

THÈSE

présentée devant

l'Université de Rennes 1

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention INFORMATIQUE

par

Damien POLLET

Équipe d'accueil : Triskell – IRISA

École doctorale : MATISSE

Composante universitaire : IFSIC

Titre de la thèse :

Une architecture pour les transformations de modèles
et la restructuration de modèles UML

Soutenue le 28 juin 2005 devant la commission d'examen

Composition du jury

Président : Françoise ANDRÉ

Rapporteurs : Stéphane DUCASSE

Tom MENS

Examineurs : Jean-Marc JÉZÉQUEL

Pierre-Alain MULLER

Gerson SUNYÉ

Remerciements

Je tiens tout d'abord à remercier Jean-Marc Jézéquel pour avoir encadré cette thèse et su guider mes recherches par ses conseils et ses encouragements.

Je remercie sincèrement Françoise André qui m'a fait l'honneur de présider le jury, ainsi que Stéphane Ducasse et Tom Mens, qui ont bien voulu évaluer et apporter à ce travail leurs critiques grandement appréciées.

Je remercie également Pierre-Alain Muller et Gerson Sunyé qui ont participé au jury.

Finalement, il m'aurait sans doute été impossible de mener cette thèse à son terme sans la présence des nombreuses personnes qui m'ont supporté, que ça soit par sympathie ou par nécessité; mes remerciements ou excuses donc à Clémentine, Gerson, Franck, Jim, Tewfik, Benoît, Yves, Noël, Didier et tous les autres membres de Triskell, ainsi qu'à Pascal, Stéphane et aux multiples Bruno, Kevin, Thomas, Nicolas...

Table des matières

Remerciements	i
I Problématique	1
1 Introduction	3
1.1 Motivation	3
1.2 Contribution	4
1.3 Plan du document	6
2 Contexte technologique et état de l'art	7
2.1 De l'approche objet à l'ingénierie des modèles	7
2.1.1 Conception orientée objets	7
2.1.2 Méthodes de développement	8
2.1.3 Sujet d'étude : le système de réunions virtuelles	9
2.1.4 Analyse avec UML	11
2.1.5 Métamodélisation	16
2.2 Automatisation de l'activité de conception	17
2.2.1 Patrons de conception	18
2.2.2 Programmation par aspects	22
2.2.3 Refactorings	24
2.3 Techniques de support	27
2.3.1 Transformations d'arbres et de graphes	27
2.3.2 Transformations de modèles	29
II Contribution	35
3 Une architecture cohérente pour les transformations de modèles	37
3.1 L'ingénierie dirigée par les modèles	37
3.2 Impacts du MDE sur le cycle de développement	40
3.2.1 Cycle de vie d'une transformation	41

3.2.2	Processus de transformation des modèles	41
3.2.3	Éléments de retour sur investissement	44
3.3	Architecture de transformation de modèles	44
3.4	Interface d'accès aux dépositaires	49
3.4.1	Niveau « méta »	49
3.4.2	Accès au modèle	51
4	Noyau sémantique : une implémentation de référence en O'Caml	55
4.1	Un MOF minimal et son API	55
4.2	Métamodèles d'expérimentation	58
4.2.1	Les machines à états	59
4.2.2	Les traces	65
4.3	Une transformation : extraction de traces d'une machine à états finis	66
4.4	Bilan et perspectives	68
4.4.1	Le coeur sémantique	68
4.4.2	Le langage MTL en pratique	69
III	Application	73
5	Application	75
5.1	Refactoring de modèles	75
5.2	Instanciation de patrons de conception	77
5.2.1	Exemple : insertion du design pattern Observer	79
5.2.2	Exemple : insertion du design pattern Command	80
5.3	Des refactorings pour UML	84
5.3.1	Exemple sur les diagrammes d'états	84
5.3.2	Équivalence d'éléments de modèle	86
5.3.3	Refactoring des diagrammes de classes	87
5.3.4	Refactoring des diagrammes d'états	89
5.3.5	État de l'art rétrospectif	92
IV	Conclusion	95
6	Conclusion	97
6.1	Cycle de vie des transformations	97
6.2	Architecture de manipulation de modèles	98
6.3	Application	98

7 Bilan et perspectives	101
7.1 Valorisation	101
7.2 Limitations et Perspectives	102
7.2.1 Support aux transformations de haut niveau	102
7.2.2 Test et validation de transformations	102
7.2.3 Traçabilité	103
V Annexes	105
Bibliographie	107
Publications	119
Glossaire	121
Résumés	123

Table des figures

2.1	Cycle de développement en spirale	10
2.2	Une partie des cas d'utilisations des réunions virtuelles	12
2.3	Diagramme des classes à l'analyse des réunions virtuelles	14
2.4	Diagramme des états d'une personne au long d'une connection au système de réunions virtuelles	15
2.5	Architecture de métamodélisation d'UML	16
2.6	Structure du design pattern Commande	20
2.7	Application de Commande au serveur de réunions virtuelles	21
2.8	Facilitation de l'extension par refactoring préliminaire	25
3.1	Définition d'une transformation par un modèle M_t	38
3.2	MDE à deux dimensions.	41
3.3	Cycle de vie d'une transformation de modèles	43
3.4	Architecture de MTL	46
3.5	Interface des dépositaires : découverte du métamodèle	50
4.1	Métamodèle des machines à états de <code>MicroStateMachine</code>	64
4.2	Machine à états <code>machine1</code> définie dans <code>MicroStateMachine</code>	64
4.3	Diagramme d'objets de la machine à états <code>mofmachine1</code>	64
4.4	Métamodèle des traces de <code>MicroTrace</code>	65
4.5	Test de transformations utilisant l'API	66
5.1	Précurseur pour le pattern Factory Method [89].	78
5.2	Précurseur du design pattern Observer dans la structure du serveur de réunions virtuelles (fig. 2.3).	79
5.3	Structure du serveur de réunions virtuelles (fig. 2.3) après ajout du design pattern Observer.	79
5.4	Version préliminaire de la gestion des messages	80
5.5	Délégation de la gestion des messages à des classes externes	82
5.6	Délégation de la gestion des messages à des classes externes	83
5.7	Diagramme d'états perfectible pour une personne	85
5.8	Étapes de refactoring du diagramme des états d'une personne	85

Table des extraits

3.1	Déclaration d'une librairie et initialisation d'un dépositaire de modèles	47
4.1	Définition de μ MOF	56
4.2	Méthodes de la facade du dépositaire	57
4.3	(api.ml) Méthodes disponibles pour les métaclasses.	57
4.4	(api.ml) Méthodes disponibles pour les méta-associations.	57
4.5	(api.ml) Méthodes disponibles pour les méta-éléments.	58
4.6	(api.ml) Accesseurs des rôles.	58
4.7	Représentation O'Caml des machines à états	59
4.8	Représentation O'Caml du métamodèle μ MOF des machines à états	60
4.9	Traduction des machines à états, de O'Caml vers μ MOF	61
4.10	Traduction des machines à états, de μ MOF vers O'Caml.	62
4.11	Les deux représentations de la machine à états en fig. 4.2	63
4.12	(microTrace.ml) Représentation des traces par un type O'Caml et en tant que métamodèle issu de μ MOF.	65
4.13	Traduction O'Caml/ μ MOF des traces	65
4.14	Extraction de traces d'une machine à états	67
4.15	Réimplémentation de l'extraction de traces avec l'API	67
4.16	Initialisation d'un dépositaire en MTL.	71
4.17	Déclaration d'une classe en MTL.	71
4.18	Création et destruction d'objets en MTL.	71
4.19	Gestion des associations en MTL.	71
5.1	Insertion du design pattern Observer	81
5.3	Généralisation d'une opération.	89

Première partie

Problématique

Chapitre 1

Introduction

1.1 Motivation

La diffusion des technologies de l'information a créé de nouveaux domaines d'activité; de nouveaux besoins et, en réponse, de nouveaux métiers sont apparus. Cette évolution se répercute sur le cycle de vie des systèmes informatiques : alors qu'à priori un système n'a à s'adapter qu'aux évolutions du domaine métier, on s'aperçoit maintenant que les technologies apparaissent, évoluent et éventuellement disparaissent plus vite que les concepts métier qu'elles servent à implémenter. Il est également fréquent que plusieurs technologies différentes se fassent concurrence pour un même domaine d'application, ce qui pose des problèmes d'interopérabilité. En fait, un système évolue suivant un axe fonctionnel, mais aussi suivant un axe technologique, et ces deux axes se révèlent être des dimensions orthogonales : de nouvelles plateformes technologiques apparaissent indépendamment des nouveaux besoins fonctionnels, et vice-versa.

La complexité technologique, la taille et la durée de vie des systèmes considérés influe aussi sur les compétences requises pour leur exploitation — au sens large : développement, déploiement et maintenance. Un système informatique est constitué de plusieurs briques logicielles coopérant entre elles ; ces briques qui peuvent être développées indépendamment les unes des autres par des fournisseurs différents, sont souvent destinées à être distribuées à grande échelle sur différentes plates-formes logicielles ou matérielles, mais aussi géographiquement. L'adoption, la mise en place puis la maintenance d'un tel système représentent donc un investissement considérable.

Ces contraintes de modularité, d'adaptabilité présentes lors de la conception et le développement de systèmes informatiques de plus en plus complexes font apparaître des problèmes techniques et méthodologiques qui ne se posent pas pour des projets de moindre envergure. Pour résoudre ces problèmes, de

nombreux outils et méthodes sont apparus qui favorisent la réutilisabilité, la réactivité et la capitalisation sur le savoir-faire : patrons de conception, canevas d'application¹, architectures à composants, programmation par aspects, refactoring. . . En facilitant la réutilisation et le changement — on parle de méthodes *agiles* [19] — ces outils aident à gérer la taille du logiciel et la variabilité des exigences.

Cependant, la complexité du développement reste un problème majeur dans le cas de systèmes de grande envergure. Pour gérer cette complexité, on se tourne vers l'idée de modélisation :

« A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality. »

— Jeff Rothenberg [105]

Aujourd'hui, le langage de modélisation unifié (UML) [121], standardisé par l'Object Management Group (OMG) a été largement accepté par l'industrie et s'est établi comme le langage commun pour l'analyse et la conception en génie logiciel orienté objet. Dans cette optique, l'OMG cherche à promouvoir plus encore l'utilisation de modèles au niveau d'abstraction le plus pertinent tout au long du processus d'analyse, de conception et de développement, en proposant l'architecture dirigée par les modèles (MDA, Model-Driven Architecture).

1.2 Contribution

Dans l'approche du MDA, l'expertise de mise en œuvre ou d'implémentation occupe une place particulière car elle est incarnée sous la forme de *transformations de modèles*. De même, puisqu'ils facilitent la manipulation de programmes, les outils et méthodes agiles se trouvent au même niveau conceptuel que les transformations de modèles (la méta-programmation), et peuvent être vus comme tels.

En fait la finalité d'une approche telle que le MDA est de fournir à tous ces outils un cadre conceptuel, une vision unificatrice, afin de pouvoir bénéficier des avantages que cela apporte : processus commun, interopérabilité, traçabilité, pérennisation des résultats. . . Il y a donc beaucoup à gagner à intégrer ces outils au sein d'une approche telle que le MDA, mais cela est impossible

1. Termes français utilisés ici pour *design patterns* et *frameworks*, respectivement.

sans une architecture de manipulation de modèles cohérente et facilement intégrable tant au niveau méthodologique qu'au niveau outillage dans un processus de développement déjà très complexe.

Constatant que le développement de transformations de modèles est une activité cruciale dans le cycle de vie d'un logiciel, puisqu'elle conditionne les possibilités d'adaptation et d'évolution de celui-ci, nous avons cherché à faciliter le développement de transformations à travers une architecture réutilisant des concepts communs et permettant aux développeurs de transformations de transposer les techniques, outils et processus qu'ils maîtrisent déjà. D'autre part, les transformations de modèles sont la concrétisation d'une expertise spécifique aux domaines des modèles manipulés, c'est-à-dire à leurs métamodèles ; leur complexité est donc liée à la taille de ces métamodèles, et croît au fur et à mesure que l'expertise s'accumule. On est donc face à une contradiction : d'une part les transformations de modèles ont pour but de cristalliser l'expérience de manipulation de modèles dans une optique de capitalisation, mais d'autre part le haut degré de technicité et la complexité structurelle des transformations rendent difficile leur maintenance et leur réutilisation.

Ces difficultés sont d'autant plus gênantes que les transformations représentent un artefact précieux pour les perspectives d'évolution du logiciel. Cependant, une grande partie des problèmes de conception et de développement qui se posent pour les transformations sont similaires à ceux qui se posent pour les modèles ; nous proposons donc d'exploiter la puissance des techniques de conception et de modélisation orientées objets et les possibilités d'expression d'UML pour développer les transformations de modèles. Pour répondre à cela nous avons défini la sémantique d'un langage de transformation proche d'UML qui permette de manipuler de manière transparente des modèles issus de différents métamodèles. En effet, si UML est le langage prédominant dans le domaine, il faut pouvoir gérer non seulement les déclinaisons d'UML créées *via* les mécanismes d'extension tels que les profils, mais aussi d'autres langages de modélisation tels que Software Process Engineering Metamodel (SPEM) [112].

Finalement, nous souhaitons montrer l'intérêt de cette approche des transformations de modèles et de l'architecture associée par le développement d'outils de modélisation agile. Les deux applications sont d'une part l'assistance à l'insertion de design patterns et d'autre part les refactorings, appliqués aux modèles UML.

1.3 Plan du document

Nous présentons ici la structure de ce document. Le chapitre 2 décrit le contexte technologique dans lequel se sont déroulés les travaux présentés ici. Nous abordons tout d'abord les caractéristiques des approches de développement par objets, puis nous présentons le langage UML et les technologies associées. Les transformations de modèles étant des méta-programmes, il est primordial de détailler l'architecture en quatre niveaux de métamodélisation d'UML afin d'en avoir une vision d'ensemble claire. Dans le reste du chapitre nous faisons un état de l'art des différentes techniques existantes se rapprochant à la métaprogrammation et aux transformations de modèles.

Le chapitre 3 présente la première partie de cette étude ; nous proposons une vision du MDA plus large que la simple « compilation » de modèles de plus en plus spécifiques à une plate-forme, exploitant plus largement les possibilités de transformation au niveau d'abstraction des modèles manipulés. Pour être concrétisable, cette vision requiert cependant un outillage dont les fondations ont une sémantique bien définie, laquelle est présentée au chapitre 4.

Le chapitre 5 présente la mise en pratique de ces travaux par laquelle nous montrons comment l'architecture proposée permet de concevoir des transformations de programme à un niveau d'abstraction plus élevé que le code source. Deux applications liées sont présentées : l'insertion de design patterns et l'application de refactorings dans des modèles UML. Nous posons ensuite les bases d'un catalogue de refactorings pour UML.

Finalement, nous concluons par un bilan de notre contribution et proposons quelques perspectives dans le chapitre 6. Les travaux effectués durant cette thèse ont donné lieu aux publications [27, 97, 115] et plus indirectement à [75, 76] ; ces références bibliographiques sont reprises page 119. Les acronymes utilisés dans ce document sont définis dans le glossaire en page 121.

Chapitre 2

Contexte technologique et état de l'art

2.1 De l'approche objet à l'ingénierie des modèles

2.1.1 Conception orientée objets

Le paradigme objet est au départ une approche cherchant à simuler le monde réel, dont les principes ont été cristallisés avec Simula67 [38]. On programme par assemblage d'entités bien définies ; chacune de ces entités ou *objets* joue un rôle bien précis : elle fournit un ensemble de fonctionnalités cohérentes et mémorise les données nécessaires à celles-ci. Les fonctionnalités du programme résultent de la coopération des objets qui communiquent en s'échangeant des messages [11]. Les systèmes à objets sont fondés sur les caractéristiques suivantes :

L'encapsulation renforce la structure modulaire d'un programme objet en masquant la structure interne de chaque objet derrière une interface bien définie. Les communications entre objets se font donc par des messages indépendants de l'implémentation de leur destinataire, ce qui augmente la séparation des responsabilités¹ entre les différents objets, et donc leur réutilisabilité.

Le polymorphisme réalisé par un mécanisme de liaison dynamique², autorise l'invocation d'un même message sur des objets de type différent. Le comportement du programme dépend donc de l'identité à l'exécution de l'interlocuteur auquel on envoie un message.

1. separation of concerns

2. Le terme « liaison *tardive* » est aussi employé.

La délégation est une utilisation particulière de la communication entre objets : le responsable d'une tâche en délègue tout ou partie à un objet subordonné.

Les classes définissent les caractéristiques d'une famille d'objets, et produisent les objets de cette famille par instanciation.

L'héritage est un moyen adopté par beaucoup de langages objets pour organiser les classes en une hiérarchie. Les classes d'objets sont définies par réutilisation et spécialisation de classes préexistantes, et via le polymorphisme les objets d'une sous-classe sont utilisables en lieu et place des objets de leurs classes ancêtres.

Ces caractéristiques montrent bien les avantages apportés par le paradigme objet par rapport à d'autres approches comme la programmation procédurale ou fonctionnelle. D'une part les concepts de modularité et de réutilisabilité ont un statut prépondérant, ce qui est un avantage pour gérer des problèmes de taille et de complexité importantes. D'autre part l'approche de la programmation par simulation d'entités réelles ou virtuelles facilite la conception des programmes dont la complexité se situe dans le flot de contrôle plutôt que dans le traitement des données : programmes réactifs, parallèles, l'exemple typique étant les systèmes d'interface graphique. L'approche objet a donc été largement adoptée dans le monde industriel et est aujourd'hui appuyée par de nombreux langages (Smalltalk [55], Objective-C [11], Eiffel [65, 82], Java [57], C# [42]) et outils : bibliothèques de classes ou de composants (Cocoa [31], Corba³ [29], COM [32], JavaBeans [85]), environnements de développement (Eclipse [45], NetBeans [88]).

2.1.2 Méthodes de développement

Si l'approche objet a des avantages certains, elle a aussi son lot d'inconvénients. Le plus perturbant est probablement que le flot de contrôle se retrouve dispersé à travers de nombreux objets sans véritable centre :

« L'objet c'est peut-être très bien, mais on ne sait plus où se trouve le programme... »

— *Anonyme*

Mais si les techniques orientées objet ont été rapidement adoptées, c'est aussi pour répondre à une forte demande en développement, à une période de grands changements dans le monde informatique : apparition des réseaux, des micro-ordinateurs, de l'informatique grand public, etc. Des méthodes d'analyse, conception et développement ont donc été conçues pour répondre

3. Common Object Request Broker Architecture

aux contraintes imposées par des développements de grande ampleur. Pour simplifier la réutilisation d'un logiciel et mémoriser les bonnes pratiques de conception, de nombreuses méthodes documentent usage et conception à travers un langage graphique ; OMT [107], OOSE [64], Booch [22], OORam [101] sont parmi les principales.

Cette diversité de méthodes plus ou moins spécialisées amène cependant des difficultés de communication entre projets ou entre outils, et le besoin d'une notation commune s'est rapidement fait sentir. Cette recherche d'un consensus entre les différents langages de modélisation a conduit Grady Booch, James Rumbaugh et Ivar Jacobson, auteurs respectifs des méthodes Booch, OMT et OOSE à proposer UML, le langage de modélisation unifié [121] — dans ce document on utilise la version 1.5, UML 2.0 n'étant pas encore officiellement publié. S'il reprend les points forts des trois langages dont il est issu, UML n'est qu'une *notation* : il ne dit rien du processus de développement associé.

La conception d'un logiciel complexe est un processus ardu, nécessitant souvent plusieurs itérations avant d'obtenir un résultat fonctionnel, réutilisable, clair et cohérent. Si les premières qualités découlent principalement d'une bonne méthodologie de conception, la cohérence et surtout la clarté d'un système sont plus subjectives et exigent qu'on identifie les structures porteuses de sens au sein de l'architecture⁴. Du point de vue de l'organisation du développement, l'encapsulation et la séparation des responsabilités — explicitées dans le paradigme objet — favorisent les approches incrémentales et le cycle de développement par raffinements successifs en spirale (fig. 2.1) à la base des méthodes agiles [19].

2.1.3 Sujet d'étude : le système de réunions virtuelles

Pour introduire les principaux éléments de la syntaxe d'UML et illustrer l'utilisation du langage dans le processus de développement, nous décrivons la conception d'un système jouet qui servira de « sujet d'étude » tout au long de ce document. Avant de passer à la conception proprement dite, présentons brièvement son cahier des charges.

4. Ceci est particulièrement vrai dans le cadre des *frameworks* : ils doivent fournir une infrastructure dans laquelle il suffit d'insérer du code métier pour obtenir une application. Ces frameworks doivent donc fournir un haut degré d'abstraction tout en conservant la majeure partie des fonctionnalités de la couche inférieure.

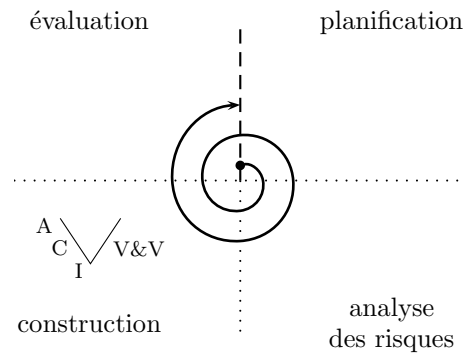


FIG. 2.1 – Cycle de développement en spirale. Chaque tour correspond à l'identification puis ajout d'une fonctionnalité simple, et garde le système dans un état fonctionnel. Suivant le contexte, la durée d'un cycle varie largement : de quelques minutes pour corriger une coquille à quelques jours pour un changement plus profond.

Le logiciel considéré est un système permettant de tenir des réunions à distance, suivant un principe similaire à l'IRC⁵ ou à Jabber⁶. Les communications se font à travers un réseau local ou Internet selon une architecture *clients-serveur* classique, et nous nous intéresserons à la partie serveur. Via leur logiciel client, les utilisateurs doivent pouvoir :

- planifier des réunions virtuelles (définition du sujet, ordre du jour, date de début et durée prévue),
- consulter les détails d'organisation d'une réunion,
- entrer et sortir virtuellement d'une réunion précédemment ouverte,
- participer aux réunions ;

et pour les réunions dont ils sont organisateurs :

- modifier les détails d'organisation,
- ouvrir et clôturer la réunion.

Suivant le mode de fonctionnement souhaité, un organisateur peut choisir des réunions parmi plusieurs types, par exemple :

- les réunions standard n'imposent pas de contrainte particulière : tout le monde peut parler à tout instant ;
- les réunions modérées ont un modérateur qui désigne les intervenants successifs parmi ceux qui demandent la parole ; la modération peut être automatique, suivant une politique donnée ;
- les réunions secrètes ne sont visibles que des participants désignés par l'organisateur lors de la création.

Les réunions peuvent aussi se voir attribuer des caractéristiques comme des restrictions de droits d'accès ou de mode de communication (texte pur ou formaté par exemple).

5. Internet Relay Chat, voir la RFC 1459 : <http://rfc.sunsite.dk/rfc/rfc1459.html>

6. <http://www.jabber.org>

Les participants à une réunion peuvent communiquer à travers divers types de données : texte pur ou mis en forme, images, flux audio ou vidéo, fichiers. Cependant le système doit rester flexible et s'adapter aux capacités de transfert et de restitution des différentes plateformes clientes existantes : téléphone mobile, assistant de poche, ordinateur portable ou station de travail ; le serveur de réunions virtuelles doit donc savoir adapter le contenu diffusé en fonction des destinataires, ou à défaut adopter une solution alternative — par exemple stocker un fichier pour consultation ultérieure — et en notifier les intéressés.

2.1.4 Analyse avec UML

Le langage UML propose plusieurs vues pour exprimer différentes facettes fonctionnelles, statiques, dynamiques et d'implémentation d'un modèle. Ces vues couvrent la plupart des phases du développement :

- les *cas d'utilisation* servent à identifier les besoins des utilisateurs (page 12) ;
- les *diagrammes de structure* servent d'abord à établir le modèle d'analyse des concepts que le logiciel manipule (page 13), puis deviennent l'une des vues essentielles à la conception détaillée par raffinements successifs jusqu'à un niveau très proche du code ;
- les *diagrammes d'états* expriment les évolutions de l'état d'une entité (une classe en général) et ses réponses en réaction à des événements extérieurs ; les *diagrammes d'activité* montrent l'enchaînement de processus et sont plus adaptés pour exprimer un flot de contrôle ;
- les interactions entre entités peuvent être spécifiées par des *diagrammes de séquences* qui arrangent les événements le long d'un axe temporel, ou par des *diagrammes de collaborations* qui mettent en relief les relations et les rôles des participants sans donner de dimension propre au temps ; les diagrammes de séquences sont utiles tant à l'expression des besoins qu'aux tests ;
- l'implémentation est documentée par des *diagrammes de composants* montrant les relations et dépendances entre sous-systèmes, ainsi que par des *diagrammes de déploiement* indiquant la configuration du système à l'exécution : répartition des ressources et des composants qui les utilisent.

Un langage de contraintes nommé OCL permet de préciser textuellement la sémantique du modèle exprimé par les vues graphiques, en définissant des invariants ou des pré/post-conditions sur les éléments du modèle. Typiquement, les contraintes OCL expriment les contraintes métier et les contrats des opérations.

Nous présentons rapidement ici l'analyse du système de réunions virtuelles, pour introduire les notations essentielles du langage UML et d'OCL.

Cas d'utilisation

La première étape de la modélisation consiste à déterminer les besoins des utilisateurs du système, et ceci plus précisément qu'en langage naturel comme dans la plupart des cahiers des charges. Un diagramme de cas d'utilisation ou *use-case* met en relation le système modélisé sous forme d'un rectangle avec des acteurs représentant l'environnement extérieur ; les utilisations possibles du système sont représentées par des ellipses portant le nom du cas d'utilisation et connectée aux acteurs concernés. L'ensemble des cas d'utilisations peut être structuré par des relations de spécialisation, inclusion, ou extension.

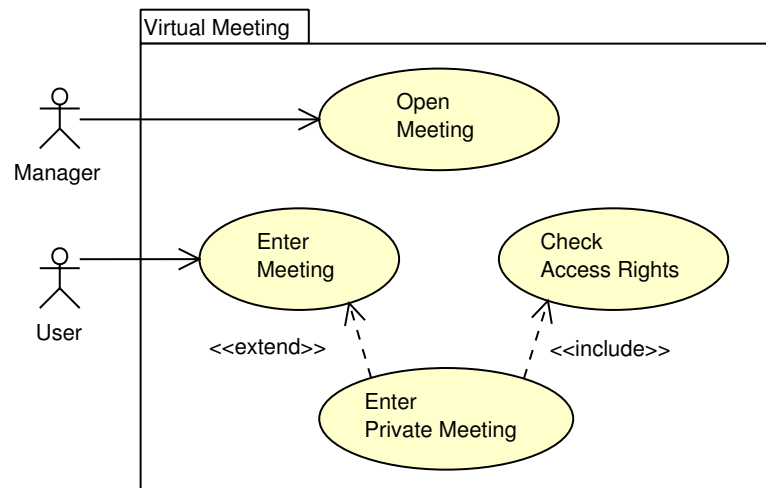


FIG. 2.2 – Une partie des cas d'utilisations des réunions virtuelles

Les *use-case* servent à identifier la frontière entre le système à concevoir et son environnement, ainsi que les fonctionnalités à travers lesquelles les acteurs extérieurs peuvent interagir avec le système.

Diagrammes statiques

UML définit plusieurs vues destinées aux aspects statiques d'un modèle : les diagrammes⁷ de classes, d'objets, de packages⁸ et de déploiement. Ces diagrammes explicitent quelles entités sont présentes dans le système et quelles sont leurs caractéristiques et leurs relations, à différents niveaux de granularité. Un diagramme de classes définit les concepts présents dans le logiciel et leurs relations : position dans la hiérarchie d'héritage, associations, attributs et comportements ; les packages servent quant à eux à organiser les différentes classes en espaces de nommage et en sous-systèmes.

Un diagramme d'analyse modélise l'*univers* du système ; les concepts représentés sont donc très abstraits mais regroupent la quasi-totalité des concepts centraux du système. Dans le cas du serveur de réunions virtuelles, figure 2.3, on retrouve un package `VirtualMeeting` regroupant en particulier les classes `Meeting` et `Person`. Précisons qu'ici une instance de `Person` n'est pas une personne physique mais son modèle, autrement dit l'objet informatique regroupant les informations que le système connaît d'un utilisateur particulier. Les relations traduisent ce qui est dit dans le cahier des charges, et à la fin de l'analyse seules les opérations essentielles sont indiquées dans les classes.

À la conception, le modèle « idéal » d'analyse sera raffiné et enrichi, pour devenir un modèle « réaliste » prenant en compte d'une part les contraintes extra-fonctionnelles — performances, flexibilité, portabilité, maintenabilité, fiabilité, sûreté, tolérance aux défaillances, etc — et d'autre part les contraintes de la programmation. À ce niveau on considère les classes non plus comme des concepts abstraits modélisant le domaine métier mais comme des éléments de programme qui doivent collaborer pour maintenir l'état du programme et réagir aux événements extérieurs. On raffine donc le modèle d'analyse, notamment en remplaçant les relations générales par des agrégations ou des accesseurs, en précisant leur navigabilité, etc.

Les packages permettent d'organiser les classes du modèle en modules relativement indépendants ou partageant une caractéristique commune : objets métier, couche applicative, couches d'infrastructure (ici `Networking`, `HttpServer` et `StreamingServer`), versions spécifiques à une plateforme d'un ensemble de classes, etc.

7. En fait, ces diagrammes sont plus séparés par l'usage que par leur définition dans le langage ; on peut tout-à-fait montrer des classes et des objets dans un package, par exemple.

8. Paquet ou paquetage en français.

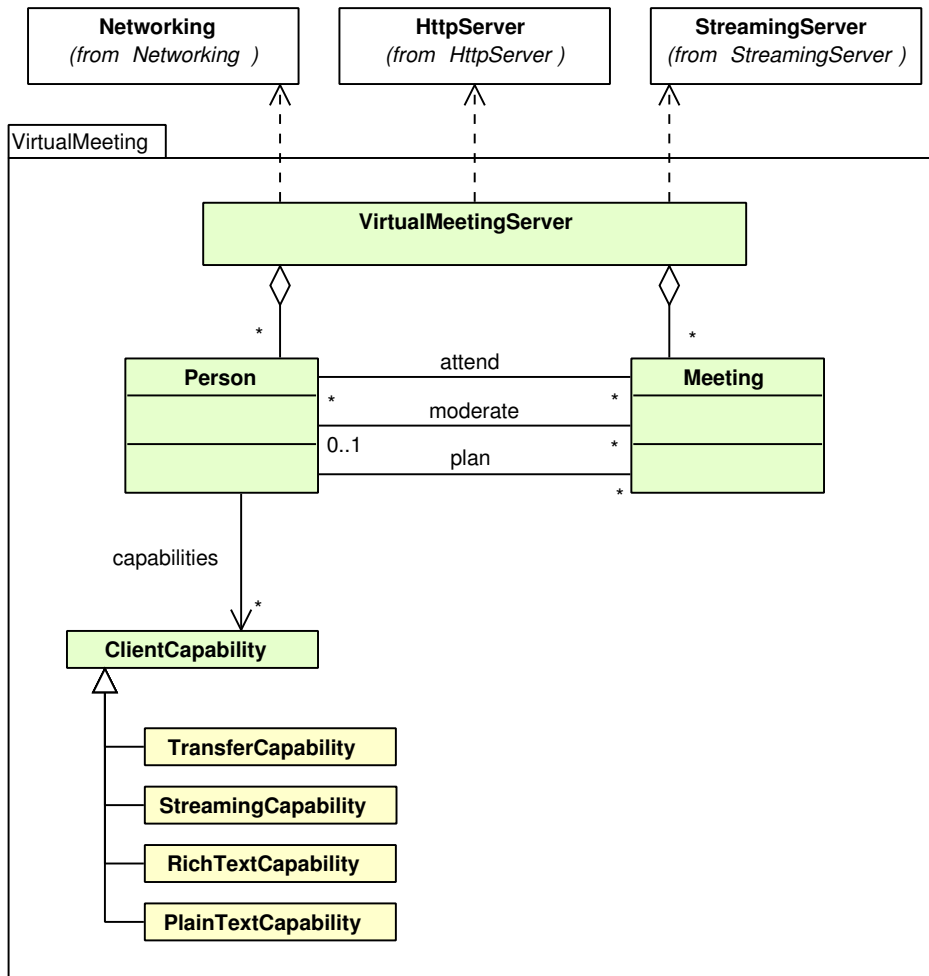


FIG. 2.3 – Diagramme de structure des classes à l'analyse des réunions virtuelles

Diagrammes d'états, de séquences et de collaborations

Les *diagrammes d'états* d'UML sont une variation des statecharts de David Harel [60]. Ils expriment les évolutions de l'état d'une entité — classe, sous-système — et ses réactions suite à la réception d'événements extérieurs ; voir par exemple en figure 2.4 le diagramme des états d'une personne dans le système de réunions virtuelles.

Les *diagrammes de séquence* arrangent les événements constituant une interaction — message, création ou destruction d'objets — le long d'un axe temporel ; les associations entre participants ne sont pas représentées.

D'un autre côté, les *diagrammes de collaboration* mettent en relief les rela-

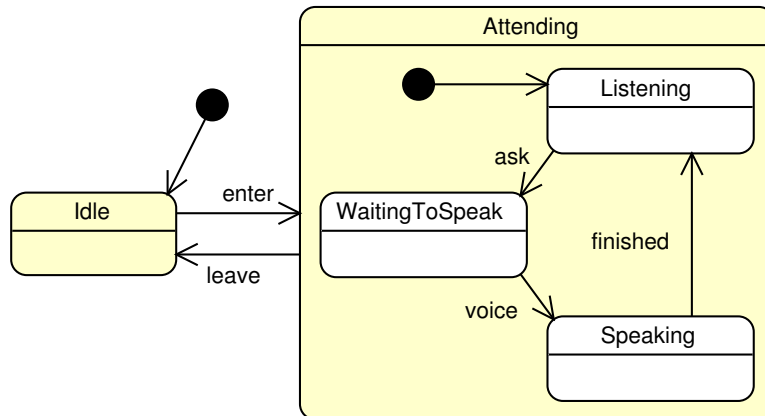


FIG. 2.4 – Diagramme des états d'une personne au long d'une connection au système de réunions virtuelles

tions entre les acteurs et le temps n'y a pas de dimension graphique propre : les événements sont numérotés chronologiquement. Visuellement, les diagrammes de collaboration se rapprochent des diagrammes d'objets, n'en conservant que les informations pertinentes pour la collaboration représentée, et précisant les rôles des participants et leurs communications.

OCL : Object Constraint Language

La syntaxe d'UML est efficace pour mettre en images la structure d'un logiciel, mais les différents types d'associations et leurs multiplicités ne peuvent pas exprimer certains invariants pourtant très simples. Ce manque est comblé par le langage de contraintes OCL [121, 126] qui permet d'exprimer des invariants et des pré/post-conditions dans le contexte d'éléments particuliers d'un modèle.

Par exemple, si les dates de début et de clôture d'une réunion sont représentées par deux attributs de `Meeting`, on explicitera la dépendance temporelle en posant l'invariant suivant :

```

context VirtualMeeting::Meeting
inv: -- relation invariante entre les valeurs de deux attributs
    scheduledStart < scheduledEnd
  
```

Les contraintes peuvent aussi servir à préciser des relations entre plusieurs éléments :

```

context VirtualMeeting::ModeratedMeeting
inv: -- le modérateur fait partie des participants
    attending->includes(moderator)
  
```

À l’analyse OCL sert à exprimer les contraintes métier tirées du cahier des charges, et à la conception il est très utile pour poser des contraintes d’implémentation ou pour spécifier des contrats sous forme de pré/post-conditions attachées aux méthodes.

2.1.5 Métamodélisation

UML est défini par une architecture en niveaux de *méta-modélisation*. On appelle M_0 le niveau des données ou des objets du monde « réel » qu’on cherche à modéliser au niveau d’abstraction supérieur M_1 . Pour exprimer les modèles de M_1 , on utilise un langage tel qu’UML, dont on définit la syntaxe par modélisation, au niveau M_2 ; le modèle d’UML est donc un *métamodèle*. Les concepts de niveau M_n sont ainsi exprimés en utilisant un langage défini dans M_{n+1} . Cet empilement de niveaux d’abstraction pourrait continuer indéfiniment, mais en pratique le langage de métamodélisation défini en M_3 est suffisamment abstrait et minimal pour se décrire lui-même ; dans l’architecture de l’OMG ce langage est le Meta-Object Facility (MOF) [84]. La figure 2.5 fait le parallèle avec le domaine des langages de programmation et l’Extended Backus-Naur Form (EBNF).

M_3	MOF	EBNF
M_2	métamodèle UML	grammaire Java
M_1	modèle UML	programme Java
M_0	concepts « réels »	données, exécutions

FIG. 2.5 – Architecture de métamodélisation d’UML. Chaque niveau M_{n+1} définit le langage servant à exprimer les concepts de niveau M_n . L’astuce est qu’à partir de M_3 , tous les niveaux supérieurs sont identiques : de même que EBNF a une grammaire describable en EBNF, MOF suffit à sa propre description.

Un modèle est une vue simplifiée de la réalité pour une application particulière : il définit le sous-ensemble des données intéressantes et des comportements possibles, de la même manière qu’un programme définit un ensemble d’exécutions ou de traces possibles. Les différents langages de modélisation sont définis par des métamodèles similaires à des grammaires et spécifiques à un domaine d’activité : conception logicielle (UML), processus (Software Process Engineering Metamodel, SPEM [112]), différents paradigmes de programmation... Dans la vision de l’OMG, MOF est le langage standard de description de métamodèles, de même qu’EBNF sert pour l’immense majorité des langages textuels. Finalement, le niveau M_3 suffit à sa propre définition : MOF et EBNF décrivent des langages, donc peuvent se décrire eux-mêmes.

XMI : XML Model Interchange

Cette architecture de métamodélisation rend possible la définition d'un format d'échange commun à tous les métamodèles MOF, donc particulièrement intéressant pour l'interopérabilité des outils de manipulation de modèles.

Ce format standard d'échange de modèles entre outils est XML Model Interchange (XMI); plutôt qu'un format, il s'agit d'une famille de formats qui standardise la génération d'une DTD à partir de tout métamodèle compatible MOF. Un modèle est donc représenté par une sérialisation sous forme d'éléments XML d'instances de son métamodèle. Les éléments XML utilisés correspondent aux classes dudit métamodèle : par exemple on peut trouver des éléments `<UML:UseCase>` ou `<UML:Class>` dans la sérialisation d'un modèle UML.

MDA : Model-Driven Architecture

L'idée à la base de l'initiative Model-Driven Architecture (MDA) de l'OMG est qu'il devrait être possible de capitaliser sur des modèles du domaine métier qui soient indépendants de toute plateforme (PIM : Platform-Independent Models) et de dériver par transformation de ces PIM des modèles spécifiques à la technologie retenue (PSM : Platform-Specific Models) et finalement du code. Mais pour certains secteurs d'activité, la valeur ajoutée d'une entreprise ne se situe pas seulement dans sa connaissance du domaine métier — le PIM — mais aussi dans l'expertise de conception et de mise en œuvre nécessaire pour faire fonctionner un système avec les contraintes d'un environnement réel — la transformation du PIM vers le PSM.

2.2 Automatisation de l'activité de conception

Pour travailler plus efficacement et réagir plus facilement aux changements de contraintes de développement, les experts de la conception de systèmes informatiques ont proposé différents outils — méthodologiques ou technologiques — rendant plus efficace ou agile la conception et le développement. Le but de ces outils est d'aider le développeur en prenant en charge de manière systématique ou automatique certains aspects de l'activité de conception. Nous les présentons ici pour montrer que les transformations de modèles permettent d'exploiter ces outils dans un cadre unique et cohérent, donc plus pratique et efficace.

2.2.1 Patrons de conception

Les *design patterns*, ou patrons de conception, sont un concept initialement proposé par Christopher Alexander [8–10, 33] dans le domaine du design et de l'architecture en génie civil ; il en donne la définition suivante :

« Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice. »

— *Christopher Alexander*

Le livre à l'origine de l'engouement pour les design patterns en informatique en dit quant à lui :

« A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context. »

— *Gamma et al. [53]*

James Coplien a défini et expliqué les concepts à la base des patterns, en mettant en parallèle la vision de Alexander et son application dans le domaine informatique [33]. Il présente notamment la structure communément adoptée pour la description d'un pattern : nom, intention, problème, contexte, forces, solution, esquisse d'implémentation, contexte résultant.

Les patterns proposés dans [53] sont classés en trois grandes catégories : création, structure et comportement. Ils s'appliquent donc essentiellement à la conception de logiciel, comme nous allons le voir dans les pages suivantes, mais il existe des patterns pour d'autres domaines comme par exemple la gestion du processus ou le travail en équipe [19].

Anti-patterns

Les design patterns d'Alexander ont pour but de cataloguer et documenter des bonnes pratiques. Les anti-patterns⁹ [26] en sont un détournement intéressant : ils montrent comment partir d'un problème pour arriver à une

9. <http://c2.com/cgi/wiki?AntiPattern>

mauvaise solution, dans le but de documenter des mauvaises situations et le moyen d'en sortir — en général par un choix plus adapté bien qu'il puisse paraître moins attractif au départ.

Un exemple d'anti-pattern est « l'anémie du modèle de domaine »¹⁰ : le modèle montre correctement la structure complexe des objets du domaine, comme le ferait un bon modèle ; le problème devient évident quand on se rend compte que les objets de domaine n'ont que très peu de comportement. Un tel modèle anémique impose les coûts d'un modèle objet sans en offrir les bénéfices : par exemple la mise en correspondance avec une base de données relationnelle requiert toute une couche logicielle ; le modèle objet n'est donc justifié que si il sert à organiser une logique complexe.

Utilisation à la conception

La phase initiale de la conception est souvent difficile car il faut passer d'un modèle très abstrait à l'architecture d'un logiciel. Juger des différentes possibilités pour aboutir à une architecture non seulement fonctionnelle, mais aussi évolutive sans être trop coûteuse, requiert expérience et intuition ; en insérant des design patterns dans le modèle d'analyse, on réutilise des solutions « clés en main » et on peut se concentrer sur les choix importants.

Par exemple, pour obtenir un modèle de conception du système de réunions virtuelles, il faut s'attaquer à plusieurs problèmes :

- traiter les messages provenant des clients, en utilisant par exemple le design pattern Commande [53, p. 233],
- gérer le parallélisme à la réception des messages puis pour les traiter (utilisation du pattern Réacteur [108]),
- créer et maintenir l'état des objets métier (Fabrique et État [53, p. 107 et 305]),
- prendre en compte les changements dynamiques des caractéristiques d'une réunion (Décorateur [53, p. 175]),
- implémenter la logique d'adaptation des médias en fonction des destinataires d'un message (Stratégie [53, p. 315]).

Prenons l'exemple du traitement des messages : le serveur va recevoir des messages par l'intermédiaire d'une couche de communication — qui ne sera pas détaillée mais qu'on pourrait implémenter par un composant SOAP¹¹ ou Corba par exemple — et il faudra décoder ces messages, vérifier leur validité et leur provenance, puis y réagir. Au vu des capacités de l'application, on peut déjà dire que les types de messages possibles seront assez nombreux

10. <http://martinfowler.com/bliki/AnemicDomainModel.html>

11. Simple Object Access Protocol (SOAP 1.1), <http://www.w3.org/TR/soap/>

— quelques dizaines — et constituent des groupes relatifs aux différents cas d’utilisation. Le pattern Commande (fig 2.6) consiste à encapsuler un message sous la forme d’un objet, pour pouvoir le manipuler ensuite de manière abstraite ; chaque commande concrète est responsable d’aller chercher les informations nécessaires. En découplant ainsi le traitement de chaque message du reste de l’application, on réduit l’impact sur le système de l’ajout d’une commande ; les cas d’utilisation et les messages correspondants pourront donc être gérés progressivement tout en gardant un système exécutable.

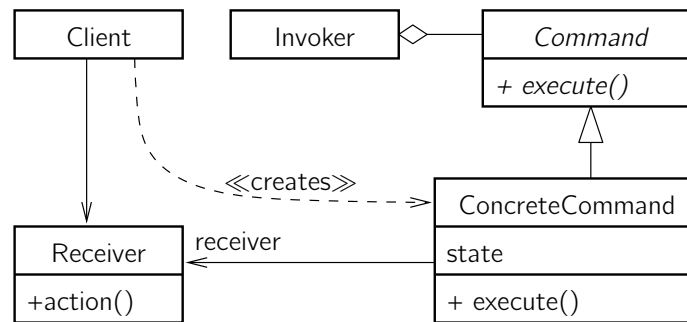


FIG. 2.6 – Structure du design pattern Commande [53, p. 233]. Parmi les classes du système, il faut identifier celles qui joueront les rôles définis ici.

Pour appliquer le pattern au systèmes de réunions virtuelles, il faut distribuer les rôles en créant au besoin de nouvelles classes (fig. 2.7). En particulier pour chaque type de message, on aura une commande concrète implémentant la réaction au message ; ces commandes concrètes se conforment à une interface spécifiée par une classe abstraite `Command`. Le rôle du *récepteur* est joué par différentes classes du système suivant chaque commande concrète : `Person`, `Meeting`... Le *client* et l'*invocateur* peuvent être joués par la même classe, `VirtualMeetingServer` en l’occurrence.

De nombreuses approches ont été proposées pour formaliser la description et l’intégration des design patterns dans un langage de modélisation [46, 47, 74, 77, 114]. D’une part, en utilisant une construction syntaxique identifiant dans un modèle les classes participant à un pattern, le concepteur peut s’abstraire de détails d’implémentation connus et se concentrer sur des tâches plus importantes. D’autre part, on a besoin d’une telle représentation abstraite des design patterns pour pouvoir automatiser leur insertion [47, 89, 90] ou leur identification [6, 79] dans des programmes existants. Un outil supportant les patterns peut vérifier que les contraintes sont respectées — par exemple qu’un sujet notifie bien tous ses observateurs quand son état change — ou libérer le programmeur de tâches rébarbatives comme l’ajout de méthodes de redirection dans un visiteur ou un décorateur.

L’approche proposée dans [74] utilise des collaborations au niveau du mé-

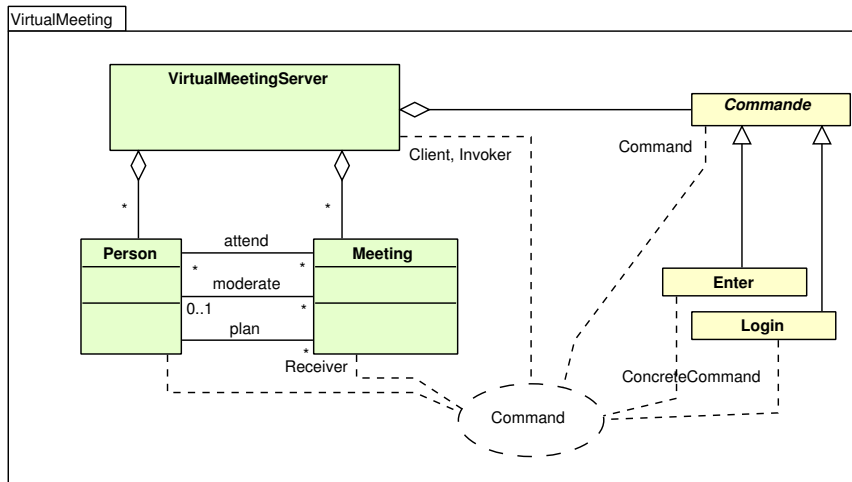


FIG. 2.7 – Application de Commande au serveur de réunions virtuelles. La présence du pattern est mise en évidence en syntaxe UML par l'ellipse en pointillés, indiquant les rôles que jouent chacune des classes.

tamodèle UML pour spécifier les rôles et relations des participants à une occurrence de pattern. Dans un design pattern, les multiplicités doivent être contraintes : par exemple un visiteur a autant de méthodes `visitElement()` qu'il y a d'éléments visitables, et il y a une implémentation de la méthode `accept(visitor)` dans chacun de ces éléments, indépendamment du nombre de visiteurs. Ces multiplicités sont exprimés par des stéréotypes «tribe» et «clan» dans [74] selon la notation de [46], et par des dimensions dans [77].

Une étude comparative de différents outils [23] montre que différentes approches du support à l'utilisation des design patterns ont été envisagées :

- au niveau d'un langage de programmation par l'utilisation de primitives représentant directement l'application de patterns dans le code ;
- par paramétrisation d'une « boîte noire » produisant le code d'implémentation d'un pattern ;
- au travers des fonctionnalités d'un environnement de génie logiciel.

Si elles donnent chacune des réponses à certains des problèmes posés, les différentes approches proposées sont fondées sur des bases différentes ; or pour être utilisables en pratique, les différents outillages doivent s'intégrer dans un cadre commun pour leur utilisation et leur exécution. L'architecture décrite dans cette thèse permet d'implanter ces différents outils au sein d'une même plate-forme d'exécution.

2.2.2 Programmation par aspects

La programmation par aspects [70] a pour but de permettre la modularisation propre du code gérant les préoccupations transverses au reste d'un programme telles que la gestion d'erreur, la synchronisation, la surveillance et l'enregistrement d'événements, le débogage. Dans les langages de programmation habituels, le code relatif à ces préoccupations se retrouve forcément disséminé à travers tout le programme, et cela nuit à la fois à la maintenabilité de ce code transverse et à la lisibilité du code principal. Le principe de la programmation par aspects est d'écrire le code transverse séparément du code principal de l'application, et de laisser à un *tisseur d'aspects* le soin d'insérer automatiquement ce code aux endroits pertinents du code principal.

Aspect-J

AspectJ [69, 71] est une des premières implémentations d'un tisseur d'aspects générique pour Java. Les aspects sont décrits avec une extension de la syntaxe Java, et tissés dans le reste du code grâce au compilateur `ajc`. Les aspects peuvent être statiques ou dynamiques : un aspect *dynamique* définit des comportements supplémentaires qui seront déclenchés à certains points bien définis de l'exécution du programme ; un aspect *statique* affecte la signature, le type du programme, en donnant de nouvelles opérations à des types existants.

Le langage de description des aspects introduit un certain nombre de constructions syntaxiques permettant de désigner les différents concepts¹² nécessaires à l'expression d'un aspect :

Les points de jointure désignent des points précis dans le flot d'exécution du programme. Suivant le modèle de points de jointure adopté on pourra coordonner le code principal et celui des aspects avec plus ou moins de précision ; les points de jointure d'AspectJ sont les appels, réceptions et exécutions¹³ de méthodes ou de constructeurs, les accès en lecture ou écriture aux champs, les exécutions de gestionnaires d'exceptions, et les initialisations de classes ou d'objets.

Les motifs réfèrent à une collection de points de jointure et à certaines valeurs à ces points de jointure. En AspectJ les motifs sont exprimés par combinaison booléenne de motifs primitifs.

12. Les termes francisés correspondent respectivement à *join point*, *pointcut*, *advice* et *aspect* dans le jargon anglophone

13. La réception est un point du flot de contrôle situé à l'intérieur de l'objet destinataire mais avant l'aiguillage vers la méthode ou le constructeur correspondant au message.

Les inserts sont des constructions similaires à des méthodes qui définissent du comportement supplémentaire à insérer aux points de jointure. En AspectJ on dispose des types d'inserts `before`, `after` et `around`, plus deux cas particuliers `after throwing` et `after returning`; si plusieurs inserts sont attachés au même point de jointure, ils seront exécutés selon une précedence et un ordonnancement précis.

Les aspects sont les unités modulaires d'implémentation transverse, composées de motifs et d'inserts mais aussi de membres Java ordinaires : à l'exécution, des instances d'aspects sont créées et peuvent encapsuler des données et du comportement, de la même manière qu'une classe Java.

Java Aspect Components

Par rapport à AspectJ, Java Aspect Components (JAC) [93] propose une approche plus dynamique des aspects, et ne définit pas de langage spécialisé : tout est fait en Java pur. En effet, JAC est un système à composants : de manière similaire aux composants EJB, les conteneurs de JAC sont des serveurs accessibles à distance, par RMI¹⁴ ou Corba [29]. Mais alors que les conteneurs EJB n'hébergent que des composants métier, c.-à-d. des objets, les conteneurs JAC peuvent aussi héberger des composants aspects. Comme les aspects ne résident pas forcément dans un même conteneur, JAC permet directement le développement d'applications *et* d'aspects distribués.

On peut écrire des aspects très proches de ceux d'AspectJ en étendant la classe Java `jac.core.AspectComponent`, mais JAC offre d'autres possibilités :

- la sémantique des classes de base peut être étendue en y attachant des méta-données structurelles ; les points d'attache (e.g. classes, méthodes, champs, collections) sont définis par un métamodèle d'exécution appelé Run-Time Type Information (RTTI) ;
- en implémentant l'interface `jac.core.BaseProgramListener`, un aspect pourra réagir à certains événements du système.

Aspects et modélisation

Les différentes vues d'UML présentent chacune un *aspect* différent du système modélisé ; dans cette optique le modèle complet est un *tissage* des vues qui le composent. De même, les travaux de Siobhán Clarke explorent les possibilités

14. Remote Method Invocation, système d'envoi de message entre objets distants en Java.

offertes par un langage de conception orienté aspects, Theme/UML [116], notamment vis-à-vis d'un langage d'implémentation tel que AspectJ [30]. Pour supporter ces approches, les outils de modélisation devraient proposer des transformations spécifiques pour le *tissage* de vues ou prenant en compte la sémantique des constructions de Theme/UML.

2.2.3 Refactorings

Dans un environnement réaliste, les logiciels doivent évoluer pour s'adapter à leur environnement ou à de nouveaux besoins. Au fur et à mesure que le logiciel est amélioré, modifié et adapté à de nouvelles exigences, le code devient de plus en plus complexe et s'éloigne de sa conception initiale, ce qui diminue la qualité du logiciel. La conséquence est que la majorité des coûts d'un logiciel sont induits par la maintenance. Ce problème n'est pas résolu en améliorant méthodes et outils de développement, car leur puissance accrue est utilisée à implémenter plus de fonctions dans le même laps de temps, ce qui ajoute à la complexité du logiciel [54].

Les *refactorings* répondent à ce problème de complexité croissante en améliorant la qualité interne du logiciel. Le terme *refactoring* a été introduit par William Opdyke dans son mémoire de thèse [91].

« Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure. »

— *Martin Fowler [51, p. xvi]*

Par extension, les refactorings sont les transformations concrètes préservant le comportement d'une application. De telles transformations n'affectent que l'apparence sans ajouter de fonctionnalité, mais permettent de mieux appréhender le système ou facilitent des modifications fonctionnelles ultérieures, comme illustré par la figure 2.8.

Le processus de refactoring se divise en plusieurs activités [80] :

1. identifier à quelles parties du logiciel devraient être refactorées ;
2. déterminer quel(s) refactoring(s) appliquer à ces endroits ;
3. garantir qu'une fois appliqué, le refactoring préserve le comportement du système ;
4. appliquer le refactoring ;
5. évaluer l'influence du refactoring sur des critères de qualité du logiciel (complexité, lisibilité, maintenabilité) ou du processus (productivité, coût, effort) ;

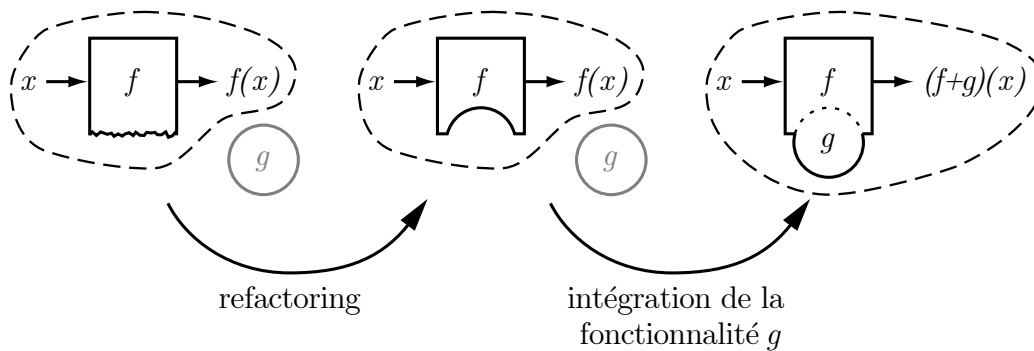


FIG. 2.8 – *Facilitation d'une extension de comportement par un refactoring préliminaire. Là où une modification nécessiterait un retour en arrière dans le processus de développement, ou serait néfaste à la structure du programme, le refactoring permet d'effectuer sans danger les modifications purement structurales préliminaires, avant d'ajouter la nouvelle fonctionnalité.*

6. maintenir la cohérence entre le code refactoré et les autres artefacts tels que documentation, spécifications, tests etc.

Dans sa thèse, William Opdyke a défini un premier catalogue de refactorings pour C++ [91], mais d'autres approches ont été proposées pour d'autres langages, notamment pour Scheme [58], Smalltalk [103, 104], Java ; plus récemment le livre de Martin Fowler [51] présente les refactorings d'un point de vue plus méthodologique et marque leur adoption dans le monde industriel.

Les publications précédentes montrent qu'il est possible de construire l'outillage de refactoring pour *a priori* n'importe quel langage objet. Cependant aucune des approches existantes n'est réellement indépendante du langage ; Tichelaar *et al.* [117] proposent un métamodèle (FAMIX) synthétisant les informations utiles au refactoring à partir de différents langages, puis valident ce métamodèle en l'utilisant pour implémenter un outil de refactoring. Selon eux, l'approche permet de réutiliser une grande partie des transformations, au prix d'une plus grande complexité et de compromis pour que les transformations soient réellement indépendantes au niveau du code. Il y a cependant des différences trop fondamentales entre langages — par exemple le typage dynamique en Smalltalk ou statique en Java — qui font que certains refactorings sont intrinsèquement spécifiques à un langage.

Un panorama du domaine

Une étude extensive des travaux sur les refactorings a récemment été menée par Tom Mens et Tom Tourwé [80] : nous la résumons ici. L'approche la plus répandue pour identifier les zones d'un programme qui bénéficieraient d'un refactoring est celle de la recherche de *bad smells* qui sont, selon Kent Beck,

« des structures dans le code qui suggèrent [...] la possibilité de refactoring » [51]; un exemple typique est la présence de code dupliqué, car cela complique la maintenance. De nombreux travaux abordent donc l'identification de *bad smells* par des approches de visualisation, de métriques ou de méta-programmation [15, 44, 49, 73, 111, 120].

La définition précise de ce qu'est la préservation du comportement est un problème récurrent : Opdyke a initialement défini cette préservation de l'ensemble des sorties du programme avant et après refactoring, pour un même ensemble d'entrées. Il suggère que cela soit assuré par des préconditions sur le programme transformé [91]. Cependant cette préservation des entrées/sorties est insuffisante pour certains domaines — par exemple des logiciels temps réel, embarqué, ou critique devront respectivement conserver des propriétés temporelles, spatiales ou de sûreté. Une approche plus pragmatique de la préservation du comportement serait de vérifier qu'une suite de tests extensive passe toujours après refactoring; malheureusement certains tests dépendants de la structure du programme qui est modifiée seront invalidés par un refactoring correct. Il faut donc adopter une méthodologie de refactoring particulière, telle que le refactoring *test-first* [41, 96]. Des approches plus formelles de la préservation sémantique d'un programmes sont possibles dans certains cas [99] mais doivent en pratique soit adopter une notion plus faible de la préservation du comportement soit limiter leur domaine d'application [78, 119].

Différentes techniques sont utilisées pour l'application de refactorings. Pour assurer la « légalité » d'un refactoring on peut lui associer des invariants, pré- et post-conditions. L'usage d'invariants a été suggéré dès les travaux de Banerjee pour les schémas de bases de données orientées objet [16]. Opdyke a adopté cette approche en y ajoutant des préconditions pour les refactorings [91], puis ces préconditions ont été spécifiées plus formellement à l'aide du calcul de prédicats du premier ordre [103]. Si on considère l'expression des refactorings, il y a une correspondance directe avec les techniques de transformation de graphes [35] : on travaille sur une représentation abstraite du programme sous forme d'un graphe. La théorie des transformations de graphes aide donc à prouver certaines propriétés des refactorings [78], ou à manipuler des programmes à différents niveaux de détails [48].

Un refactoring doit conserver deux propriétés du programme transformé : une propriété de *correction* et une propriété de *préservation*. La correction assure que le programme fonctionne sans erreur; cela inclue sa validité syntaxique, mais aussi sémantique : le programme ne doit pas déclencher d'erreur à l'exécution. Malheureusement cela est indécidable dans le cas général [59]. La propriété de préservation peut être vérifiée statiquement ou dynamiquement.

L'évaluation des préconditions d'un refactoring [91, 103] peut être considérée comme une approche statique. Toutefois, la logique du premier ordre impose des approximations conservatives, par rapport à ce qui est possible avec des formalismes de transformation de graphes [78] : préservation d'accès, de mise à jour, ou encore d'appel. La préservation du typage est aussi une approche statique de la préservation du comportement d'un programme. Pour pouvoir vérifier des aspects plus précis de la préservation de comportement, il faut prendre en compte des informations dynamiques [20].

2.3 Techniques de support

Les activités de conception décrites précédemment ne sont réellement efficaces que si leur application est automatisée; elles doivent donc être supportées par outillage adapté. On pourrait exploiter les technologies existantes dédiées à la transformation de structures d'arbres ou de graphes pour manipuler des programmes ou des modèles (sect. 2.3.1). Cependant, des technologies plus spécifiques à la transformation de modèles permettent une approche moins syntaxique du problème (sect. 2.3.2).

2.3.1 Transformations d'arbres et de graphes

Les arbres et les graphes sont des structures déjà largement utilisées pour représenter des programmes en vue de leur manipulation : arbres syntaxiques, graphes de flot de contrôle, etc.

XSLT : Extensible Stylesheet Language Transformations

Au départ, eXtensible Stylesheet Language (XSL) est une famille de recommandations¹⁵ du W3C pour la présentation de documents XML. Les principaux membres de cette famille sont XSL Formatting Objects (XSL-FO), un vocabulaire dédié à la spécification de sémantiques de présentation, et XSL Transformations (XSLT), le langage des feuilles de style exprimant la transformation d'un document XML vers sa présentation en XSL-FO; XSLT utilise un troisième langage, XPath¹⁶, pour accéder ou référer aux parties du document XML transformé. Le W3C travaille¹⁷ également sur XPath 2 et son extension

15. <http://www.w3.org/Style/XSL/>

16. <http://www.w3.org/TR/xpath>

17. Voir <http://www.w3.org/TR/xquery> et <http://www.w3.org/TR/xpath20>, spécifications à l'état de *working draft* au 04/04/2005.

XQuery, pour sélectionner des sous-arbres de documents XML et construire de nouveaux documents à partir de cette sélection.

En pratique on peut utiliser l'outillage XSLT sur tout type de document XML, en entrée comme en sortie ; en particulier, il est possible de transformer des programmes via une représentation XML telle que JavaML [14], un vocabulaire XML pour la syntaxe abstraite de programmes Java. Mais si XSLT a été conçu pour transformer de telles structures arborescentes, il s'avère être peu adapté à la manipulation directe de graphes via une sérialisation en XML, comme par exemple XMI pour les modèles issus de métamodèles MOF. Cette approche pose en effet de nombreux problèmes pratiques [27, 95] :

- la syntaxe XSLT est très verbeuse et peu lisible, au point de compromettre la maintenabilité de transformations pourtant relativement simples ;
- les expressions de navigation dépendent de la représentation XMI du modèle, elle même dérivée du MOF ; il faut donc avoir une connaissance détaillée de ces deux technologies pour écrire des transformations XSLT ;
- un programme XSLT s'applique à la représentation arborescente en XMI d'un modèle qui a en général une structure de graphe ; le code de transformation doit donc effectuer des opérations d'ordre syntaxique en plus des opérations sémantiques qui sont utiles pour la transformation.

On pourrait comparer cela à utiliser des outils traitant des fichiers séquentiels tels que la commande unix `awk`¹⁸ pour modifier un document XML. Il reste toutefois possible d'exploiter XSLT non pas comme langage de spécification des transformations, mais comme environnement d'exécution des transformations ; c'est la solution retenue par Mikael Peltier [95] exposée en section 2.3.2.

Transformations de graphes

PROGRES (PROgramming with Graph REwriting Systems) [109] est un langage de transformations de graphes dirigés, attribués, à nœuds et arcs étiquetés, parfois appelés graphes DIANE, pour *DI*rected, *A*ttributed, *N*ode and *E*dge *l*abeled. Ces graphes se différencient donc d'UML par la présence de nœuds attribués à la place des objets classifiés, et d'arcs orientés étiquetés à la place des associations binaires entre objets.

La syntaxe de PROGRES est mi textuelle, mi graphique : les parties gauches et droites des règles de transformation peuvent être exprimées sous forme graphique au milieu du texte de la transformation. Chaque opération de manipulation d'un graphe, est composée d'étapes basiques de recherche de motifs et de transformation des sous-graphes correspondants. Bien que basée principalement sur des règles, la programmation avec PROGRES peut aussi

18. <http://www.gnu.org/software/gawk/gawk.html>

se faire de manière impérative, à travers des structures de contrôle similaires à celles de Prolog :

- l'opérateur `&` correspond à la séquence d'évaluation de gauche à droite des clauses ;
- la construction `choose...else...end` correspond à la sélection des clauses composant un prédicat ;
- les opérateurs `and` et `or` sont les contreparties non-déterministes de `&` et `choose` ;
- finalement les règles peuvent être appelées récursivement.

Les transformations, ou *transactions* dans la terminologie de PROGRES, sont atomiques, préservent la cohérence du graphe manipulé, et peuvent faire des choix non-déterministes avec la possibilité d'initier un retour arrière plus tard si nécessaire.

Il existe de nombreux systèmes de transformation de graphes, entre autres Fujaba [52] ou AGG [4], qui se basent sur les grammaires de graphes et des règles de réécriture de graphes. VIATRA (VISual Automated model TRAnSformations) [124] combine cette approche de transformations par règles avec l'ordonnancement visuel de ces règles par des diagrammes d'états. Ces outils montrent comment les transformations de graphes sont utilisables dans le domaine de la manipulation de modèles et du MDA.

2.3.2 Transformations de modèles

Pour implémenter des refactorings, on a besoin d'informations précises sur la structure du programme manipulé, typiquement obtenues via des techniques d'analyse statique. Les premières implémentations des refactoring manipulaient des structures de données ad-hoc suivant les possibilités d'introspection et de réflexion offertes par le langage considéré — termes en Scheme, arbre abstrait Smalltalk — ou via des représentations spécifiques telles que Cstructure qui est optimisée pour que les recherches mais aussi les mises à jour des données d'analyse statique de programmes C soient efficaces [86].

Cependant, le besoin premier est dans tous les cas la définition d'une représentation du langage manipulé en fonction des besoins de transformation, pour avoir un accès direct aux informations pertinentes. Ces informations sont essentiellement de nature sémantique, même si en pratique il est aussi utile de manipuler des informations syntaxiques. Les systèmes ou langages de transformation de modèles sont un outillage spécialisé pour la manipulation de structures sémantiquement riches. En cela, ils fournissent la plate-forme permettant d'unifier les approches de manipulation de programmes ou de graphes décrites précédemment sur une base technologique commune.

MétaGen

Le système MétaGen [21] a été développé pour prendre en compte les problèmes de communication au sein d'une équipe de développement, en réaction aux cadres jugés trop rigides que proposent les outils de représentation des connaissances et méthodes à objets disponibles à l'époque. L'approche proposée pour MétaGen est de « rendre flexible les moyens de communication eux-mêmes, en adaptant le langage utilisé [*par l'outillage d'*] une méthode de métamodélisation » [102]. C'est une approche générative : du code exécutable est dérivé à partir d'un modèle de haut niveau, accessible à un utilisateur spécialiste du domaine d'application :

- les règles de transformation incarnent l'expertise d'implémentation du type de modèle considéré ;
- le modèle utilisateur est transformé vers un second modèle qui représente une implémentation possible de l'application à base de frameworks ;
- l'utilisation de frameworks rend réaliste la production d'un code complexe et fonctionnel à partir du modèle d'implémentation.

MTrans

MTrans [95] est un langage dédié de spécification de transformations de modèles. Les langages dédiés sont définis comme suit :

« A Domain-Specific Language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain »

— *Van Deursen et al.* [40]

Le langage MTrans est un langage de règles ; on peut résumer ses principaux objectifs ainsi :

- on écrit les transformations en utilisant directement les concepts définis dans les métamodèles sources et cibles, mais toutefois indépendamment d'un métamodèle particulier ;
- les transformations sont *n*-aires, c'est-à-dire ayant pour sources et destination plusieurs modèles correspondant chacun à leur métamodèle ;
- les transformations peuvent être génériques, c.-à-d. applicable dans plusieurs contextes, par exemple les différentes versions d'UML.
- des aspects plus pratiques tels que l'extensibilité, la simplicité d'abord de la syntaxe, sa concision etc.

QVT : Query Views Transformations

Pour répondre au besoin d'un langage de manipulation de modèles, l'OMG a publié un appel à propositions (RFP, *Request For Proposal*) intitulé MOF 2.0 Query/Views/Transformations (QVT) [100]. Ce RFP doit aboutir à la standardisation des outils sémantiques permettant de mettre en correspondance des modèles exprimés dans des langages issus du MOF :

- l'aspect interrogation est nécessaire pour filtrer ou sélectionner des éléments de modèle, soit de manière *ad-hoc*, soit pour identifier les éléments qui seront à la source d'une transformation ;
- les vues sont des modèles qui révèlent un aspect particulier ou permettent de travailler sur un sous-ensemble d'un autre modèle ;
- les transformations définissent ou établissent une correspondance entre différents modèles, et en particulier des vues.

QVT a donc pour but d'outiller le domaine des modèles, de la même manière que XPath et XSLT sont utilisés pour mettre en correspondance des documents XML issus de différentes DTD ou schémas.

Cependant, RFP QVT [100] n'est pas forcément la solution définitive pour les problèmes de transformation de modèles, car le domaine de l'ingénierie des modèles manque encore de maturité. D'abord, vu les différences considérables entre les réponses soumises au RFP, il sera probablement difficile d'atteindre un consensus pour un langage d'interrogation, de vues et de transformation de modèles MOF à la fois unifié et bien fondé. En fait, il est probablement trop tôt pour standardiser quoi que ce soit car les besoins ne sont pas encore bien définis ou même compris. D'autre part, l'expérience des standards OMG précédents montre qu'il faut un délai considérable pour que les utilisateurs aient à leur disposition des solutions interopérables. Dans le cas de QVT, les disparités de points de vue entre les soumissions et les problèmes de « design by consensus » souvent posés par le mode de fonctionnement de l'OMG¹⁹ augurent d'un standard qui sera long à élaborer, probablement multiforme, et difficile à accepter.

Pourtant, il existe dans l'industrie un besoin réel pour des techniques permettant de gérer l'évolution de systèmes informatiques complexes, besoin auquel le MDA et plus généralement l'ingénierie des modèles semble apporter des solutions. La problématique est donc qu'un standard doit émerger de propositions très différentes, alors qu'on ne peut encore que spéculer sur les utilisations concrètes possibles de ce standard. L'architecture que nous proposons dans le chapitre suivant est donc une approche de conciliation, suffisamment

19. La logique de standardisation semble aller à l'encontre des intérêts des implémenteurs commerciaux qui pourtant forment la majorité des membres de l'OMG.

générique et minimale pour supporter les propositions actuelles sans recouvrement fonctionnel, et suffisamment flexible pour supporter les évolutions futures des besoins.

Besoins pour un langage de transformation

Pour être utile et pratique, une architecture de transformation doit répondre aux besoins suivants :

Réutilisation : même si on peut utiliser une transformation *ad-hoc* pour modifier ponctuellement un modèle particulier, la plupart des transformations sont conçues pour être réutilisées avec plusieurs modèles. En général, une transformation est donc paramétrée par les modèles qu'elle manipule, et le contenu concret de ces modèles n'est connu que lorsque la transformation est appliquée.

Composition : Dans la perspective du MDA, le modèle du produit final est obtenu par raffinement successifs des PIM puis des PSM ; cet enchaînement est une composition de transformations. De même, on veut être en mesure de réutiliser le code d'une transformation existante en l'appelant depuis une transformation englobante.

On doit donc pouvoir, au sein de l'environnement et du langage de transformation, exprimer du code de contrôle, pour articuler différentes transformations entre elles.

Généricité : Certaines transformations, comme par exemple un générateur de documentation, peuvent être appliquées indépendamment du degré de précision du modèle auquel elles sont appliquées, voire indépendamment des différences entre versions mineures d'un même métamodèle (UML 1.1 à 1.5 par exemple). Cependant cela nécessite que l'outil de transformation soit capable de s'accomoder de telles différences tant qu'elles ne gênent pas la transformation.

Configuration : D'un projet à un autre, ou pour adapter un logiciel à différentes variantes d'une même plateforme, on a souvent besoin d'appliquer les mêmes transformations en n'adaptant que quelques parties spécifiques ; de même, dans le contexte des lignes de produits, une transformation générique sera spécialisée pour un domaine d'application restreint.

Maintenance : À partir du moment où on considère que les transformations sont des produits logiciels qui seront réutilisés, il faut pouvoir les maintenir et les adapter en fonction de l'évolution du domaine et des besoins des utilisateurs.

Il existe d'autres besoins tels que la bidirectionnalité ou les mises à jour incrémentales. La bidirectionnalité rend possible l'application d'une transformation dans les deux sens, c.-à-d. on peut retrouver les modèles d'origine à partir du résultat. Les mises à jour incrémentales désignent la possibilité de reporter sur le modèle résultat des changements faits sur le modèle source après l'application de la transformation. Si on ne souhaite pas se limiter à des transformations réversibles, il faut recourir sur des mécanismes de traçabilité des transformations ou calculer et mémoriser les différences entre modèles; cela rejoint des travaux récents [5] mais reste dans les perspectives de ce travail.

D'autre part, Sendall et Kozaczynski ont identifié les caractéristiques désirables d'un langage de transformation de modèles [110]; selon eux, pour supporter le développement dirigé par les modèles, un tel langage devrait :

- être exécutable ;
- avoir une implémentation efficace ;
- être expressif mais pas ambigu aussi bien pour les transformations qui modifient des modèles existants (ajout, modification ou suppression d'éléments) que pour les transformations qui créent des modèles de toutes pièces ;
- donner une description précise, concise et claire des transformations pour favoriser la productivité lors du développement :
 - en différenciant clairement la description des règles de sélection dans le modèle source et les règles de production du modèle cible,
 - en offrant des constructions graphiques pour les cas où on a une telle représentation est plus concise et intuitive qu'une notation textuelle,
 - en rendant implicites les concepts qu'on peut déduire intuitivement du contexte ;
- donner le moyen de combiner les transformations par au moins des opérateurs de séquence, de condition, et de répétition, pour construire des transformations composites ;
- permettre de définir les conditions requises pour qu'une transformation soit exécutée.

Langage spécifique ou généraliste ? Pour répondre à ces besoins il faut donc à priori définir un langage spécifique ; cependant la frontière entre langages spécifiques à un domaine et langages de programmation généralistes est floue : selon Mernik *et al.* [81], il y a toute une échelle de spécialisation entre des extrêmes tels que BNF d'un côté et C++ de l'autre. Les langages PostScript et PDF [2, 3] illustrent ces différents degrés de spécialisation : ils sont tous deux spécialisés pour la description de documents dans le domaine de la mise en page et de l'impression. Cependant PostScript est un langage de

programmation exécutable, alors que PDF est descriptif; on peut donc dire que PostScript est un langage *moins* spécialisé que PDF. Un framework ou une librairie pour un langage de programmation généraliste est aussi une forme — qu'on pourrait qualifier d'embryonnaire — de langage spécialisé.

Dans [115], nous nous sommes concentrés sur la définition d'un certain type de transformations : l'adaptation à UML des refactorings de William Opdyke [91]. Nous avons spécifié chaque transformation par des pré- et post-conditions exprimées en OCL au niveau du métamodèle. OCL tel que défini dans la spécification UML [121] est un langage d'expressions; bien qu'il soit adapté pour exprimer des contraintes entre éléments de modèle, on ne peut pas l'utiliser pour *modifier* un modèle. Pourtant, on peut exprimer avec suffisamment de précision les conditions d'applicabilité d'un refactoring, ainsi que l'état du modèle après transformation. Il serait donc pratique de pouvoir exprimer les étapes concrètes de la transformation sans changer de formalisme, en généralisant OCL par l'ajout de fonctionnalités de modification [97].

Deuxième partie

Contribution

Chapitre 3

Une architecture cohérente pour les transformations de modèles

Pour supporter le MDA, l'OMG a construit une architecture de gestion des métadonnées basée sur un unique méta-méta-modèle, le MOF [84], et sur une librairie de métamodèles M_2 tels que UML [121] ou SPEM [112] avec lesquels les utilisateurs peuvent décrire leurs modèles M_1 . Dans l'optique du MDA, un modèle spécifique capture chaque aspect d'un système aux différents stades de son développement. Pour cela, le MDA s'appuie sur des langages standards comme UML, XMI et MOF ; mais il manque encore l'environnement de gestion des modèles, ainsi qu'une définition précise de ce qu'est un modèle et un langage de manipulation de modèles.

Les travaux de standardisation dans ce domaine sont en cours avec les activités liées à QVT (cf. section 2.3.2). L'idée principale pour QVT est d'uniformiser le plus possible les interrogations, transformations et vues de modèles. Si l'initiative réussit cela pourrait être un avantage par rapport au domaine de XML où plusieurs langages différents (XSLT, XQuery) s'attaquent à l'interrogation et à la transformation de documents.

3.1 L'ingénierie dirigée par les modèles

Étudier la manipulation de modèles dans le contexte du MDA a plusieurs avantages. En se basant sur des standards ouverts, on peut réutiliser des travaux externes ; l'éventail de métamodèles disponibles au niveau M_2 recouvre différentes catégories d'aspects : processus et produits, statiques et dynamiques, fonctionnels et non-fonctionnels, objets et relationnels, PIM et PSM . . . Le problème est ici d'exprimer comment on transforme un modèle M_a en un autre M_b . Notre vision est qu'on peut considérer les modèles comme des entités de première classe, et donc qu'on décrit une transformation de modèles par un

autre modèle M_t (fig. 3.1). Cette approche se justifie dans un cadre où les transformations sont assez complexes pour devenir un investissement technologique, car elle permet d'utiliser des méthodes de développement éprouvées. Ce schéma pourrait par exemple être réalisé en sérialisant M_a et M_b en XMI et en compilant M_t vers XSLT ou XQuery.

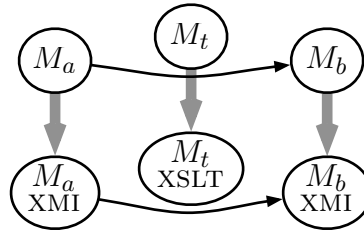


FIG. 3.1 – Une transformation entre deux modèles M_a et M_b peut être définie par un autre modèle M_t .

Il faut donc considérer que, de même que M_a et M_b , M_t est un modèle et est naturellement écrit dans le langage de son métamodèle. Cela a plusieurs conséquences. La première est que lorsqu'on compile M_t vers XSLT on a accès à MM_a et MM_b , métamodèles de M_a et M_b respectivement. La seconde conséquence est que le métamodèle de M_t est l'hypothétique langage avec lequel on peut décrire toutes sortes de transformations de modèles.

Nous défendons ici la thèse que les transformations doivent être des modèles de premier rang dans le monde MDA ; ceci par exemple en adoptant une approche orientée objet et en exploitant l'expressivité d'UML en tant que métamodèle du langage de transformation. UML est à la fois un langage de modélisation et un outil pour la gestion de projet, et il dispose de concepts utiles à l'analyse, la conception et le développement de transformations :

- la macro-organisation en composants des transformations est exprimée par les diagrammes de packages, et de composants ;
- les diagrammes de classes révèlent la structure et les motifs de la conception : une transformation est exprimée par des opérations organisées en classes et packages. La spécialisation et la liaison dynamique peuvent servir à gérer la variabilité, par exemple à l'aide des design patterns [53] ;
- les diagrammes d'activité expriment le processus de transformation en capturant les dépendances entre sous-tâches de transformation, et peuvent servir à combiner plusieurs transformations entre elles ;
- les diagrammes de déploiement spécifient les aspects spécifiques à la plateforme, comme par exemple quel CASE tool sera utilisé pour gérer quels types de modèles.

D'un point de vue « génie logiciel » il y a de nombreux avantages à modéliser des transformations en utilisant UML ; mais tel quel, UML n'est pas utilisable

pour modéliser complètement des transformations de modèles. En effet le problème d'appliquer concrètement ces transformations à des modèles reste non résolu, car UML n'a pas de sémantique d'exécution fixée. Les approches impérative et déclarative sont toutes deux en compétition dans les soumissions pour QVT [100] (cf. section 2.3.2).

Transformation par programmes impératifs Dans le paradigme impératifs, les transformations sont des programmes d'un langage impératif, et modifient le modèle à travers des effets de bord. Il faut dans ce cas donner une sémantique aux corps des méthodes et aux actions dans le modèle de transformation, en joignant un langage d'action à UML.

Il y a plusieurs possibilités pour un tel langage : (1) un langage de programmation tel que Java, Python ou le langage intégré à un CASE tool, (2) Action Semantics [12], ou (3) OCL étendu par des fonctionnalités impératives. Bien que la possibilité (1) soit probablement la plus rapide à mettre en place, en générant du code à partir d'un environnement de développement UML existant, ce n'est certainement pas la meilleure approche puisqu'elle réduit à néant l'intérêt d'une transformation indépendante de la plateforme : toute transformation suffisamment détaillée deviendrait spécifique à une plateforme ou un CASE tool particulier. Action Semantics et OCL 2.0 ont une bonne intégration avec le métamodèle UML. Les transformations de modèles sont très similaires à des méta-programmes¹ ; en fait ils manipulent des modèles, par exemple dans le cas d'un modèle UML, ils modifient des Instances dont les classes sont Classifier, Package, State, etc. Ces manipulations peuvent être exprimées en utilisant Action Semantics, mais OCL a déjà été largement adopté comme le langage privilégié pour traverser des modèles et sélectionner des éléments. Nous croyons donc qu'il s'intégrerait mieux avec une extension de syntaxe pour la manipulation de modèles.

Transformation par règles déclaratives Dans le paradigme déclaratif, les transformations sont définies par composition de règles décrites par leur pré- et post-conditions. Les préconditions définissent des motifs d'intérêt dans les modèles sources ; les postconditions définissent le motif correspondant dans le modèle destination. Cette approche couvre les spécifications OCL et les systèmes à réécriture de graphes. Comme le programmeur d'une transformation peut raffiner les conditions autant que nécessaire avec un langage de

1. Un méta-programme est un programme qui en manipule un autre, par exemple en le générant.

contraintes tel qu'OCL, l'approche déclarative est précise ; par son haut niveau d'abstraction, elle tend aussi à être expressive et indépendante d'une technologie particulière.

Cohabitation des deux approches L'approche déclarative est bien adaptée pour écrire des transformations abstraites ou incomplètes, dans les premières étapes du développement. De telles transformations devront être raffinées jusqu'à ce qu'elles deviennent exécutables. Dans les cas simples, ce raffinement peut être fait de manière efficace par un mécanisme d'inférence tel que Prolog. Par contre, pour des règles complexes il aurait probablement des temps de calcul irréalistes ou des résultats contre-intuitifs², mais il pourrait cependant aider le développeur en restreignant ses choix, dans un mode semi-automatique.

Une méthode plus pragmatique est de spécifier le comportement de manière explicite. Ici, les pré- et post-conditions de la règle de transformation deviennent des contrats pour l'implémentation impérative.

Si on considère que les transformations manipulent des modèles issus du MOF, les transformations primitives — création d'instances et de liens, affectation des valeurs d'attributs, etc — sont réduites à un nombre fini et peuvent être définies par un formalisme déclaratif. Les deux approches peuvent donc coexister au niveau des composants de transformation : une transformation impérative peut utiliser des sous-transformations déclaratives, et dans l'autre sens une transformation impérative peut être encapsulée pour fournir une interface déclarative.

3.2 Impacts du MDE sur le cycle de développement

Il y a une analogie entre d'une part la proportion de code développé manuellement ou par génération de code, et d'autre part des transformations *ad-hoc* ou modélisées précisément. Pour la plupart des proportions entre travail manuel et automatisé, on arrive au bout du compte à des résultats similaires, mais à des coûts différents. Le développement manuel n'est en général effectué qu'une fois, alors que le celui d'un générateur de code est fait dans un but de réutilisation. Ce second cas est plus coûteux initialement, puisqu'il nécessite une bonne compréhension des points communs et des variations

2. Il s'avère aussi qu'en pratique la programmation logique requiert souvent une grande expertise du fonctionnement du moteur d'inférence utilisé pour limiter l'explosion combinatoire et obtenir des programmes efficaces.

dans la ligne de produits générés ; cependant sur le long terme il pourra apporter un meilleur retour sur investissement. Dans le cas des transformations de modèles, le coût initial de la métamodélisation et du développement des règles de transformation est une forme de capitalisation pour la robustesse et la réutilisation.

3.2.1 Cycle de vie d'une transformation

En appliquant le MDA à lui même, on peut parler de transformations indépendantes ou spécifiques vis-à-vis d'une plateforme : respectivement PIT (Transformation Indépendante de la Plateforme) ou PST (Transformation Spécifique à une Plateforme). On ajoute donc deux étapes dans le cycle de développement :

1. exprimer une transformation de modèle indépendamment d'un outil quelconque ;
2. obtenir de cette expression un outil concret qui correspond à une transformation spécifique à un outil ou à une technologie.

Nous avons donc un nouveau cycle de développement des transformations, qui peut être vu comme étant orthogonal au cycle de développement des applications, tel qu'illustré en figure 3.2.

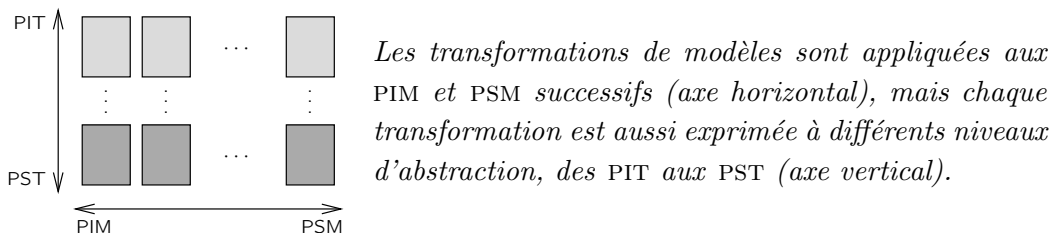


FIG. 3.2 – MDE à deux dimensions.

3.2.2 Processus de transformation des modèles

La figure 3.3 page 43 illustre un cycle de vie typique pour le développement d'une tranformation ; dans la figure 3.2 cela correspond à une chaine de transformations verticale, partant d'une PIT pour obtenir une PST. Chaque colonne du diagramme représente un acteur intervenant dans ce cycle de vie :

Le développeur du framework de transformation crée les abstractions utiles au développeur de transformations pour simplifier l'expression de transformations.

Le développeur de transformations spécifie et développe une transformation entre modèles du domaine, aux niveaux PIT et PST ; ici, le *domaine* comprend tout ce qui est pertinent à un niveau d'abstraction donné au long du cycle de développement de l'application, comme par exemple l'expertise d'implémentation du modèle métier dans la technologie du moment.

L'utilisateur de transformations applique la transformation au cours du cycle de développement de l'application ; la transformation appliquée est une PST parce qu'elle doit être exprimée dans le formalisme de l'outil utilisé.

Selon la figure 3.3, durant chaque activité les acteurs utilisent et produisent des artefacts (*workproducts*) ainsi que des modèles ou métamodèles :

L'environnement de modélisation de PIT est un ensemble d'éléments tels que des profils UML ou des outils, nécessaires pour modéliser des transformations au niveau des PIT.

La transformation PIT vers PST recouvre également un ensemble de profils et d'outils requis pour transformer des PIT en PST.

Le métamodèle du domaine représente la description d'un métamodèle à un niveau d'abstraction donné (PIM ou PSM). Pour exprimer une transformation entre différents domaines, le développeur de transformations utilise les métamodèles de chacun de ces domaines.

Le modèle de transformation PIT est le modèle d'une transformation de modèles, indépendant de tout outil ou technologie. Par exemple ce genre de transformation peut être modélisée par un diagramme d'activité UML, par OCL ou tout autre formalisme standard et largement accepté.

Le modèle de transformation PST est le modèle spécifique à un outil ou une technologie donnée d'une transformation de modèles.

La transformation PST recouvre l'ensemble des éléments dont a besoin l'utilisateur de transformations pour appliquer une transformation à un modèle.

D'une part, le développeur du framework de transformation définit l'environnement de modélisation qui sera utilisé par le développeur de transformations. Ce dernier définit une transformation de modèles au niveau PIT, par exemple pour passer du PIM de conception à un PSM pour les EJB, mais sans considérations de l'outil qui sera finalement utilisé pour effectuer la transformation.

D'autre part, le développeur du framework de transformation fournit les transformations PIT vers PST, que le développeur de transformations peut appliquer pour chaque plateforme voulue. Par exemple, on peut produire à partir d'une PIT son implémentation en XSLT, dans le langage de script d'un outil donné, ou encore produire la documentation de la transformation.

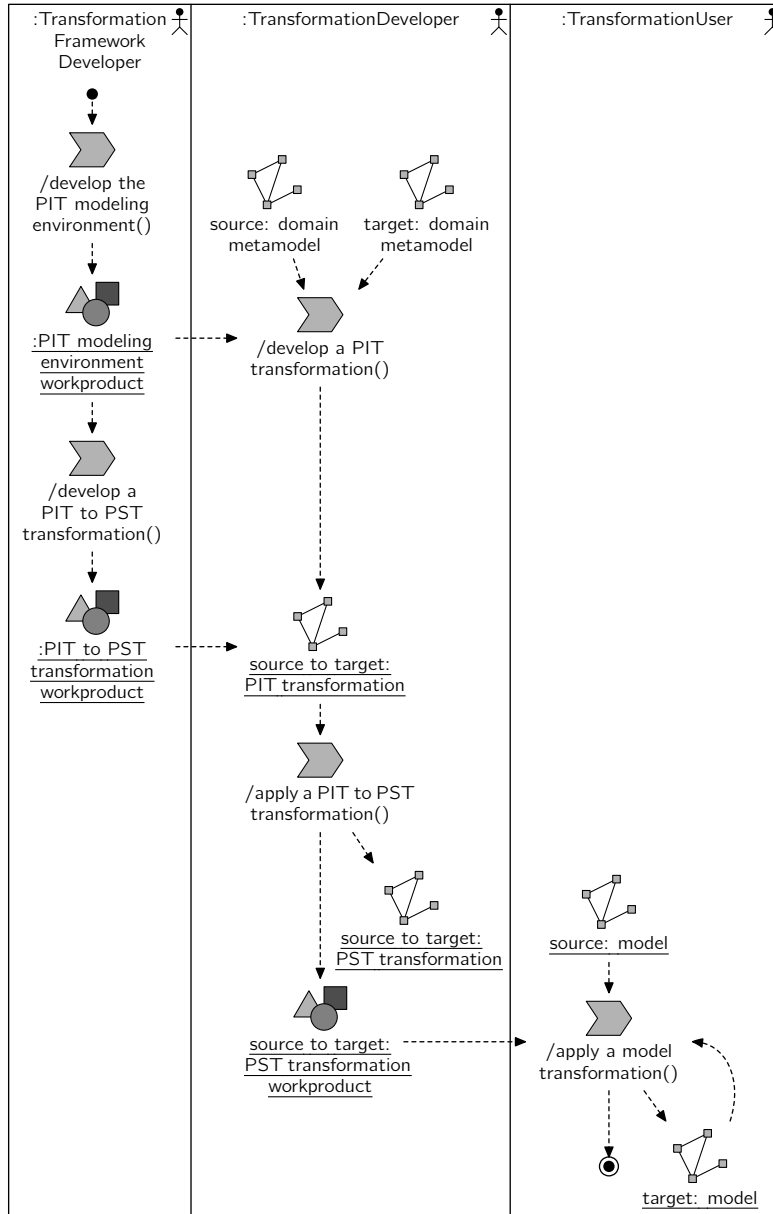


FIG. 3.3 – Diagramme SPEM du cycle de vie d'une transformation de modèles.

Alors que le processus de développement d'un framework de transformation n'a en général lieu qu'une fois, quand l'outillage de développement des applications est configuré et adapté, le processus de développement d'une transformation est mis en place à chaque passage à chaque niveau de modélisation à un autre dans le cycle de développement d'une application. Le processus d'utilisation d'une transformation se déroule encore plus fréquemment et chaque développeur pourra être amené à utiliser une transformation, à n'importe quelle étape du développement. Ces transformations, livrées en tant que *PST transformation workproducts*, peuvent contenir la documentation, les outils d'assistance, ou tout autre artefact facilitant le travail du développeur final ou son intégration dans le processus englobant.

3.2.3 Éléments de retour sur investissement

Les arguments pour choisir l'approche « PIT vers PST » du MDE à deux dimensions sont principalement une extension des arguments en faveur de l'approche « PIM vers PSM » du MDA monodimensionnel :

- L'expression des transformations est durable : l'idée est d'être indépendant de l'évolution technologique (versions des langages, des outils, etc).
- Le niveau d'abstraction auquel on décrit une transformation est élevé : le développeur de transformations peut concentrer ses efforts sur la transformation elle-même et pas sur les techniques, langages ou pratiques d'un outil.

Cependant, cette approche nécessite des investissements, d'une part pour le développement de chaque transformation PIT vers PST, et d'autre part pour l'adaptation de ces transformations à chaque outil avec lequel ces transformations seront exprimées. Ces coûts surviennent à la construction du framework de transformation.

3.3 Architecture de transformation de modèles

Pour répondre au cahier des charges posé à la section 2.3.2, nous avons participé à la définition d'un Langage de Transformation de Modèles (MTL), et plus précisément à l'architecture réflexive qui rend MTL indépendant des métamodèles manipulés.

L'architecture de transformation de modèles de MTL s'articule autour du concept de *dépositaire de modèle*, ou conteneur de modèles ; ce dépositaire donne à la *logique de transformation* l'accès en lecture ou en écriture aux modèles qu'il contient. Un dépositaire est aussi responsable de l'interfaçage

avec l'extérieur du « monde des modèles », car la logique de transformation ne travaille que sur des éléments de modèles issus de l'architecture MOF. Par exemple, la persistance d'un modèle sous forme d'un fichier XMI ou la génération des fichiers de code source d'un projet sont sous la responsabilité d'un dépositaire, car les fichiers et leur contenu textuel ne sont pas des éléments de modèles. Il est donc possible d'accéder à des « espaces technologiques » non-MOF moyennant qu'un dépositaire expose cet espace et ses entités comme un métamodèle et des modèles MOF. Par exemple un dépositaire XML exposerait des documents sous la forme de modèles issus d'un métamodèle DOM³ ou, de manière plus *ad-hoc*, en utilisant le métamodèle représentant une DTD particulière.

Pour résumer, un dépositaire a donc au moins les caractéristiques suivantes :

- stockage des informations d'un modèle ;
- un moyen systématique d'interroger ces informations, c'est-à-dire indépendamment du métamodèle du modèle qu'il contient ;
- prise en charge des modifications locales du modèle, éventuellement avec des restrictions.

Et les caractéristiques suivantes sont également souhaitables même si elles ne sont pas essentielles :

- la persistance du modèle, qui en pratique est déjà gérée par les CASE tools, avec XMI ;
- l'échange de modèles à travers XMI : sauvegarde, chargement...
- la possibilité de gérer plusieurs versions d'un modèle, pour le contrôle de versions et la traçabilité durant l'évolution des modèles ; toutefois cela peut également être abordé au niveau des transformations [5] ;
- la possibilité de configurer le dépositaire pour n'importe quel métamodèle issu du MOF.

Accès aux modèles

On veut pouvoir exprimer des transformations entre plusieurs modèles ; or dans le cas général chacun de ces modèles peut être issu d'un métamodèle différent. Par contre le MOF est en principe unique et stable⁴ ; on sait donc à priori que tout métamodèle issu du MOF sera composé de méta-classes, structuré par des méta-associations etc. L'approche retenue utilise donc une

3. Document Object Model, voir <http://www.w3.org/DOM/>

4. En pratique il évolue, mais il tend à se stabiliser ; quoi qu'il en soit, le niveau d'abstraction est tel que la version précise du MOF importe peu pour cette utilisation ; les problèmes qui se posent sont plutôt d'ordre pratique, par exemple obtenir la description de tel ou tel métamodèle dans la version du MOF souhaitée.

interface que les dépositaires doivent implémenter, et qui se situe au niveau MOF.

À travers cette interface commune aux dépositaires, le moteur de transformation peut donc d'une part découvrir quelles sont les méta-classes qui structurent le métamodèle que le dépositaire connaît, et d'autre part demander la création ou la modification d'instances de ces méta-classes (fig. 3.4). Le langage de transformation peut ainsi rester indépendant de tout métamodèle, sans pour autant forcer l'utilisateur à utiliser des concepts de niveau MOF dans le code de ses transformations. Dans le cas de MTL, tout se passe comme si les classes des métamodèles étaient disponibles depuis le programme de transformation, et les éléments de modèle apparaissent comme des instances que la transformation crée et assemble.

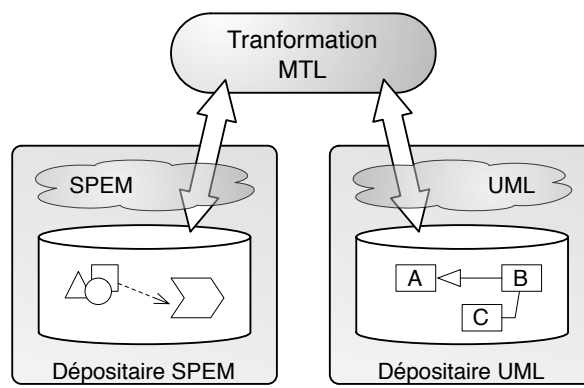


FIG. 3.4 – Architecture de MTL. À travers une interface spécifiée à partir du MOF, la logique de transformation peut découvrir les métamodèles reconnus par chaque dépositaire et piloter la modification des modèles.

Au delà du besoin inhérent à l'architecture MDA de transformer des modèles issus de métamodèles différents, cette approche par *pilotage* de dépositaires se justifie par le gain d'interopérabilité entre les outils de modélisation disponibles. En utilisant un CASE tool comme dépositaire on peut tirer parti de ses fonctionnalités spécifiques et le connecter à d'autres outils ; cela correspond bien à la vision MDA : utiliser les formalismes et outils adaptés pour chaque au niveau d'abstraction.

L'extrait 3.1 montre les premières lignes d'une transformation simpliste. La librairie est l'unité de déploiement d'une transformation ; on utilise ici MDR (Metadata Repository, voir <http://mdr.netbeans.org/>) comme dépositaire. MDR gère n'importe quel métamodèle MOF, aussi on le configure en lui indiquant le fichier XMI contenant la version allégée d'UML utilisée dans l'exemple en plus de celui contenant le modèle à transformer.

EXT. 3.1 – *Déclaration d'une librairie et initialisation d'un dépôt de modèles (HelloWorld.mtl).*

```

library HelloWorld

mdrdriver := new MDRDriver::MDRModelManager();
mdrdriver.init();           // create and initialize a repository driver
5
model theModel : RepositoryModel;
theModel := mdrdriver.getModelFromXMI (
  'SimpleUmlMM_MOF.xml',    // metamodel as XMI
  'SimpleUmlMM',           // root package in metamodel
10 'SimpleUML_source_model',
  'hello_world.xmi',       // XMI to load as model
  null                      // no output file
);

```

Structuration objet

Nous avons choisi de faire de MTL un langage de programmation généraliste orienté objet, disposant de constructions spécifiques à la manipulation de modèles. Un langage très spécifique aurait certes l'avantage de pouvoir fournir des notations et abstractions plus adaptées à son domaine qu'un langage généraliste [81]. Par contre, un langage généraliste sera plus flexible, et pourra être étendu plus facilement ; or c'est ce point qui nous intéresse plus particulièrement. En effet le domaine des méthodologies et outils de modélisation évolue vers plus de complexité et de diversité : il existe déjà une multitude de technologies et de plateformes de transformation, et on a donc besoin de transformations flexibles qui puissent s'interfacer facilement avec d'autres outils, proposer des interfaces de paramétrage à leurs utilisateurs, etc.

UML dispose déjà d'un langage spécialisé : OCL. Ce langage a été conçu à l'origine pour exprimer des contraintes métier, donc dans des modèles de niveau M_1 . Cependant, du fait de l'architecture réflexive d'UML, OCL peut tout-à-fait être utilisé aux niveaux d'abstractions plus élevés, et l'est d'ailleurs en particulier pour exprimer les règles de bonne formation du métamodèle UML lui-même [121]. Cette utilisation d'OCL relève de la métaprogrammation [114, 115] ; par exemple les contraintes spécifiées dans [115] sont semblables à des contrats de programmes de transformation de modèles. Pour garder une certaine uniformité, MTL reprend donc des constructions d'OCL et des concepts proches de ceux déjà présents en UML, comme par exemple les types de base, les collections et leurs opérations, etc. Le but est aussi, à terme, que le développement de transformations s'intègre à l'outillage MDA et UML et que les transformations puissent être développées directement à partir d'un modèle UML.

EXT. 3.2 – Déclaration d'une classe de transformation (*HelloWorld.mtl*).

```

class HelloWorld {
15  run() : theModel::Class {
      c : theModel::Class;
      c := new theModel::Class();
      c.name := 'HelloWorld';
      return c;
20  }
}

```

L'extrait 3.2 montre la déclaration d'une classe simple. La notation des noms complets de types est reprise d'OCL, avec la différence que la première composante identifie le dépositaire dans lequel se situe le type en question. Les dépositaires jouant ce rôle d'espace de nommage, il est possible de manipuler plusieurs modèles issus d'un même métamodèle, sans devoir casser le cloisonnement entre différents modèles.

Langage d'actions

Vus de manière simplifiée, les modèles sont des graphes typés et attribués ; une transformation peut donc modifier :

- l'existence des nœuds, en les créant ou les supprimant avec des types corrects vis-à-vis du métamodèle ;
- la valeur des attributs des nœuds ;
- l'existence d'arêtes, qui correspondent aux liens des associations du métamodèle.

La difficulté en ce qui concerne un langage de transformation de modèles est d'offrir au développeur de transformations le moyen d'enchaîner ces modifications simples quel que soit le métamodèle manipulé, sans toutefois lui imposer de connaître la représentation au niveau du MOF de ce métamodèle.

Du point de vue du développeur de transformations, on manipule en MTL les éléments de modèle stockés dans un dépositaire de la même manière qu'on manipule les objets du programme de transformation lui même : les classes du métamodèle géré par le dépositaire apparaissent dans la transformation comme des classes MTL, et les éléments de modèle comme des objets MTL. Ainsi, instancier une classe provenant d'un dépositaire crée l'élément de modèle correspondant, qu'on peut ensuite modifier en affectant ses attributs. Les associations sont gérées par le langage, et pour créer ou supprimer des liens entre éléments de modèle, le développeur dispose des primitives `associate` et `dissociate`.

La différence majeure avec les objets d'un programme Java par exemple, est que les objets gérés par un dépositaire sont persistants : dans la plupart des cas on souhaite obtenir le résultat de la transformation sous une forme exploitable avec d'autres outils : par exemple un fichier XMI qui sera chargé dans un CASE tool. Cependant un dépositaire peut aussi publier sous l'apparence d'un modèle des données d'une autre forme : par exemple, un fichier de code source pourrait être publié sous l'apparence d'un modèle d'arbre syntaxique abstrait.

3.4 Interface d'accès aux dépositaires

3.4.1 Niveau « méta »

L'interface d'accès est un canal de communication à double sens entre le moteur de transformation et chacun des dépositaires dont il a besoin. D'une part le moteur de transformation interroge le dépositaire en se référant à des données connues à priori ou obtenues via des requêtes antérieures ; d'autre part le dépositaire publie son contenu en réponse. Cette section détaille les interfaces Java utilisées à l'initiative du moteur de transformation pour interroger le dépositaire ; la figure 3.5 en illustre la structure. On peut remarquer que les opérations disponibles n'ont pas une sémantique « exploratoire » ; en effet, les transformations sont écrites pour un métamodèle donné, et certaines informations comme par exemple les noms des classes du métamodèle — métaclasse ici — sont connus à priori par le développeur, et inscrits dans le code de la transformation.

API Il s'agit du point d'entrée de l'interface, que le moteur de transformation interroge pour découvrir la structure du métamodèle, et pour manipuler le modèle sous-jacent. Même si un dépositaire peut gérer plusieurs modèles simultanément, le moteur de transformation obtient une instance implémentant cette interface par modèle. Attention, même si les méthodes décrites ci-dessous renvoient un objet, cela ne signifie pas pour autant que cet objet existe ou a été créé dans le modèle : les objets renvoyés sont destinés à désigner les concepts du modèle, et pas à en créer de nouveaux. L'accès au contenu concret d'un modèle est retardé le plus possible pour n'effectuer que les traitements nécessaires.

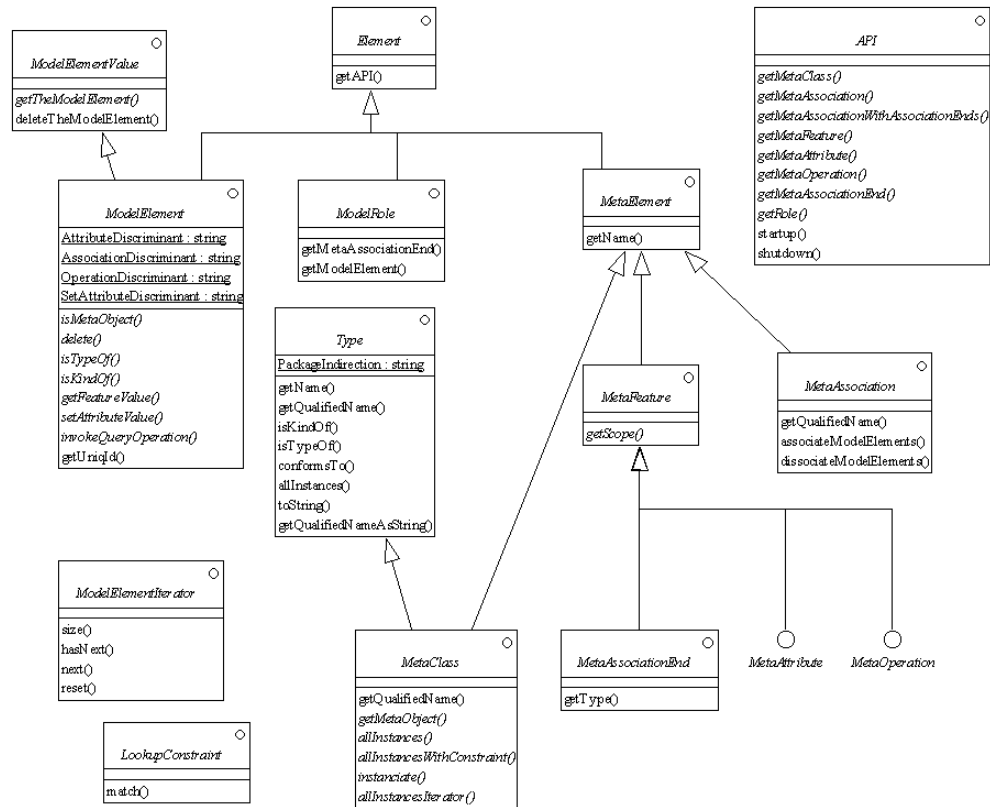


FIG. 3.5 – Structure de l'interface d'accès aux dépôts.

`getMetaClass(name:Sequence{String}):MetaClass`

Retourne une classe du métamodèle ayant le nom qualifié passé en argument. Le nom qualifié explicite l'ensemble des parents de la classe dans la hiérarchie des espaces de noms du métamodèle.

`getMetaFeature(name, scope):MetaFeature`

`getMetaAttribute(name, scope):MetaAttribute`

`getMetaOperation(name, scope):MetaOperation`

Renvoie un membre accessible depuis la métaclasse `scope`, et identifié par son nom ; `name` et `scope` sont respectivement de type `String` et `MetaClass`. La méthode `getMetaFeature()` permet de demander un membre de type quelconque : extrémité d'association, attribut ou méthode.

`getRole(element:ModelElement, end:MetaAssociationEnd):ModelRole`

Renvoie une paire associant un élément de modèle et l'extrémité d'une méta-association.

`getMetaAssociationEnd(role, type, scope):MetaAssociationEnd`

`getMetaAssociation(name:Sequence{String}):MetaAssociation`

`getMetaAssociationWithAssociationEnds(ends):MetaAssociationEnd`

Recherche d'une association du métamodèle, par son nom qualifié, ou connaissant ses extrémités; l'argument `ends` est un ensemble d'extrémités d'association, c.-à-d. de type `Set{MetaAssociationEnd}`.

`startup(arguments : Sequence{Value})`

`shutdown(arguments : Sequence{Value})`

Points de contrôle pour l'initialisation et à la finalisation d'un dépositaire.

Element L'interface commune aux éléments contenus dans un dépositaire, c'est-à-dire à tous les éléments de modèle ou de métamodèle.

`getAPI()`

Tout élément fait partie d'un modèle particulier, et appartient donc au dépositaire correspondant. On peut connaître ce dépositaire en appelant `getAPI()`.

MetaElement

`getName()`

Tous les éléments d'un métamodèle peuvent être nommés. Comme une transformation est conçue pour un métamodèle ou une famille de métamodèles spécifique, les noms des méta-éléments sont connus *à priori* par le programmeur. Ils apparaissent dans la transformation sous forme d'identifiants de types, d'associations et d'attributs.

MetaFeature et ses descendantes

`getScope()`

Retourne la métaclasse dans laquelle est défini cette méta-feature.

`getType()`

Retourne le type du méta-élément auquel on accède en naviguant cette méta-feature.

3.4.2 Accès au modèle

Cette section détaille les classes des objets renvoyés par le dépositaire en réponse au moteur de transformation.

MetaClass Les méta-classes sont les classes constituant le métamodèle ; les éléments du modèle sont représentés par des instances de ces méta-classes.

`getQualifiedName()`

Retourne le nom qualifié complet de la méta-classe destinataire de ce message.

`getMetaObject()`

Certains éléments de modèle peuvent avoir des données statiques — au sens Java, c'est-à-dire partagées entre toutes les instances. Du point de vue de l'interface d'accès aux dépositaire, ces données sont contenues dans un élément du modèle représentant la métaclasse, et accessible par `getMetaObject()`.

`allInstances()`

`allInstancesWithConstraint(constraint)`

Renvoie la collection des instances de cette méta-classe ou de ses descendantes. On peut restreindre cette collection en utilisant la version avec contrainte de la méthode.

`instanciate(contextualElement, arguments)`

Crée puis ajoute un élément de modèle dans le dépositaire.

MetaAssociation

`getQualifiedName()`

Retourne le nom qualifié complet de la méta-association destinataire de ce message.

`associateModelElements(contextualElement, roles)`

Crée un lien entre les éléments de modèle désignés par les rôles passés en paramètres, et pour la méta-association concernée.

`dissociateModelElements(contextualElement, roles)`

Détruit le lien entre les éléments de modèle désignés par les rôles passés en paramètres.

ModelElement

`isMetaObject()`

Indique si cet élément de modèle est le représentant d'une méta-classe.

`isTypeOf(classifier)`

Indique si l'élément de modèle est une instance directe de la classe passée en paramètre. Les *méta-objets* ne sont pas instances de la méta-classe qu'ils représentent.

`isKindOf(classifier)`

Indique si l'élément de modèle est une instance de la classe passée en paramètre, ou d'une de ses super-classes. Renvoie faux pour les *méta-objets*, de la même manière que `isTypeOf()`.

`getFeatureValue()`

Évalue le membre passé en paramètre, c'est-à-dire renvoie la valeur d'un attribut, parcourt une association, ou évalue une opération.

`setAttributeValue(contextualElement, attribute, value)`

Change la valeur de l'attribut passé en paramètre. Si l'objet récepteur du message est un méta-objet, cela revient à changer la valeur d'un attribut « statique » pour toutes les instances de la méta-classe représentée.

`invokeQueryOperation(contextualElement, feature, argument)`

Un dépositaire peut implémenter certaines opérations du métamodèle qu'il publie, et donner accès à ces opérations au moteur de transformation via `invokeQueryOperation()`. Il appartient au dépositaire d'implémenter ces opérations sans effet de bord.

`delete()`

Supprime l'élément du modèle.

ModelRole

`getMetaAssociationEnd()`

`getModelElement()`

Les instances de `ModelRole` sont de simples paires associant un élément de modèle à l'extrémité d'une association. Elles servent à désigner sans ambiguïté comment ajouter ou supprimer un lien quand on appelle `AssociateModelElements()` ou `DissociateModelElements()`. Ces deux méthodes ne sont donc que de simples accesseurs.

Chapitre 4

Noyau sémantique : une implémentation de référence en O’Caml

La description des interfaces composant l’API présentée dans le chapitre précédent est nécessaire pour standardiser les points d’accès à un dépositaire de modèles. Cependant il est souvent difficile d’appréhender le fonctionnement d’un système en se basant seulement sur la documentation technique, et d’autant plus lorsque comme ici le domaine recouvre plusieurs niveaux *méta*. De plus, l’API que nous proposons laisse une part de travail importante à la charge du dépositaire¹ : il faut en effet donner accès au métamodèle des données que le dépositaire contient, et représenter ces données en termes d’éléments de modèle correspondant à ce métamodèle.

Nous avons donc choisi d’implémenter de manière minimaliste un dépositaire et des transformations de modèles. Cette implémentation est réalisée en langage fonctionnel, de façon à rester proche de l’expression mathématique du problème. Par rapport à la définition d’une sémantique formelle, cette approche a l’avantage de fournir un démonstrateur simplement exécutable.

4.1 Un MOF minimal et son API

Le module `Mof` définit la représentation en O’Caml d’un méta-métamodèle simplifié (fig. 4.1) qu’on désignera par μMOF ici. Ce méta-métamodèle est inspiré du MOF, mais n’en conserve que les éléments principaux : classifieurs, attributs, associations, généralisation ; les associations et liens sont binaires. L’objectif n’est pas ici d’implanter de façon fidèle le MOF de la norme, mais

1. Plus précisément à la charge du pilote qui adapte le dépositaire à l’API.

d’obtenir un méta-métamodèle assez complet pour représenter les métamodèles décrits plus loin, ainsi que des modèles issus de ces métamodèles.

EXT. 4.1 – (*mof.ml*) Définition de μ MOF, un MOF simplifié en O’Caml. Du MOF officiel, μ MOF ne conserve que les classes, attributs et associations, la relation de généralisation, les instances et liens.

```

10 type className = string
   type roleName = string
   type datatype = Integer | Boolean | String
   type mult = Range of int*int | One | Maybe | Any | Some

15 type attribute = Attr of string * datatype
   type assoend = AEnd of className * mult * roleName
   type association = {
     asso: string; (* name *)
     src: assoend; dest: assoend;
20   ordered: bool; (* defined by order in link list *)
     aggreg: bool (* src is the composite *)
   }

   type classif = { name: className; attrs: attribute list }
25
   (* associative list, subclass → super *)
   type generalization = (className*className) list
   type metamodel = {
     mm: string; (* name *)
30   classes: classif list;
     gen: generalization;
     assocs: association list
   }

35 type value = Int of int | Bool of bool | Str of string
   type instance = Inst of classif * (attribute * value) list
   (* Link( source, destination, association ) *)
   type link = Link of instance * instance * association

40 (* a model, represented using MOF instances *)
   type mminstance = Model of metamodel * instance list * link list

```

Le module `Api` est une implémentation de l’API d’accès aux dépositaires décrite dans le chapitre précédent, pour des modèles représentés avec μ MOF. Afin de rendre cette implémentation purement fonctionnelle, la hiérarchie de classes de la version Java est mise à plat, et les méthodes sont remplacées par des fonctions ayant deux arguments supplémentaires explicitant le contexte d’appel : le dépositaire concerné et l’objet destinataire du message.

La première partie de l’API (fig. 4.2) sert à obtenir les points d’entrée à partir desquels on pourra naviguer le « contenu » du dépositaire, c’est-à-dire le modèle et son métamodèle. Les informations nécessaires pour obtenir ces points d’entrée sont connues *a priori* par le moteur de transformation : elles sont disponibles statiquement dans le code source de chaque transformation.

Par exemple le nom d'une métaclasse provient d'une déclaration de type, ou le nom d'un rôle d'une opération de navigation.

EXT. 4.2 – (*api.ml*) *Fonctions représentant les méthodes de la facade du dépositaire. Ces méthodes donnent des points d'entrée à partir desquels on peut ensuite parcourir le modèle.*

```
(* Mof.mminstance → Mof.classifName → Mof.classif *)
let getMetaClass (Model(mm,_,_)) name = classForName mm name

let getMetaAttribute rep name scope =
25 List.find (fun (Attr(n,dt)) → n = name) scope.attrs
let getRole rep (Inst(c,_)) aend = aend_r aend
(* getMetaAssociationEnd with role for class typ, viewed from class scope *)
let getMetaAssociationEnd (Model (mm,_,_)) role typ scope =
  let ends_match (AEnd(srccn,_,r)) (AEnd(destcn,_,_)) =
30 (r=role)&&(srccn=typ.name)&&(destcn=scope.name) in
  let rec find_assoend = function
    | a::l → if (ends_match a.src a.dest) then a.src
              else if (ends_match a.dest a.src) then a.dest
              else find_assoend l
35 | [] → raise Not_found
  in find_assoend mm.assocs
let getMetaAssociation (Model (mm,_,_)) name =
  List.find (fun a → a.asso = name) mm.assocs
let getMetaAssociationWithEnds (Model (mm,_,_)) aends =
40 List.find (fun a → (a.src,a.dest) = aends) mm.assocs
```

Connaissant une métaclasse, le programme de transformation peut ensuite interroger le dépositaire pour créer de nouvelles instances de cette métaclasse ou obtenir l'ensemble de celles qui existent déjà (fig. 4.3). Comme ces instances de métaclasses représentent des éléments de modèle, cela revient à naviguer ou modifier effectivement le modèle contenu dans le dépositaire.

EXT. 4.3 – (*api.ml*) *Méthodes disponibles pour les métaclasses.*

```
(* getQualifiedName *)
let getClassname rep mc = mc.name
let allInstances (Model(mm,il,_) ) mc = List.filter (fun (Inst(c,_)) → c = mc) il
55 let allInstancesWithConstraint rep mc cn =
  List.filter cn (allInstances rep mc)
let instanciate (Model(mm,il,ll)) mc ctx args =
  let i = Inst(mc,[]) in (Model(mm,i::il,ll)),i
```

EXT. 4.4 – (*api.ml*) *Méthodes disponibles pour les méta-associations.*

```
(* getQualifiedName *)
let getAssociationName rep ma = ma.asso
70 let associateModelElements (Model(mm,il,ll)) ma ctx mroles =
  let new_links = match mroles with
    | [r1;r2] → [ Link(
      (List.assoc ma.src mroles),(List.assoc ma.dest mroles),ma) ]
    | _ → failwith "only binary associations are supported"
75 in (Model(mm,il,ll@new_links))
```

```

let dissociateModelElements (Model(mm,il,ll)) ma ctx mroles =
  let esrc = (List.assoc ma.src mroles)
  and edest = (List.assoc ma.dest mroles) in
  let new_ll =
80   List.filter (fun (Link(i1,i2,a))
    → not (i1=esrc)&&(i2=edest)&&(a=ma)) ll in
  (Model(mm,il,new_ll))

```

EXT. 4.5 – (*api.ml*) Méthodes disponibles pour les méta-éléments.

```

let getFeatureValue rep (Inst(c,values)) ctx mf =
95  snd (List.find (fun (a,v) → a = mf) values)
let delete (Model(mm,il,ll)) me =
  let new_il = List.filter (fun i → i<>me) il
  and new_ll = List.filter (fun (Link(s,d,_)) → (s<>me) && (d<>me)) ll in
  (Model(mm,new_il,new_ll))
100 let isTypeOf rep (Inst(c,_)) mc = (c = mc)
  let rec isKindOf (Model(mm,il,ll) as rep) (Inst(c,_ as me) mc =
    (isTypeOf rep me mc) or
    (isKindOf rep me (classForName mm (List.assoc mc.name mm.gen)))
  let setAttributeValue (Model(mm,il,ll)) (Inst(c,av) as me) ctx ma v =
105  let new_me = Inst(c, (ma,v)::(List.remove_assoc ma av)) in
  let new_il = List.map (fun i → if i = me then new_me else i) il in
  let subst_me x = if x = me then new_me else x in
  let new_ll = List.map (fun (Link(s,d,a)) → Link((subst_me s),(subst_me d),a)) ll in
  Model(mm,new_il,new_ll), new_me
110 let invokeQueryOperation (Model(mm,il,ll)) (Inst(c,av) as me) ctx mf v =
  let readAttributeValue a = List.assoc a av in
  let collectAssoEnd ae =
    List.map link_d
      (List.filter (fun (Link(s,d,a)) → s = me && a.dest = ae) ll)
115  in
  match mf with
    AttrF a → Scal (readAttributeValue a)
  | EndF ae → Coll (collectAssoEnd ae)
  | _ → failwith "metamodel-specific_code"
120
let getUniqId rep me = me

```

EXT. 4.6 – (*api.ml*) Accesseurs des rôles.

```

130 let getRoleMetaAEnd rep (ae,_) = ae
  let getRoleModelElt rep (_,me) = me

```

4.2 Métamodèles d’expérimentation

Nous avons ensuite défini des métamodèles simples pour des domaines bien connus, afin de reprendre un problème classique d’informatique et valider le fait qu’on puisse l’exprimer facilement en termes de transformations de modèles, à travers l’API définie au chapitre 3. D’autre part, pour vérifier que cette transformation de modèle correspond bien au problème posé, nous l’avons implémentée parallèlement en O’Caml, de manière *ad-hoc*.

Chaque métamodèle a donc deux représentations : la première sous la forme d'un modèle μMOF , c'est-à-dire une valeur O'Caml de type `Mof.mminstance` et la seconde sous la forme de simples types O'Caml. Ces deux représentations sont mises en correspondance par deux fonctions `to_mof` et `from_mof` effectuant le changement de représentation dans les deux sens. Ces deux fonctions représentent par ailleurs des exemples de mappings qu'un dépositaire publiant un contenu « non MOF » devrait implémenter (*cf.* section 3.3, page 49).

Le problème choisi ici est l'extraction de traces d'exécution d'une machine à états finis. Les métamodèles impliqués ont en effet une implémentation O'Caml triviale et une représentation μMOF raisonnablement concise, sans pour autant limiter l'intérêt des transformations du point de vue de l'API. Nous décrivons d'abord le métamodèle des machines à états en section 4.2.1, puis celui des traces en section 4.2.2.

4.2.1 Les machines à états

Le premier de ces métamodèles définit la structure de machines à états finies, dont les états et les transitions sont étiquetés respectivement par des entiers et des chaînes. Ces machines à états sont d'abord représentées par les types O'Caml définis dans l'extrait 4.7.

EXT. 4.7 – (*microStateMachine.ml*) *Représentation O'Caml des machines à états. Le type `MicroStateMachine.t` est un triplet regroupant l'état initial, puis la liste des états d'acceptation, et finalement les transitions et leurs destinations pour chaque état de la machine.*

```
(* OCaml representation of state machines *)
10 type state = State of int
    type transition = Transition of string

    (* map types for a few useful domains *)
    module SMap = MyMap.Make(struct type t = state let compare = compare end)
15 module TMap = MyMap.Make(struct type t = transition let compare = compare end)
    module InstMap = MyMap.Make(struct type t = Mof.instance let compare = compare end)
    module LinkSet = Set.Make(struct type t = Mof.link let compare = compare end)

    (* initial state * list of final states *
20 {each state => {transition=>destination}}
    (empty map for states with no outgoing transition) *)
    type t = StateMachine of state * state list * (state TMap.t) SMap.t

    (* accessor functions *)
25 let init_state (StateMachine(i,_,_)) = i
    let accept_state (StateMachine(_,a,_)) = a
    let is_accept statemachine state =
        List.exists (fun s -> s = state) (accept_state statemachine)
```

```

30 let equals (StateMachine(i1,a1,s1)) (StateMachine(i2,a2,s2)) =
    (i1 = i2) && (listeq a1 a2) && (SMap.equals (TMap.equals (=)) s1 s2)

```

Le métamodèle des machines à états est ensuite défini dans les termes de μ MOF (extrait 4.8); la figure 4.1 donne la structure de ce métamodèle sous forme d’un diagramme de classes UML.

EXT. 4.8 – (*microStateMachine.ml*) Représentation O’Caml du métamodèle μ MOF des machines à états. On définit d’abord les métaclasse et leurs attributs par des valeurs des types *Mof.classif* et *Mof.attribute*, respectivement; on peut ensuite déclarer la représentation de chacune des associations puis finalement assembler le tout en une valeur du type *Mof.metamodel*.

```

(* MOF representation of state machines *)
40 let machine_cl = { Mof.name="StateMachine"; Mof.attrs=[] }
    let id_at = Mof.Attr("id",Mof.Integer)
    let state_cl = { Mof.name="State"; Mof.attrs=[ id_at ] }
    let label_at = Mof.Attr("label",Mof.String)
    let trans_cl = { Mof.name="Transition"; Mof.attrs=[ label_at ] }
45 let state_as = { Mof.asso="states";
    Mof.src=Mof.AEnd("StateMachine",Mof.One,"containermachine");
    Mof.dest=Mof.AEnd("State",Mof.Some,"state");
    Mof.ordered=false; Mof.aggreg=true
    }
50 let init_as = { Mof.asso="init";
    Mof.src=Mof.AEnd("StateMachine",Mof.Maybe,"");
    Mof.dest=Mof.AEnd("State",Mof.One,"initstate");
    Mof.ordered=false; Mof.aggreg=false
    }
55 let accept_as = { Mof.asso="accept";
    Mof.src=Mof.AEnd("StateMachine",Mof.Maybe,"");
    Mof.dest=Mof.AEnd("State",Mof.Some,"acceptstate");
    Mof.ordered=false; Mof.aggreg=false
    }
60 let trans_as = { Mof.asso="transitions";
    Mof.src=Mof.AEnd("StateMachine",Mof.One,"ownermachine");
    Mof.dest=Mof.AEnd("Transition",Mof.Some,"transition");
    Mof.ordered=false; Mof.aggreg=true
    }
65 let source_as = { Mof.asso="source";
    Mof.src=Mof.AEnd("Transition",Mof.Any,"strans");
    Mof.dest=Mof.AEnd("State",Mof.One,"srcstate");
    Mof.ordered=false; Mof.aggreg=false
    }
70 let dest_as = { Mof.asso="destination";
    Mof.src=Mof.AEnd("Transition",Mof.Any,"dtrans");
    Mof.dest=Mof.AEnd("State",Mof.One,"deststate");
    Mof.ordered=false; Mof.aggreg=false
    }
75 let statemachines_mm = { Mof.mm="StateMachines";
    Mof.classes=[ machine_cl; state_cl; trans_cl ];
    Mof.gen=[ (* no generalization *) ];
    Mof.assocs=[ state_as; init_as; accept_as; trans_as; source_as; dest_as ]
    }

```

On peut ensuite exprimer la correspondance entre les deux représentations des machines à états; pour cela on définit deux fonctions inverses l'une de l'autre. La première de ces fonctions, appelée `to_mof`, construit la représentation μMOF à partir de la représentation O'Caml d'une machine à états (extrait 4.9).

EXT. 4.9 – (*microStateMachine.ml*) Traduction des machines à états, de la représentation O'Caml vers la représentation μMOF . Dans un premier temps on construit la représentation μMOF des états et transitions en mémorisant de quelle valeur O'Caml chaque instance μMOF est issue. Cette mise en correspondance permet ensuite d'établir la topologie du modèle en créant les liens entre instances, pour chaque méta-association présente dans le métamodèle des machines à états.

```

85 (* Conversion to MOF representation *)
let state_to_mof (State s) = Mof.Inst(state_cl,[ id_at,Mof.Int s ])
let transition_to_mof (Transition t) = Mof.Inst(trans_cl,[ label_at,Mof.Str t ])

(* MicroStateMachine.t → Mof.mminstance *)
90 let to_mof (StateMachine(init,accept,sts_map)) =
  let machine_mof = Mof.Inst(machine_cl,[ ]) in
  let state_mappings = (* translate all states first *)
    SMap.fold
      (fun s tm states_acc → SMap.add s (state_to_mof s) states_acc)
95     sts_map
    SMap.empty in
  let map_a_trans src_state t dest_state (trans_acc, links_acc) =
    let wt = (transition_to_mof t) in
    let updated_trans_acc = TMap.add t wt trans_acc in
100    let updated_links_acc =
      LinkSet.add
        (Mof.Link(wt, (SMap.find src_state state_mappings), source_as))
        (LinkSet.add
          (Mof.Link(wt, (SMap.find dest_state state_mappings), dest_as))
105          links_acc) in
      (updated_trans_acc, updated_links_acc) in
  let trans_mappings, end_links =
    SMap.fold
      (fun src_state ts_map (trans_acc, links_acc) →
110        (TMap.fold (map_a_trans src_state) ts_map (trans_acc, links_acc)))
      sts_map
      (TMap.empty, LinkSet.empty) in
  let init_link = Mof.Link(machine_mof, (SMap.find init state_mappings), init_as)
  and accept_links =
115    List.map
      (fun s → Mof.Link(machine_mof, (SMap.find s state_mappings), accept_as))
      accept
  and state_links = SMap.elements
      (SMap.map
        (fun mofstate → Mof.Link(machine_mof, mofstate, state_as))
        state_mappings)
  and trans_links = TMap.elements
      (TMap.map
        (fun moftrans → Mof.Link(machine_mof, moftrans, trans_as))
125        trans_mappings)
  @ (LinkSet.elements end_links)

```

```

130   in Mof.Model(
      statemachines_mm,
      machine_mof :: (SMap.elements state_mappings) @ (TMap.elements trans_mappings),
      init_link :: accept_links @ state_links @ trans_links )

```

L’autre fonction, `from_mof`, effectue le changement de représentation inverse : de μ MOF vers O’Caml (extrait 4.10).

EXT. 4.10 – (*microStateMachine.ml*) Traduction des machines à états, de la représentation μ MOF vers la représentation O’Caml. Suivant un principe similaire à `to_mof`, la transformation commence par construire les valeurs composant la machine à états avant de les regrouper dans les structures représentant la machine complète.

```

(* Conversion to OCaml representation *)
let state_from_mof = function
  | Mof.Inst(cl,[at,Mof.Int s])
    when (cl=state_cl && at=id_at) → State s
140 | _ → failwith "unwrapstate:_not_a_state"

let trans_from_mof = function
  | Mof.Inst(cl,[at,Mof.Str t])
    when (cl=trans_cl && at=label_at) → Transition t
145 | _ → failwith "unwraptrans:_not_a_transition"

(* Mof.mminstance → MicroStateMachine.t *)
let from_mof (Mof.Model(metamodel,instances,links)) =
  let machine_mof = List.find (fun (Mof.Inst(c,_)) → c = machine_cl) instances in
150 let state_mappings =
    List.fold_left
      (fun m i → InstMap.add i (state_from_mof i) m)
      InstMap.empty
      (List.filter (fun (Mof.Inst(c,_)) → c = state_cl) instances)
155 and trans_mappings =
    List.fold_left
      (fun m i → InstMap.add i (trans_from_mof i) m)
      InstMap.empty
      (List.filter (fun (Mof.Inst(c,_)) → c = trans_cl) instances)
160 and init_state =
    match List.find (fun (Mof.Link(m,_,a)) → m=machine_mof && a=init_as) links
    with (Mof.Link(_,i,_)) → i
  and accept_states =
    List.map
165 (fun (Mof.Link(_,i,_)) → i)
      (List.filter (fun (Mof.Link(m,_,a)) → m=machine_mof && a=accept_as) links) in
  let src_state_trans =
    match List.find (fun (Mof.Link(t,_,a)) → t=trans && a=source_as) links
    with (Mof.Link(_,i,_)) → InstMap.find i state_mappings
170 and dest_state_trans =
    match List.find (fun (Mof.Link(t,_,a)) → t=trans && a=dest_as) links
    with (Mof.Link(_,i,_)) → InstMap.find i state_mappings in
  let states_without_trans =
    InstMap.fold
175 (fun i s acc_map → SMap.add s TMap.empty acc_map)
      state_mappings
      SMap.empty
  in

```

```

180 StateMachine(
  (InstMap.find init_state state_mappings),
  (List.map (fun s → InstMap.find s state_mappings) accept_states),
  (InstMap.fold
    (fun i t sts_acc →
      let src, dest = (src_state i),(dest_state i) in
185      (SMap.add src
        (TMap.add t dest (SMap.find src sts_acc))
        sts_acc))
    trans_mappings
    states_without_trans )

```

EXT. 4.11 – (*microStateMachine.ml*) La machine à états de la figure 4.2, représentée d'abord par une valeur O'Caml (*machine1*) puis par un modèle μ MOF (*mofmachine1*).

```

(* Sample models *)
let machine1 =
  let (s1,s2,s3) = (State 1),(State 2),(State 3) in
  let (t1,t2,t3) = (Transition "t1"),(Transition "t2"),(Transition "t3") in
200 StateMachine (s1, [s2], SMap.from_assoc [
  (s1, TMap.from_assoc [ (t1,s2); (t3,s3) ]);
  (s2, TMap.from_assoc [ (t2,s3) ]);
  (s3, TMap.from_assoc [])
  ])
205
let mofmachine1 =
  let sm = Mof.Inst(machine_cl, [])
  and s1 = Mof.Inst(state_cl, [ id_at,Mof.Int 1 ])
  and s2 = Mof.Inst(state_cl, [ id_at,Mof.Int 2 ])
210 and s3 = Mof.Inst(state_cl, [ id_at,Mof.Int 3 ])
  and t1 = Mof.Inst(trans_cl, [ label_at,Mof.Str "t1" ])
  and t2 = Mof.Inst(trans_cl, [ label_at,Mof.Str "t2" ])
  and t3 = Mof.Inst(trans_cl, [ label_at,Mof.Str "t3" ]) in
  Mof.Model( statemachines_mm, [sm;s1;s2;s3;t1;t2;t3], [
215 Mof.Link(sm,s1,init_as); Mof.Link(sm,s2,accept_as);
  Mof.Link(sm,s1,state_as); Mof.Link(sm,s2,state_as); Mof.Link(sm,s3,state_as);
  Mof.Link(sm,t1,trans_as); Mof.Link(t1,s1,source_as); Mof.Link(t1,s2,dest_as);
  Mof.Link(sm,t2,trans_as); Mof.Link(t2,s2,source_as); Mof.Link(t2,s3,dest_as);
  Mof.Link(sm,t3,trans_as); Mof.Link(t3,s1,source_as); Mof.Link(t3,s3,dest_as); ] )

```

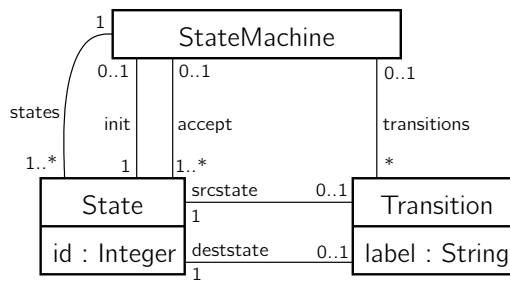


FIG. 4.1 – Métamodèle des machines à états de *MicroStateMachine*.

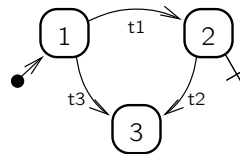


FIG. 4.2 – Machine à états *machine1* définie dans *MicroStateMachine* (extrait 4.11).

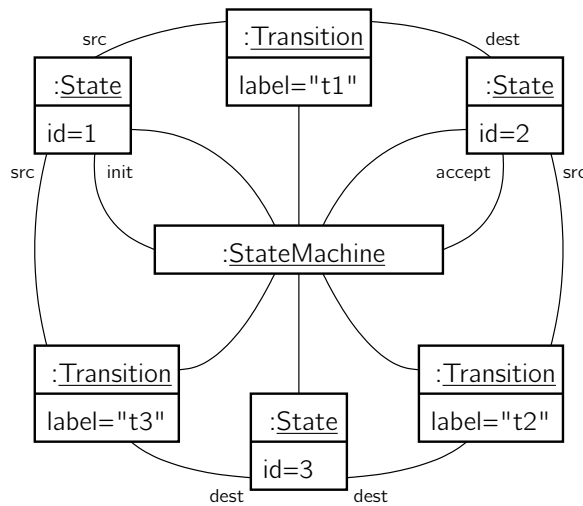


FIG. 4.3 – Diagramme d’objets de la machine à états *mofmachine1* définie dans *MicroStateMachine* (extrait 4.11).



FIG. 4.4 – Métamodèle des traces de MicroTrace.

4.2.2 Les traces

Le second métamodèle définit ce qu'est une trace, c'est-à-dire une séquence d'événements; chaque événement est étiqueté par une chaîne. La figure 4.4 montre le diagramme UML de ce métamodèle, et l'extrait 4.12 en détaille les deux représentations O'Caml et μ MOF.

EXT. 4.12 – (*microTrace.ml*) Représentation des traces par un type O'Caml et en tant que métamodèle issu de μ MOF.

```

(* OCaml representation of traces *)
type t = Trace of string list
10
(* MOF representation of traces *)

let trace_cl = { Mof.name="Trace"; Mof.attrs=[] }
let value_at = Mof.Attr("value",Mof.String)
15 let event_cl = { Mof.name="Event"; Mof.attrs=[ value_at ] }
let appears_as = {
  Mof.asso="appears";
  Mof.src=Mof.AEnd("Trace",Mof.One,"trace");
  Mof.dest=Mof.AEnd("Event",Mof.Some,"event");
20 Mof.ordered=true; Mof.aggreg=true
}
let traces_mm = {
  Mof.mm="Traces";
  Mof.classes=[ trace_cl; event_cl ];
25 Mof.gen=[ (* no generalization *) ];
  Mof.assocs=[ appears_as ]
}

```

De la même manière que les machines à états, on définit les deux fonctions `to_mof` et `from_mof` réalisant la traduction entre les deux représentations des traces (extrait 4.13).

EXT. 4.13 – (*microTrace.ml*) Traduction des traces entre la représentation par une valeur O'Caml et la représentation par un modèle.

```

(* to_mof : MicroTrace.t → Mof.mminstance *)
40 let to_mof (Trace l) =
  let t = Mof.Inst(trace_cl,[]) in
  let wrapevt s = Mof.Inst(event_cl,[ value_at, Mof.Str s ]) in
  let wrapplnk e = Mof.Link(t,e,appears_as) in

```

```

let el = List.map wrapevt l in
45 let ll = List.map wraplnk el in
   Mof.Model(traces_mm, t::el, ll)

(* from_mof : Mof.mminstance → MicroTrace.t *)
let from_mof (Mof.Model(mm,il,ll)) =
50 let tr = List.find (fun (Mof.Inst(c,_)) → c = trace_cl) il in
   let isLinked t e a =
       List.exists (fun (Mof.Link(tt,ee,aa)) → tt = t && ee = e && aa = a)
         ll
     in
55 let evts =
       List.filter (fun e → (match e with
         (Mof.Inst(c,_)) → (c = event_cl && isLinked tr e appears_as)))
         il
     in
60 Trace (List.map
         (fun (Mof.Inst(_,[_],Mof.Str s)) → s)
         evts)

```

4.3 Une transformation : extraction de traces d’une machine à états finis

Finalement, pour tester le bon fonctionnement de l’API, on peut comparer les résultats d’une transformation de modèles avec une spécification de cette transformation. D’une part, on implémente la transformation de modèles proprement dite par manipulation de la représentation μMOF du modèle à l’aide de l’API. D’autre part, on spécifie cette transformation en l’implémentant de manière spécifique, c’est à dire sur les types O’Caml définis précédemment. Comme la spécification est exécutable, elle peut servir d’oracle pour tester le bon fonctionnement de la transformation, et ce à travers les fonctions `to_mof` et `from_mof`. La figure 4.5 illustre ce protocole de test ; on y retrouve un schéma similaire à la figure 3.1, page 38.

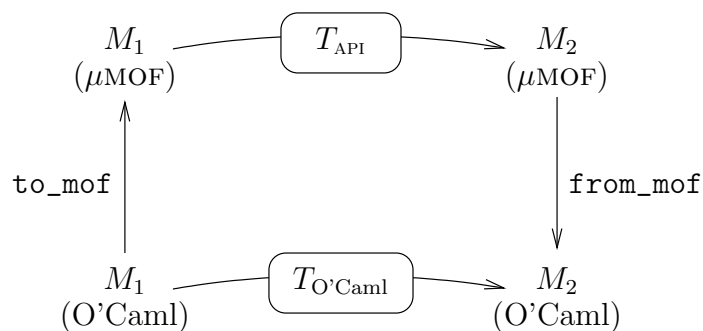


FIG. 4.5 – On exerce l’API en comparant les sorties de transformations de modèles avec celles de leurs spécifications exécutables.

La transformation présentée ici génère le modèle d'une trace, partant du modèle d'une machine à états. Elle construit la trace des événements correspondant une séquence de transitions tirée de la machine à états.

EXT. 4.14 – (*sm2t.ml*) Une transformation : extraction de traces à partir d'une machine à états. Partant de l'état initial de la machine, cette transformation tire systématiquement la première transition disponible.

```
(* may raise Not_found *)
let firstStep (StateMachine (_,_,sts)) src =
20   try
      List.hd (TMap.to_assoc (SMap.find src sts))
    with Failure "hd" → raise Not_found

(* extract a trace of length maxlen from a state machine,
25  choosing transitions with next *)
let trace (StateMachine(init,accept,sts_map)) next maxlen =
  let rec rtrace s tacc = function
    | 0 → tacc
    | n → try
30       let (Transition e), s2 = next (StateMachine(init,accept,sts_map)) s in
          rtrace s2 (MicroTrace.append tacc (MicroTrace.Trace [e])) (n-1)
        with Not_found → tacc
  in
  rtrace init MicroTrace.empty maxlen
```

EXT. 4.15 – (*sm2t.ml*) Réimplémentation de la transformation d'extraction de traces, sur la représentation μ MOF des deux métamodèles manipulés, et en utilisant l'API.

```
let mof_firstStep rep src =
  let machine_cl = Api.getMetaClass rep "StateMachine" in
60  let state_cl = Api.getMetaClass rep "State" in
      let trans_cl = Api.getMetaClass rep "Transition" in
          let source_as = Api.getMetaAssociation rep "source" in
              let dest_as = Api.getMetaAssociation rep "destination" in
                  let src_ae = Api.getMetaAssociationEnd rep "srcstate" state_cl trans_cl in
65  let dest_ae = Api.getMetaAssociationEnd rep "deststate" state_cl trans_cl in
                      let src_cn i =
                          match Api.invokeQueryOperation rep i () (Api.EndF src_ae) ()
                            with Api.Coll c → List.exists (fun i → i = src) c in
                          let step_trans = List.hd (Api.allInstancesWithConstraint rep trans_cl src_cn) in
70  let step_dest = Api.ocl_hd (
                          Api.invokeQueryOperation rep step_trans () (Api.EndF dest_ae) ())
                      in
                      (step_trans, step_dest)

75 let mof_trace rep next maxlen =
  let machine_cl = Api.getMetaClass rep "StateMachine" in
      let state_cl = Api.getMetaClass rep "State" in
          let machine rep = List.hd (Api.allInstances rep machine_cl) in
              let init_ae = Api.getMetaAssociationEnd rep "initstate" state_cl machine_cl in
80  let initstate = Api.ocl_hd (
              Api.invokeQueryOperation rep (machine rep) () (Api.EndF init_ae) ()) in
```

```

let rec rtrace s tacc =
  let trace_cl = Api.getMetaClass tacc "Trace" in
  let event_cl = Api.getMetaClass tacc "Event" in
85  let appears_as = Api.getMetaAssociation tacc "appears" in
  let trace_ae = Api.getMetaAssociationEnd tacc "trace" trace_cl event_cl in
  let event_ae = Api.getMetaAssociationEnd tacc "event" event_cl trace_cl in
  let value_at = Api.getMetaAttribute tacc "value" event_cl in
  let add_event trep t =
90    let label =
      match
        Api.invokeQueryOperation rep t () (Api.AttrF value_at) ()
        with Api.Scal v → v in
      let trace_inst = List.hd (Api.allInstances trep trace_cl) in
95      let trep2, ne = Api.instantiate trep event_cl () () in
      let trep3, ne2 = Api.setAttributeValue trep2 ne () value_at label in
      Api.associateModelElements trep3 appears_as ()
      [(trace_ae, trace_inst); (event_ae, ne2)]
    in function
100  | 0 → tacc
    | n → try
      let t, s2 = next rep s in
      rtrace s2 (add_event tacc t) (n-1)
      with Not_found → tacc
105  in
  rtrace initState MicroTrace.emptyoftrace maxlen

```

4.4 Bilan et perspectives

4.4.1 Le coeur sémantique

Pour qu’une interface de programmation soit implémentable, et pour que les implémentations s’y conformant soient interopérables, cette interface de programmation doit être définie précisément et sans ambiguïté. Malheureusement l’une des grandes difficultés liées à la méta-programmation, et amplifiée ici par l’architecture à quatre « niveaux méta » de l’OMG, est justement de savoir à quel niveau se trouvent les concepts qu’on manipule. Dans le domaine des transformations de modèles, cette difficulté est encore augmentée car on retrouve des concepts nommés similairement à tous les niveaux : classes, instances, associations... Pour réduire à néant ces possibilités de confusion entre concepts différents, une définition formelle est donc nécessaire.

Nous avons cependant choisi d’éliminer une approche par des formalismes purement mathématiques au profit de l’implémentation d’un prototype exécutable en langage O’Caml. Nous nous appuyons donc sur la sémantique de ce langage d’implémentation pour définir celle de l’interface d’accès aux modèles. La formalité de l’approche est assurée par le système de typage statique strict d’O’Caml, et par le choix d’un style fonctionnel² plutôt qu’impératif. On obtient ainsi un programme qu’on pourrait au besoin aisément

transcrire sous une forme mathématique, et qui joue simultanément le rôle de démonstrateur.

4.4.2 Le langage MTL en pratique

Dans ce chapitre, nous avons défini la sémantique de l'interface d'accès aux dépositaires de modèles. Cependant, la contrainte d'indépendance vis-à-vis des métamodèles et des dépositaires rend cette interface peu pratique pour une utilisation directe. En pratique on développe les transformations dans un langage de haut niveau appelé MTL. L'environnement d'exécution de MTL découvre la structure de chaque métamodèle utilisé en s'appuyant sur une implémentation en Java de l'interface d'accès aux dépositaires spécifiée précédemment en O'Caml. L'interface d'accès elle-même n'est directement utile qu'aux développeurs des pilotes de dépositaires (voir la figure 3.3, page 43, colonne de gauche). Ce langage MTL est un langage impératif orienté objet, avec une syntaxe concrète textuelle inspirée d'OCL pour la navigation dans les modèles et de Java pour la partie impérative. Pour faciliter la modélisation de transformations, des concepts des diagrammes de classes UML sont repris dans le langage : packages, classes, associations, visibilitées...

Pour un programme de transformation, un métamodèle est similaire à une librairie de classes : on manipule les éléments d'un modèle de la même manière que tout objet MTL. Concrètement, les modèles apparaissent à la fois comme des conteneurs et des espaces de noms : après initialisation d'un dépositaire, on obtient un espace de noms donnant accès aux classes du métamodèle. Ces classes sont déjà instanciées si un modèle a été chargé dans le dépositaire, et on peut accéder à ces instances en à l'aide de l'opération `allInstances()`, par exemple. L'initialisation et l'obtention de l'espace de noms sont spécifiques à chaque dépositaire et chaque modèle : en particulier, les noms de fichiers XMI à charger y apparaissent ; ce code est donc souvent écrit dans un « pilote » qui appelle la transformation ensuite (extrait 4.16).

On déclare des classes MTL de la même manière qu'en Java (extrait 4.17). Ces classes servent exclusivement à structurer le programme de transformation ; il n'est pas possible d'ajouter ainsi des classes à un métamodèle. En effet, les dépositaires ne sont pas forcément extensibles à ce niveau, même si certains dépositaires (MDR par exemple) sont capables de charger des métamodèles arbitraires³.

-
2. O'Caml n'est pas un langage purement fonctionnel, il offre des constructions impératives même si sa syntaxe encourage un style déclaratif.
 3. Dans ce cas on peut appliquer une transformation au métamodèle, puis charger celui-ci pour obtenir un *nouveau* dépositaire.

Quand on crée un élément dans un modèle, on spécifie ce modèle en explicitant l'espace de noms (UMLrep dans les extraits 4.16 et 4.18). Même si on manipule deux modèles UML, par exemple, tout se passe comme si les classes du métamodèle existaient une fois dans chaque espace de noms. Les objets créés sans espace de noms sont internes à MTL et leur durée de vie est limitée à l'exécution de la transformation. Le typage est statique pour les objets internes à MTL, et dynamique pour les objets des dépositaires, c.-à-d. les éléments de modèle.

Les associations sont gérées directement par le langage, que ce soit entre objets MTL ou éléments de modèle : les mots-clés `associate` et `dissociate` créent ou détruisent un lien identifié entre deux objets (extrait 4.19). Le lien précis est identifié par les rôles et les classes des objets ; en effet, d'une part deux mêmes objets peuvent être liés plusieurs fois par des associations différentes, et d'autre part les associations peuvent être redéfinies à plusieurs niveaux de la hiérarchie de classes.

Pour résumer, MTL est un langage de transformation proche des langages orientés objets « classiques », il permet d'appliquer des méthodologies et des patrons de conception familiers. Il n'impose pas de contraintes sur la forme ou l'arité des transformations : n'importe quel nombre de modèles peuvent être utilisés en lecture ou écriture au cours de la transformation. Les programmes MTL peuvent être vus comme des modèles, situés au même niveau que les modèles transformés. Le langage en lui-même est indépendant des métamodèles ; des pilotes découplent l'environnement d'exécution des technologies de dépositaires de modèles.

EXT. 4.16 – Initialisation d'un dépositaire en MTL.

```

model UMLrep : RepositoryModel;
model mdrdriver : MDRDriver;

main() : Standard::Void {
5 // on obtient un objet "facade" pour le pilote MDR
  mdrdriver := new MDRDriver::MDRModelManager();
  mdrdriver.init();
  // MDR charge les modèles depuis des fichiers XMI
  UMLrep := mdrdriver.getModelFromXMI(
10   'metamodelFilename.xmi',
    'UML', // nom du package racine dans le métamodèle UML 1.4
    'UML1.4_model',
    'inputModelFilename.xmi', 'outputModelFilename.xmi');
  aTransformation := new PrivatizeAttributeTransformation();
15 aTransformation.run(UMLrep); // on passe le conteneur à la transformation
}

```

EXT. 4.17 – Déclaration d'une classe en MTL.

```

class PrivatizeAttributeTransformation extends Transformation {
  UML : RepositoryModel; // déclaration d'attribut

  run (UMLrep : RepositoryModel) {
5 // ... code de la transformation
  }
  // ... autres méthodes, utilisées par run()
}

```

EXT. 4.18 – Création et destruction d'objets en MTL.

```

newOperation : UMLrep::Core::Operation;
// création dans le modèle UMLrep
// une exception serait levée si Operation n'existait pas
newOperation := new UMLrep::Core::Operation();
5
anOperation.name.toOut(); // affiche le nom de l'opération
// une exception est levée si l'attribut n'existe pas
newOperation.name := 'fancyNewName'; // affectation

10 // destruction de l'élément et de ses constituants (ici, name)
newOperation.delete();

```

EXT. 4.19 – Gestion des associations en MTL.

```

// déclaration d'une association entre deux classes A et B
association { theA : A (0 1); // rôle : classe multiplicité
              myBs : B -1 ordered; }

5 // pour deux instances a et b des classes A et B et jouant les rôles theA et myBs de l'association
associate ( // la syntaxe est identique pour dissociate
  theA := a : A,
  myBs := b : B
);

```


Troisième partie

Application

Chapitre 5

Application

Les transformations de modèles ont de nombreuses applications possibles au cours du cycle de vie d'une application ; une bonne partie de ces applications peuvent d'ores et déjà être classées comme suit :

- les transformations affectant l'ensemble d'un modèle, ou produisant un modèle entièrement nouveau, par exemple un changement de représentation, l'extraction d'une vue, ou le tissage d'un aspect ;
- les modifications locales, utilisées en tant que raccourcis de conception ou de programmation, à la manière du préprocesseur en C, des macros en Lisp, ou des fonctions de méta-programmation en Ruby ;
- les modifications à plus large échelle ou fortement dépendantes de la sémantique du modèle, telles que l'application de refactorings ou l'introduction de design patterns.

En pratique, que ce soit pour l'introduction de design patterns ou l'application de refactorings, les manipulations de modèles concrètes sont semblables. En fait, on peut même introduire la plupart des design patterns dans un modèle en appliquant une séquence de refactorings [89]. C'est donc autour de ce troisième point que s'articule cette partie : nous discutons tout d'abord de l'intérêt d'appliquer des refactorings aux modèles UML, que nous illustrons par quelques exemples d'insertion de design patterns. La section 5.3 concerne les refactorings proprement dits et propose un ensemble de refactorings pour les modèles UML.

5.1 Refactoring de modèles

Les activités de conception logicielle ne se limitent pas à la création de nouvelles applications en partant de rien ; très souvent le concepteur doit modifier et faire évoluer le comportement et les fonctionnalités d'une application existante.

Il est maintenant reconnu comme étant une bonne pratique de diviser une évolution en deux étapes [51, 89] :

1. Sans introduire de nouveau comportement au niveau conceptuel, restructurer le logiciel pour améliorer des facteurs de qualité tels que la maintenabilité, la lisibilité, l'efficacité etc.
2. Tirer parti de la conception améliorée pour modifier le comportement du logiciel.

Cette première étape appelée *refactoring* (cf. sec. 2.2.3) est maintenant considérée comme essentielle pendant le développement et la maintenance d'un logiciel. Les travaux sur les refactorings se sont dès le départ dirigés vers la transformation de programmes orientés objet [63, 91, 103] ; William Opdyke donne deux raisons à cela :

- Comparée à des approches du développement plus traditionnelles, la programmation orientée objet facilite les refactorings car elle explicite les informations structurelles nécessaires.
- Le refactoring est plus particulièrement important dans la programmation orientée objet. Certains membres de la communauté accordent une grande importance à la conception et re-conception du logiciel pour le rendre plus réutilisable. Dans certains cas, la meilleure façon d'améliorer le design d'un programme est de le réécrire ; dans d'autres cas, sa restructuration peut être d'un abord plus aisé.

Brant et Roberts [24] présentent les refactorings comme un outil essentiel pour gérer l'évolution d'un logiciel. Selon eux, puisque les méthodes traditionnelles de développement fondées sur le cycle de vie en cascade placent la maintenance en dernière phase du cycle de vie d'un logiciel, elles ne prennent pas en compte l'évolution de ce logiciel. Ils remarquent aussi que d'autres méthodes s'inspirant plutôt du cycle de vie en spirale telles que Joint Application Development [128] ou plus récemment Extreme Programming [19], offrent un meilleur support pour l'évolution du logiciel. Ces méthodes encouragent l'utilisation des langages de quatrième génération¹ et des environnements de développement intégrés, et sont donc plus appropriées à l'utilisation des refactorings. Puisque UML semble être plus proche de l'esprit des méthodes du premier type que des méthodes plus agiles, on pourrait s'attendre à ce que l'intégration des refactorings à UML n'en vaille pas la peine.

Malgré cette apparente incompatibilité méthodologique, nous pensons que les refactorings peuvent être intégrés profitablement dans les outils UML [115]. Les méthodes ont changé depuis les premières observations de Brooks [25],

1. Langages de programmation conçus dans un but spécifique; voir http://en.wikipedia.org/wiki/Fourth-generation_programming_language

et la frontière entre les deux familles est maintenant moins stricte. Les méthodes récentes, comme par exemple Catalysis [43] qui utilise UML comme notation, prennent en compte l'évolution des logiciels et donc l'évolution de la conception. De plus, comme certains outils² offrent maintenant la possibilité de créer des modèles de conception à partir du code source, les refactorings pourraient servir à modifier ce code et à améliorer la conception d'applications existantes.

Cependant, en pratique il est difficile de mesurer l'impact réel des modifications sur les différentes facettes de la conception comme sur l'implémentation. C'est particulièrement vrai d'UML : ses multiples vues structurelles et dynamiques peuvent partager de nombreux éléments de modèle ; par exemple, quand une méthode est supprimée d'un diagramme de classes, il est difficile de dire à première vue, sans l'aide d'un outil, quel est l'impact sur les diagrammes de séquence ou d'activités, les collaborations, les contraintes OCL etc. Pourtant, UML a un avantage clé par rapport aux autres langages de conception : sa syntaxe est définie par un métamodèle qui définit comment s'intègrent les différentes vues ; on peut donc utiliser le métamodèle pour contrôler l'impact d'une modification, ce qui est primordial si cette modification doit préserver le comportement initial de l'application.

Une utilisation intéressante des refactorings est l'insertion de design patterns dans un diagramme de conception UML ; nous développons maintenant quelques exemples où les refactorings sont utilisés pour améliorer la conception de l'application présentée en section 2.1.3.

5.2 Instanciation de patrons de conception

Comme il est rappelé à la section 2.2.1, les patrons de conception ou *design patterns* sont des solutions génériques à des problèmes de conception récurrents. Si le choix de tel ou tel patron pour résoudre un problème donné dépend de facteurs difficilement quantifiables, l'insertion du schéma de solution dans le modèle de conception est une affaire relativement systématique, et qui gagnerait donc à être automatisée.

Mél O'Cinnéide propose une méthodologie pour l'application de design patterns [89], basée sur l'application de refactorings primitifs. Ces refactorings sont ensuite combinés en minitransformations qui à leur tour permettent d'exprimer la transformation d'application d'un design pattern. O'Cinnéide a

2. p. ex. Objecteering (www.objecteering.com), Poseidon (www.gentleware.com), Together (www.borland.com/together) ou VisualParadigm (www.visual-paradigm.com)

illustré ses travaux par la réalisation d'un outil³ manipulant des programmes Java. L'utilisation des refactorings pour l'insertion de design patterns a d'ailleurs récemment fait l'objet d'un livre[68]. Nous illustrons ici comment ces transformations peuvent être adaptées à la manipulation de modèles.

Les transformations étant composées à partir de refactorings, elles ne peuvent pas changer le comportement de l'application ; le comportement doit donc être déjà présent sous forme d'un *précurseur*. O'Cinnéide définit ces précurseurs comme étant « *une structure logicielle exprimant l'objectif du design pattern de manière simpliste mais n'étant pas pour autant une erreur de conception* ». Par exemple il donne le diagramme reporté en figure 5.1 pour décrire le précurseur du design pattern Factory Method [53, p. 107].

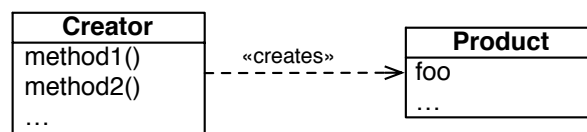


FIG. 5.1 – Précurseur pour le pattern Factory Method [89].

O'Cinnéide juge sa méthodologie difficilement applicable dans le cas du design pattern Observer [53, p. 293], faute d'avoir trouvé un précurseur pertinent pour la transformation [89, p. 147]. En effet, la dépendance entre sujet et observateurs potentiels peut être implémentée de nombreuses manières différentes, et les précurseurs possibles sont soit trop vagues soit improbables en pratique⁴. Mais dans un modèle, il est possible de préciser cette dépendance sous une forme abstraite, par exemple via une contrainte OCL ou tout simplement la notation UML de l'occurrence d'un design pattern.

De plus, en se basant sur [74] on dispose d'un formalisme de description des design patterns au niveau du métamodèle, et suffisamment précis pour qu'il soit exploitable par un outil d'application automatique. Étant donné un design pattern donné par un tel modèle, les problèmes qui se posent pour son introduction sont les suivants :

- paramétrer l'application du pattern en affectant explicitement les rôles du pattern aux classes du modèle, ou identifier parmi les classes présentes celles qui joueront les rôles du pattern ;
- sélectionner une variation d'implantation parmi les différentes possibilités, ou paramétrer l'application du pattern par les informations nécessaires.

3. DPT : Design Patterns Tool

4. Par exemple, un mécanisme d'observation avec l'abonnement et la notification, mais un seul observateur.

5.2.1 Exemple : insertion du design pattern Observer

Dans le système de réunions virtuelles décrit en 2.1.3, quand un participant entre, sort, ou prend la parole dans une réunion, tous les participants de la réunion doivent être notifiés. Durant la phase de conception il faudra donc introduire le design pattern Observer [53, p. 293] ; en passant, l'utilisation du pattern est ici préférable à une solution ad-hoc parce que les participants peuvent arriver ou repartir à tout moment de la réunion, mais aussi parce qu'ils peuvent n'être intéressés que par certains types d'événements. À l'analyse, la présence d'Observer est donc spécifiée dans le modèle du système de réunions virtuelles par le précurseur en figure 5.2.

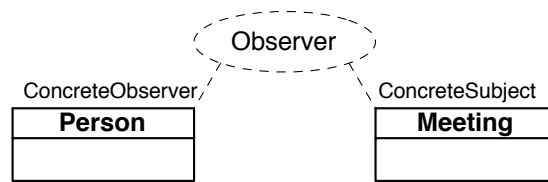


FIG. 5.2 – Précurseur du design pattern Observer dans la structure du serveur de réunions virtuelles (fig. 2.3).

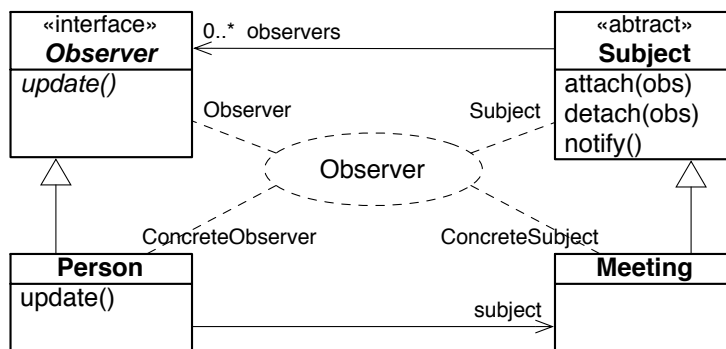


FIG. 5.3 – Structure du serveur de réunions virtuelles (fig. 2.3) après ajout du design pattern Observer.

Pour expliciter la structure de ce design pattern et obtenir la figure 5.3, il faut ajouter la super-classe abstraite Subject à Meeting et l'interface Observer à Person. Dans la transformation MTL présentée dans les extraits 5.1 et 5.2, ces deux éléments sont respectivement assemblés par les méthodes buildSubject et buildObserver. Les relations d'héritage sont ensuite mises en place par addGeneralization. Le mot-clé associate établit un lien dans le modèle, dont les extrémités et la méta-association sont identifiées par chacun des arguments :

```
// création d'un lien entre instanceA et instanceB :
associate( roleA := instanceA : typeA,
           roleB := instanceB : typeB )
```

Dans cet exemple on se contente de déclarer les méthodes, mais dans les cas où le langage d'implémentation est connu il serait tout à fait possible de fournir une implémentation raisonnable pour `attach`, `detach` et `notify`. On pourrait également insérer des appels à `notify` dans tous les accesseurs en écriture de `Meeting`, mais cela dépend de la sémantique de chaque application et n'est donc pas souhaitable dans le cas général. De même, l'implémentation de `update` restera à la charge du développeur.

5.2.2 Exemple : insertion du design pattern Command

Dans le système de réunions virtuelles, la classe `VirtualMeetingServer` a pour rôle de traiter les messages en provenance des logiciels clients. Ces messages sont acheminés *via* la couche de gestion des communications implémentée dans le package `Networking`. Il faut en particulier reconnaître le type de chaque message pour pouvoir traiter les données spécifiques à chaque type. Dans un premier temps ces traitements peuvent être mis sous la responsabilité de `VirtualMeetingServer`, par exemple sous la forme des opérations spécifiées en figure 5.4. Évidemment, au fur et à mesure qu'on prend en compte de nouveaux aspects du cahier des charges, le nombre de types de messages augmente et avec lui le nombre de méthodes dans `VirtualMeetingServer`.

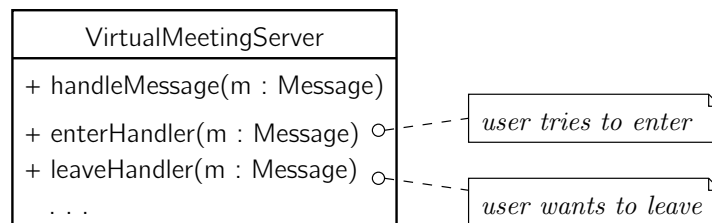


FIG. 5.4 – *Version préliminaire de la gestion des messages*

Une meilleure façon de faire est d'utiliser le design pattern Commande, en déléguant la gestion des messages à une classe spécialisée pour chaque type, plutôt qu'à une méthode de `VirtualMeetingServer`. Modifier le modèle actuel pour faire apparaître Commande ne changera pas le comportement de `VirtualMeetingServer` du point de vue d'un appelant de `handleMessage()` ; par contre, on pourra ensuite ajouter et tester indépendamment les structures de gestion de chaque type de message.

Pour arriver au modèle final en figure 5.6, on va chercher dans un premier temps à obtenir le modèle en figure 5.5, dans lequel la gestion des messages concrets est déléguée à un ensemble de classes externes au package `VirtualMeetingServer`. La marche à suivre pour obtenir ce résultat fait in-

EXT. 5.1 – *Insertion du design pattern Observer (Observer.mt1).*

```

library Observer;
model m : RepositoryModel;

addObserverPattern(theSubject : m::Class; theObserver : m::Class)
5 {
  observerI : m::Interface;
  subject : m::Class;

  observerI := buildObserver(); addGeneralization(theObserver, observerI);
10 subject := buildSubject(observerI); addGeneralization(theSubject, subject);
  addAssociation(subject, observerI, 'observers');
  addAssociation(theObserver, theSubject, 'subject');
}

15 buildObserver() : m::Interface
{
  result : m::Interface;
  update : m::Operation;

20 result := new m::Interface(); result.name := 'Observer';
  update := new m::Operation(); update.name := 'update';
  associate( method := update : m::Operation, owner := result : m::Interface );
  return result;
}

25 buildSubject(observerI : m::Interface) : m::Class
{
  result : m::Class;
  attach : m::Method;
30 detach : m::Method;
  notify : m::Method;

  result := new m::Class(); result.name := 'Subject';
  attach := new m::Method(); attach.name := 'attach';
35 addParameter(attach, 'obs', observerI);
  associate( method := attach : m::Method,
            owner := result : m::Interface );
  detach := new m::Method(); detach.name := 'detach';
  addParameter(detach, 'obs', observerI);
40 associate( method := detach : m::Method,
            owner := result : m::Interface );
  notify := new m::Method(); notify.name := 'notify';
  associate( method := notify : m::Method,
            owner := result : m::Interface );
45 return result;
}

```

EXT. 5.2 – Fonctions auxiliaires pour l'insertion du design pattern Observer (suite de *Observer.mtl*).

```

addGeneralization(childElement : m::GeneralizableElement,
                  parentElement : m::GeneralizableElement)
{
50  gen : m::Generalization;

    gen := new m::Generalization();
    associate( child := childElement : m::GeneralizableElement,
               generalization := gen : m::Generalization );
55  associate( parent := parentElement : m::GeneralizableElement,
               specialization := gen : m::Generalization );
}

addParameter(owner : m::BehavioralFeature, name : String, type : m::Classifier)
60 {
    param : m::Parameter;

    param := new m::Parameter(); param.name := name;
    associate( behavioralFeature := owner : m::BehavioralFeature,
65               parameter := param : m::Parameter );
    associate( type := type : m::Classifier,
               typedParameter := param : m::Parameter );
}

70 addAssociation(src : m::Classifier, dest : m::Classifier
                 destRole : String)
{
    asso : m::Association;
    srcEnd : m::AssociationEnd;
75  destEnd : m::AssociationEnd;

    asso := new m::Association();
    srcEnd := new m::AssociationEnd();
    destEnd := new m::AssociationEnd(); destEnd.name := destRole;
80  associate( connection := srcEnd : m::AssociationEnd,
               association := asso : m::Association );
    associate( connection := destEnd : m::AssociationEnd,
               association := asso : m::Association );
    associate( participant := src : m::Classifier,
85               association := srcEnd : m::AssociationEnd );
    associate( participant := dest : m::Classifier,
               association := destEnd : m::AssociationEnd );
}

```

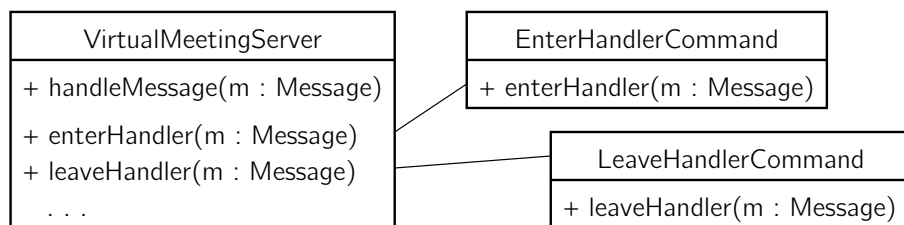


FIG. 5.5 – Étape intermédiaire de délégation de la gestion des messages à des classes externes : chaque méthode *-Handler()* délègue son comportement à la méthode *execute()* de classe *-HandlerCommand* correspondante.

tel-00538536, version 1 - 22 Nov 2010

tervenir un bon nombre de refactorings successifs ; tout d'abord, pour chaque méthode concernée dans `VirtualMeetingServer` :

1. on crée une classe vide, nommée convenablement ;
2. on associe entre la nouvelle classe à `VirtualMeetingServer` ;
3. à travers cette association, on déplace la méthode en laissant une méthode de délégation dans `VirtualMeetingServer` ;

Il faut maintenant regrouper toutes les classes déléguées sous une interface commune et simplifier la manière dont elles sont identifiées :

1. on renomme les méthodes déplacées en `execute(...)` pour correspondre à l'interface des commandes ;
2. on crée la super-classe `MessageHandlerCommand`, commune à toutes les futures commandes concrètes ;
3. l'opération générique `execute(...)` peut ensuite être spécifiée au niveau de `MessageHandlerCommand` ;
4. similairement les associations peuvent être remontées d'un niveau ;
5. on peut finalement remplacer la collection d'associations par une agrégation unique mais qualifiée.

On obtient le système présenté en figure 5.6 ; extérieurement et même du point de vue de la méthode `handleMessage(...)` `VirtualMeetingServer` fonctionne de la même manière qu'en figure 5.4.

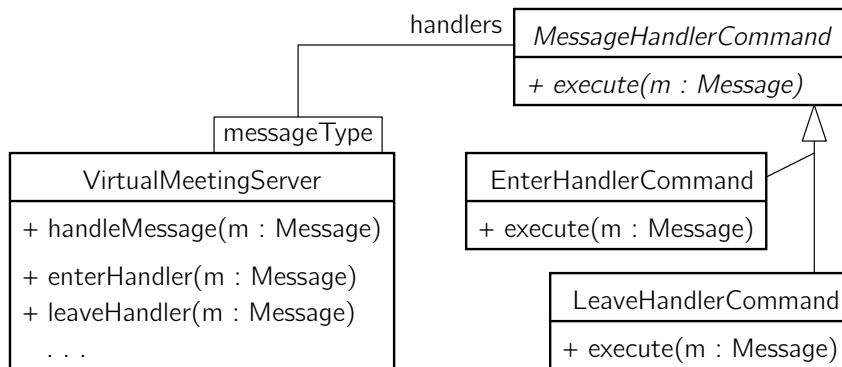


FIG. 5.6 – Délégation de la gestion des messages à des classes externes

La dernière étape consiste à éliminer les méthodes `msgHandler(...)`, qui sont spécifiques à certains types de messages et qui ne devraient donc pas apparaître dans `VirtualMeetingServer`. Ces méthodes ne font maintenant plus qu'un appel à `execute(...)` sur l'instance correspondant à une clé spécifique dans l'association `handlers`. On peut incorporer les appels à `execute(...)` directement dans `handleMessage(...)`, puis éliminer les méthodes car elles

ne sont plus appelées. Finalement, en choisissant pour clé de l'association `handlers` une valeur extractible génériquement de tout message, on peut même factoriser tous les appels à `execute(...)` et ainsi supprimer toute référence à un type concret de message dans `VirtualMeetingServer`.

Nous venons de voir comment on peut insérer des design patterns dans un diagramme de conception UML. Quand la fonctionnalité est déjà présente sous la forme d'un précurseur, l'insertion du design pattern correspondant se réduit à l'application d'une séquence de refactorings. Cependant, l'insertion de design patterns n'est qu'une utilisation particulière des refactorings ; la section suivante les présente dans un cadre plus général.

5.3 Des refactorings pour UML

Les refactorings des diagrammes de classes UML utilisés dans les exemples précédents correspondent aux refactorings les plus structuraux utilisés pour du code. Pour des aspects comportementaux, on utilise les autres vues d'UML ; par exemple les diagrammes d'états se prêtent particulièrement bien au refactoring car ils ont une sémantique précise. Nous illustrons ceci avec l'exemple suivant, avant de présenter un catalogue de refactorings pour UML 1.x [115].

5.3.1 Exemple sur les diagrammes d'états

La figure 5.7 est l'un des diagrammes d'états du système de réunions virtuelles ; il spécifie les états que peut prendre la classe `Person` introduite à la figure 2.3, page 14. Bien que ce diagramme soit correct, il ne fait pas clairement apparaître l'existence de deux modes de fonctionnement : la personne est en réunion ou non. On peut donc expliciter ces modes en ajoutant un macro-état `Attending` qui regroupera les trois états `Listening`, `WaitingToSpeak` et `Speaking` ; ce nouvel état sera connecté à `Idle` par les transitions `enter` et `leave`, cette dernière n'apparaissant plus qu'une fois dans le diagramme.

La transformation est effectuée en quatre étapes :

1. création de l'état composite, nommé `Attending` et englobant entièrement le diagramme de départ ;
2. déplacement de l'état `Idle` et du pseudo-état initial hors de `Attending` ;
3. fusion des transitions `leave` en une seule transition de haut niveau partant de la frontière de `Attending` ;

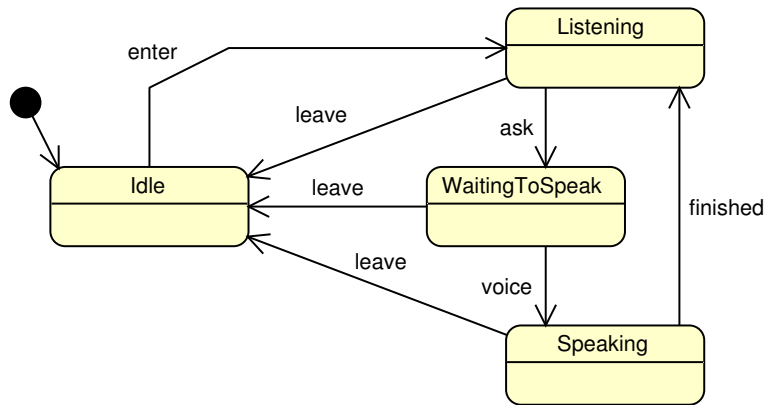


FIG. 5.7 – Diagramme d'états perfectible pour une personne connectée au système de réunions virtuelles

4. finalement, la transition `enter` peut être divisée en une transition de haut niveau de `Idle` vers `Attending` et à l'intérieur de ce dernier une transition initiale désignant `Listening`.

À la suite de ces modifications on obtient le diagramme déjà vu à la section 2.1.4, page 15, et rappelé en figure 5.8.

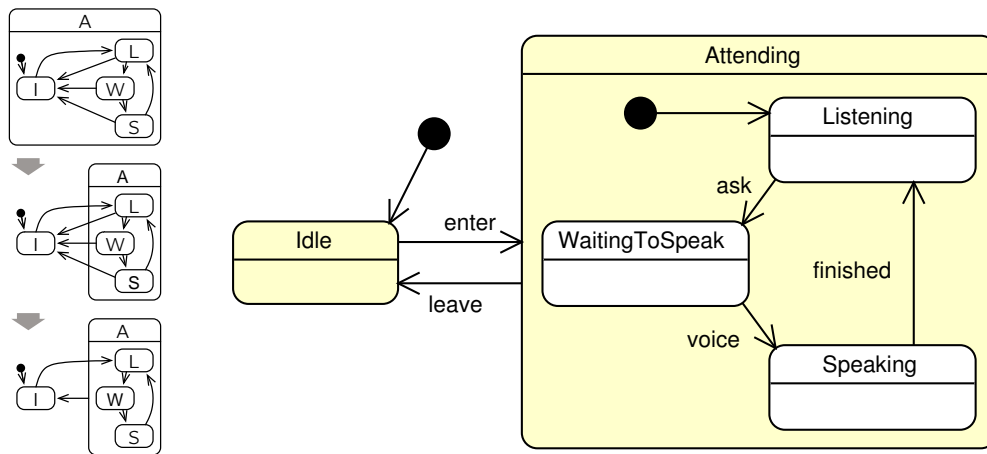


FIG. 5.8 – Étapes de refactoring et diagramme final (cf. fig. 2.4) des états d'une personne connectée au système de réunions virtuelles

Telles qu'elles ont été appliquées, les transformations précédentes sont des refactorings ; elles préservent le comportement⁵ pour les raisons suivantes :

- la création d'un état englobant n'ajoute pas de comportement et n'interfère pas non plus avec celui déjà spécifié par le statechart initial ;

5. Il existe plusieurs sémantiques pour les statecharts, celle que nous considérons ici est celle d'UML 1.x.

- déplacer l'état `Idle` hors de `Attending` est possible car ce dernier n'a ni action d'entrée ni action de sortie ; l'ordre d'exécution des actions est donc inchangé ;
- les transitions quittant `Attending` peuvent être fusionnées car elles sont équivalentes : elles portent la même étiquette, ont la même destination, et partent de tous les sous-états de `Attending` ;
- ici, le remplacement de la transition `enter` est possible car il n'y a pas d'autre transition entrante qui puisse être affectée.

5.3.2 Équivalence d'éléments de modèle

La notion d'équivalence d'éléments de modèle apparaît régulièrement dans les définitions de refactorings ; en effet, cette notion correspond à la préservation du comportement dans le domaine des refactorings de programmes. Or les approches formelles de la préservation de comportement apparaissent irréalistes, parce que peu de langages ont des spécifications suffisamment formelles, mais aussi à cause de la complexité des preuves à produire [89]. De fait, les travaux existants proposent une approche semi-formelle [91] voire totalement informelle [51].

En pratique on est souvent réduit à faire confiance aux jeux de tests. Cette approche élimine une partie de la question de la conservation du comportement, si on considère que les tests constituent une spécification du logiciel. Elle semble donc bien justifiée dans une méthodologie où le développement est dirigé par les tests⁶[96], car les tests sont alors écrits dans une perspective de spécification plus que de vérification.

Dans le cas du refactoring de modèles, définir formellement l'équivalence d'éléments de modèle est un problème difficile, et qui plus est pas forcément justifié : en effet un modèle est par nature incomplet, particulièrement à l'analyse et pendant les phases initiales de la conception. Fréquemment le modèle manipulé n'a pas de *comportement* proprement dit que le refactoring puisse préserver — un diagramme de classes ne spécifiant que des opérations et pas de méthodes par exemple. Parfois aussi ce qu'on veut préserver lors d'un refactoring n'est pas le comportement mais une propriété dépendante du contexte d'application de la transformation ; dans ce cas il n'y a donc pas une notion d'équivalence unique. Plusieurs approches pragmatiques sont envisageables pour répondre à cette problématique :

6. On écrit d'abord un test, puis on « corrige » le code en conséquence pour que le test passe.

- dans tous les cas il faut bien sûr assurer la cohérence du modèle vis-à-vis des contraintes sémantiques du métamodèle UML : les *Well-Formedness Rules* de la spécification du langage ;
- dans les cas où le modèle est très abstrait ou incomplet, une équivalence structurelle convient : par exemple la correspondance des signatures pour les méthodes ;
- la transformation peut insérer des contraintes dans le modèle, laissant la charge au développeur de les vérifier ou de les respecter pendant des modifications ultérieures, à la manière des obligations de preuve en langage B [1].

5.3.3 Refactoring des diagrammes de classes

Les refactorings présentés par Opdyke dans sa thèse [91], et qui ont été par la suite améliorés et implémentés par Roberts [103], ainsi que les transformations de restructuration présentées par ailleurs [20, 28, 58, 63] s'appliquent essentiellement à trois concepts : les classes, les méthodes et les variables. Nous avons donc naturellement commencé la transposition des refactorings existants vers UML, en nous intéressant aux diagrammes de classes.

Les refactorings présentés ici peuvent être classés en quatre types d'opérations de base : ajout, suppression, généralisation et spécialisation d'éléments de modèle. Les deux derniers types déplacent des éléments à travers la hiérarchie d'héritage, le long des relations de généralisation. La plupart des éléments de modèle composant un diagramme de classes peuvent avoir une connexion directe à des éléments d'autres vues ; certains refactorings s'appliquant aux diagrammes de classe ont donc potentiellement un impact sur d'autres vues d'UML.

Ajout et suppression d'éléments

L'ajout de membres (attributs et méthodes) et d'associations à une classe est possible quand le nouveau membre ou la nouvelle association n'a la même signature qu'aucun autre membre ou association de la classe considérée, d'une super-classe ou d'une sous-classe de celle-ci.

La suppression d'associations et de membres n'est possible que si l'élément supprimé n'est pas référencé dans le modèle entier ; une méthode peut par exemple être référencée dans un diagramme d'interaction si elle est liée à un message ou un stimulus, ou encore dans un diagramme d'états si elle est liée à une action ou un événement.

L'ajout et la suppression de classes devient particulièrement intéressant quand la hiérarchie d'héritage est prise en compte. On peut insérer une instance de `GeneralizableElement` au milieu d'une relation de généralisation, entre deux éléments parents ; l'élément inséré ne doit pas introduire de comportement, et surtout être du même type que les deux autres — des `Class` dans la plupart des cas mais ce ne sont pas les seuls `GeneralizableElement` concrets apparaissant dans un modèle. La suppression d'une instance de `GeneralizableElement` a l'effet contraire : on supprime un élément inutile pour relier ses sous-classes directement à ses super-classes ; l'élément ne doit alors être référencé dans d'autres diagrammes, ni directement ni indirectement — par le biais d'instances, de membres etc.

Généralisation et spécialisation

Le refactoring de généralisation peut être appliqué aux éléments possédés par les classes, comme les attributs, les `AssociationEnd`, les méthodes, les opérations, ou même les diagrammes d'états. Il consiste à la fusion de plusieurs éléments en un seul qui est ensuite transféré vers une super-classe commune. Puisque les membres privés ne sont pas accessibles depuis les sous-classes, on ne peut pas les déplacer ainsi. Cette transformation implique que toutes les sous-classes directes de la super-classe partagent un élément équivalent ; pour les attributs, les `AssociationEnd` ou les opérations cette équivalence peut être vérifiée structurellement, mais le problème est plus difficile pour les méthodes et les statecharts. Le corps de la transformation dans le cas des opérations est présenté à l'extrait 5.3.

Le refactoring de spécialisation est l'inverse du précédent : il envoie un élément d'une classe dans toutes ses sous-classes. Informellement, il préserve le comportement si la classe d'origine n'est pas le *contexte de référence* de l'élément, c.-à-d. si l'élément n'est utilisé (p.ex. par des contraintes) que via des instances de sous-classes de la classe d'origine. Ce contexte de référence peut être obtenu dans les diagrammes d'objets, ou si le corps des méthodes est disponible, par analyse des actions de lecture et d'écriture des attributs. On peut obtenir le contexte de référence des `AssociationEnd` grâce aux diagrammes d'objets et de collaborations, et celui des méthodes grâce aux diagrammes d'interaction et d'états.

D'autres problèmes peuvent survenir si on néglige l'existence de l'héritage multiple en UML. Il faut vérifier que les classes qui vont recevoir l'élément transféré n'ont pas de sous-classe commune, c'est-à-dire que le schéma d'héritage traditionnellement problématique en diamant ne se présente pas ; si

EXT. 5.3 – Généralisation d'une opération.

```

library Generalizations;
model m : RepositoryModel;

generalizeOperations(genOp : m::Operation, destClass : m::Class)
5 {
  subclassIt : Standard::Iterator;
  subclass : m::Class;
  opIt : Standard::Iterator;
  op : m::Operation;

10
  subclassIt := destClass.specialization.getNewIterator(); subclassIt.start();
  while subclassIt.isOn() {
    subclass := subclassIt.item().child;
    opIt := subclass.feature.getNewIterator(); opIt.start();
15
    while opIt.isOn() {
      op := opIt.item();
      if op.name.[=](genOp.name) {
        dissociate( feature := op : m::Feature,
20
                    owner := subclass : m::Classifier );
        op.delete();
        opIt.next();
      }
      subclassIt.next();
    }
25
  }
  associate( feature := genOp : m::Feature,
            owner := destClass : m::Classifier );
}

```

il existe une sous-classe commune, après la transformation elle hériterait de deux membres équivalents, ce qui serait une erreur conceptuelle.

5.3.4 Refactoring des diagrammes d'états

Les diagrammes d'états sont un sujet intéressant du point de vue du refactoring car ils explicitent le comportement des classes et des méthodes. La plupart des difficultés qu'on rencontre pour définir des refactorings de diagrammes d'états provient de l'activation de comportement attaché aux états, telles que les actions *do*, *entry* ou *exit*. La première est exécutée quand son état est actif. L'action *entry* est exécutée lorsqu'un état est activé; dans le cas particulier d'un état composite, son action d'entrée est exécutée avant les actions d'entrée de ses sous-états. L'action *exit* est exécutée lorsqu'un état devient inactif; dans le cas d'un état composite, son action de sortie est exécutée après les actions de sortie de ses sous-états.

États simples

Fold Incoming/Outgoing Actions Ces transformations remplacent l'ensemble d'actions attachées aux transitions entrantes ou sortantes d'un état, par une action d'entrée ou de sortie attachée à cet état. Cela implique que les actions attachées à toutes les transitions entrantes ou sortantes soient équivalentes. De plus, les états source et cible de chaque transition doivent avoir le même conteneur : autrement dit elles ne doivent pas traverser la frontière d'un état composite, car celui-ci pourrait déclencher une action d'entrée ou de sortie. Essentiellement, deux actions sont équivalentes si elles appellent la même opération,instancient la même classe, envoient le même signal, etc. Dans le cas de séquences d'actions, elles sont équivalentes si elles sont composées d'actions de base équivalentes. Finalement, l'état concerné ne doit pas avoir d'action en entrée ou en sortie. Les pré- et post-conditions pour Fold Incoming Actions sont présentées ci-dessous ; celles de Fold Outgoing Actions s'en déduisent trivialement et ne seront donc pas détaillées.

```
context State::foldIncomingActions
pre:
  self.entry -> isEmpty() and
  self.incoming.effect -> forAll(a,b:Action |
    a.isEquivalentTo(b)) and
  self.incoming.source -> forAll(s:State |
    s.container = self.container)
post:
  self.entry -> notEmpty() and
  self.incoming.effect -> isEmpty() and
  self.incoming.effect@pre -> forAll(a:Action |
    a.isEquivalentTo(self.entry))
```

Unfold Entry/Exit Action Ces transformations sont symétriques aux précédentes. Elles remplacent une action d'entrée ou de sortie attachée à un état par l'ensemble d'actions correspondantes attachées aux transitions entrantes ou sortantes de l'état. Les transitions concernées ne doivent pas avoir d'actions attachées et ne doivent pas traverser la frontière d'un état composite.

Ces transformations, ainsi que celles présentées avant, pourraient être appliquées à des transitions entre états ayant différents conteneurs. Dans ces cas, tous les états composites qu'une transition traverse ne doivent avoir d'action de sortie si la transition en sort, ou d'action d'entrée si la transition y entre.

Group States Ici on regroupe des états dans un nouveau composite. La transformation s'applique à une collection non vide d'états, appartenant tous au même *container*. On lui passe en paramètre le nom de l'état composite à créer. Le *container* est toujours un état composite : les règles de bonne formation de la spécification des diagrammes d'états UML précise que la racine d'une machine à états est toujours un état composite unique.

Une fois la transformation effectuée, la machine à états contient un nouvel état composite qui englobe tous les états de la collection passée en paramètre. Ce nouvel état n'a aucune transition directe, qu'elle soit entrante, sortante ou interne, ni aucune action d'entrée, de sortie, ou d'activation (*do*). Il s'agit du refactoring utilisé à la première étape dans l'exemple de la figure 5.8.

États composites

Fold Outgoing Transitions Cette transformation remplace l'ensemble de transitions quittant chacune un sous-état d'un état composite, et arrivant sur le même état destination passé en paramètre, par une seule transition allant du composite à la destination, comme nous faisons à la troisième étape en figure 5.8. Les actions attachées aux transitions seront fusionnées et doivent donc être équivalentes.

Unfold Outgoing Transitions On remplace une transition quittant un état composite par un ensemble de transitions partant de chacun de ses sous-états, et arrivant à la même destination que la transition originale. Toutes ces transitions doivent avoir des action et événements équivalents. L'ordre des actions d'entrée, de sortie et de transition reste inchangé.

Move State into Composite L'insertion d'un état dans un composite est une transformation plutôt complexe ; plusieurs contraintes doivent être vérifiées avant et après exécution. Puisque la transformation ne doit pas ajouter de nouvelles transitions à l'état inséré, ce dernier doit avoir une transition équivalente à chaque transition quittant le composite. Les transitions entrantes du composite sont toutes liées directement ou implicitement à un sous-état déjà présent et n'affecteront donc pas l'état inséré. Si le composite a une action *do*, alors l'état inséré doit en avoir une équivalente, qui sera supprimée par la transformation.

Si le composite a une action d'entrée, les transitions de l'état inséré vers les sous-états du composite ne doivent pas avoir d'action, car après transformation ces transitions recevront une copie de l'action d'entrée du composite.

Indépendamment, si une transition vers l'état inséré provient de l'extérieur du composite, ce dernier ne peut pas avoir d'action d'entrée.

Si la destination d'une transition provenant de l'état inséré n'est pas un sous-état du composite, ce dernier ne doit pas avoir d'action de sortie. Si le composite a une action d'entrée, alors les transitions depuis ses sous-états vers l'état inséré ne doivent pas avoir d'action, car elles recevront une copie de l'action de sortie du composite.

Move State out of Composite Extraire un sous-état de son composite englobant est aussi une tâche complexe qui mérite clarification. Ce refactoring a été utilisé pour la situation simple de l'exemple en figure 5.8. Le sous-état peut avoir des transitions internes ou externes, c'est-à-dire avec des états qui sont à respectivement l'intérieur ou à l'extérieur du composite.

Les transitions internes posent problème quand le composite a des actions d'entrée ou de sortie, car ces actions ne sont pas déclenchées par ce type de transitions, mais le seront une fois le sous-état extrait du composite. Dans ce cas, l'extraction ne peut se faire que si les transitions internes portent une action équivalente à l'action correspondante du composite : l'action d'entrée pour les transitions quittant l'état extrait, et l'action de sortie pour les transitions arrivant sur l'état extrait. Après transformation, les actions attachées à ces transitions seront donc supprimées pour ne pas faire double emploi avec l'action du composite.

La présence d'actions d'entrée et de sortie pose aussi problème pour les transitions externes qui déclenchent ces actions en traversant la frontière du composite ; il faut que ces actions soient toujours déclenchées après la transformation. La solution consiste donc pour chacune des transitions externes à attacher une action équivalente à l'action d'entrée ou de sortie du composite, pour les transitions respectivement entrantes et sortantes de l'état extrait.

Si le sous-état est lié au pseudo-état initial du composite cela signifie qu'il est en fait destination de toutes les transitions entrantes du composite ; après transformation, on doit donc transférer ces transitions à l'état extrait.

5.3.5 État de l'art rétrospectif

Depuis sa publication [115], cette contribution sur les refactorings de modèles a donné lieu à d'autres travaux dans la communauté, que nous récapitulons ici.

Tout d'abord, Correa et Werner [36] étendent [115] par de nouveaux refactorings pour OCL et pour les modèles UML précisés par des contraintes OCL.

Le métamodèle UML n'est pas adapté pour maintenir la correspondance modèle-code lorsque l'une de ces représentations est modifiée : en particulier le corps des méthodes est considéré comme étant spécifique à chaque implémentation ; or certains refactorings, comme par exemple Extract Method, nécessitent justement d'identifier une portion précise du code d'une méthode. Pieter Van Gorp *et al.* [56] proposent donc une extension du métamodèle UML plus adaptée aux besoins du refactoring. En particulier, cette extension représente les actions à un plus haut niveau d'abstraction que UML 1.5, qui intègre Action Semantics. Ces modifications à UML sont inspirées de travaux précédents sur la formalisation des refactorings par des techniques de réécriture de graphes [78] ; l'article concerne le refactoring de code source mais la structure des graphes manipulés se rapproche d'un métamodèle tel qu'UML.

La préservation de la consistance et propriétés de comportement est un problème important pour le refactoring. Engels *et al.* [61], abordent ce problème pour des systèmes de composants communicants dans le cadre de UML-RT. Van Der Straeten *et al.* [122, 123] introduisent des propriétés de consistance du comportement hérité entre diagrammes de classes et diagrammes de séquences UML 2.0. Cette consistance leur permet de donner une définition de la préservation de comportement d'un refactoring.

Whittle [127] introduit un système à règles pour la transformation d'une version simplifiée des diagrammes de classes UML. Il propose une méthode de vérification des refactorings de modèles par identification des différences entre modèle source et résultat. Les différences et unions de modèles sont également abordées par Alanen et Porres [5] ; ils identifient sept opérations de base permettant d'exprimer la différence — ou *delta* — entre deux modèles, puis proposent les algorithmes d'extraction et de fusion de deltas. Ces algorithmes sont indépendants d'un métamodèle, et constituent donc la base d'un système de contrôle de versions et de résolution de conflits pour l'ingénierie dirigée par les modèles. Porres [98] propose par ailleurs un outil de transformation de modèles basé sur le langage Python⁷.

7. <http://www.python.org>

Quatrième partie

Conclusion

Chapitre 6

Conclusion

Au cours de cette thèse, nous nous sommes intéressés au problème de l'adaptation des grands systèmes informatiques. Cette adaptation est nécessaire car ces systèmes sont subissent deux contraintes opposées :

- une contrainte de *stabilité*, car ils sont voués à une grande durée d'exploitation, pendant laquelle le domaine métier évolue assez peu en regard de la technologie ;
- une contrainte de *flexibilité*, car ils devront répondre dans une dimension fonctionnelle à l'apparition de besoins nouveaux et dans une dimension technologique à l'évolution rapide des plates-formes sur lesquelles ils reposent ou avec lesquelles ils doivent s'interfacer.

Pour gérer la complexité de ces systèmes on a recours à la modélisation, qui permet de masquer la complexité d'un système en travaillant au niveau d'abstraction pertinent.

Les contributions majeures de cette thèse s'inscrivent donc dans le cadre particulier des transformations de modèles, car elles sont au cœur de l'outillage pour l'ingénierie des modèles.

6.1 Cycle de vie des transformations

Dans le chapitre 3, nous nous sommes d'abord intéressés au cycle de vie d'une transformation de modèles. Nous avons montré que ces transformations capturent l'expertise du domaine métier, et évoluent donc parallèlement à ce dernier.

Nous avons proposé une approche réflexive du développement des transformations de modèles, qui facilite l'adoption des méthodes largement reconnues par l'industrie pour l'analyse et la conception orientées objet. Cette approche pragmatique permet d'envisager que la modélisation, le développement et la maintenance des transformations les plus complexes réutilise l'outillage de

manipulation de modèles déjà exploité pour développer les systèmes informatiques dédiés au domaine métier.

6.2 Architecture de manipulation de modèles

L'approche précédente nécessite un outil de transformation de modèles flexible et intégrable simplement dans le processus de développement habituel. Nous proposons une architecture de transformation de modèles qui se découpe en trois parties principales :

- Des *dépositaires* jouant le rôle de conteneurs de modèles. Ce rôle est tout indiqué pour les CASE tool disponibles, car ils fournissent en plus l'interface d'édition des modèles.
- Un *langage* pour exprimer les transformations. Nous avons choisi pour cela de développer un langage impératif orienté objet, et disposant de primitives spécialisées pour la manipulation de modèles. Ce choix est justifié par la possibilité avec un langage généraliste de supporter d'autres paradigmes de transformation plus spécialisés mais moins flexibles.
- Une *interface* qui doit être implémentée par chaque dépositaire. À travers cette interface, l'interprète du langage de transformation peut découvrir la structure du métamodèle spécifique au dépositaire et étendre l'ensemble des classes disponibles dynamiquement.

Cette architecture, décrite au chapitre 3, a pour avantage d'être suffisamment générique et flexible pour supporter différents paradigmes de transformation de modèles, et en particulier l'application réflexive de l'approche MDE au développement de transformations de modèles. L'interface d'accès aux dépositaires est le point central de cette architecture ; son fonctionnement est spécifié par l'implémentation de référence décrit au chapitre 4. Cette interface relie le langage aux dépositaires ; ceux-ci sont implémentés par les outils CASE existants pour divers métamodèles, via un *driver* implémentant l'interface. Finalement, le langage MTL est impératif et orienté objet, ce qui rend possible l'implémentation d'un système à règles de transformation, par exemple.

6.3 Application

Finalement, nous avons appliqué dans le chapitre 5 les transformations de modèles à deux techniques de génie logiciel, l'application de design patterns et les refactorings. Les design patterns peuvent être placés dans un diagramme avec la notation des collaborations paramétrées, puis rendues

explicités de manière automatique lors du passage à l'étape de conception détaillée. Quant aux refactorings, ils sont intrinsèquement des transformations de programme ou de modèles, et comme ils nécessitent des vérifications rhédibitoires à faire manuellement, leur automatisation est particulièrement souhaitable. Ces exemples montrent que dès l'étape de conception et à un niveau d'abstraction élevé on peut automatiser certaines tâches du développement.

Chapitre 7

Bilan et perspectives

7.1 Valorisation

Tout d'abord, une implémentation fonctionnelle et *open source* du langage MTL, basé sur ces travaux, est disponible [87]. Le compilateur MTL s'intègre à l'environnement de développement Eclipse¹ sous la forme d'un plugin, et s'appuie sur des outils tiers tels que MetaData Repository ou Eclipse Modeling Framework (MDR & EMF²) comme modules dépositaires de modèles.

À travers MTL, les travaux présentés dans cette thèse ont également été valorisés en collaboration avec des industriels, et en particulier dans le cadre du projet de recherche MOTOR au sein de la convention CARROLL³ entre trois partenaires⁴ : Thales Research & Technology (TRT), le Commissariat à l'Énergie Atomique (CEA) et l'Institut National de Recherche en Informatique et en Automatique (INRIA). Ce projet a eu pour but de participer aux activités de standardisation l'OMG pour le développement dirigé par les modèles (QVT), et d'étudier la complémentarité de différentes approches de la transformation de modèles. Dans ce cadre, MTL se positionne en tant qu'infrastructure permettant de mettre en œuvre et de combiner des approches par règles ou impératives, en prenant en charge l'interopérabilité avec les dépositaires de modèles et en fournissant un environnement de programmation généraliste orienté objet.

D'autre part, le projet régional Amadeus⁵ fédère plusieurs établissements d'enseignement et de recherche de Bretagne autour de la modélisation objet et de la transformation de modèles. L'objectif de ce projet est d'étudier comment des techniques de génie logiciel telles que la programmation par contrat,

1. <http://www.eclipse.org>

2. <http://mdr.netbeans.org>, <http://www.eclipse.org/emf>

3. <http://www.carroll-research.org>

4. <http://www.thalesgroup.com>, <http://www-drt cea.fr>, <http://www.inria.fr>

5. <http://enstb.org:9673/amadeus>

la vérification ou les méthodologies de développement peuvent être adaptées au cadre du développement dirigé par les modèles.

7.2 Limitations et Perspectives

7.2.1 Support aux transformations de haut niveau

L'architecture proposée ici rend possible l'écriture de programmes de manipulations de modèles, indépendamment des métamodèles, et surtout de manière portable vis-à-vis des dépositaires de modèles. Mais l'approche proposée est somme toute assez bas niveau ; d'un côté c'est un avantage parce que l'architecture est flexible et qu'on peut l'adopter comme une plate-forme commune pour utiliser différents paradigmes de transformation. C'est aussi un gros inconvénient parce que l'outillage pour ces paradigmes n'est pas disponible actuellement dans notre approche : par exemple le parcours du modèle doit être fait explicitement, alors que des systèmes comme VIATRA [124] prennent cela en charge naturellement car ils utilisent une approche de programmation logique. Pour ces différents paradigmes de transformation, des bibliothèques de transformations configurables et réutilisables, ainsi qu'un framework permettant d'assembler et de combiner ces transformations seraient donc très utiles [125].

7.2.2 Test et validation de transformations

Notre approche considère les transformations de modèles comme des produits logiciels qui doivent être conçus et développés avec des techniques de génie logiciel pertinentes. En particulier, on peut s'assurer du bon fonctionnement d'une transformation en soumettant celle-ci à une batterie de tests. Cela requiert cependant qu'on sache dire si un jeu de tests est d'une qualité suffisante, par exemple en termes de couverture ou de robustesse.

Les travaux existants [17, 18, 50] sur le test de logiciel orienté objet et l'optimisation de cas de test par analyse de mutation pourraient être adaptés au test de transformations de modèles. Un tel outil de test automatique repose sur une procédure de génération de données de test et sur un oracle. Le générateur doit produire des données pour le système sous test. Cet oracle va décider si le système passe ou non un test, en fonction des données fournies au système et de la réponse de celui-ci. Pour l'analyse de mutation, on a de plus besoin d'introduire des modifications locales dans les transformations.

7.2.3 Traçabilité

Lors de la maintenance d'un logiciel il est souvent utile de connaître en détail l'ensemble des modifications effectuées d'une version à une autre. Quand on a affaire à du code source, on peut utiliser des outils tels que les commandes Unix `diff` ou `patch`, qui permettent respectivement d'extraire ou d'appliquer les différences entre deux fichiers. La structure de graphe des modèles rend l'extraction de telles différences plus difficiles que sur un simple fichier texte ou un document XML. Pour résoudre ce problème, Alanen et Porres [5] proposent des algorithmes de calcul de différence et d'union de modèles. Cependant, ils soulèvent l'aspect sémantique de ces différences : l'application d'un *delta* structurellement correct à un modèle doit aussi respecter les règles de bonne formation du métamodèle.

On peut aussi considérer la tracabilité comme une approche de la transformation de modèle : plutôt que de modifier ou produire des *modèles résultats*, une transformation pourrait établir un ensemble de différences entre les modèles manipulés. Il importe alors peu que ces différences soit établies par un programme impératif, par des règles de transformations ou par une séquence d'actions de l'utilisateur d'un CASE tool. Les informations de tracabilité peuvent également servir à propager des changements dans des modèles évoluant en parallèle, et donc rendre possible une approche de l'ingénierie des modèles moins linéaire que celle par raffinements successifs du MDA.

La tracabilité est également intéressante au niveau du processus de développement, par exemple pour identifier quels éléments d'un modèle contribuent à un cas d'utilisation donné. Cela laisse de nombreuses questions ouvertes, car il faut concevoir un métamodèle pour représenter les informations de tracabilité : quelles sont les informations pertinentes, comment lier des modèles issus de différents métamodèles, comment exploiter concrètement un modèle de tracabilité ?

Annexes

Bibliographie

- [1] Jean-Raymond Abrial. *The B-book : assigning programs to meanings*. Cambridge University Press, 1996. ISBN 0-521-49619-5.
- [2] Inc. Adobe Systems, editor. *PostScript(R) Language Reference*. Pearson Education, 3rd edition, 1999. ISBN 0201379228.
- [3] Inc. Adobe Systems, editor. *PDF Reference : Version 1.4*. Addison-Wesley, 3rd edition, 2001. ISBN 0201758393.
- [4] AGG. Attributed graph grammar system home page. <http://tfs.cs.tu-berlin.de/agg>.
- [5] Marcus Alanen and Ivan Porres. Difference and union of models. In Stevens et al. [113], pages 2–17. ISBN 3-540-20243-9.
- [6] Hervé Albin-Amiot, Pierre Cointe, and Yann-Gaël Guéhéneuc. Un méta-modèle pour coupler application et détection des design patterns. In Dao and Huchard [39], pages 41–58.
- [7] Omar Aldawud, Grady Booch, Siobhán Clarke, Tzilla Elrad, Bill Harrison, Mohamed Kandi, and Alfred Strohmeier, editors. *Workshop on Aspect-Oriented Modeling with UML (AOSD-2002)*, Enschede, the Netherlands, March 2002. <http://lglwww.epfl.ch/workshops/aosd-uml/papers.html>.
- [8] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, New York, 1979.
- [9] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, and Shlomo Angel. *The Oregon Experiment*. Oxford University Press, New York, 1975.
- [10] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, New York, 1977.

- [11] *The Objective-C Programming Language*. Apple Computer, Inc., February 2004. <http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/>.
- [12] AS. Action semantics for the UML – response to the OMG RFP ad/98-11-01. OMG Document, August 2000.
- [13] Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors. *UML 2004 – The Unified Modeling Language. Modeling Languages and Applications*, volume 3273 of LNCS, Lisbon, October 2004. Springer Verlag. ISBN 3-540-23307-5.
- [14] Greg J. Badros. JavaML : A markup language for java source code. In Herman and Vezza [62]. ISBN 1-930792-00-X. <http://www.cs.washington.edu/homes/gjb/JavaML/>.
- [15] M. Balazinska, E. Merlo, M. Daguenaïs, B. Lagüe, and Kostas Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proc. 7th Working Conf. Reverse Engineering*, pages 98–107. IEEE Press, November 2000.
- [16] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *Proc. 1987 ACM SIGMOD int'l conf. Management of data*, pages 311–322, San Francisco, California, 1987. ACM Press.
- [17] Benoît Baudry. *Assemblage testable et validation de composants*. PhD thesis, Université de Rennes 1, 2003.
- [18] Benoît Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. From genetic to bacteriological algorithms for mutation-based testing. *Software, Testing, Verification & Reliability journal (STVR)*, November 2004.
- [19] Kent Beck. *Extreme Programming Explained : Embracing Change*. Addison-Wesley, 1999. ISBN 0-201-61641-6. <http://www.extremeprogramming.org>. — Une méthode de développement agile faisant un usage important du refactoring.
- [20] Paul L. Bergstein. Maintenance of object-oriented systems during structural schema evolution. *Theory and Practice of Object Systems*, 3 (3) : 185–212, 1997.
- [21] Gilles Blain, Nicolas Revault, Houari A. Sahraoui, and Jean-François Perrot. A metamodeling technique. In D. de Champeaux, H. Kaindl, J. Laubsch, and A. Shappert, editors, *OOPSLA '94 Workshop on 'AI and OO Software Engineering'*, Portland, OR,

- October 1994. <http://www-poleia.lip6.fr/~revault/papers/OOPSLA94-AIOOSE-BRSP.htm>.
- [22] Grady Booch. *Object-oriented Analysis and Design with Applications*. Pearson Education, 2nd edition, September 1993. ISBN 0-8053-5340-2.
- [23] Isabelle Borne and Nicolas Revault. Comparaison d'outils de mise en œuvre de design patterns. *L'Objet*, 5(2) : 243–266, 1999.
- [24] John Brant and Don Roberts. Refactoring techniques and tools (plenary talk). In *Smalltalk Solutions*, New York, NY, 1999.
- [25] Frederick P. Brooks. *The Mythical Man-Month : Essays on Software Engineering*. Addison-Wesley, 1982.
- [26] William J. Brown, Raphael C. Malveau, William H. Brown, Hays W. McCormick III, and Thomas J. Mowbray. *AntiPatterns : Refactoring Software, Architecture, and Projects in Crisis*. John Wiley & Sons, Inc., March 1998. ISBN 0-471-19713-0.
- [27] Jean Bézivin, Nicolas Farcet, Jean-Marc Jézéquel, Benoît Langlois, and Damien Pollet. Reflective model driven engineering. In Stevens et al. [113], pages 175–189. ISBN 3-540-20243-9. <http://www.irisa.fr/triskell/publis/2003/Bezivin03.pdf>.
- [28] Eduardo Casais. *Managing Evolution in Object Oriented Environments : An Algorithmic Approach*. PhD thesis, University of Geneva, 1991.
- [29] CCM. CORBA Component Model, version 3.0. OMG Document formal/2002-06-65, June 2002. <http://www.omg.org/technology/documents/formal/components.htm>.
- [30] Siobhán Clarke and Robert J. Walker. Towards a standard design language for AOSD. In Ossher and Kiczales [92]. ISBN 1-58113-469-X.
- [31] Cocoa. Cocoa frameworks home page. <http://developer.apple.com/cocoa/>.
- [32] COM. Component Object Model. <http://www.microsoft.com/com/>.
- [33] James O. Coplien. *Software Patterns*. SIGS Management Briefings. SIGS Books & Multimedia, 1996. — Une très bonne introduction aux design patterns.
- [34] James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design*. Addison-Wesley, Reading, MA, 1995. ISBN 0-201-6073-4.

- [35] Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Proceedings of the First International Conference on Graph Transformation (ICGT)*, volume 2505 of LNCS, Barcelona, Spain, October 2002. Springer Verlag. ISBN 3-540-44310-X.
- [36] Alexandre Correa and Cláudia Werner. Applying refactoring techniques to UML/OCL models. In Baar et al. [13], pages 173–187. ISBN 3-540-23307-5.
- [37] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *OOPSLA '03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.
- [38] Ole-Johan Dahl, Bjørn Myhrhaug, and Krysten Nygaard. Simula 67 common base language. Technical report, Norwegian Computing Centre, Oslo, 1970.
- [39] Michel Dao and Marianne Huchard, editors. *Langages et Modèles à Objets (LMO '02)*, volume 8 of *L'Objet*, August 2002. Éditions Hermes Science.
- [40] Arie van Deursen and Paul Klint. Little languages, little maintenance? *Journal of Software Maintenance*, 10 : 75–93, 1998.
- [41] Arie van Deursen and Leon Moonen. The video store revisited : Thoughts on refactoring and testing, May 2002.
- [42] Peter Drayton, Ben Albahari, and Ted Neward. *C# in a Nutshell*. O'Reilly & Associates, 2nd edition, August 2003. ISBN 0-596-00526-1.
- [43] Desmond D'Souza and Alan Wills. *Objects, Components and Frameworks With UML : The Catalysis Approach*. Addison-Wesley, 1998. ISBN 0201310120.
- [44] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Yang and White [130], pages 109–118.
- [45] Eclipse. Eclipse development platform home page. <http://eclipse.org/>.
- [46] Amnon H. Eden. *Precise Specification of Design Patterns and Tool Support in Their Application*. PhD thesis, University of Tel Aviv, 1999.
- [47] Amnon H. Eden, Amiram Yehudai, and Joseph Gil. Precise specification and automatic application of design patterns. In *Proc. 12th Int'l Automated Software Engineering Conf. (ASE'97)*, pages 143–152, November 1997.

- [48] Niels van Eetvelde and Dirk Janssens. A hierarchical program representation for refactoring. In *Proc. UniGra'03 workshop*, Electronic Notes in Theoretical Computer Science. Elsevier, Inc., 2003.
- [49] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proc. 9th Working Conference on Reverse Engineering*, pages 97–108. IEEE Press, October 2002.
- [50] Franck Fleurey, Jim Steel, and Benoit Baudry. Validation in model-driven engineering : Testing model transformation. In *Proc. of the SIVOES-Modeva workshop, SIVOES (Specification Implementation and Validation Of Embedded Systems)-MoDeVa (Model Design and Validation)*, Rennes, November 2004.
- [51] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring : improving the design of existing code*. Object Technology. Addison-Wesley, 1999. ISBN 0-201-48567-2. <http://www.refactoring.com>. — Un catalogue de refactorings au niveau langage, avec des exemples en Java.
- [52] Fujaba. Fujaba tool suite home page. <http://www.fujaba.de>.
- [53] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. — L'ouvrage de référence définissant les 23 design patterns du « GoF » (Gang of Four, surnom des auteurs).
- [54] Robert L. Glass. Maintenance : Less is not more. *IEEE Software*, 15 (4) : 67–68, 1998.
- [55] Adele Goldberg and Dave Robson. *Smalltalk-80 : The Language*. Addison-Wesley, 1989. ISBN 0-201-13688-0.
- [56] Pieter van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards automating source-consistent UML refactorings. In Stevens et al. [113], pages 144–158. ISBN 3-540-20243-9.
- [57] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1996. ISBN 0-201-63451-1. <http://java.sun.com/docs/Books/jls/>.
- [58] William G. Griswold. *Program restructuring as an aid to software maintenance*. PhD thesis, Université de Washington, August 1991.
- [59] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2) : 120–131, February 1996.

- [60] David Harel. Statecharts : a visual formalism for complex systems. *Science of computer programming*, 8(3) : 231–274, June 1987. ISSN 0167-6423.
- [61] Gregor Hengels, Reiko Heckel, Jochen M. Küster, and Luuk Goenewegen. Consistency-preserving model evolution through transformations. In Jézéquel et al. [66], pages 212–226. ISBN 3-540-44254-5.
- [62] Ivan Herman and Albert Vezza, editors. *Ninth International World Wide Web conference*, Amsterdam, May 2000. ISBN 1-930792-00-X. <http://www9.org/>.
- [63] Walter L. Hürsch. *Maintaining consistency and behavior of object-oriented systems during evolution*. PhD thesis, Université de Northeastern, Boston, August 1995.
- [64] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering – A use case driven approach*. Addison-Wesley, 1992. — Référence de la méthode OOSE.
- [65] Jean-Marc Jézéquel. *Object-oriented software engineering with Eiffel*. Eiffel in practice. Addison-Wesley, 1996.
- [66] Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors. *UML 2002 – The Unified Modeling Language. Model Engineering, Concepts, and Tools*, volume 2460 of LNCS, Dresden, October 2002. Springer Verlag. ISBN 3-540-44254-5.
- [67] Jean-Marc Jézéquel, Michel Train, and Christine Mingins. *Design patterns and contracts*. Addison-Wesley, 1999.
- [68] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, August 2004. ISBN 0-3212-1335-1.
- [69] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeff Palm, and William G. Griswold. An overview of aspectj. In Jørgen Lindskov Knudsen, editor, *Proc. ECOOP'2001*, volume 2072 of LNCS, pages 327–353. Springer Verlag, 2001.
- [70] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proc. ECOOP'97*, volume 1241 of LNCS, pages 220–242. Springer Verlag, 1997. <http://www.parc.com/research/csl/projects/aspectj/>.

- [71] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, and Jeff Palm. *The AspectJ Programming Guide*. The AspectJ Team, Xerox Corporation, Palo Alto Research Center, 2001. <http://eclipse.org/aspectj/>.
- [72] Donald E. Knuth. *The T_EXbook*. Addison-Wesley, 1984.
- [73] Michele Lanza and Stéphane Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In Dao and Huchard [39], pages 135–149.
- [74] Alain Le Guennec, Gerson Sunyé, and Jean-Marc Jézéquel. Precise modeling of design patterns. In *Proceedings of UML 2000*, volume 1939 of LNCS, pages 482–496. Springer Verlag, 2000. <http://www.irisa.fr/triskell/publis/2000/LeGuennec00a.pdf>.
- [75] Angeles Manjarrés, Simon Pickin, Gerson Sunyé, Damien Pollet, and Jean-Marc Jézéquel. OO analysis patterns as UML metalevel collaborations. In *Proceedings of the 22nd SGAI International Conference on Knowledge Based Systems and Applied Artificial Intelligence : Research and Development in Intelligent Systems XIX (ES2002)*, BCS Conference Series. Springer Verlag, December 2002.
- [76] Angeles Manjarrés, Gerson Sunyé, Damien Pollet, Simon Pickin, and Jean-Marc Jézéquel. AI analysis patterns as UML meta-model constructs. In ACM Press, editor, *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE 2002)*, pages 237–238, July 2002. <http://www.irisa.fr/triskell/publis/2002/Manjarres02a.pdf>.
- [77] David Mapelsden, John Hosking, and John Grundy. Design pattern modelling and instantiation using DPML. In James Noble and John Potter, editors, *Proceedings of TOOLS 2002*, volume 10, Sydney, 2002.
- [78] Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising behaviour preserving program transformations. In Corradini et al. [35], pages 286–301. ISBN 3-540-44310-X.
- [79] Tom Mens and Tom Tourwé. A declarative evolution framework for object-oriented design patterns. In *Proc. Int'l Conf. Software Maintenance*, pages 570–579. IEEE Press, 2001.
- [80] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2), February 2004.

- [81] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. Technical Report SEN-E0309, Centrum voor Wiskunde en Informatica, Amsterdam (NL), December 2003.
- [82] Bertrand Meyer. *Eiffel : The Language*. Prentice Hall, 1992.
- [83] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.
- [84] MOF. Meta-Object Facility specification, version 1.4. OMG Document formal/2002-04-03, April 2002. <http://www.omg.org/technology/documents/formal/mof.htm>.
- [85] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly & Associates, 2nd edition, 2000.
- [86] John D. Morgenthaler. *Static analysis for a software transformation tool*. PhD thesis, Université de Californie, San Diego, 1997.
- [87] MTL. MTL project home page. <http://modelware.inria.fr>.
- [88] NetBeans. Netbeans IDE home page. <http://www.netbeans.org/>.
- [89] Mél Ó Cinnéide. *Automated Application of Design Patterns : A Refactoring Approach*. PhD thesis, University of Dublin, Trinity College, October 2000.
- [90] Mél Ó Cinnéide and Paddy Nixon. A methodology for the automated introduction of design patterns. In Yang and White [130], pages 463–472.
- [91] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, 1992. <ftp://st.cs.uiuc.edu/pub/papers/refactoring/>.
- [92] Harold Ossher and Gregor Kiczales, editors. *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, April 2002. ACM Press. ISBN 1-58113-469-X.
- [93] Renaud Pawlak. *La programmation orientée aspect interactionnelle pour la construction d'applications à préoccupations multiples*. PhD thesis, CNAM, laboratoire Cedric, 2002.
- [94] Renaud Pawlak, Laurence Duchien, Gerard Florin, Fabrice Legond-Aubry, Lionel Seinturier, and Laurent Martelli. A UML notation for aspect-oriented software design. In Aldawud et al. [7]. <http://lglwww.epfl.ch/workshops/aosd-uml/Allsubs/pawlak.pdf>.

- [95] Mikaël Peltier. *Techniques de transformation de modèles basées sur la métamodélisation*. PhD thesis, Université de Nantes, 2003.
- [96] Jens Uwe Pipka. Refactoring in a “test first”-world. In XP02 XP02 [129].
- [97] Damien Pollet, Didier Vojtisek, and Jean-Marc Jézéquel. OCL as a core UML transformation language. Workshop on Integration and Transformation of UML models (WITUML 2002), June 2002. <http://ctp.di.fct.unl.pt/~ja/wituml02.htm>.
- [98] Ivan Porres. Model refactorings as rule-based update transformations. In Stevens et al. [113], pages 159–174. ISBN 3-540-20243-9.
- [99] Maurizio Proietti and Alberto Pettorossi. Semantics preserving transformation rules for prolog. In *Proc. ACM Symp. Partial evaluation and semantics-based program evaluation (PEPM'91)*, volume 26, pages 274–284, May 1991.
- [100] QVT. MOF 2.0 Query / Views / Transformations RFP. OMG Document ad/2002-04-10, April 2002.
- [101] Trygve Reenskaug. *Working with Objects : the OORAM Software Engineering Method*. Manning Publications, 1996. ISBN 0-13-452930-8.
- [102] Nicolas Revault. *Principes de métamodélisation pour l'utilisation de canevas d'applications à objets*. PhD thesis, Université de Paris 6, 1996.
- [103] Donald B. Roberts. *Practical analysis for refactoring*. PhD thesis, université de l'Illinois, Urbana-Champaign, 1999.
- [104] Donald B. Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Theory and practice of object systems*, 3(4) : 253–263, 1997.
- [105] Jeff Rothenberg. The nature of modeling. In Lawrence E. Widman, Kenneth A. Loparo, and Norman R. Nelson, editors, *Artificial Intelligence, Simulation, and Modeling*, pages 75–92. John Wiley & Sons, Inc., New York, 1989. ISBN 0-471-60599-9.
- [106] Grzegorz Rozenberg, editor. *Handbook on Graph Grammars : Applications*, volume 2, 1999. World Scientific.
- [107] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object Oriented Modeling and Design*. Prentice Hall, 1991. — Référence de la méthode OMT.

- [108] Douglas C. Schmidt. Reactor : an object behavioral pattern for demultiplexing and dispatching handles for synchronous events. In Coplien and Schmidt [34]. ISBN 0-201-6073-4. <http://www.cs.wustl.edu/~schmidt/PDF/reactor-siemens.pdf>.
- [109] Andy Schürr, A. Winter, and A. Zündorf. Progres : Language and environment. In Rozenberg [106], pages 487–550.
- [110] Shane Sendall and Wojtek Kozaczynski. Model transformation – the heart and soul of model-driven software development. *IEEE Software*, 20(5), 2003.
- [111] Frank Simon, Frank Steinbruckner, and Claus Lewerentz. Metrics based refactoring. In *Proc. European Conf. Software Maintenance and Reengineering*, pages 30–38. IEEE Press, 2001.
- [112] SPEM. Software Process Engineering Metamodel specification, version 1.0. OMG Document formal/02-11-14, November 2002. <http://www.omg.org/technology/documents/formal/spem.htm>.
- [113] Perdita Stevens, Jon Whittle, and Grady Booch, editors. *UML 2003 – The Unified Modeling Language. Modeling Languages and Applications*, volume 2863 of LNCS, San Francisco, October 2003. Springer Verlag. ISBN 3-540-20243-9.
- [114] Gerson Sunyé, Alain Le Guennec, and Jean-Marc Jézéquel. Design patterns application in UML. In Elisa Bertino, editor, *Proceedings of ECOOP 2000*, volume 1850 of LNCS, pages 44–62. Springer Verlag, 2000.
- [115] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring UML models. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of LNCS, pages 134–148, Toronto, October 2001. Springer Verlag. ISBN 3-540-42667-1. <http://www.irisa.fr/triskell/publis/2001/Sunye01b.pdf>.
- [116] Theme/UML. Theme/UML. Web site. http://www.dsg.cs.tcd.ie/index.php?category_id=355.
- [117] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings of ISPSE 2000*, pages 157–167. IEEE Press, 2000.
- [118] Lance Aiji Tokuda. *Evolving Object-Oriented Designs with Refactorings*. PhD thesis, University of Texas at Austin, December 1999.

- [119] Lance Aiji Tokuda and Don Batory. Evolving object-oriented designs with refactorings. *Automated Software Engineering*, 8(1) : 89–120, 2001.
- [120] Tom Tourwé and Tom Mens. Identifying refactoring opportunities using logic meta programming. In *Proc. 7th European Conf. Software Maintenance and Reengineering (CSMR '03)*, pages 91–100. IEEE Press, March 2003.
- [121] UML. Unified Modeling Language specification, version 1.5. OMG Document formal/03-03-01, March 2003. <http://www.omg.org/technology/documents/formal/uml.htm>.
- [122] Ragnhild Van Der Straeten, Viviane Jonckers, and Tom Mens. Supporting model refactorings through behavior inheritance consistencies. In Baar et al. [13], pages 305–319. ISBN 3-540-23307-5.
- [123] Ragnhild Van Der Straeten, Joycelyn Mens, Tom Simmonds, and Viviane Jonckers. Using description logic to maintain consistency between UML models. In Stevens et al. [113], pages 326–340. ISBN 3-540-20243-9.
- [124] Dániel Varró. Automatic program generation for and by model transformation systems. In Hans-Jörg Kreowski and Peter Knirsch, editors, *Proc. AGT 2002 : Workshop on Applied Graph Transformation*, pages 161–173, Grenoble, France, April 2002.
- [125] Dániel Varró and András Pataricza. Generic and meta-transformations for modeltransformation engineering. In Baar et al. [13], pages 290–304. ISBN 3-540-23307-5.
- [126] Jos Warmer and Anneke Kleppe. *The Object Constraint Language : precise modeling with UML* . Object Technology. Addison-Wesley, 1998. ISBN 0-201-37940-6.
- [127] Jon Whittle. Transformations and software modeling languages : Automating transformations in UML. In Jézéquel et al. [66], pages 227–242. ISBN 3-540-44254-5.
- [128] Jane Wood and Denise Silver. *Joint Application Development*. John Wiley & Sons, Inc., New York, 2nd edition, 1995.
- [129] XP02. *Proceedings of the 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering (XP2002)*, Alghero, Sardinia, Italy, May 2002.
- [130] Hongji Yang and Lee White, editors. *Proceedings of the International Conference on Software Maintenance (ICSM '99)*, September 1999. IEEE Press.

Publications

- [27] Jean Bézivin, Nicolas Farcet, Jean-Marc Jézéquel, Benoît Langlois, and Damien Pollet. Reflective model driven engineering. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 – The Unified Modeling Language. Modeling Languages and Applications*, volume 2863 of LNCS, pages 175–189, San Francisco, October 2003. Springer Verlag. ISBN 3-540-20243-9. <http://www.irisa.fr/triskell/publis/2003/Bezivin03.pdf>.
- [75] Angeles Manjarrés, Simon Pickin, Gerson Sunyé, Damien Pollet, and Jean-Marc Jézéquel. OO analysis patterns as UML metalevel collaborations. In *Proceedings of the 22nd SGA International Conference on Knowledge Based Systems and Applied Artificial Intelligence : Research and Development in Intelligent Systems XIX (ES2002)*, BCS Conference Series. Springer Verlag, December 2002.
- [76] Angeles Manjarrés, Gerson Sunyé, Damien Pollet, Simon Pickin, and Jean-Marc Jézéquel. AI analysis patterns as UML meta-model constructs. In ACM Press, editor, *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE 2002)*, pages 237–238, July 2002. <http://www.irisa.fr/triskell/publis/2002/Manjarres02a.pdf>.
- [97] Damien Pollet, Didier Vojtisek, and Jean-Marc Jézéquel. OCL as a core UML transformation language. Workshop on Integration and Transformation of UML models (WITUML 2002), June 2002. <http://ctp.di.fct.unl.pt/~ja/wituml02.htm>.
- [115] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. Refactoring UML models. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of LNCS, pages 134–148, Toronto, October 2001. Springer Verlag. ISBN 3-540-42667-1. <http://www.irisa.fr/triskell/publis/2001/Sunye01b.pdf>.

Glossaire

Acronymes

API : Application Programming Interface – Ensemble des points d’entrée d’une librairie de fonctions ou d’objets.

DOM : Document Object Model – Interface de manipulation de documents XML ou HTML.

DSL : Domain-Specific Language – Langage destiné à un domaine d’application précis, et abandonnant la généralité au profit d’autres avantages (simplicité, formalité, optimisations).

DTD : Document Type Definition – Un des langages de définition de la structure d’un document XML.

EBNF : Extended Backus-Naur Form – Langage de description de grammaires.

EJB : Enterprise Java Beans – Technologie à composants Java.

JAC : Java Aspect Components – Outil de programmation par aspects dynamiques en Java.

MDA : Model-Driven Architecture – Architecture centrée sur les modèles promue par l’OMG.

MDE : Model-Driven Engineering

MDR : Meta-Data Repository – Composant logiciel conteneur de modèles et métamodèles.

MOF : Meta-Object Facility – Langage de la famille OMG, pour la description de métamodèles.

MTL : Model Transformation Language

OCL : Object Constraint Language – Langage de navigation et d’expression de contraintes inclus dans UML 1.5.

OMG : Object Management Group

OMT : Object Modeling Technique

OOSE : Object-Oriented Software Engineering

PIM : Platform-Independent Model

PSM : Platform-Specific Model

PIT : Platform-Independent Transformation

PST : Platform-Specific Transformation

QVT : Query / Views / Transformations – RFP publiée par l'OMG pour la conception du MOF 2.0.

RFP : Request For Proposal

SPEM : Software Process Engineering Metamodel

UML : Unified Modeling Language

W3C : World-Wide Web Consortium

XMI : XML Metadata Interchange – Standard pour la génération de structures de documents XML à partir de métamodèles, en vue de sérialiser des modèles.

XML : eXtensible Markup Language

XSL : XML Stylesheet Language – Recommandation du W3C pour la manipulation de documents XML.

XSL-FO : XSL Formatting Objects – Sous-ensemble de XSL dédié à la présentation via différents média de documents XML.

XSLT : XSL Transformations – Sous-ensemble de XSL dédié à la transformation de documents XML. XSLT est lui-même un langage XML.

Abstract

Model engineering attempts to solve how we can evolve complex software systems. Indeed, those systems must follow the evolution of new requirements and technologies, and this evolution is faster and faster compared to the business domain evolution. We thus propose to reuse the domain expertise independantly of any underlying technology, through model transformation techniques.

The contribution presented in this document is an architecture for manipulating models which is independent of any specific metamodel. During development of model transformations, this architecture supports proven techniques of object-oriented software engineering. A reference implementation in functional programming specifies the semantics of the interface for accessing models.

Our approach is based on a MOF-level interface (MOF : Meta-Object Facility) for model manipulation. The associated programming language supports direct manipulation of model elements, because the metamodel structure dynamically extends the set of types available to the model transformation program. From a methodological point of view, we show that model transformations capture the implementation expertise for a business domain to a given technology ; it is therefore useful to model and develop complex transformations using sound software engineering and model engineering techniques. We illustrate this in practice using transformations for design pattern introduction and refactoring in UML models (UML : Unified Modeling Language).

Keywords software engineering, model-driven engineering, metamodels, refactoring, design patterns

Résumé

Avec l'ingénierie des modèles on cherche à résoudre le problème de l'évolution des grands systèmes informatiques. En effet, ces systèmes doivent s'adapter à l'évolution des besoins et des technologies, et cette évolution est de plus en plus rapide par rapport à celle du domaine métier. On souhaite donc réutiliser l'expertise de ce domaine indépendamment des technologies sur lesquelles on s'appuie, grâce à des techniques de manipulation de modèles.

La contribution présentée dans ce manuscrit est une architecture de manipulation de modèles indépendante d'un quelconque métamodèle. Cette architecture favorise la réutilisation des techniques reconnues de génie logiciel orienté objet pour le développement de transformations de modèles. La sémantique de l'interface d'accès aux modèles est spécifiée par une implémentation de référence en langage fonctionnel.

Notre approche est fondée sur une interface de niveau MOF (Meta-Object Facility) pour la manipulation de modèles. Le langage associé permet de manipuler des éléments de modèle directement, car la structure du métamodèle étend dynamiquement l'ensemble des types accessibles au programme de transformation. D'un point de vue méthodologique, on montre que les transformations de modèles synthétisent l'expertise d'implantation d'un domaine métier vers une technologie donnée ; il est donc utile de modéliser les transformations les plus complexes pour les développer en appliquant récursivement les techniques de génie logiciel et d'ingénierie des modèles. La mise en pratique illustre ce point et montre le fonctionnement de l'architecture de manipulation de modèles avec des transformations pour l'introduction de design patterns et la restructuration de modèles UML (Unified Modeling Language).

Mots clés génie logiciel, ingénierie dirigée par les modèles, métamodèles, restructuration, patrons de conception