ko)	6 004	1 650	2 117	1 373
Taille, sans particule, compressés (Ko)	2 039	530	855	632
Ratio sans / avec particule	34,0 %	32,1 %	40,4 %	46,0 %
Temps de génération, avec particule (s)	1 050	371	379	260
Temps de génération, sans particule (s)	1 036	386	370	255
Ratio sans / avec particule	98,7 %	104,0 %	97,6 %	98,1 %
Tous fichiers générés				
Taille, avec particule (Ko)	206 726	59 077	70 329	47 502
Taille, sans particule (Ko)	80 778	21 823	31 487	23 302
Ratio sans / avec particule	39,1 %	36,9 %	44,8 %	49,1 %
Taille, avec particule, compressés (Ko)	6 514	1 721	2 403	1 624
Taille, sans particule, compressés (Ko)	2 495	594	1 111	856
Ratio sans / avec particule	38,3 %	34,5 %	46,2 %	52,7 %
Temps de génération, avec particule (s)	1 555	399	640	459
Temps de génération, sans particule (s)	1 529	415	630	453
Ratio sans / avec particule	98,3 %	104,0 %	98,4 %	98,7 %
Importation des fichiers XMI de base				
Ratio de taille, sans / avec particule	74,0 %	73,6 %	75,5 %	76,0 %
Temps d'importation, avec particule (s)	24,3	4,6	16,7	13,0
Temps d'importation, sans particule (s)	23,6	4,5	15,8	12,7
Ratio de temps d'importation, sans / avec	97,1 %	97,8 %	94,6 %	97,7 %

Tableau IV. Statistiques concernant l'utilisation d'une particule d'indentation pour la génération des fichiers XMI de base et XMI/RCR

- le temps d'importation des fichiers XMI de base, en utilisant à nouveau la configuration matérielle et logicielle décrite à la section 7.4, est sensiblement le même, que les fichiers aient été générés avec une particule d'indentation ou non.

Les fichiers générés sans indentation, bien qu'occupant moins d'espace disque, sont plus difficiles à examiner par un utilisateur lorsqu'ils sont affichés à l'aide d'un éditeur de texte standard. Cependant, ils peuvent être visualisés à l'aide d'un navigateur possédant une feuille de style XML; à ce moment, le navigateur effectue automatiquement l'indentation des balises, qu'elle soit présente ou non dans le fichier original, au détriment d'un temps d'affichage plus long. Par conséquent, l'utilisation d'une particule d'indentation pour la génération des fichiers XMI de base et XMI/RCR pourra être laissée au soin de l'usager des scripts d'exportation.

7.6 Comparaison de la taille des fichiers XMI générés par deux approches différentes

Cette section présente une comparaison de la taille des fichiers XMI générés par deux approches, la première étant Datrix et le convertisseur ASG2XMI implanté dans l'atelier SPOOL, la deuxième étant la génération des fichiers XMI de base à partir des AST de DISCOVER, à l'aide des scripts d'exportation décrits dans ce mémoire.

Les fichiers source considérés pour la première approche sont un ensemble de 38 fichiers tirés du système ET++ pour lesquels Datrix a pu générer des fichiers `.asg`, par la suite convertis en fichiers XMI par l'application ASG2XMI de l'atelier SPOOL. L'ensemble des fichiers source `.c` et `.C` du système ET++ sont retenus pour la deuxième approche. En ce qui concerne les fichiers XMI considérés, les fichiers XMI de base générés par le processus d'exportation DISCOVER possèdent à toutes fins pratiques le même contenu que les fichiers produits par l'approche Datrix / ASG2XMI : ceux-ci sont donc retenus pour effectuer la comparaison.

	Exportation Datrix / ASG2XMI	Exportation DISCOVER Tcl / Access
Taille des fichiers source (`.c`, `.C`) utilisés (octets)	251 876	1 476 688
Taille des fichiers XMI générés (octets)	52 536 823	11 325 595
Ratio taille des fichiers XMI / taille des fichiers source	208,6	7,7

Tableau V. Génération des fichiers XMI : comparaison de deux approches

D'après les ratios taille des fichiers XMI / taille des fichiers source obtenus au tableau V, il apparaît que le processus d'exportation utilisant les AST de DISCOVER produit des fichiers environ 27 fois plus compacts qu'avec Datrix et le convertisseur ASG2XMI, pour le cas précis impliquant le système ET++. Ce résultat s'explique par le fait que Datrix ne traite que des fichiers source préparés par un préprocesseur, lesquels sont augmentés de tous les fichiers de déclarations (`.h`) identifiés par des directives préprocesseur `#include`. Datrix produit ensuite des fichiers `.asg` en format TA-like à l'intérieur desquels tous les arcs doivent référer à des nœuds présents dans le même fichier. Le convertisseur ASG2XMI reçoit ultérieurement ces fichiers `.asg` et génère des fichiers XMI contenant de multiples fois les éléments présents dans les fichiers de déclarations. Le processus d'importation rejette éventuellement les déclarations redondantes en ne créant qu'une seule fois les objets correspondants dans le dépôt SPOOL.

À l'opposé, DISCOVER conserve uniquement les objets présents dans l'unité de compilation courante pour chaque AST représentant un fichier source ou un fichier de déclarations. DISCOVER stocke les liens vers les objets déclarés dans des fichiers externes sous la forme de références texte. Ceci justifie la création de tables de symboles globales et leur remplissage préalable à l'émission des fichiers XMI dans le cadre du processus d'exportation impliquant les AST de DISCOVER. Ces tables de symboles contribuent directement à l'émission de fichiers XMI plus compacts.

7.7 Objectifs atteints

Les sous-sections suivantes décrivent les objectifs atteints parmi ceux énoncés dans le cadre du chapitre 4, *« Énoncé du problème »*, en lien avec le contenu des fichiers générés et la performance des processus d'exportation et d'importation, en plus de discuter des résultats par rapport aux hypothèses émises à la section 1.3, *« Objectifs de la recherche »*.

7.7.1 Contenu des fichiers XMI de base

L'inspection visuelle des fichiers XMI de base, générés pour un système donné, permet de constater la présence des éléments suivants :

- chaque fichier XMI de base renferme les balises représentant les identifiants des stéréotypes XMI utilisés et des types primitifs propres au langage du système;
- dans le premier fichier XMI de base généré, on retrouve la hiérarchie des fichiers et des répertoires du système;
- pour chacun des fichiers du système sont inscrits les fichiers inclus par les directives `#include` ainsi que les fichiers qui incluent le fichier courant;
- dans les fichiers XMI de base subséquents, on retrouve, à l'intérieur de balises identifiant chacun des fichiers du système, les différentes structures (`class`, `struct`, `union`, `interface`) respectivement définies à l'intérieur de ces fichiers;
- pour chacune des classes exportées, des balises représentant les superclasses, les sous-classes et les interfaces implantées sont émises;
- pour chacune des interfaces, les classes les implantant sont inscrites;
- pour chacune des structures, on retrouve les structures et fonctions amies (`friend`) ainsi que les structures englobées *(inner classes)*;
- pour chacune des structures, les attributs sont émis à l'aide des balises requises;
- les énumérations sont inscrites avec leurs littéraux;

- les alias sont inscrits avec les types qu'ils représentent;

- chacune des fonctions est exportée avec les noms et types de ses paramètres, son type de retour et les exceptions qu'elle lance;

- les énumérations, alias et fonctions appartenant à des structures sont exportés à l'intérieur des balises des structures correspondantes; ceux qui n'appartiennent à aucune structure sont émis à l'intérieur d'une classe utilitaire propre au fichier du système à l'intérieur duquel leurs déclarations sont rencontrées;

- les types non résolus *(forward types)*, les types complexes (types constants, pointeurs, références, tableaux) et les types fonctions rencontrés lors de la génération de chaque fichier sont inscrits en fin de fichier, à l'intérieur de leurs classes utilitaires respectives.

7.7.2 Contenu des fichiers XMI/RCR

L'inspection visuelle des fichiers XMI/RCR, générés pour un système donné, fait ressortir la présence des éléments suivants :

- un premier ensemble de fichiers XMI/RCR contient les déclarations des objets globaux de tout le système, ainsi que les valeurs d'initialisation qui leur sont associées;

- les fichiers XMI/RCR subséquents renferment les expressions d'initialisation associées aux variables statiques des structures (`class`, `struct`, `union` et `interface`) et aux littéraux des énumérations, ainsi que le détail des corps des fonctions et des blocs d'initialisation statiques des structures;

- les éléments identifiés aux points précédents sont exportés à l'aide de balises XMI/RCR, lesquelles représentent fidèlement les énoncés et expressions présents dans le code source;

- les variables statiques, énumérations, blocs d'initialisation et fonctions appartenant à des structures sont émis à l'intérieur de balises représentant les structures

correspondantes; ceux qui n'appartiennent à aucune structure sont exportés à l'intérieur d'une classe utilitaire propre au fichier du système à l'intérieur duquel leurs définitions sont rencontrées;

- les types non résolus, les types complexes, les types fonctions, les objets globaux non résolus et les fonctions non résolues rencontrés lors de la génération de chaque fichier sont inscrits en fin de fichier, à l'intérieur de leurs classes utilitaires respectives;
- chaque utilisation d'un objet local à une fonction ou global au système possède un lien pointant vers la déclaration de l'objet correspondant.

7.7.3 Performance du processus d'exportation des fichiers

Les fichiers XMI sont exportés à un débit moyen de 124 Ko/s. Le processus d'exportation pour le système le plus complexe, ET++, a duré 26 minutes.

La taille des fichiers générés demeure raisonnable, particulièrement si la particule d'indentation est supprimée, laquelle est modifiable avant le déclenchement du processus d'exportation. Les fichiers présentent un haut niveau de redondance dû au fait que les balises utilisées, disponibles en nombre restreint, y sont répétées de nombreuses fois. Ils peuvent donc être compressés efficacement dans des archives `.zip` standard, les fichiers compressés possédant une taille de 20 à 35 fois inférieure aux fichiers originaux. Ceci revêt une importance particulière puisque l'importateur de l'atelier SPOOL a été conçu pour traiter des fichiers préalablement stockés dans des archives `.zip`.

7.7.4 Performance du processus d'importation des fichiers XMI de base

Les fichiers XMI de base ont été traités avec succès lors du processus d'importation dans le dépôt SPOOL, à un débit moyen de 420 Ko/s. Les fichiers les plus volumineux, ceux du système ET++, ont été importés en 24 secondes. La réussite du processus

d'importation a pu être validée par l'utilisation des outils de visualisation de l'atelier SPOOL avec les bases de données créées lors des tests.

Les éléments suivants ont cependant été omis ou traités différemment par le processus d'importation, puisqu'ils ne sont pas supportés en tant que tels par le dépôt SPOOL dans sa version actuelle :

- les interfaces sont importées en tant que classes;
- les liens modélisant l'implantation des interfaces par les classes ne sont pas pris en compte;
- les balises identifiant les exceptions lancées par les fonctions sont ignorées;
- les types fonctions sont importés en tant que classes afin de permettre l'établissement des liens de typage, le dépôt SPOOL ne supportant pas les liens de type dirigés vers les opérations.

7.7.5 Validation des hypothèses émises

Lors de la présentation des objectifs de la recherche, nous formulions comme hypothèse que le parcours dirigé des arbres syntaxiques abstraits (AST) permet d'obtenir toutes les informations requises pour l'identification des composantes logicielles de haut niveau, celles présentes dans les corps des méthodes ainsi que les relations de dépendance entre toutes ces composantes. La mise au point des scripts d'exportation produisant les fichiers XMI de base et XMI/RCR, ainsi que l'inspection visuelle des fichiers générés, ont permis de repérer toutes les informations conceptuelles permettant la conduite d'analyses complètes et détaillées, énumérées à la section 4.2, *« Éléments constitutifs d'un système considérés »*. Par conséquent, nous concluons que l'examen approfondi des AST permet de récupérer les modèles conceptuels dans leur intégralité, à partir de la compilation du code source.

L'autre hypothèse formulée était que le profil RCR, un mécanisme d'extension du métamodèle UML, peut être mis à profit pour l'apport de nouvelles classes de stockage

au sein du dépôt de SPOOL et, parallèlement, pour la définition de structures d'encodage additionnelles, intégrées au format d'échange XMI. Lors de la conception des scripts d'exportation générant les fichiers d'échange, il fut constaté que les métaéléments du profil RCR modélisaient de façon appropriée les différentes structures rencontrées lors du balayage exhaustif des AST; en effet, seulement quelques légers ajustements ont été apportés au profil et présentés à la section 7.1, *« Ajustements au profil RCR »*, afin de fournir une représentation adéquate pour certains éléments. De plus, lors de l'élaboration du modèle du dépôt étendu SPOOL/RCR, il fut aisé de concevoir de nouvelles classes de stockage destinées à recevoir les microcomposantes logicielles présentes dans les corps des méthodes en effectuant une transposition des métaéléments du profil RCR. Par conséquent, nous concluons que le profil RCR est un mécanisme approprié et efficace pour l'extension d'un dépôt conceptuel basé sur UML, tel le dépôt de SPOOL, et d'un format d'échange dérivé de UML, en l'occurrence XMI.

7.8 Leçons retenues

Le langage Tcl/Access, utilisé pour accéder aux AST de DISCOVER, ne propose pas de structures de données versatiles telles celles offertes par le langage Java. Par exemple, lors de la programmation des scripts d'exportation, des structures de listes parallèles ont dû être utilisées pour implanter des vecteurs contenant des objets complexes, afin de permettre la recherche rapide d'un objet dont un des attributs possède une valeur prédéterminée. Tcl/Access offre cependant une vitesse d'exécution remarquable, comparable à celle des langages compilés, tout en permettant une mise au point rapide des scripts rédigés. Néanmoins, la conception modulaire des scripts d'exportation rend possible la traduction des algorithmes dans d'autres langages afin qu'ils puissent s'appliquer à d'autres parseurs qui rendent accessibles les AST générés lors du processus de compilation.

Afin de faciliter la mise au point des scripts, il est primordial d'implanter des fonctionnalités de journalisation qui permettent d'enregistrer les choix effectués par les algorithmes lors de leur exécution. À titre d'exemple, le journal des opérations permet de détecter si des nœuds d'AST inconnus jusqu'à présent ont été rencontrés, afin que de

nouvelles routines de traitement puissent être implantées à leur intention. L'inscription d'un estampillage temporel à chaque ligne du journal permet d'identifier les étapes nécessitant le plus de temps, afin que les algorithmes mis en cause puissent être optimisés.

Les tables de symboles globales, construites avant l'émission des fichiers XMI, sont d'une importance capitale pour la génération de fichiers comprenant des informations de qualité. En particulier, l'établissement de liens entre les utilisations des variables et fonctions et leurs déclarations permet de conduire de nombreuses analyses dans l'atelier SPOOL. Lors de la rencontre d'éléments non résolus, c'est-à-dire de l'utilisation d'éléments dont la déclaration n'a pas été rencontrée au préalable, il est nécessaire de les enregistrer dans une table de symboles en leur assignant un identifiant généré automatiquement. Ceci permet d'effectuer des liens convergents vers des éléments non résolus rencontrés plusieurs fois.

Le traitement des noms des énumérations anonymes et des structures anonymes requiert une attention particulière. Ces noms sont générés automatiquement par DISCOVER et servent à faire le pont entre les déclarations et les utilisations de ces éléments. Un traitement spécifique doit être appliqué afin d'obtenir la structure dont fait partie une énumération ou une structure anonyme.

Les AST de DISCOVER présentent quelquefois des structures invalides, résultant d'erreurs de *parsing* où DISCOVER a tout de même récupéré le maximum d'informations possible. Un nœud d'AST `if_stmt` ne possédant qu'un seul nœud enfant est un bon exemple de ce type de nœud. Lorsque les scripts d'exportation rencontrent ces nœuds, ils émettent les informations présentes dans les AST, malgré le fait qu'elles soient incomplètes, et inscrivent au journal la présence du nœud déficient.

7.9 Conclusion

Ce chapitre a présenté diverses discussions reliées à la conception des scripts d'exportation des fichiers XMI de base et XMI/RCR, à l'analyse de leurs extrants et des processus leur étant associés.

Dans un premier temps, les ajustements effectués au profil RCR ont été décrits. Ces derniers facilitent la modélisation des informations contenues dans les AST de DISCOVER, qui sont transférées ultérieurement dans le dépôt de l'atelier SPOOL. Une courte discussion justifie l'utilisation littérale des balises RCR au lieu des balises XMI stéréotypées correspondantes.

Par la suite, les résultats des expérimentations conduites avec les fichiers XMI de base et XMI/RCR ont été exposés, notamment en ce qui concerne la génération des fichiers, l'importation des fichiers XMI de base dans le dépôt de SPOOL, les impacts de l'utilisation d'une particule d'indentation et la comparaison, au niveau de leur taille, des fichiers XMI de base avec ceux produits par l'approche Datrix / ASG2XMI.

Pour clore le chapitre, les objectifs atteints ont été présentés, ainsi que les leçons retenues de la conception des scripts d'exportation.

Le chapitre suivant constitue la conclusion de ce mémoire.

8

Conclusion

Le chapitre 8, *« Conclusion »*, présente une synthèse de ce mémoire, en plus de discuter des travaux futurs reliés à l'étude courante et d'offrir une réflexion finale.

Ce chapitre contient les sections suivantes :

Section	*Page*
8.1 Synthèse	86
8.2 Travaux futurs	91
8.3 Réflexion finale	93

8.1 Synthèse

Ce mémoire a présenté les résultats de la recherche effectuée dans le cadre du projet SPOOL, relativement à la conception et à l'implantation d'une nouvelle passerelle alimentant le dépôt de code source étendu de l'atelier de génie logiciel, permettant l'analyse détaillée de systèmes de taille industrielle. La nouvelle passerelle possède la capacité de traiter les systèmes écrits en Java, en plus d'apporter des informations détaillées concernant les corps des méthodes, afin de permettre à l'atelier de supporter de nouvelles activités d'investigation logicielle telles le découpage orienté objet, la détection de clones et la récupération de microconceptions.

Deux hypothèses furent formulées pour la conduite des travaux. La première stipule que le parcours dirigé des arbres syntaxiques abstraits, représentation intermédiaire entre le code source et le code exécutable produite par un compilateur, permet d'identifier toutes les composantes systémiques requises pour la réalisation d'analyses approfondies. La deuxième concerne les capacités du profil RCR, un mécanisme d'extension du

métamodèle UML; il est attendu que le profil saura fournir des solutions adéquates et efficaces pour la modélisation du détail des corps des méthodes, tant dans la définition de nouvelles classes de stockage au sein du dépôt de SPOOL que dans la création de structures d'encodage additionnelles intégrées au format d'échange XMI.

À titre de rappels, les concepts et technologies reliés de près à l'étude courante ont été exposés. Au nombre de ceux-ci, on retrouve :

- lex et yacc, des utilitaires Unix supportant la construction de compilateurs;
- les arbres syntaxiques abstraits (AST), fournissant une représentation intermédiaire entre le code source et le code objet au cœur d'un compilateur;
- les graphes sémantiques abstraits (ASG), AST augmentés d'informations sémantiques, dont les liens entre les utilisations et les déclarations des variables;
- UML, un langage graphique pour la visualisation, la construction, la spécification et la documentation des composantes des systèmes, aidant à la communication des idées entre les différents membres d'une équipe de développement;
- XMI, un format d'échange intégrant trois standards industriels, XML, UML et MOF, visant à faciliter l'échange de métamodèles basés sur UML;
- le profil RCR, offrant une solution compacte et efficace pour la modélisation détaillée des corps des méthodes d'un système, à l'aide de métaéléments stéréotypés, dérivés des métaéléments UML.

Une étude des produits commerciaux et projets de recherche assimilés à l'étude courante par les buts visés et les technologies employées a permis de souligner les caractéristiques des produits et projets suivants :

- GEN++, un générateur d'analyseurs de code source C++, permettant une exploration dirigée des ASG par l'entremise de son langage de spécification de requêtes, GENOA;

- Datrix, une suite d'outils visant à évaluer la qualité du logiciel, dont les parseurs traitent le code source pour en extraire les ASG et les émettre sous la forme de fichiers d'échange;

- CPPX, un compilateur C++ modifié qui génère en sortie une base d'informations récupérable par les ateliers de génie logiciel;

- Columbus, un cadre d'application de rétro-ingénierie logicielle, effectuant l'analyse, la construction de représentations internes, le filtrage et l'exportation des composantes tirées du code source des systèmes;

- DISCOVER, un atelier de génie logiciel comportant une suite d'outils aidant à la compréhension des systèmes, dont des parseurs qui stockent, dans un dépôt interne accessible par programmation, l'information détaillée tirée du code source;

- la passerelle Datrix-SPOOL, traitant les fichiers d'échange générés par les parseurs de la suite d'outils Datrix en vue d'alimenter le dépôt orienté objet de l'atelier SPOOL.

Une comparaison des produits et projets examinés, basée sur sept critères reliés aux fonctionnalités souhaitées de l'atelier SPOOL, soit la prise en charge des systèmes rédigés en C, C++ et Java, le traitement des composantes logicielles de haut niveau et du détail des corps des méthodes, ainsi que l'établissement et l'exploitation de liens entre l'utilisation et la déclaration des variables, des types et des fonctions, a permis d'identifier les parseurs de la suite DISCOVER comme étant la solution la plus appropriée permettant d'apporter les nouvelles fonctionnalités requises pour SPOOL, soit le traitement du langage Java et l'apport d'informations détaillées concernant les corps des méthodes, dont les liens entre l'utilisation et la déclaration des microcomposantes logicielles.

L'énoncé du problème, étudié dans le cadre de ce mémoire, a précisé la nature de la problématique abordée, à savoir l'alimentation d'un dépôt de code source permettant l'analyse détaillée de systèmes logiciels de taille industrielle. De plus, il a fait ressortir les aspects suivants de la solution désirée :

- les éléments constitutifs d'un système étant retenus, et ceux étant exclus;
- les langages de programmation considérés : C, C++ et Java;
- les attentes techniques associées aux différents processus qui conduisent à l'alimentation du dépôt;
- les attentes techniques concernant les fichiers d'échange générés au cours des opérations conduites.

Une ébauche de la solution retenue a été présentée. Celle-ci met en œuvre :

- les parseurs de la suite DISCOVER, pour le *parsing* du code source C, C++ et Java;
- des scripts d'exportation spécialisés, rédigés en Tcl/Access et exécutables dans l'environnement DISCOVER, qui parcourent les AST stockés dans son dépôt et émettent les éléments constitutifs considérés pour un système sous la forme de fichiers d'échange XMI et XMI/RCR; le profil RCR est ici mis à contribution pour modéliser les éléments composant les corps des fonctions;
- un importateur XMI/RCR, capable de traiter les balises présentes dans les fichiers d'échange et de créer les objets correspondants dans le dépôt étendu de l'atelier SPOOL;
- un dépôt SPOOL/RCR, version améliorée du dépôt SPOOL actuel, spécialement conçu pour stocker les éléments propres au langage Java et le détail des corps des fonctions.

Les composantes de la solution présentée permettent à l'atelier SPOOL d'analyser les systèmes logiciels écrits en Java, en plus des langages C et C++ déjà supportés, mais aussi, et surtout, elles rendent accessibles à l'ensemble des outils de visualisation de l'atelier l'information détaillée au sujet des corps des fonctions; les scripts d'exportation contribuent significativement à la solution en construisant de façon fiable les liens propres aux ASG tels les références entre les utilisations et les déclarations des variables et des fonctions.

Les composantes clés de la solution retenue ont été exposées, en débutant par les scripts d'exportation exécutables dans l'environnement DISCOVER. Un ensemble d'algorithmes permet la génération de fichiers XMI de base, qui contiennent les éléments principaux d'un système. Cet ensemble de fichiers est traitable par l'importateur XMI faisant partie de la passerelle Datrix-SPOOL. Un second ensemble d'algorithmes a pour tâche d'exporter le détail des corps des fonctions sous la forme de fichiers XMI/RCR, qui requièrent un importateur spécialisé discuté plus avant. L'architecture du dépôt SPOOL/RCR a ensuite été présentée de façon élaborée, suivie par les stratégies de conception de l'importateur XMI/RCR associé.

Différentes discussions ont permis de rapporter certains aspects de l'implantation de la solution retenue, notamment :

- les ajustements effectués au profil RCR afin de faciliter sa mise en œuvre;
- les considérations relatives à l'utilisation littérale des balises RCR lors de la génération des fichiers XMI/RCR;
- les résultats associés aux tests de génération des fichiers XMI de base et XMI/RCR; ces tests ont impliqué quatre systèmes de taille industrielle : ET++ (rédigé en C++), Apache (C), SPOOL (Java) et jKitGo (Java);
- les résultats colligés lors de l'importation des fichiers XMI de base pour les quatre systèmes énumérés précédemment;
- les impacts de l'utilisation d'une particule d'indentation lors de la génération des fichiers d'échange;
- la comparaison de la taille des fichiers XMI de base générés par deux approches différentes, notamment celle impliquant les parseurs Datrix et l'utilitaire de conversion ASG2XMI, et celle représentée par la solution retenue;
- la validation des objectifs et hypothèses mis de l'avant lors de l'énoncé du problème;
- les leçons retenues du travail d'implantation effectué.

8.2 Travaux futurs

Cette section présente quelques travaux futurs découlant directement de l'implantation technique accomplie dans le cadre du projet de recherche.

8.2.1 Implantation du dépôt SPOOL/RCR

Afin d'être en mesure d'emmagasiner toutes les informations contenues dans les fichiers d'échange XMI/RCR, le dépôt de l'atelier SPOOL devra être modifié afin d'inclure les nouvelles classes de stockage présentées dans le cadre de la section 6.2, *« Architecture du dépôt SPOOL/RCR »*. De plus, de nouvelles méthodes utilitaires devront être implantées afin d'accéder aux composantes de ces classes. Les fabriques d'objets seront modifiées pour permettre l'instanciation et l'initialisation des nouveaux métaéléments.

Certaines classes de stockage, rendues désuètes par l'intégration du profil RCR, devront être enlevées du modèle du dépôt : par exemple, les classes `MAction`, `MCallAction`, `MCreateAction`, etc. Les applications de l'atelier subiront les modifications appropriées afin de tirer l'information dont elles ont besoin des nouveaux métaéléments accumulant les informations propres aux classes supprimées.

8.2.2 Implantation de l'importateur XMI/RCR

Un nouvel importateur, capable de traiter tous les types de balises rencontrés dans les fichiers XMI de base et XMI/RCR, devra être réalisé afin de charger le contenu des fichiers d'échange dans le dépôt étendu. À cet effet, la section 6.3 de ce document, *« Stratégies de conception de l'importateur XMI/RCR »*, pourra être mise à profit.

8.2.3 Ajustements aux scripts d'exportation

Lors de l'exportation d'autres systèmes que ceux étudiés dans le cadre de ce projet de recherche, de nouveaux types de nœuds d'AST pourraient être rencontrés, nécessitant du même coup la rédaction de routines de traitement jumelées dans les scripts

d'exportation. De plus, la découverte de nouvelles structures de nœuds enfants, associées aux types de nœuds déjà considérés, pourra réclamer des corrections mineures aux scripts actuels.

8.2.4 Développement de nouveaux outils d'analyse et de visualisation pour l'atelier SPOOL

L'intégration du détail des corps des fonctions dans le nouveau dépôt de l'atelier SPOOL, ainsi que l'apport des liens entre les utilisations et les déclarations des variables et des fonctions, nécessitent le développement de nouveaux outils d'analyse et de visualisation, incorporés à l'atelier.

Au nombre des analyses rendues possibles par l'entreposage d'informations détaillées sur le code source, citons : le découpage orienté objet [Chen_2001, Larsen_1996], l'analyse d'impact des changements [Chaumun_2001, Kabaili_2001], la récupération de microconceptions [Coplien_1992, Marinescu_2002] et la détection de clones [Baxter_1998, Laguë_1997]. De plus, des graphes de flot d'exécution interne aux fonctions et d'utilisation des objets globaux peuvent être produits par des outils de visualisation conçus à cet effet.

8.2.5 Application des algorithmes de génération des fichiers XMI et XMI/RCR à d'autres processeurs frontaux

Afin d'assurer une utilisation répandue des algorithmes de génération des fichiers d'échange XMI de base et XMI/RCR, ceux-ci ont été conçus de telle façon qu'ils peuvent être appliqués à d'autres processeurs frontaux donnant accès aux AST résultant de la compilation du code source. Les algorithmes pourront être adaptés à des processeurs du domaine public, ou à d'autres applications traitant des fichiers source rédigés en de nouveaux langages et générant des AST similaires. La poursuite du développement des scripts d'exportation sera cependant facilitée dans l'environnement DISCOVER, grâce à l'utilisation du langage de scriptage Tcl/Access qui permet une mise au point aisée et rapide des programmes.

8.3 Réflexion finale

La production automatisée d'arbres syntaxiques abstraits de qualité, uniquement à partir des fichiers source des systèmes, constitue la pierre angulaire de l'analyse statique rigoureuse du code source. La possibilité de sauvegarder ces structures riches en informations et de les recréer dans leur intégralité rend possible la navigation dirigée et le recueil de données dérivées utiles pour les activités de rétroconception, de compréhension et de réingénierie.

L'intégration technique du profil RCR à un atelier de génie logiciel basé sur UML fournit une preuve tangible de la commodité et de l'efficacité des mécanismes d'extension d'UML, notamment les stéréotypes, pour la modélisation des corps des méthodes des systèmes. L'utilisation du format d'échange XMI met en valeur une solution versatile et efficiente pour la transmission de modèles complexes, tout en évitant la création d'un nouveau format d'encodage spécialisé qui ne bénéficie pas de support étendu dans l'industrie.

À la lumière du travail effectué dans le cadre du projet de recherche SPOOL, l'auteur souhaite voir se disséminer la connaissance des arbres syntaxiques abstraits et assister à la naissance de nouvelles applications destinées à leur étude, à défaut de pouvoir se consacrer à cette tâche avec opiniâtreté.

Bibliographie

[Apache]

The Apache Software Foundation, Forest Hill, Maryland, United States of America. *Apache HTTP Server Project.*
Site Web : http://httpd.apache.org/

[Baxter_1998]

BAXTER, Ira D., YAHIN, Andrew, MOURA, Leonardo, SANT'ANNA, Marcelo, BIER, Lorraine. « Clone Detection Using Abstract Syntax Trees ». In *Proceedings of the International Conference on Software Maintenance (ICSM'1998)*, pages 368-377, Bethesda, Maryland, United States of America, March 1998. IEEE.
Disponible à l'adresse : http://www.semdesigns.com/Company/Publications/ICSM98.pdf

[Bédard_2002]

BÉDARD, Jean-François. *La passerelle DISCOVER-SPOOL : algorithmes de génération des fichiers d'échange.* Rapport technique GELO-150, Laboratoire de génie logiciel (GÉLO), Département d'informatique et de recherche opérationnelle, Université de Montréal, Québec, Canada, avril 2002.

[Chaumun_2001]

CHAUMUN, M. Ajmal, KABAILI, Hind, KELLER, Rudolf K., LUSTMAN, François. « A Change Impact Model for Changeability Assessment in Object-Oriented Systems ». In *Science of Computer Programming*, Elsevier Science Publishers, 2001. À paraître.
Disponible à l'adresse : http://www.iro.umontreal.ca/~labgelo/Publications/Papers/scp-2001.pdf

[Chen_2001]

CHEN, Zhenqiang, XU, Baowen. « Slicing Object-Oriented Java Programs ». In *ACM SIGPLAN Notices*, vol. 36, no. 4, pages 33-40, April 2001.

[Chikofsky_1990]

CHIKOFSKY, Elliot J., CROSS, James H., II. « Reverse Engineering and Design Recovery: A Taxonomy ». In *IEEE Software*, vol. 7, no. 1, pages 13-17, January 1990.
Disponible à l'adresse : http://www.infosys.tuwien.ac.at/Teaching/Courses/SWV/papers/chikofsky90.pdf

[Coplien_1992]

COPLIEN, James O. *Advanced C++ Programming Styles and Idioms.* Addison Wesley Professional, United States of America, 1992.

[CPPX]

CPPX: Open Source C++ Fact Extractor. Software Architecture Group, University of Waterloo, Ontario, Canada.
Site Web : http://www.swag.uwaterloo.ca/~cppx/

[Datrix_2000]

Bell Canada's Quality Engineering and Research Group, Montréal, Québec, Canada. *Datrix.* 2000.
Site Web : http://www.iro.umontreal.ca/labs/gelo/datrix/

[Datrix_ASG_2000]

Bell Canada's Quality Engineering and Research Group, Montréal, Québec, Canada. *DATRIX™ Abstract Semantic Graph Reference Manual, version 1.4.* May 2000.
Disponible à l'adresse : http://www.iro.umontreal.ca/labs/gelo/datrix/refmanuals/asgmodel-1.4.pdf

[Demeyer_1999]

DEMEYER, Serge, DUCASSE, Stéphane, TICHELAAR, Sander. « Why Unified is not Universal: UML Shortcomings for Coping with Round-trip Engineering ». In *Proceedings of the Second International Conference on the Unified Modeling Language (UML'99)*, Bernhard Rumpe, editor. Springer-Verlag, 1999, LNCS 1723.
Disponible à l'adresse : iamwww.unibe.ch/~ducasse/WebPages/ARTICLES/Deme99d.pdf

[Devanbu_1998]

DEVANBU, Premkumar. *GEN++: C++ Analysis – Without the Pain.* Department of Computer Science, College of Engineering, University of California, Davis, United States of America, 1998.
Site Web : http://www.cs.ucdavis.edu/~devanbu/genp/

[ET++_1993]

ET++ Application Framework Distribution. Department for Graphics and Parallel Processing, Johannes Kepler Universität, Linz, Austria, 1993.
Site Web : http://www.gup.uni-linz.ac.at:8001/research/debugging/distribution/et++.html

[FrontEndART]

FrontEndART: the Analyser Front End Company for the Software (Reverse) Engineering Community, Szeged, Hungary. *Columbus/CAN.*
Site Web : http://www.frontendart.com/

[GCC]

Free Software Foundation, Boston, Massachusetts, United States of America. *GCC: GNU Compiler Collection.*
Site Web : http://gcc.gnu.org/

[GXL]

GXL: Graph eXchange Language. GUPRO : Generische Umgebung zum PROgrammverstehen (Generic Understanding of PROgrams). Universität Koblenz-Laudau, Germany.
Site Web : http://www.gupro.de/GXL/

[Holt_2002]

HOLT, Richard C. *An Introduction to TA: the Tuple-Attribute Language.* Department of Computer Science, University of Waterloo and Toronto, Ontario, Canada, 2002.
Disponible à l'adresse : http://plg.uwaterloo.ca/~holt/papers/ta.html

[HTML]

World Wide Web Consortium, Massachusetts Institute of Technology, Cambridge, Massachusetts, United States of America. *HTML: HyperText Markup Language.*
Site Web : http://www.w3.org/MarkUp/

[jKitGo]

Instantiations, Tualatin, Oregon, United States of America. *jKitGo.*
Site Web : http://www.instantiations.com/go-code/demo112/com/objectshare/gf/doc/readme.htm

[Kabaili_2001]

KABAILI, Hind, KELLER, Rudolf K., LUSTMAN, François. « A Change Impact Model Encompassing Ripple Effect and Regression Testing ». In *Proceedings of the Fifth International Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, pages 25-33, Budapest, Hungary, June 2001. Tenu en conjonction avec la *15th European Conference on Object-Oriented Programming (ECOOP'2001).*
Disponible à l'adresse : http://www.iro.umontreal.ca/~labgelo/Publications/Papers/ecoop-2001.pdf

[Keller_2001]

KELLER, Rudolf K., BÉDARD, Jean-François, ST-DENIS, Guy. « Design and Implementation of a UML-based Design Repository ». In *Proceedings of the Thirteenth Conference on Advanced Information Systems Engineering (CAiSE'01)*, pages 448-464, Interlaken, Switzerland, June 2001.
Disponible à l'adresse : `http://www.iro.umontreal.ca/~labgelo/Publications/Papers/caise-2001.pdf`

[Keller_2002]

KELLER, Rudolf K., SCHAUER, Reinhard, ROBITAILLE, Sébastien, LAGUË, Bruno. « Pattern-Based Design Recovery with SPOOL ». In *Advances in Software Engineering: Comprehension, Evaluation, and Evolution*, chapter 6, pages 113-135, Hakan Erdogmus and Oryal Tanir, editors. Springer, 2002.
Texte préliminaire disponible à l'adresse : `http://www.iro.umontreal.ca/~labgelo/Publications/Papers/cbook-2001-dpr.pdf`

[Laguë_1997]

LAGUË, Bruno, PROULX, Daniel, MAYRAND, Jean, MERLO, Ettore M., HUDEPOHL, John. « Assessing the Benefits of Incorporating Function Clone Detection in a Development Process ». In *Proceedings of the International Conference on Software Maintenance (ICSM'1997)*, pages 314-321, Bari, Italy, October 1997. IEEE.

[Larsen_1996]

LARSEN, Loren, HARROLD, Mary Jean. « Slicing Object-Oriented Software ». In *Proceedings of the 18th International Conference on Software Engineering (ICSE'96)*, pages 495-505, Berlin, Germany, March 1996. IEEE.
Disponible à l'adresse : `http://www.lorenlarsen.com/articles/icse96.pdf`

[Levine_1995]

LEVINE, John R., MASON, Tony, BROWN, Doug. *lex & yacc, Second Edition.* O'Reilly & Associates, United States of America, 1995.
Site Web de l'ouvrage chez l'éditeur : `http://www.oreilly.com/catalog/lex/`

[Marinescu_2002]

MARINESCU, Floyd, ROMAN, Ed. *EJB Design Patterns: Advanced Patterns, Processes, and Idioms.* John Wiley & Sons, United States of America, 2002.
Site Web de l'ouvrage : `http://www.theserverside.com/books/EJBDesignPatterns/index.jsp`

[MKS]

MKS Inc., Waterloo, Ontario, Canada. *DISCOVER*.
Site Web : http://www.mks.com/products/discover/

[ODMG]

Object Data Management Group: The Standard for Storing Objects. Burnsville, Minnesota, United States of America.
Site Web : http://www.odmg.org/

[OMG]

OMG: Object Management Group. Needham, Massachusetts, United States of America.
Site Web : http://www.omg.org/

[OMG_Catalog]

Object Management Group, Needham, Massachusetts, United States of America. *Catalog of OMG Modeling Specifications*.
Site Web : http://www.omg.org/technology/documents/modeling_spec_catalog.htm

[POET]

Poet Software Corporation. San Mateo, California, United States of America.
Site Web : http://www.poet.com/

[Rigi]

Rigi: A Visual Tool for Understanding Legacy Systems. Department of Computer Science, University of Victoria, British Columbia, Canada.
Site Web : http://www.rigi.csc.uvic.ca/

[Robitaille_2000]

ROBITAILLE, Sébastien, SCHAUER, Reinhard, KELLER, Rudolf K. « Bridging Program Comprehension Tools by Design Navigation ». In *Proceedings of the International Conference on Software Maintenance (ICSM'2000)*, pages 22-32, San Jose, California, United States of America, October 2000. IEEE.
Disponible à l'adresse : http://www.iro.umontreal.ca/~labgelo/Publications/Papers/icsm-2000.pdf

[Rosenblum_1991]

ROSENBLUM, David S., WOLF, Alexander L. « Representing Semantically Analyzed C++ Code with Reprise ». AT&T Bell Laboratories, Murray Hill, New Jersey, United States of America. In *Proceedings of the USENIX C++ Conference*, Washington, DC, United States of America, 1991.
Disponible à l'adresse : `http://citeseer.nj.nec.com/rosenblum91representing.html`

[Sander_1995]

SANDER, Georg. *VCG: Visualization of Compiler Graphs.* Chair for Programming Languages and Compiler Construction, Universität des Saarlandes, Saarbrücken, Germany, 1995.
Site Web : `http://rw4.cs.uni-sb.de/~sander/html/gsvcg1.html`

[Schauer_1999]

SCHAUER, Reinhard, ROBITAILLE, Sébastien, MARTEL, François, KELLER, Rudolf K. « Hot Spot Recovery in Object-Oriented Software with Inheritance and Composition Template Methods ». In *Proceedings of the International Conference on Software Maintenance (ICSM'99)*, pages 220-229, Oxford, England, United Kingdom, August 1999. IEEE.
Disponible à l'adresse : `http://www.iro.umontreal.ca/~labgelo/Publications/Papers/icsm99.pdf`

[Schauer_2002]

SCHAUER, Reinhard, KELLER, Rudolf K., LAGUË, Bruno, KNAPEN, Gregory, ROBITAILLE, Sébastien, ST-DENIS, Guy. « The SPOOL Design Repository: Architecture, Schema, and Mechanisms ». In *Advances in Software Engineering: Comprehension, Evaluation, and Evolution*, chapter 13, pages 269-294, Hakan Erdogmus and Oryal Tanir, editors. Springer, 2002.
Texte préliminaire disponible à l'adresse : `http://www.iro.umontreal.ca/~labgelo/Publications/Papers/cbook-2001-rep.pdf`

[SPOOL]

SPOOL: Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems. Laboratoire de génie logiciel (GÉLO), Département d'informatique et de recherche opérationnelle, Université de Montréal, Québec, Canada.
Site Web : `http://www.iro.umontreal.ca/labs/gelo/spool/`

[StDenis_2001]

ST-DENIS, Guy. *RCR : un profil UML pour la rétroconception, la compréhension et la réingénierie de logiciels.* Mémoire de maîtrise, Département d'informatique et de recherche opérationnelle, Université de Montréal, Québec, Canada, avril 2001. Disponible à l'adresse : http://www.iro.umontreal.ca/~labgelo/Publications/Theses/msc-stdenis.pdf

[UML]

Object Management Group, Needham, Massachusetts, United States of America. *Unified Modeling Language (UML).*
Site Web : http://www.omg.org/technology/documents/formal/uml.htm

[W3C]

W3C: World Wide Web Consortium. Massachusetts Institute of Technology, Cambridge, Massachusetts, United States of America.
Site Web : http://www.w3.org/

[XMI]

Object Management Group, Needham, Massachusetts, United States of America. *XML Metadata Interchange (XMI).*
Site Web : http://www.omg.org/technology/documents/formal/xmi.htm

[XML]

World Wide Web Consortium, Massachusetts Institute of Technology, Cambridge, Massachusetts, United States of America. *eXtensible Markup Language (XML).*
Site Web : http://www.w3.org/XML/

[XML4J]

IBM/alphaWorks, IBM Corporation, White Plains, New York, United States of America. *XML Parser for Java.*
Site Web : http://www.alphaworks.ibm.com/tech/xml4j