Université de Montréal

RCR: un profil UML pour la rétroconception, la compréhension et la réingénierie de logiciels

par

Guy St-Denis

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

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

Avril, 2001

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé:

RCR: un profil UML pour
la rétroconception, la compréhension
et la réingénierie de logiciels

présenté par:

Guy St-Denis

a été évalué par un jury composé des personnes suivantes:

Président-rapporteur:	Dr. François Lustman
Directeur de recherche:	Dr. Rudolf K. Keller
Membre du jury:	Dr. Sébastien Roy

Mémoire accepté le:.....

Sommaire

L'adoption du Unified Modeling Language (UML) par la communauté d'analyse, de conception et de développement de logiciels a favorisé le progrès à plusieurs niveaux. L'unicité du langage a multiplié la fréquence et la qualité des échanges entre les intervenants impliqués dans l'élaboration d'un logiciel. De façon similaire, l'unicité du métamodèle a stimulé le développement d'outils diversifiés et compatibles, favorisant ainsi les collaborations et la réutilisation du code et des données.

Cette situation contraste fortement avec celle qui prévaut chez la communauté de maintenance des logiciels. La fragmentation de cette communauté en groupuscules isolés perdure en raison de l'absence d'une norme pour la modélisation des logiciels déjà implantés. L'incompatibilité des outils rend difficile ou impossible la collaboration et la réutilisation.

Devant ce bilan peu réjouissant, plusieurs de la communauté de maintenance ont préconisé un rapprochement, voire même l'unification des outils et des processus de maintenance avec ceux de développement. Vu l'imposante présence des outils et des modèles basés sur UML dans la communauté de développement, il appert que l'intégration des outils de la communauté de maintenance devra passer par l'utilisation de modèles UML.

Un obstacle potentiel à cette intégration provient du fait que le métamodèle UML a été conçu pour la modélisation des logiciels en cours de développement et non pour les logiciels déjà en place. La faiblesse principale d'UML vis-à-vis les logiciels implantés est son incapacité de modéliser explicitement le code source qui définit le corps des méthodes. Or, ces détails sont essentiels pour la poursuite des activités de maintenance telles que la rétroconception, la compréhension et la réingénierie.

Ce mémoire présente le profil RCR, une extension au métamodèle UML qui introduit 52 nouveaux métaéléments permettant la modélisation détaillée de logiciels conçus dans des langages impératifs classiques ou orientés objet. Les entités modélisées par les métaéléments du profil RCR comprennent les blocs, les énoncés, les expressions, les variables et les opérateurs primitifs qui constituent les expressions complexes.

Le profil RCR a été conçu pour être à la fois général, expressif et efficace. La généralité du profil permet la modélisation pour plus d'un langage de programmation. Son expressivité de base capte déjà certaines nuances intéressantes du code, mais le modèle peut être raffiné davantage au besoin. L'efficacité du profil RCR signifie que les modèles créés n'exigent pas de ressources de stockage démesurées et ils facilitent l'accès à l'information pertinente.

En somme, le profil RCR apporte au métamodèle UML standard la richesse de contenu nécessaire pour la réalisation des tâches de rétroconception, de compréhension et de réingénierie de logiciels. Ce faisant, le profil RCR ouvre la porte au rapprochement de la communauté de développement et de la communauté de maintenance de logiciels.

Mots clés: génie logiciel, rétroconception, compréhension, réingénierie, UML, métamodèle, profil

Table des matières

Liste des tableaux

Liste des figures

Liste des sigles et abréviations

AGL	Atelier de génie logiciel
API	Application Programming Interface
CNRC	Conseil national de recherches du Canada
CORBA	Common Object Request Broker Architecture
CRGL	Consortium pour la recherche en génie logiciel
CRSNG	Conseil de recherches en sciences naturelles et en génie du Canada
DTD	Document Type Definition
IDL	Interface Definition Language
MOF	Meta-Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
OMT	Object Modeling Technique
OOSE	Object-Oriented Software Engineering
RCR	Rétroconception Compréhension Réingénierie
SPOOL	Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Remerciements

Ce mémoire représente l'aboutissement
d'une intéressante aventure qui débuta en 1995.
En écrivant ces dernières lignes, je réalise à quel point cet
accomplissement n'eut été possible sans la présence, l'appui,
l'aide et l'amour des personnes suivantes
que je remercie sincèrement.

Mon directeur, Rudolf K. Keller.

Ma copine, Louise.

Ma famille.

Mes amis et amies.

Mes collègues au GÉLO et chez Bell Canada.

Mes professeurs et le personnel du DIRO.

Les musiciens et musiciennes pour la bande sonore inoubliable!

Cette recherche a été réalisée avec l'aide financière du Fonds FCAR
et du Consortium pour la recherche en génie logiciel (CRGL).
Le CRGL est co-financé par
le Conseil national de recherches du Canada (CNRC),
le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG),
et plusieurs compagnies privées dont notre partenaire de recherche, Bell Canada.

Chapitre 1: Introduction

1.1 Motivation

Avant l'arrivée du Unified Modeling Language (UML) [OMG_2000], la communauté d'analyse et de conception orientée objet était fragmentée en clans campés autour des trois méthodologies populaires de l'époque: Booch [Booch_1994], OMT [Rumbaugh_1991] et OOSE [Jacobson_1992]. Chaque tribu possédait son propre dialecte et utilisait ses propres outils. Parce que les outils locaux et les outils étrangers étaient incompatibles, la collaboration et la réutilisation étaient difficiles sinon impossibles. Plutôt que d'être investies dans le développement de techniques et d'outils d'avant-garde, les ressources (humaines, techniques, temporelles, monétaires, etc.) étaient gaspillées à reproduire trois versions d'un même produit ou résultat, à gérer les incompatibilités constantes et pire encore, à tenter de convertir l'ennemi! L'introduction d'UML comme solution unifiée de modélisation et son acceptation subséquente par la majorité des intervenants eurent des retombées très positives. La forte popularité de cette norme favorise le développement d'outils plus innovateurs qui ont l'avantage important d'être compatibles entre eux, et donc complémentaires. De plus, l'unification des dialectes a grandement facilité les discussions, les échanges et les collaborations entre les membres de la communauté.

Les effets du mouvement rassembleur UML ne se sont pas fait sentir cependant chez la communauté de maintenance: elle demeure à ce jour tout aussi fragmentée et souffre des malaises énumérés précédemment. L'absence d'une solution universellement acceptée pour la modélisation des systèmes logiciels déjà implantés perpétue l'incompatibilité des outils de cette communauté et menace sérieusement les progrès dans ce domaine comme le témoigne Woods *et al.* dans cet article [Woods_1998]:

> "Not only does the lack of interoperability make it more difficult to construct integrated environments, but it also implies that repositories of reverse engineering tools (such as the one being formed by the IEEE Computer Society Technical Council on Software's Committee on Reverse Engineering and Re-engineering), are doomed to having only a very limited impact despite their stated goal of avoiding unnecessary duplication of effort by having a central site for tools, publications, and data. To make these tools truly useful they need to be interoperable, not merely sharable."

À l'occasion du passage au vingt-et-unième siècle, plusieurs auteurs ont dressé le bilan des tendances passées et futures dans le domaine de la rétroconception [Müller_2000] et résument l'état présent comme suit:

> "Given that reverse engineering tools seem to be a key to aiding program understanding, how effective are today's offerings in meeting this goal? In both academic and corporate settings, reverse engineering tools have a long way to go before becoming an effective and integral part of the standard toolset a typical software engineer calls upon in day-to-day usage. Perhaps the biggest challenge to increased effectiveness of reverse engineering tools is wider adoption: tools can't be effective if they aren't used, and most software engineers have little knowledge of current tools and their capabilities. While there is a relatively healthy market for unit-testing tools, code debugging utilities, and integrated development environments, the market for reverse engineering tools remains quite limited."

Ces mêmes auteurs proposent à deux reprises le besoin vital d'intégrer les outils et les processus de la rétroconception avec leurs pairs du domaine de conception et de développement:

> "To increase the adoption rate of reverse engineering tools, vendors need to address several issues. The tools need to be better integrated with common development environments on the popular platforms."
>
> (...)
>
> "We need to integrate forward and reverse engineering processes for large evolving software systems and achieve the same appreciation for product and process improvement for long-term evolution as for the initial development phases."

Aujourd'hui, les environnements de conception et les environnements de développement sont intégrés grâce à UML. Il s'ensuit que l'intégration prônée des outils de rétroconception (et de maintenance en général) avec les outils de la communauté de conception et de développement doit inévitablement passer par UML.

Il existe cependant un obstacle potentiel à cette intégration qui découle du fait que UML a été conçu comme langage de modélisation pour les nouveaux systèmes logiciels en cours d'élaboration par opposition aux logiciels déjà implantés qui sont l'objet des activités de rétroconception, de compréhension et de réingénierie. En particulier, le métamodèle UML modélise bien la structure des logiciels mais ne prévoit pas de façon simple et efficace de modéliser explicitement le code source qui définit l'implantation concrète du corps d'une méthode.

L'objectif de ce mémoire est de présenter une solution qui permet l'utilisation d'UML pour modéliser explicitement le code qui définit le corps d'une méthode. Ces modèles plus détaillés peuvent alors être utilisés pour les activités de maintenance, et en particulier pour la rétroconception, la compréhension et la réingénierie de ces systèmes logiciels.

1.2 Contexte de la recherche

Ce travail a été réalisé dans le cadre du projet SPOOL. Le projet SPOOL (Spreading Desirable Properties into the Design of Object-Oriented, Large-Scale Software Systems) est une collaboration de l'équipe d'évaluation de qualité logicielle de Bell Canada et du groupe de génie logiciel GÉLO à l'université de Montréal. Ce projet s'inscrit au sein des activités initiées par le Consortium pour la recherche en génie logiciel (CRGL), un regroupement d'intervenants industriels, académiques et gouvernementaux qui vise l'élargissement du savoir en matière de génie logiciel au Canada.

L'objectif principal du projet SPOOL est d'identifier et de promouvoir les meilleures pratiques de conception et de développement qui contribuent à la qualité des systèmes logiciels orientés objet. La recherche s'articule autour de quatre axes complémentaires: les métriques de conception orientée objet, l'analyse de l'impact des changements, la traçabilité des transformations et le génie logiciel basé sur les patrons de conception.

Les besoins de recherche ont mené au développement de l'atelier SPOOL pour faciliter les activités de rétroconception, de compréhension et de réingénierie. L'architecture générale de l'atelier est conçue de manière à faciliter la création et l'intégration d'applications adaptées à des tâches spécifiques comme la rétroconception, la navigation, les analyses, le calcul de métriques et la visualisation. L'intégration est assurée à l'aide d'un dépôt central contenant les modèles des systèmes logiciels étudiés ainsi que les résultats des analyses. Le métamodèle adopté pour la modélisation des systèmes est dérivé du métamodèle UML 1.1 et adapté aux besoins concrets du projet SPOOL. Une description détaillée de l'environnement réalisé est fournie dans [Schauer_2001].

L'implication de l'auteur dans le projet SPOOL a d'abord consisté en la validation partielle du modèle d'impact de changement [Chaumun_1998] dans le contexte d'un projet de fin d'études de premier cycle. Ensuite, un outil d'analyse d'impact de

changements basé sur le modèle d'impact de Chaumun a été implanté et utilisé dans le cadre d'une expérimentation visant à identifier des indicateurs de changeabilité dans les logiciels orientés objet. Enfin, une passerelle améliorée permettant l'importation de modèles dans le dépôt a été réalisée.

1.3 Contribution principale

La contribution principale de notre recherche est le profil RCR. Le profil RCR décrit comment étendre le métamodèle UML avec les mécanismes d'extensions standard afin de permettre la modélisation détaillée du code source qui définit le corps des méthodes d'un système logiciel écrit dans un style de programmation impératif ou orienté objet. En particulier, les langages de programmation visés par le profil RCR sont C, C++ et Java.

Concrètement, le profil RCR définit 52 nouveaux métaéléments qui sont utilisés pour modéliser les entités suivantes: les blocs, les énoncés, les expressions, les variables et les pas d'évaluation. Les quatre premières entités modélisent les entités équivalentes dans le code source. Les pas d'évaluation ont été conçus pour le profil RCR pour représenter de manière explicite et concise l'application d'opérateurs prédéfinis dans les langages de programmation.

Le profil RCR étend l'utilité des modèles UML au domaine de la maintenance et élimine un obstacle majeur à l'unification des activités de développement et de maintenance. En particulier, le modèle UML d'un logiciel créé pendant sa phase d'élaboration peut maintenant être enrichi et ensuite réutilisé pour les activités de rétroconception, de compréhension et de réingénierie.

1.4 Publications

Les activités de l'auteur au sein du projet SPOOL ont donné lieu aux publications suivantes.

L'expérimentation sur les indicateurs de changeabilité dans les logiciels orientés objet a généré plusieurs publications dont une sur l'évaluation de la fiabilité de logiciels orientés objet [Kabaili_1999] et une autre sur l'évaluation de la changeabilité [Chaumun_2000]. Deux autres ont rapporté les résultats d'une étude de la cohésion des classes [Kabaili_2000, Kabaili_2000a].

Les travaux sur la passerelle du dépôt de SPOOL ont inspiré une publication discutant des formats d'échanges de modèles [Saint-Denis_2000]. L'utilisation de la technologie XML pour la passerelle a permis à l'auteur de collaborer à la rédaction d'un article discutant de l'utilité de XML dans la conception de classes auto-documentées et auto-testables [Deveaux_2000]. Finalement, l'architecture d'un dépôt basé sur UML est discutée dans un chapitre de livre [Schauer_2001] ainsi que dans un article de conférence [Keller_2001].

1.5 Structure du mémoire

Ce mémoire est organisé comme suit:

Le chapitre 2 propose des rappels des activités de rétroconception, de compréhension et de réingénierie de logiciels. Le chapitre conclut avec un rappel de UML.

Le chapitre 3 présente l'état de l'art en ce qui concerne la modélisation détaillée des systèmes logiciels avec UML. Le chapitre est subdivisé en trois volets dédiés à la communauté UML, la communauté de développement et la communauté de maintenance.

Le chapitre 4 précise la nature du problème qui doit être résolu. Les choix préliminaires qui délimitent l'étendue du problème sont présentés d'abord. Ensuite, deux mécanismes techniques dont l'existence est présupposée par la solution sont présentés. Le chapitre conclut en énumérant les contraintes qui doivent être respectées par la solution ainsi que les besoins qui doivent être satisfaits.

Le chapitre 5 présente le profil RCR. Un survol introduit les principaux concepts ainsi que les hiérarchies des nouveaux métaéléments. La section suivante explique les deux stratégies sous-jacentes de la solution. La troisième section présente chaque nouveau métaélément individuellement. Finalement, la dernière section résume le contenu du chapitre.

Le chapitre 6 propose cinq discussions au sujet du profil RCR. La première discussion explique comment le profil RCR satisfait les besoins et repecte les contraintes énumérées au chapitre 4. La deuxième discussion compare le métamodèle UML étendu par le profil RCR à d'autres métamodèles similiaires. La troisième discussion illustre comment le profil RCR est utilisé pour modéliser le code source. La quatrième discussion évoque les applications qui peuvent profiter du profil RCR. La dernière discussion explore les stratégies de mise en oeuvre du profil RCR dans le cas général et dans le cas particulier du projet SPOOL.

Le chapitre 7 conclut le mémoire avec une synthèse résumant les principaux éléments du mémoire, un aperçu des travaux futurs et une réflexion finale.

L'annexe A qui suit la bibliographie présente une stratégie pour la modélisation des types dérivés avec UML.

Chapitre 2: Rappels

Cette section propose des rappels et des précisions utiles à la compréhension du mémoire. Nous présentons d'abord l'activité de rétroconception. Ensuite, nous discutons de la compréhension de logiciels en rappelant l'utilité des analyses statiques et des métriques. Nous poursuivons avec un résumé des types de réingénierie de logiciels que l'on retrouve en pratique. Et finalement, nous terminons ce chapitre avec un exposé sur les diagrammes et le métamodèle UML.

2.1 Rétroconception

L'*abstraction* est un concept important dans la définition et la discussion de rétroconception. Pour cette raison, rappelons brièvement que l'abstraction est une stratégie pour faciliter la compréhension et pour gérer la complexité. Le procédé d'abstraction consiste à créer un modèle d'une entité ou d'un concept, c'est-à-dire une représentation qui limite notre attention aux éléments principaux en cachant les détails secondaires. Les représentations sont décrites à l'aide d'un langage approprié qui définit l'univers du discours d'un niveau d'abstraction donné. Quand il s'agit de systèmes logiciels, le niveau d'abstraction s'abaisse lorsque la représentation tend vers l'implantation matérielle du logiciel. À la limite, la séquence de bits constituant le code exécutable d'un logiciel pourrait être sa représentation au plus bas niveau d'abstraction. À l'autre extrémité de l'axe d'abstraction se trouvent les descriptions informelles, formulées non pas en termes informatiques mais plutôt dans le langage du domaine d'application, décrivant les objectifs généraux du logiciel. Les plans d'architecture générale, les documents de conception détaillée et le code source sont des représentations du système logiciel à des niveaux d'abstraction différents qui sont situés quelque part entre ces deux pôles.

Dans le domaine du logiciel, la définition couramment acceptée du terme *rétroconception*[1] (*reverse engineering*, en anglais) provient de Chikofsky et Cross [Chikofsky_1990]. Selon eux, la rétroconception est un processus d'analyse d'un système logiciel ayant comme objectifs principaux a) l'identification des composantes du système et de leurs interrelations, et b) la création de représentations du système sous d'autres formes ou à des niveaux d'abstraction plus élevés. En somme, on fait appel à la rétroconception pour obtenir une représentation désirée d'un logiciel lorsqu'elle est incomplète, inexacte, inaccessible ou inexistante.

Rien dans cette définition n'exclut la possibilité d'appliquer ce processus d'analyse à partir d'un niveau d'abstraction quelconque ou de l'utiliser récursivement. Conséquemment, les *composantes du système* dont il s'agit dans la définition de Chikofsky et Cross varient en fonction du degré d'abstraction traité. Prenons par exemple le modèle d'un logiciel exprimé en termes de *structures de données* et de *fonctions*. À un niveau d'abstraction plus élevé, le modèle du même système peut s'exprimer en termes de *modules* et d'*interfaces* qui englobent les composantes du niveau inférieur (i.e., les structures de données et les fonctions). Le principe d'abstraction crée une hiérarchie de composantes où celles des niveaux d'abstraction supérieurs agrègent celles des niveaux inférieurs.

La rétroconception repose essentiellement sur la *collection, l'analyse et la synthèse de données*. La collection consiste à recueillir les données pertinentes du système qui permettront éventuellement d'identifier ses composantes et leurs rôles ou de construire une représentation désirée. Une liste non-exhaustive des données utiles inclut les artefacts résultant de l'exécution du logiciel (e.g., trace, fichier, interface usager, signal, etc.), le code (e.g., exécutable, objet, source, etc.), la documentation (e.g., guide d'usager, manuel de maintenance, plan de conception, cahier des charges, etc.), les témoignages (e.g., utilisateur, mainteneur, développeur, concepteur, etc.) et les connaissances a priori (e.g., domaine du problème ou d'application, patrons de conception, styles de programmation, algorithmes, etc.). Lorsque la récolte des

[1] Les termes *rétro-ingénierie* et *désossage* sont aussi utilisés.

données est jugée satisfaisante, les activités d'analyse et de synthèse sont amorcées. L'analyse s'oriente surtout sur l'identification des composantes et de leurs interrelations, tandis que la synthèse cherche principalement à faire apparaître les composantes d'un niveau d'abstraction supérieur en puisant à même les résultats de l'analyse précédente.

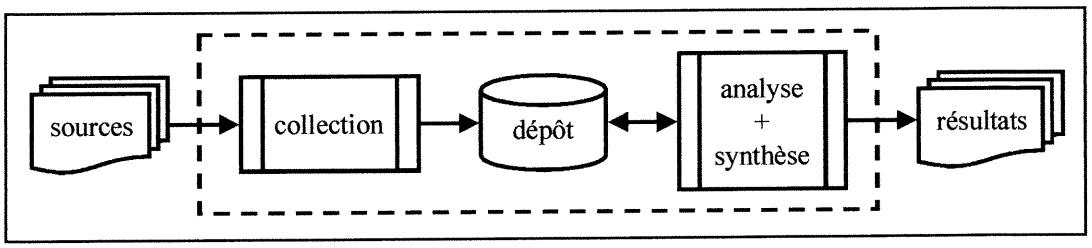


Figure 1: Architecture générique d'un environnement de rétroconception (Généralisation du modèle présenté dans [Chikofsky_1990])

À l'heure actuelle, la rétroconception de logiciels est une activité effectuée par l'humain avec l'assistance d'outils informatiques pour combattre les difficultés associées à la taille et à la complexité des systèmes traités. La Figure 1, adaptée de [Chikofsky_1990], illustre l'architecture générique d'un environnement de rétroconception typique. L'activité de collection, appuyée par les outils appropriés (e.g., parseur, analyseur sémantique, etc.), alimente le dépôt de données. De leur côté, les activités d'analyse et de synthèse font appel aux outils nécessaires pour la manipulation et l'examen des données du dépôt pour générer les résultats voulus. Au besoin, les activités d'analyse et de synthèse peuvent elles-mêmes alimenter le dépôt avec des résultats stratégiques qui serviront de données pour les opérations ultérieures.

2.2 Compréhension

Plusieurs chercheurs [vonMayrhauser_1994, Biggerstaff_1994, Robitaille_2000, etc.] se sont penchés sur la question de la *compréhension* de systèmes logiciels à cause de son importance fondamentale et des défis importants à relever. Une définition adéquate pour les besoins de ce mémoire caractérise la compréhension d'un logiciel

comme la création d'un modèle mental exact et complet de ce dernier. Ce modèle mental est construit progressivement par l'acquisition et l'intégration de nouvelles connaissances liées au système logiciel étudié. Les connaissances recherchées ainsi que les stratégies [vonMayrhauser_1994] utilisées pour les obtenir varient selon les individus, leurs motivations et le système logiciel traité. Comme dans le cas de la rétroconception, l'humain fait appel à l'informatique pour surmonter les obstacles de la complexité et du volume de données à traiter. Les outils assistent l'humain de deux façons: ils facilitent l'acquisition d'information pertinente et ils optimisent sa présentation pour maximiser son assimilation.

L'acquisition d'information s'effectue à partir d'un ensemble de sources similaires à celles utilisées pour la rétroconception: les artefacts d'exécution, le code, les modèles, la documentation, les témoignages et les connaissances a priori. Dans la pratique cependant, l'exécution, le code source et les modèles d'un système logiciel sont privilégiés en raison de leur disponibilité, de leur plus grande exactitude et surtout de leur richesse en information. Les analyses qui s'intéressent au système logiciel lors de son exécution sont dites *dynamiques*, tandis qu'on nomme *statiques* celles qui s'effectuent à partir du code source. Certaines de ces analyses peuvent être réalisées à partir de modèles suffisamment détaillés du système. En plus de ces analyses, le calcul de *métriques* permet de quantifier diverses propriétés du logiciel, enrichissant ainsi le modèle mental en évolution. Comme pour les analyses, des modèles appropriés peuvent servir de base pour le calcul de métriques. Les paragraphes qui suivent résument les analyses statiques et les métriques que l'on peut retrouver lors de l'étude d'un système logiciel.

2.2.1 Analyses statiques

La liste des analyses statiques est très vaste. Le survol suivant permet néanmoins d'apprécier la gamme d'information qui peut être obtenue. Le *graphe de dépendances*, comme son nom le suggère, révèle les dépendances sémantiques et syntaxiques qui existent entre les éléments du système. Les dépendances comprennent, entre autres,

l'instanciation, l'héritage, la définition, l'accès, la réalisation, l'appartenance et l'invocation (i.e., *graphe d'appel*). Le *graphe de flot de données* est utilisé pour identifier le cheminement des données à travers le système et le *graphe de flot de contrôle* permet de tracer les pas d'exécution du logiciel. Le *slicing* [Weiser_1981] tente de définir un sous-programme minimal possédant un sous-ensemble de comportements identique au programme complet. On l'utilise par exemple pour déterminer les énoncés qui affectent la valeur d'une variable à un point donné dans l'exécution du logiciel. Le *graphe de collaboration* [Schauer_1998] permet de déterminer l'existence et la nature de collaborations (e.g., design pattern, idiom, etc.) entre les composantes du logiciel. L'*analyse de pointeur* [Yong_1999] détermine pour un pointeur donné l'ensemble d'éléments du logiciel qui peuvent être la cible du pointeur. La *prédiction d'impact de changement* [Chaumun_1999] cherche à identifier les éléments affectés par une modification apportée au système. L'*exploration* d'un système logiciel [Sim_1999], soit par recherche explicite d'éléments ou par navigation, est une forme d'analyse plus flexible qui donne à chaque individu la liberté de choisir sa propre stratégie d'acquisition de connaissances. Certaines analyses se spécialisent dans la détection d'anomalies et de défauts potentiels comme le code mort, l'usage de variables non-initialisées, le clonage [Baker_1995, Mayrand_1996], les chaînes de récursion infinies, etc. D'autres analyses intègrent la notion d'évolution en comparant divers aspects du même logiciel parmi différentes versions.

2.2.2 Métriques

Les auteurs Fenton et Neil rapportent dans [Fenton_2000] l'existence de milliers de métriques qui ont été proposées depuis plus de 35 ans pour quantifier ou pour prédire une variété d'attributs associés au logiciel à travers son cycle de vie. La classification des métriques qu'ils suggèrent apparaît sous forme abrégée dans le Tableau 1. Les entités pertinentes dans le cycle vital du logiciel sont réparties parmi les groupes Produit, Processus et Ressources. Les métriques quantifient directement les attributs *internes* et elles tentent de prédire les attributs *externes*. Dans le contexte de ce

mémoire, les métriques intéressantes pour la compréhension de logiciel sont celles qui quantifient les attributs internes du code et du design. Ce choix est motivé par l'observation que les métriques associées aux attributs externes sont dérivées des métriques d'attributs internes.

Entité	**Attribut**	
	interne	**externe**
PRODUIT		
Spécifications	...	...
Design	taille, réutilisation, modularité, couplage, cohésion, héritage, fonctionnalité, etc.	changeabilité, complexité, maintenabilité, etc.
Code	taille, réutilisation, modularité, couplage, fonctionnalité, complexité algorithmique, structure de flot de contrôle, etc.	fiabilité, utilisabilité, maintenabilité, réutilisabilité, etc.
Cas de test	...	...
etc.	...	...
PROCESSUS		
Elaboration de la spécification	...	...
Conception détaillée	...	...
Tests	...	...
etc.	...	...
RESSOURCES		
Personnel	...	...
Equipe	...	...
Organismes	...	...
Logiciel	...	...
Matériel	...	...
Bureaux	...	...
etc.	...	...

Tableau 1: Classification des métriques
(Version abrégée du tableau dans [Fenton_2000])

2.3 Réingénierie

La *réingénierie*[2] (*reengineering*, en anglais) est définie dans [Chikofsky_1990] comme étant le processus d'examen et d'altération d'un système afin de le reconstituer sous une nouvelle forme suivi de l'implantation de cette nouvelle forme. Dans leur discussion du terme, les auteurs suggèrent que l'étendue des transformations apportées au système est relativement grande ou profonde. Dans ce mémoire, nous élargissons leur définition de réingénierie en permettant les transformations plus légères, mais en omettant les tâches de maintenance ponctuelles et superficielles (e.g., la correction d'une erreur de syntaxe dans le code source).

Les paragraphes qui suivent présentent les principales formes de réingénierie rencontrées que l'on distingue par la nature des transformations et les objectifs qui les motivent. Les figures illustrent comment chaque type de réingénierie transforme un système logiciel en indiquant à l'aide de hachures les aspects du logiciel qui sont typiquement impactés. Ces aspects sont groupés en trois familles qui correspondent aux principaux niveaux d'abstraction d'un logiciel tels que présentés à la section 2.1:

- l'*architecture de haut niveau* comprend les modules et les interfaces
- les *composantes détaillées* incluent les structures de données et les méthodes
- l'*implantation* désigne la réalisation concrète du logiciel exprimée en code source

[2] Les termes *reconception*, *reconfiguration* et *remise à neuf* sont également utilisés.

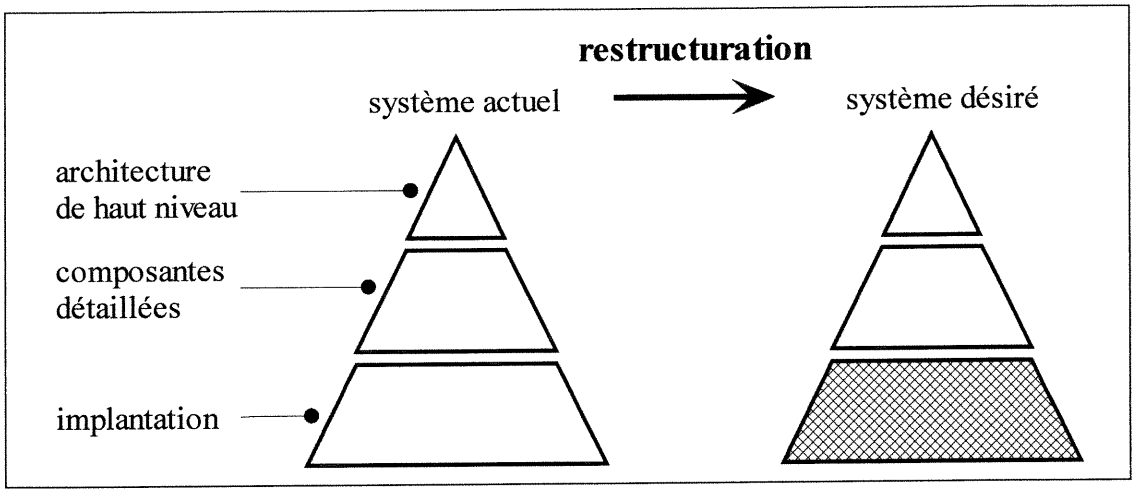


Figure 2: Réingénierie d'un système par restructuration

La *restructuration* (*restructuring* [Chikofsky_1990]) est une technique de maintenance préventive qui implique des transformations structurelles relativement légères du code sans altérations comportementales du logiciel tel que vu par l'usager. Un exemple de restructuration consiste à éliminer l'usage des énoncés *goto* en utilisant un style de programmation structurée.

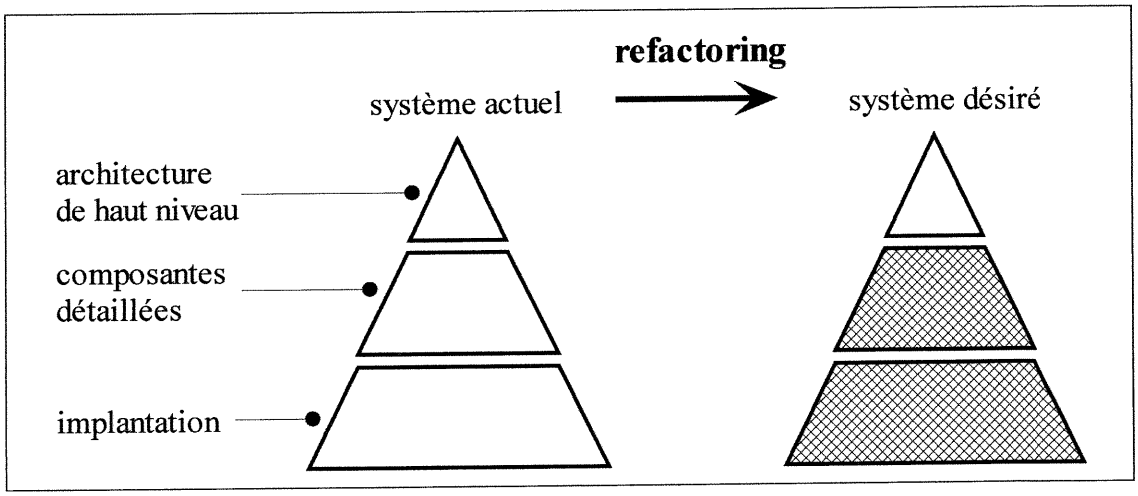


Figure 3: Réingénierie d'un système par refactoring

Le *refactoring* [Opdyke_1992] ressemble à la restructuration à la différence près que les transformations qui sont effectuées sont généralement plus complexes et elles exigent une compréhension plus approfondie de la sémantique du code. Par exemple, le refactoring *Tease Apart Inheritance* qui apparaît dans [Fowler_1999] implique la

réorganisation de la hiérarchie de classes pour éliminer un dédoublement de fonctionnalité parmi les sous-classes.

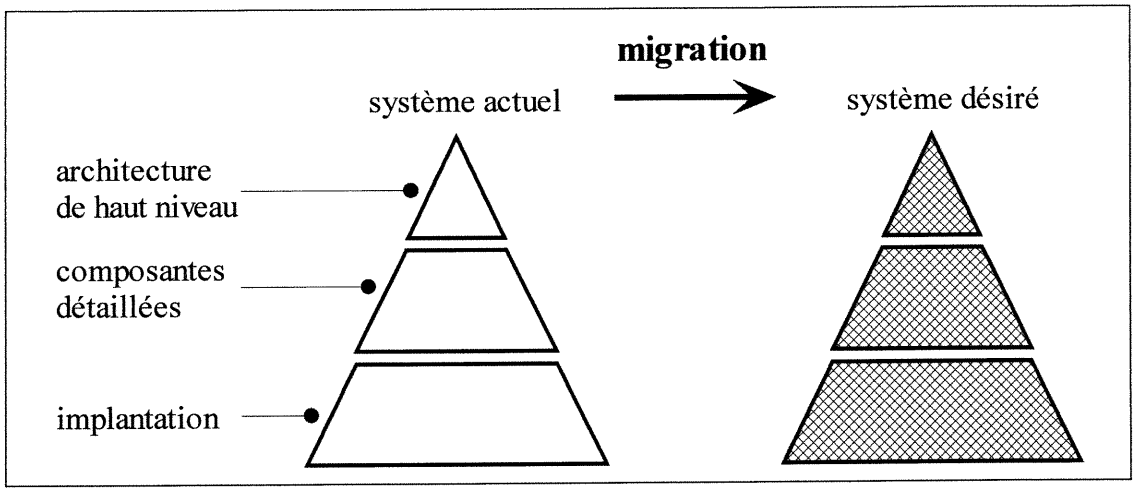


Figure 4: Réingénierie d'un système par migration

La *migration* est une activité de réingénierie caractérisée par l'évolution d'une propriété globale du logiciel. La propriété affectée peut être le langage de programmation (e.g., migration Fortran à C), la plate-forme d'exécution (e.g., migration Macintosh à Windows), le style de programmation (e.g., migration impératif à orienté objet), etc. La majorité des migrations impliquent de profondes transformations du logiciel.

La réingénierie comprend également une catégorie de transformations qui ressemblent au refactoring ou à la migration mais elles étendent ou modifient la fonctionnalité externe du logiciel.

2.4 UML

2.4.1 Origines et concepts

Le *Unified Modeling Language* (UML) est défini dans [OMG_2000][3] comme un langage graphique de modélisation pour visualiser, spécifier, construire et documenter les artefacts d'un système logiciel. Né de la collaboration de Grady Booch, James Rumbaugh et Ivar Jacobson vers 1995, UML intègre et étend les concepts et la notation des méthodologies de développement orienté objet les plus populaires de l'époque: Booch [Booch_1994], OMT [Rumbaugh_1991] et OOSE [Jacobson_1992]. Suite à cet effort initial, la coalition UML Partners constituée des plus importantes entreprises[4] en informatique est créée pour assurer l'évolution saine du langage et pour encourager son adoption dans la communauté de développement.

Parallèlement à ces activités, le consortium Object Management Group (OMG)[5] cherche à enrichir son architecture technique [OMG_1997] d'une infrastructure supportant l'analyse et la conception orientée objet. Les composantes recherchées incluent un métamodèle formel définissant la sémantique des modèles créés, la spécification de mécanismes pour l'échange de modèles entre outils et si possible, une notation visuelle. Un appel à propositions [OMG_1996] est lancé aux membres en juin 1996. L'évaluation des soumissions s'ensuit et à l'automne 1997, l'OMG adopte formellement la version 1.1 de la spécification UML soumise par les UML Partners et assume dorénavant la responsabilité exclusive de son évolution. Aujourd'hui, UML est reconnu comme norme *de facto* et progresse définitivement vers le statut de norme *de jure* [Kobryn_1999].

---

[3] Ce travail s'appuie sur cette définition de UML, à moins d'avis contraire. La terminologie originale anglaise est utilisée dans le texte pour éviter l'introduction d'imprécisions syntaxiques et sémantiques.
[4] La liste inclut Microsoft, IBM, Oracle, Hewlett-Packard, TI, Unisys, i-Logix, IntelliCorp, ICON Computing, MCI Systemhouse, Rational Software, Digital Equipment, ObjectTime, Platinum Technology, Ptech, Taskon, Reich Technologies, Sterling et Softeam.
[5] L'OMG est un consortium sans but lucratif de plus de 800 membres industriels, académiques, gouvernementaux ou indépendants visant à faciliter l'intégration d'applications dans les environnements distribués hétérogènes. Voir <http://www.omg.org/>.

Le langage UML s'articule autour de deux pôles: le métamodèle et les diagrammes. Le métamodèle définit les éléments utilisés pour construire le modèle d'un système, tandis que les diagrammes facilitent l'abstraction en présentant des vues complémentaires du modèle construit. La prochaine section présente les principaux types de diagrammes utilisés dans la modélisation UML. Ensuite, nous décrivons sommairement l'organisation et le contenu du métamodèle. Dorénavant, nous utiliserons le terme *métaélément* pour désigner un élément du métamodèle.

2.4.2 Diagrammes

Les diagrammes et leur documentation associée sont les principaux artefacts visibles de la modélisation UML. Chacun des huit types de diagrammes s'intéresse à un aspect particulier du système complet. Ces aspects sont modélisés par des sous-ensembles de métaéléments. Chaque type de diagramme utilise une notation graphique particulière pour transmettre l'information contenue dans les métaéléments sous-jacents. Cependant, un diagramme donné n'est pas tenu de présenter la totalité de l'information des métaéléments; certains détails peuvent être omis selon la stratégie de modélisation choisie par l'analyste.

Les diagrammes sont répartis en deux grandes familles associées aux deux dimensions fondamentales d'un système: sa structure (i.e. *structure diagrams*) et son comportement (i.e. *behavior diagrams*). La liste des diagrammes principaux apparaît dans le Tableau 2. Les sections suivantes présentent un exemple typique de chacun de ces diagrammes et décrivent leur utilité dans la modélisation d'un système logiciel.

structure diagrams
class diagram
implementation diagrams
component diagram
deployment diagram
behavior diagrams
use case diagram
statechart diagram
activity diagram
interaction diagrams
sequence diagram
collaboration diagram

Tableau 2: Diagrammes principaux de UML

2.4.2.1 Use Case Diagram

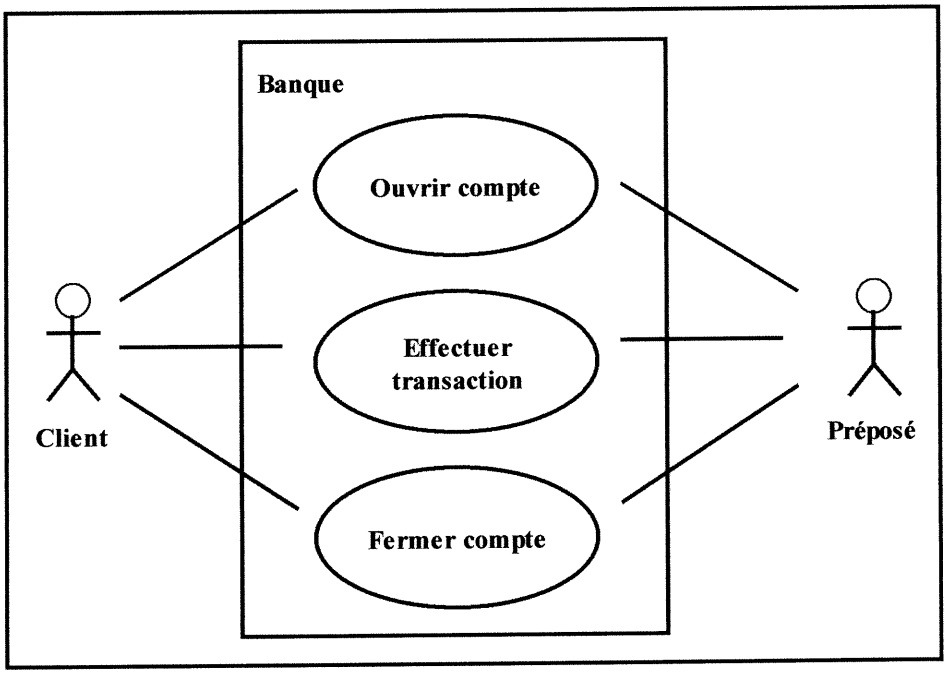


Figure 5: Exemple de *use case diagram*

Le *use case diagram* est principalement constitué d'acteurs et de cas d'utilisation (*use case*) et sert à illustrer les relations qui existent entre ces éléments. Dans l'exemple de la Figure 5, les deux acteurs *Client* et *Préposé* interviennent dans les cas d'utilisation

Ouvrir compte, *Effectuer transaction* et *Fermer compte*. Ces cas d'utilisation appartiennent au système *Banque*.

2.4.2.2 Class Diagram

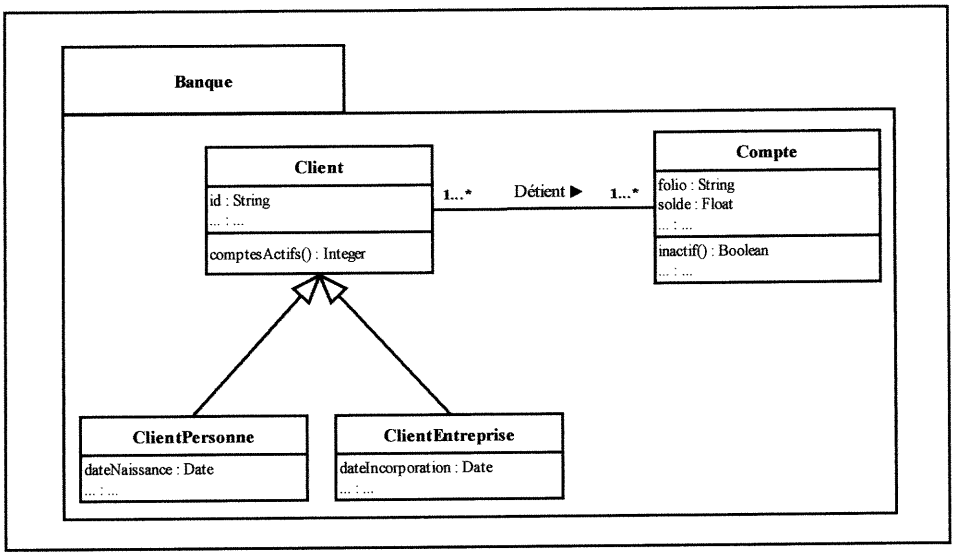


Figure 6: Exemple de *class diagram*

Le *class diagram* est utilisé pour visualiser l'architecture d'un système. Il est principalement constitué d'éléments structuraux tels que les classes, les interfaces et les modules (*packages*). Il peut aussi contenir des instances de classes[6]. La composition interne des éléments et leurs interrelations font partie de l'information rendue par ce genre de diagramme. L'exemple de la Figure 6 fournit une vue du contenu du package *Banque*. On y retrouve les classes *Client*, *ClientPersonne*, *ClientEntreprise* et *Compte*. Les flèches à tête creuse indiquent que les classes *ClientPersonne* et *ClientEntreprise* héritent de la superclasse *Client*. La ligne et le texte entre la classe *Client* et la classe *Compte* dénote l'association binaire *Détient* qui relie un ou plusieurs *Client* à un ou plusieurs *Compte*. Le contenu de la classe *Compte* est décrit dans les deux compartiments sous le nom de la classe. Le premier compartiment révèle une liste incomplète d'attributs comprenant *folio* et *solde* qui

[6] Un diagramme constitué majoritairement d'instances de classes est nommé *object diagram*.

sont de types String et Float, respectivement. Le second compartiment contient une liste incomplète d'opérations qui inclut *inactif* ayant Boolean comme type de retour.

2.4.2.3 Statechart Diagram

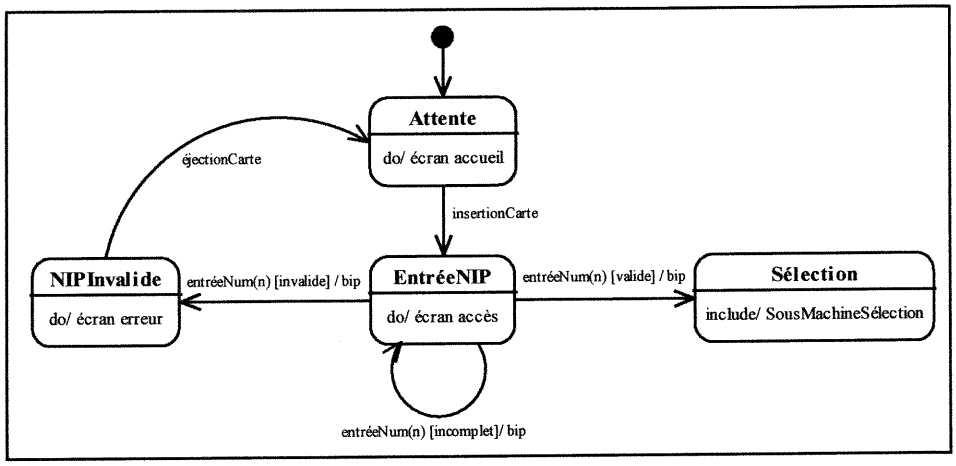


Figure 7: Exemple de *statechart diagram*

Le *statechart diagram* expose le comportement d'une entité du système en décrivant les transitions d'états et les actions effectuées suite aux événements qui surviennent durant la vie de l'entité. Les éléments typiques d'une machine à états s'y retrouvent (e.g., état, événement, transition) ainsi que des extensions qui simplifient la modélisation (e.g. pseudo-états, machines imbriquées, etc.). L'exemple de la Figure 7 représente la machine à états d'un guichet automatique bancaire (GAB). Sur ce diagramme figurent quatre états: *Attente*, *EntréeNIP*, *NIPInvalide* et *Sélection*. Les actions effectuées par le GAB dans chacun de ses états sont inscrites dans le compartiment inférieur de l'état. Par exemple, le GAB affiche son écran d'accueil lorsqu'il est dans l'état *Attente*. L'action *include/SousMachineSélection* de l'état *Sélection* dénote l'invocation de la machine à états imbriquée *SousMachineSélection*. Les transitions sont représentées par les flèches annotées. Les annotations indiquent l'événement, la condition et l'action effectuée. Par exemple, la transition entre les états *EntréeNIP* et *NIPInvalide* est déclenchée lorsque le NIP complété par l'entrée d'un numéro (*entréeNum(n)*) est *invalide*. Un *bip* sonore accompagne la transition.

2.4.2.4 Activity Diagram

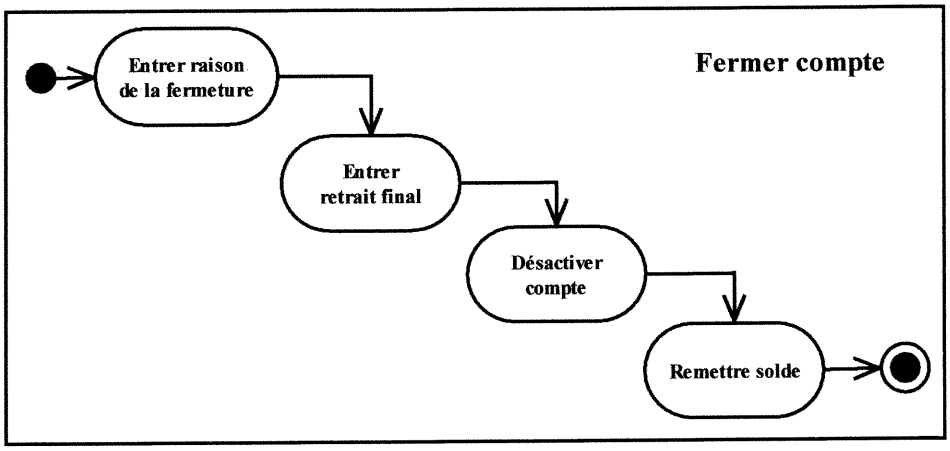


Figure 8: Exemple d'*activity diagram*

L'*activity diagram* s'apparente au *statechart diagram*, mais contrairement à ce dernier qui s'intéresse surtout aux événements externes asynchrones, l'*activity diagram* modélise principalement le flot de contrôle interne d'une entité comme un cas d'utilisation, une classe ou une méthode. L'état générique du *statechart diagram* est remplacé ici par un état d'action qui comporte une action simple effectuée immédiatement après la transition entrante. La transition de sortie est déclenchée lorsque l'action simple termine. L'exemple de la Figure 8 est associé avec le cas d'utilisation *Fermer compte*. Les quatre états anonymes sont traversés séquentiellement en commençant par celui au coin supérieur gauche du diagramme. Pour chaque état, l'action à compléter est inscrite à l'intérieur de son symbole.

2.4.2.5 Sequence Diagram

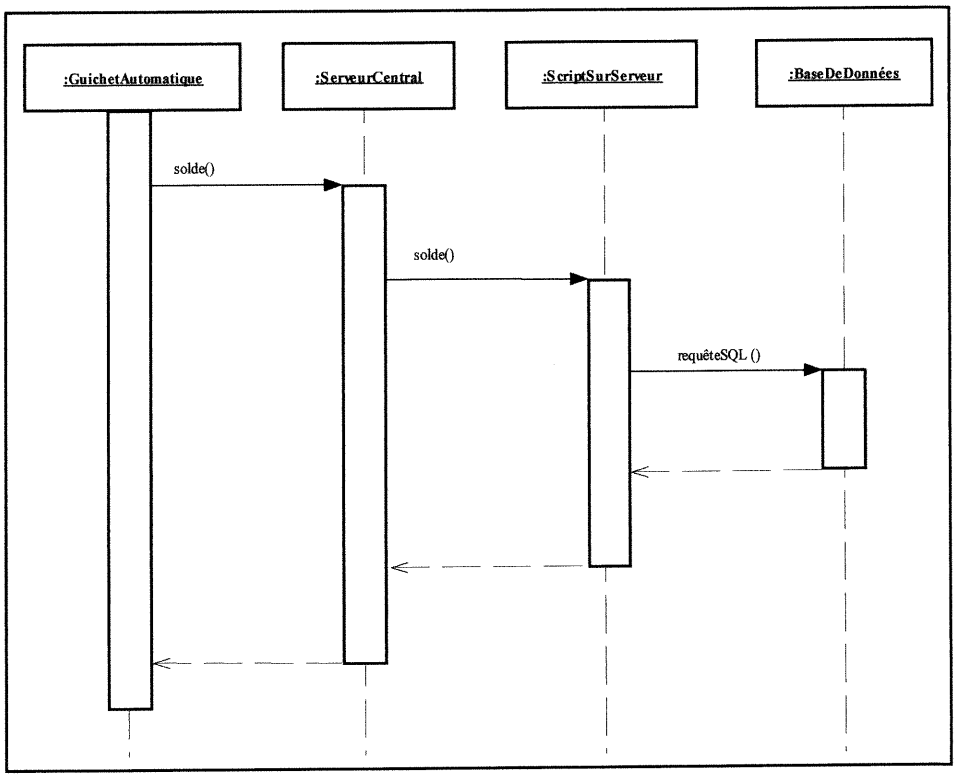


Figure 9: Exemple de *sequence diagram*

Le *sequence diagram* est utilisé pour présenter l'interaction séquentielle des entités du système. Ce diagramme est typiquement organisé selon deux axes perpendiculaires: le premier représente le temps et le second est populé d'instances qui participent à l'interaction modélisée. Les stimuli qui sont échangés entre les objets s'étendent de la ligne de vie de l'objet source jusqu'à la ligne de vie de l'objet destination. L'exemple de *sequence diagram* de la Figure 9 décrit une interaction possible entre quatres objets du système: *GuichetAutomatique*, *ServeurCentral*, *ScriptSurServeur* et *BaseDeDonnées*. Les flèches correspondent aux stimuli transmis entre objets: les flèches pleines dénotent l'invocation de procédures (e.g., *solde()*, *requêteSQL()*) et les flèches en traits représentent leur retour. Les rectangles verticaux correspondent aux périodes où l'objet exécute directement une action ou bien l'objet est en attente de la complétion d'une sous-action imbriquée.

2.4.2.6 Collaboration Diagram

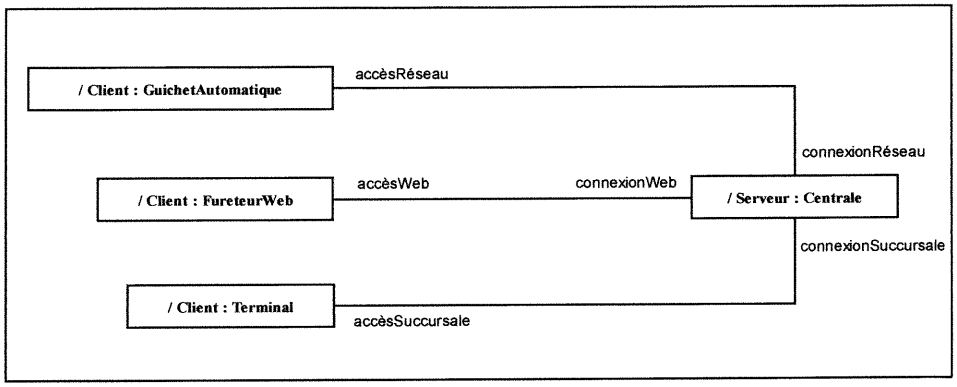


Figure 10: Exemple de *collaboration diagram*

Le *collaboration diagram* sert à illustrer l'organisation d'entités qui réalisent un comportement particulier à l'intérieur du système. La variante de base de ce type de diagramme présente les rôles des classes et des associations à l'intérieur d'une collaboration spécifique. D'autres variantes de diagramme illustrent l'*interaction* (i.e., *interaction diagram*), c'est-à-dire un ensemble de communications possibles entre les classes ou entre les objets. L'exemple de la Figure 10 expose une collaboration entre les classes *GuichetAutomatique*, *FureteurWeb*, *Terminal* et *Centrale*. La notation spéciale "/ *rôle* : *classe*" indique le rôle que joue une classe donnée dans la collaboration. Dans ce cas-ci, les trois premières classes jouent le rôle de *Client* tandis que *Centrale* joue le rôle de *Serveur*. Dans les trois associations, *GuichetAutomatique*, *FureteurWeb* et *Terminal* sont associés à des points d'accès provenant du réseau, du Web ou d'une succursale, respectivement; tandis que la classe *Centrale* représente une connexion réseau, Web ou succursale selon le type de client.

2.4.2.7 Component Diagram

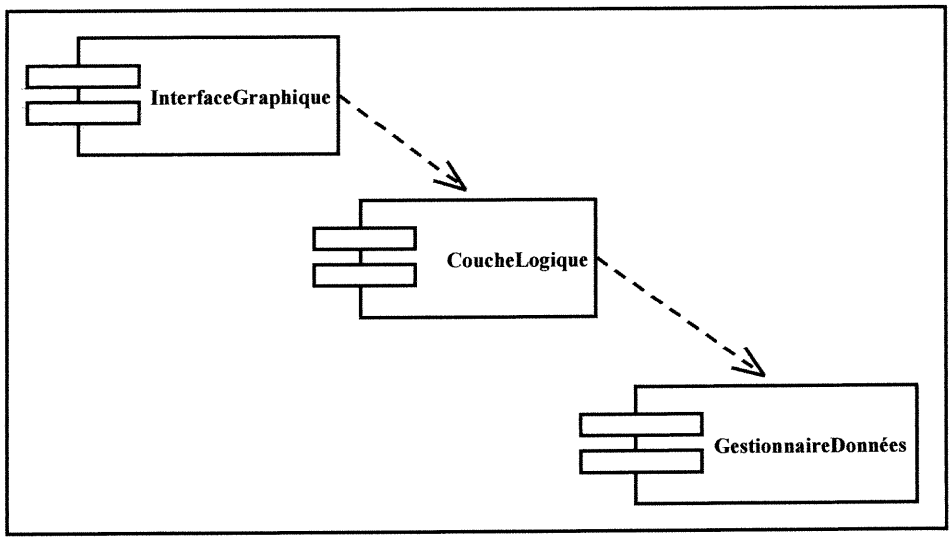


Figure 11: Exemple de *component diagram*

Le *component diagram* est utilisé pour exposer les dépendances qui existent entre des composantes issues de l'implantation matérielle du système. Les composantes comprennent le code source (e.g., fichier *.C), le code binaire (e.g., librairie) ou le code exécutable (e.g., application), ainsi que d'autres entités comme les bases de données. Le *component diagram* de la Figure 11 présente trois composantes: *InterfaceGraphique*, *CoucheLogique* et *GestionnaireDonnées*. On y retrouve également deux dépendances: *InterfaceGraphique* dépend de *CoucheLogique*, qui elle-même dépend de *GestionnaireDonnées*.

2.4.2.8 Deployment Diagram

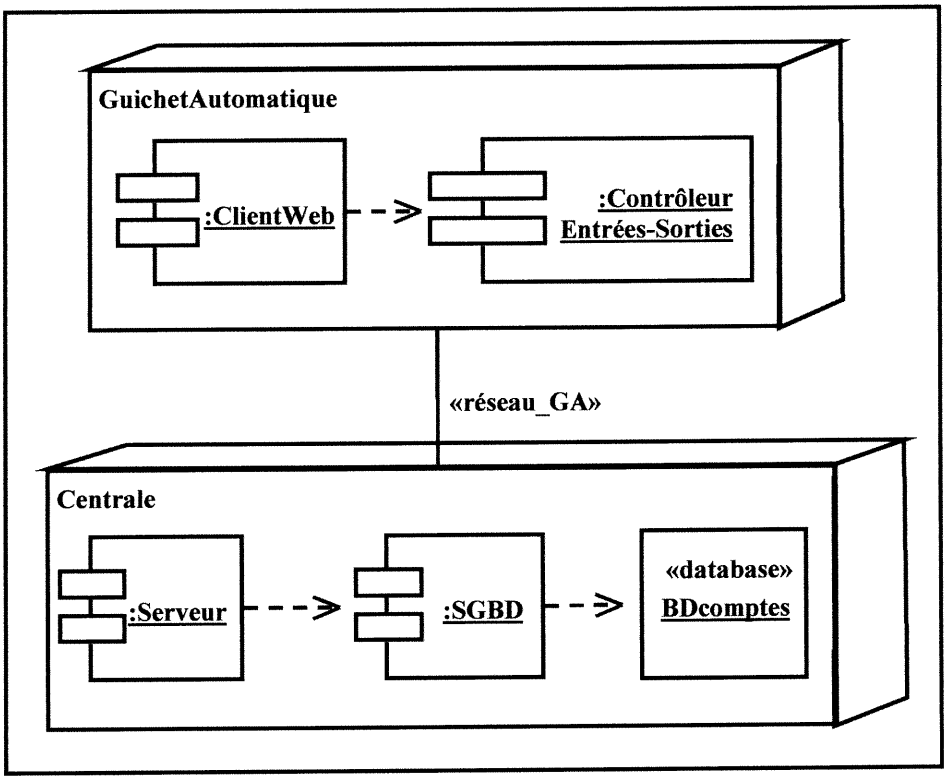


Figure 12: Exemple de *deployment diagram*

Le *deployment diagram* présente une vue des unités de traitement impliquées dans la réalisation d'un système. Le diagramme permet de visualiser la distribution des instances (e.g., composantes et objets) parmi les unités de traitement ainsi que les liens de communication qui existent. L'exemple de la Figure 12 illustre une configuration constituée de deux unité de traitement: *GuichetAutomatique* et *Centrale*. *GuichetAutomatique* contient deux composantes: *ClientWeb* qui dépend de *ContrôleurEntrées-Sorties*. Pour sa part, *Centrale* contient deux composantes (*Serveur*, *SGBD*) et un objet (*BDComptes*). Les deux unités de traitement communiquent entre elles via le *réseau_GA*.

2.4.3 Métamodèle

La création de modèles et de schémas est une activité fondamentale de l'informatique. Reconnaissant l'importance de cette activité et cherchant à satisfaire son objectif

d'interopérabilité universelle, l'OMG s'est doté de l'architecture de métamodèlisation à quatre niveaux telle qu'illustrée à la Figure 13. Le méta-métamodèle du niveau M3 est formellement défini dans la spécification du *Meta Object Facility* (MOF) [OMG_2000a] adopté par l'OMG en 1997. En plus de définir le méta-métamodèle, cette spécification définit également la procédure de génération d'interfaces en IDL[7] pour tout métamodèle dérivé du MOF facilitant ainsi l'interopérabilité de modèles et de schémas. Dans le cas présent, le métamodèle UML occupe le niveau M2 de cette architecture et sert à définir les modèles et les schémas du niveau M1.

[7] Le Interface Definition Language (IDL) est utilisé pour définir les interfaces universelles des entités qui existent dans l'Object Management Architecture de l'OMG.

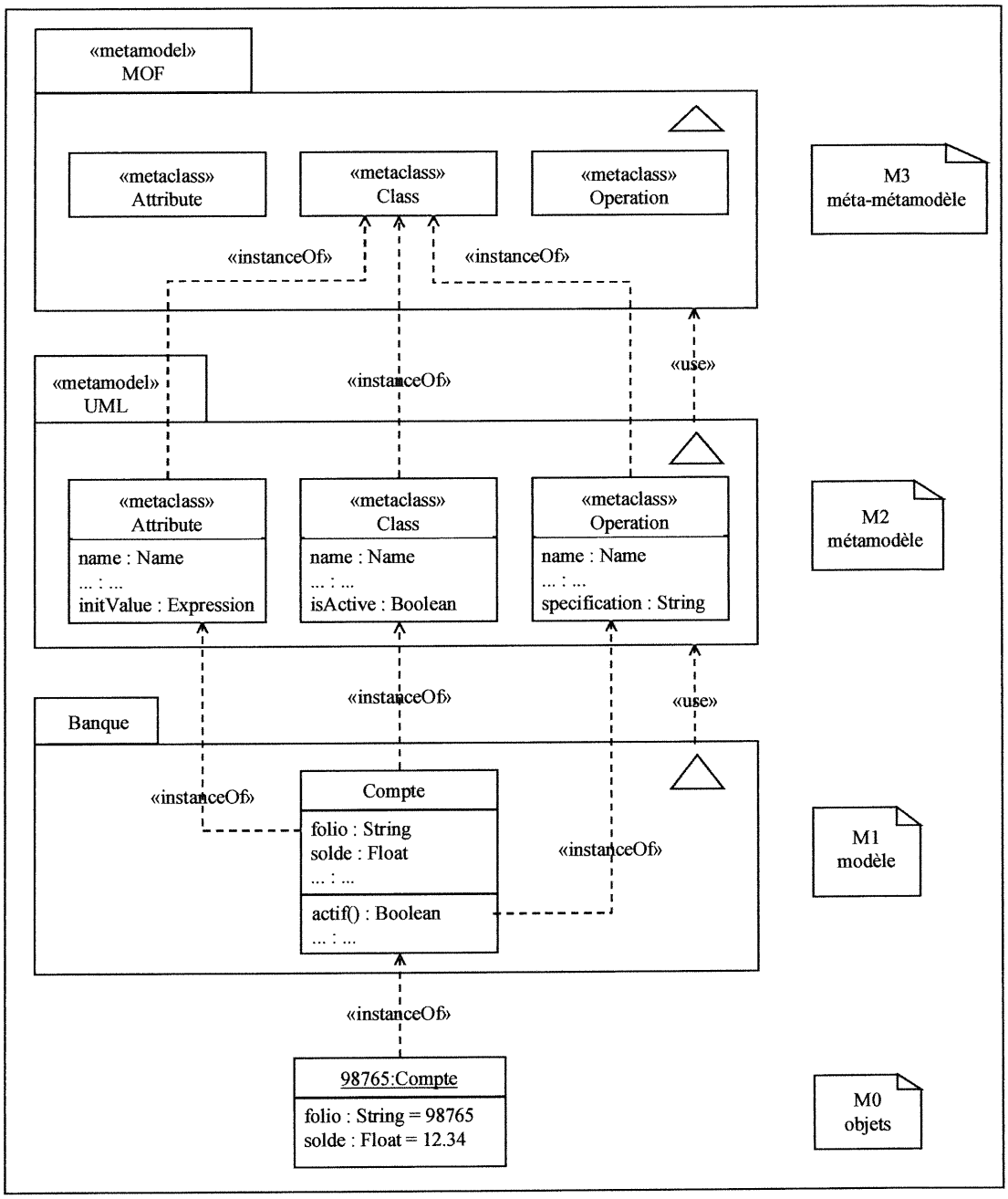


Figure 13: Architecture de métamodélisation à quatre niveaux de l'OMG (Adapté de [Kobryn_1999])

Les métaéléments du métamodèle UML sont définis par une hiérarchie de classes abstraites et concrètes enrichie de liens sémantiques. La sémantique est décrite à l'aide de texte en langage naturel et les aspects plus formels (e.g., invariants, conditions, etc.) sont exprimés en *Object Constraint Language* (OCL) [OMG_2000], un langage déclaratif qui accompagne UML. Le métamodèle est partitionné en

packages organisés selon le diagramme de la Figure 14. Les sous-sections suivantes décrivent brièvement le contenu de chacun de ces packages.

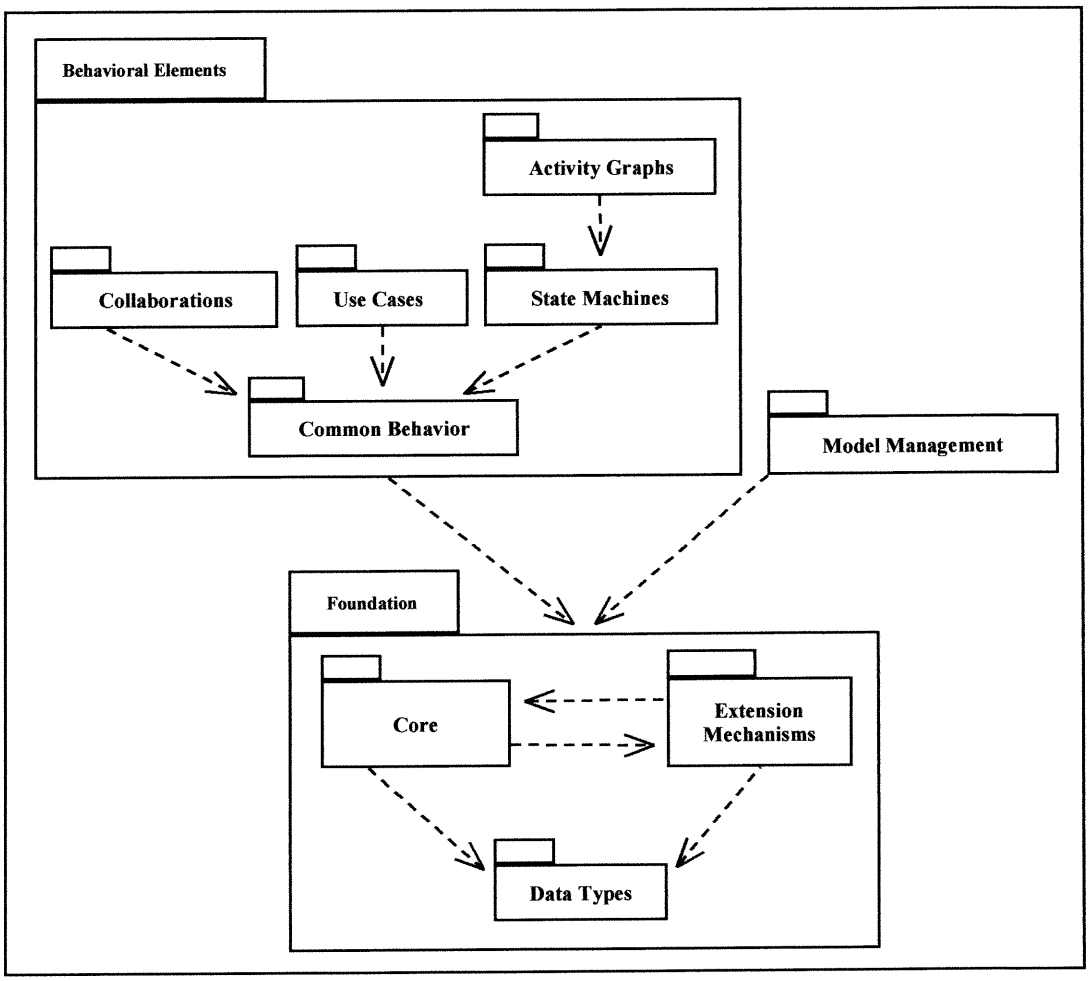


Figure 14: Packages du métamodèle UML

2.4.3.1 *UML.Foundation.Core*

Le package *UML.Foundation.Core* constitue le noyau du métamodèle UML. Les métaéléments qui s'y trouvent définissent les entités et les relations de base pour définir un modèle. La charpente est constituée d'entités telles que *ModelElement*, *Namespace*, *GeneralizableElement*, *Classifier*, *Class*, *Interface*, *DataType*, *Feature*, *StructuralFeature*, *BehavioralFeature*, *Attribute*, *Operation*, *Method* et *Parameter*. Les relations sont modélisées par *Association*, *Dependency*, *Generalization*, *Flow*,

Abstraction, *Permission*, etc. On y retrouve également des métaéléments auxiliaires tels que *TemplateParameter*, *Comment* et *PresentationElement*.

2.4.3.2 *UML.Foundation.DataTypes*

Le package *UML.Foundation.DataTypes* a la particularité de contenir les métaéléments qui ne sont pas des descendants de *ModelElement*. Spécifiquement, ces pseudo-métaéléments sont destinés exclusivement à la définition du métamodèle UML lui-même; ils ne peuvent pas être utilisés comme éléments à part entière dans les modèles[8]. Ce groupe comprend *Integer*, *UnlimitedInteger*, *String*, *Time*, *Boolean*, *AggregationKind*, *ChangeableKind*, *VisibilityKind*, *Multiplicity* ainsi que *Expression* et ses descendants (e.g. *ActionExpression*, *BooleanExpression*, *ProcedureExpression*, *TypeExpression*, etc.).

2.4.3.3 *UML.Foundation.ExtensionMechanisms*

Le package *UML.Foundation.ExtensionMechanisms* contient les métaéléments nécessaires pour étendre le métamodèle. Les trois mécanismes d'extension prévus sont la classification, les propriétés et les contraintes. La classification consiste à étiqueter un élément afin de signaler son appartenance à un groupe d'éléments possédant une sémantique distincte. Concrètement, l'étiquetage est réalisé par l'association du métaélément *Stereotype* à un autre métaélément. Le deuxième mécanisme d'extension, les propriétés, consiste à associer une ou plusieurs propriétés supplémentaires à un métaélément. Le métaélément *TaggedValue* réalise ce type d'extension. Le dernier procédé d'extension associe une ou plusieurs nouvelles contraintes à un métaélément. Ce mécanisme fait appel au métaélément *Constraint*.

---

[8] La spécification UML 1.3 le mentionne explicitement: "Note that these data types are the data types used for defining UML and not the data types to be used by a user of UML. The latter data types will be instances of the DataType metaclass defined in the metamodel." ([OMG_2000], p. 2-78).

2.4.3.4 *UML.BehavioralElements.CommonBehavior*

Le package *UML.BehavioralElements.CommonBehavior* rassemble les métaéléments essentiels à la modélisation des aspects dynamiques d'un système. Les instances d'entités sont modélisées par *Instance* et ses sous-classes *DataValue*, *ComponentInstance*, *NodeInstance* et *Object*, tandis que l'instance d'une association correspond à *Link*. La communication entre instances est réifiée par *Stimulus*. L'exécution d'instructions qui affectent l'état d'un modèle est modélisée par la famille d'*Action* qui comprend *CreateAction*, *CallAction*, *ReturnAction* et *DestroyAction*. Et finalement, *Signal* et *Exception* dénotent les communications asynchrones.

2.4.3.5 *UML.BehavioralElements.Collaborations*

Le package *UML.BehavioralElements.Collaborations* regroupe les métaéléments qui permettent de modéliser l'organisation d'entités d'un système qui peuvent interagir entre elles. Les métaéléments *Collaboration* et *Interaction* correspondent à deux perspectives proposées par UML pour modéliser une société d'entités. Les rôles sont modélisés par *ClassifierRole*, *AssociationRole* et *AssociationEndRole* et les messages échangés lors d'une interaction correspondent à *Message*.

2.4.3.6 *UML.BehavioralElements.UseCases*

Le package *UML.BehavioralElements.UseCases* renferme les métaéléments nécessaires pour créer les modèles des cas d'utilisation. On y retrouve les deux entités principales, *Actor* et *UseCase*, ainsi que les métaéléments *Include*, *Extend* et *ExtensionPoint* qui enrichissent un cas d'utilisation existant.

2.4.3.7 *UML.BehavioralElements.StateMachines*

Le package *UML.BehavioralElements.StateMachines* s'intéresse aux machines à états. Le coeur du package réside dans le métaélément *StateMachine*. S'ajoutent à celui-ci les entités de la famille *State*, dont *Pseudostate*, *SynchState*, *StubState*, *SimpleState*, *CompositeState* et *FinalState*. Le reste du package inclut *Transition*,

Guard et le groupe *Event* qui comprend *SignalEvent*, *CallEvent*, *TimeEvent* et *ChangeEvent*.

2.4.3.8 *UML.BehavioralElements.ActivityGraphs*

Le package *UML.BehavioralElements.ActivityGraphs* est associé au graphe d'activité, un cas particulier d'une machine à états. On y retrouve *ActivityGraph* qui peut être partitionné avec *Partition*. Les états spécialisés d'un graphe d'activité incluent *ActionState*, *CallState* et *ObjectFlowState*. Un objet et son état à un moment donné sont modélisés avec *ClassifierInState*.

2.4.3.9 *UML.ModelManagement*

Le package *UML.ModelManagement* contient les métaéléments destinés à la gestion des éléments d'un système. Les trois groupements possibles sont *Model*, *Package* et *Subsystem*. Le métaélément *ElementImport* réifie la relation entre un élément et le package qui l'importe.

2.4.4 Profil

Le *profil* est un mécanisme proposé par l'OMG [ADTF_1999] pour définir une spécialisation d'un métamodèle pour un domaine d'application particulier. Cette approche est dite légère (*lightweight*) comparativement à l'approche plus lourde (*heavyweight*) qui consiste à étendre le métamodèle en modifiant sa définition au niveau du méta-métamodèle. Concrètement, un profil comprend un ou plusieurs des items suivants:

- Un sous-ensemble non-strict de métaéléments existants pour délimiter la portée du profil.
- Des règles de bonne forme (*well-formedness rules*) supplémentaires définies au moyen de contraintes exprimées en OCL.
- Des éléments standard (*standard elements*) supplémentaires, c'est-a-dire des instances des métaéléments *Stereotype*, *TaggedValue* et *Constraint*.
- Des descriptions en langage naturel de concepts sémantiques supplémentaires.

- Des notations graphiques supplémentaires.

2.4.5 XMI

La technologie XML Metadata Interchange (XMI) a été adoptée par l'OMG [OMG_2000b] comme mécanisme d'échange de données pour les outils de modélisation et les dépôts de données. Cette solution s'appuie sur l'intégration du langage de balisage XML [W3C_2000] et le MOF. Sommairement, l'approche consiste à encoder les données à échanger dans un document XML qui est conforme à une certaine définition de document (*document type definition*, ou DTD). La spécification XMI définit premièrement la marche à suivre pour générer le DTD d'un métamodèle dérivé du MOF et deuxièmement comment produire un document XML à partir d'une instance de modèle.

Chapitre 3: État de l'art

Ce chapitre présente l'état de l'art dans la modélisation détaillée de systèmes logiciels avec UML. Nos observations proviennent des trois communautés jugées pertinentes à ce travail: la communauté UML et les communautés complémentaires de développement et de maintenance de logiciel. La communauté UML s'intéresse aux aspects théoriques et pratiques du langage lui-même: sa définition, sa structure, ses applications, son évolution, ses forces, ses faiblesses, etc. La communauté de développement s'intéresse aux techniques et aux outils qui aident à concevoir et à implanter les systèmes logiciels. La communauté de maintenance rassemble les intervenants des domaines de rétroconception, de compréhension et de réingénierie de systèmes logiciels.

3.1 Communauté UML

3.1.1 Spécification UML

Dans [Rumbaugh_1999], les créateurs originaux d'UML déclarent sous la rubrique *Programming Language Responsibilities*:

> "UML must work with various implementation languages without incorporating them explicitly. (...) The representation of detailed language properties for implementation raises the problem of capturing information about implementation properties without building their semantics into UML. Our approach was to capture language properties that go beyond UML's built-in capabilities by means of stereotypes and tagged values."

Cette citation rappelle que les mécanismes d'extension (i.e., *Stereotype*, *TaggedValue*, *Constraint*) sont nécessaires pour modéliser les détails d'implémentation qui

dépassent les capacités intégrées du langage. Afin de clairement identifier les limites de la modélisation détaillée avec UML, nous examinons les trois principaux mécanismes qu'offre le langage pour intégrer le code source au modèle du logiciel. Nous nommons ces stratégies *Comment*, *Method.body* et *ActivityGraph*. Le code source utilisé pour les démonstrations suivantes apparaît dans l'Exemple 1[9].

```
Point Droite::intersection(Droite d)
{
  if (pente == d.pente) return Point(INF, INF);
  float x = (d.delta - delta) / (pente - d.pente);
  float y = (pente * x) + delta;
  return Point(x, y);
}
```

Exemple 1: Méthode adaptée de [Booch_1999]

3.1.1.1 Stratégie *Comment*

La première stratégie prévue dans UML consiste à placer le corps de la méthode dans un commentaire qui lui est attaché. La Figure 15 illustre comment cette stratégie peut apparaître dans un *class diagram*. La Figure 16 indique la configuration des éléments du modèle sous-jacent qui réalisent cette stratégie: un objet *Comment* qui contient[10] la chaîne de caractères définissant le corps de la méthode est associé à l'objet *Method* par l'association *AnnotatedElement–Comment*.

[9] Lors d'une implantation réelle, ce code doit tenir compte des éventuelles limites de représentation des nombres réels.

[10] Selon le métamodèle, *Comment* n'a aucun attribut. Pour demeurer cohérents avec les autres exemples présentés dans cette section, nous lui accordons l'attribut *text* de type String pour contenir le texte du commentaire.

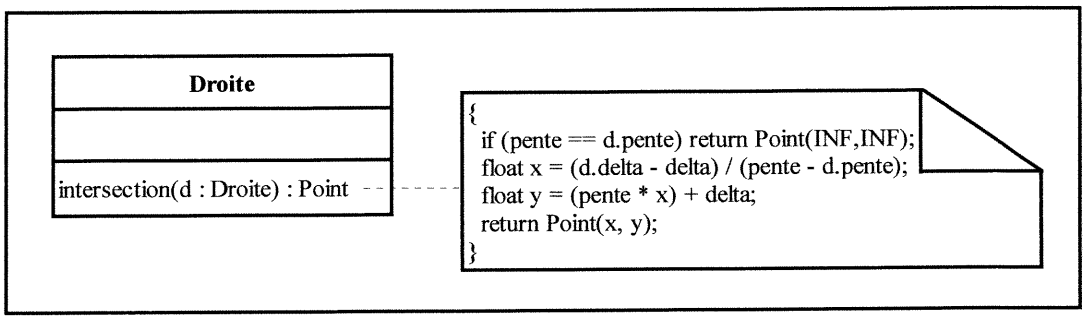


Figure 15: Corps de méthode dans *Comment*
(tel que vu dans le *class diagram*)

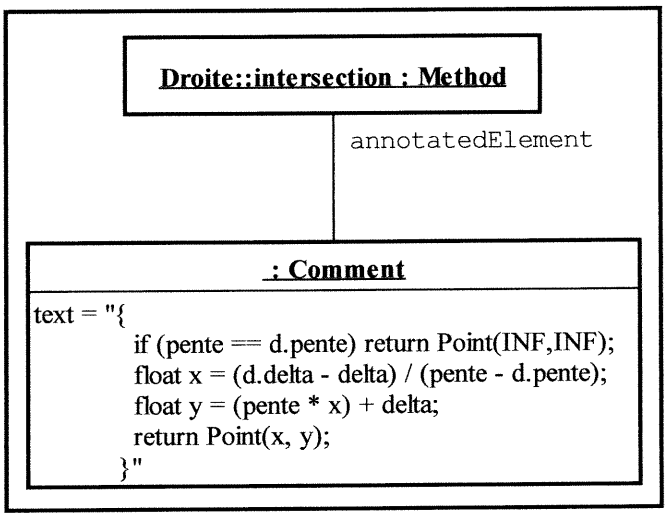


Figure 16: Corps de méthode dans *Comment*
(tel qu'implanté dans le modèle sous-jacent)

3.1.1.2 Stratégie *Method.body*

La deuxième stratégie, plus intuitive que les deux autres, consiste à utiliser l'attribut *body* du métaélément *Method* pour stocker le corps de la méthode. L'attribut *body* est de type *ProcedureExpression*, c'est-à-dire un énoncé qui produit un changement de valeurs dans son environnement lorsqu'il est évalué. Dans le métamodèle, *ProcedureExpression* hérite de *Expression* qui possède deux attributs: *body*[11] contient la chaîne de caractères définissant l'expression, et *language* indique le langage à utiliser pour l'évaluer. Le modèle sous-jacent, illustré à la Figure 17, se résume en fait à l'objet *Method* seulement; son attribut *body*, une instance de *ProcedureExpression*,

[11] Notons que les attributs *Method.body* et *ProcedureExpression.body* sont distincts et indépendants.

contient le texte du code ainsi que le nom du langage de programmation. Il n'y a pas de notation graphique particulière dans les diagrammes associée à cette stratégie.

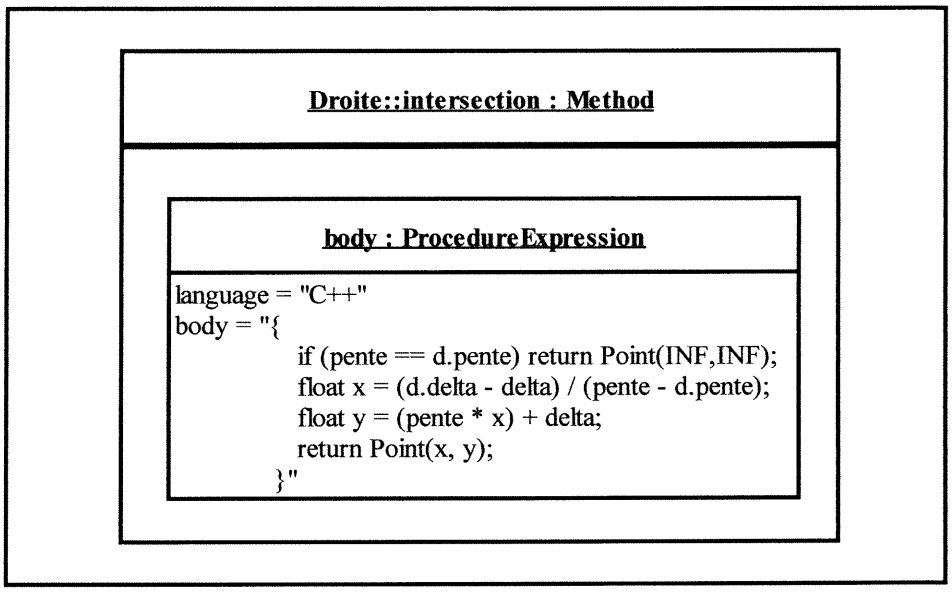


Figure 17: Corps de méthode dans *Method.body*
(tel qu'implanté dans le modèle sous-jacent)

3.1.1.3 Stratégie *ActivityGraph*

La troisième approche consiste à associer un *ActivityGraph* et une méthode. L'usage de l'*ActivityGraph* dans ce contexte s'apparente à l'usage d'un flowchart pour décrire les étapes d'un algorithme. La Figure 18 présente un *activity diagram* qui décrit les étapes majeures de l'exécution de la méthode *Droite::intersection*. Sommairement, l'*ActivityGraph* sous-jacent est constitué d'états de type *ActionState* et de transitions *Transition*. Chaque état *ActionState* est constitué d'une action d'entrée qui est exécutée lorsque l'état est atteint. Lorsque l'action termine, les transitions sortantes de l'état sont éligibles à être tirées. Les conditions de garde sont alors évaluées: la transition tirée est choisie parmi l'ensemble des transitions dont les conditions de garde sont vraies. Les actions prédéfinies dans le métamodèle sont *CreateAction*, *DestroyAction*, *CallAction*, *ReturnAction*, *SendAction*, *TerminateAction*, *UninterpretedAction* et *Action*, la superclasse des actions. *Action* possède l'attribut *recurrence*, de type *IterationExpression*, qui permet de spécifier une boucle régissant

la répétition de l'action. Il existe également des états spéciaux, de type *PseudoState*, qui représentent l'état de départ, les branchements, les fusions, etc. Le métaélément *Transition* relie deux états de l'*ActivityGraph*. Chaque *Transition* possède une condition de garde *Guard*, ainsi qu'une *Action* qu'elle peut déclencher lorsque la transition est tirée.

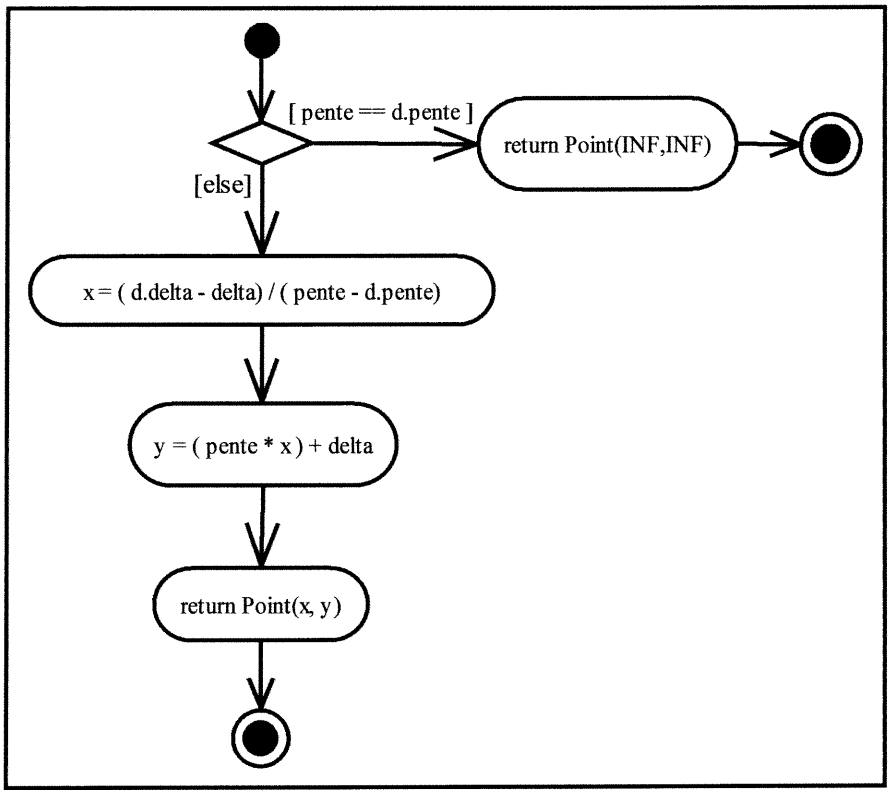


Figure 18: Corps de méthode dans *ActivityGraph*
(tel que vu dans l'*activity diagram*)
(Source: adapté de [Booch_1999])

La Figure 20 présente l'instanciation d'un fragment (voir Figure 19) de l'*ActivityGraph* complet de la Figure 18. Dans la Figure 20, le coeur de la stratégie apparaît dans l'association *BehaviorContext* qui lie un *ActivityGraph* à l'objet *Method*. Le reste de la figure présente les éléments qui constituent l'*ActivityGraph*: les objets *Transition* et l'état *top*. L'état *top* est de type *CompositeState* puisqu'il contient tous les sous-états de l'*ActivityGraph*, c'est-à-dire les instances de *ActionState* et de *PseudoState*. Le premier *PseudoState* correspond à l'état initial de la machine à états. Par convention, la transition sortante est immédiatement tirée et l'état suivant, un

PseudoState de type *choice*[12], est atteint. Cet état et ses deux *Transition* sortantes correspond à l'énoncé *if* du corps de la méthode. La condition de garde de la transition du haut reprend l'expression booléenne du *if*, c'est-à-dire "pente == d.pente". La transition du bas utilise la condition de garde spéciale *else* qui garantit que la transition associée sera tirée si aucune autre transition est tirable. Les états *ActionState* encapsulent les autres énoncés du corps de la méthode dans des instances de la famille *Action* qui déterminent l'action d'entrée de chaque état. Dans le premier cas, l'énoncé "return Point(INF, INF)" est représenté par une instance de *ReturnAction*. La valeur retournée par l'énoncé *return* est spécifiée par son argument actuel *Argument* qui encode l'expression "Point(INF, INF)" dans *Expression*, stockant du même coup le nom du langage à utiliser pour l'évaluation. Dans le second cas, l'action *UninterpretedAction* est choisie parce que l'expression "x = (d.delta - delta) / (pente - d.pente)" ne correspond à aucune des *Action* prédéfinies[13]. Concrètement, l'attribut *UninterpretedAction.script*, qui est de type *ActionExpression*, est utilisé pour stocker l'expression.

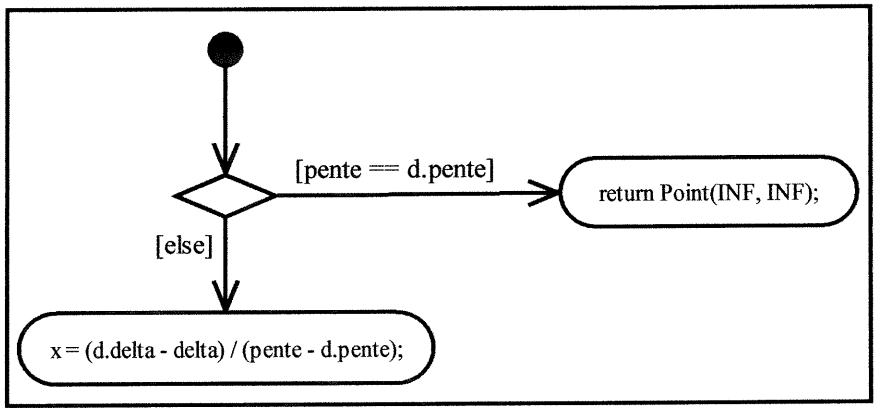


Figure 19: Fragment de l'*ActivityGraph* de la Figure 18

[12] La distinction entre les *PseudostateKind choice* et *junction* est subtile. Nous croyons que *choice* se rapproche plus de l'énoncé *if* dans ce cas-ci.

[13] *AssignmentAction* est mentionné dans [Rumbaugh_1999], mais n'apparaît plus dans le métamodèle de la plus récente spécification [OMG_2000].

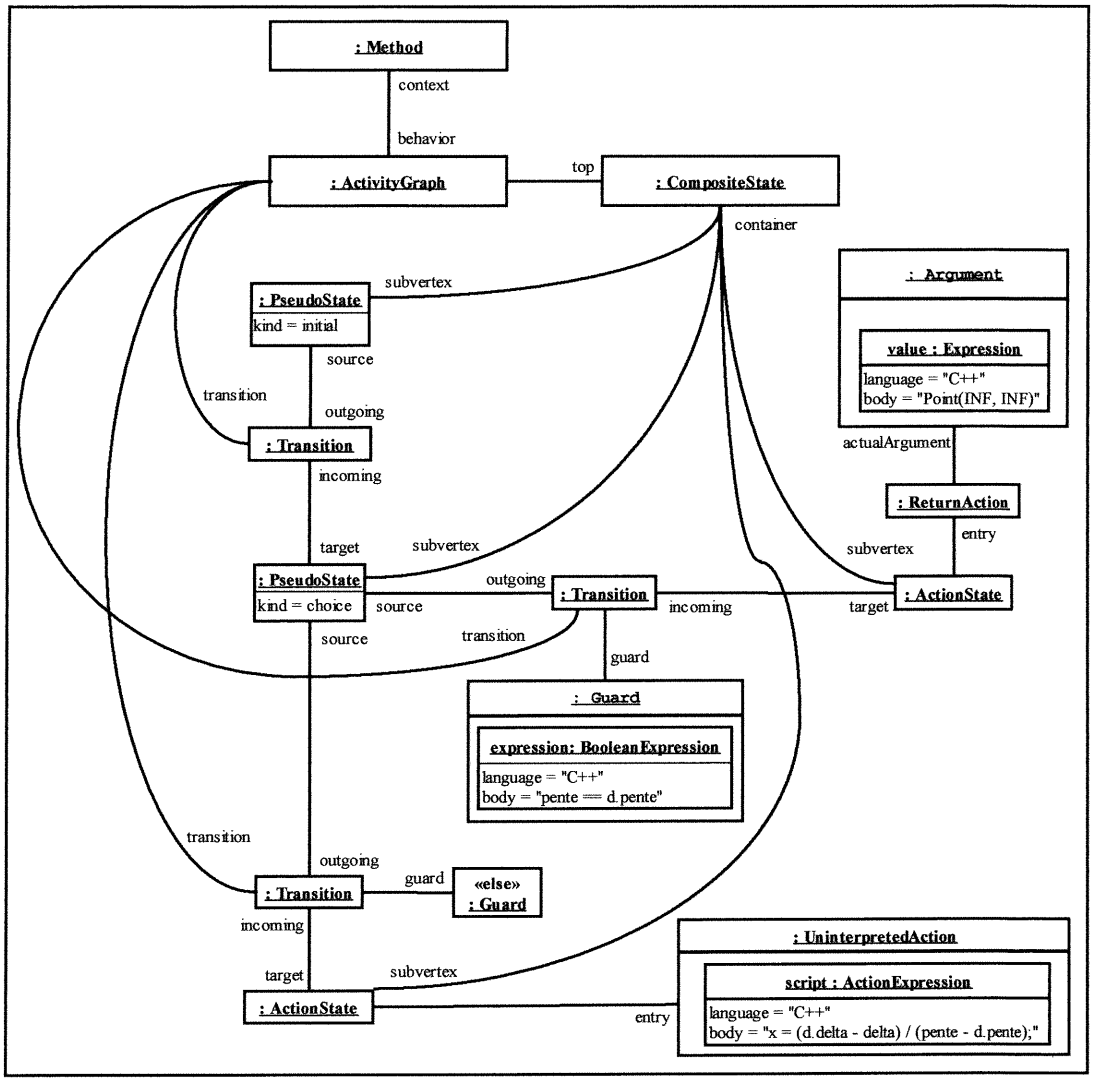


Figure 20: Corps de méthode dans *ActivityGraph*
(Figure 19 telle qu'implantée dans le modèle sous-jacent)

3.1.1.4 Autres stratégies

Les concepts associés au métaélément *Collaboration* et ses diagrammes (i.e., le *collaboration diagram*, l'*interaction diagram* et le *sequence diagram*) peuvent être utilisés pour modéliser la structure et la dynamique d'une société d'objets existant au sein d'une méthode, mais cette approche n'est pas généralisable à des fragments de code arbitraires dans lesquels les objets n'apparaissent pas.

3.1.2 UML Action Semantics

En novembre 1998, l'OMG lance un appel à propositions [OMG_1998] à ses membres intitulé *Action Semantics for the UML*. Cet appel vise à corriger le manque de précision sémantique des descriptions du comportement des métaéléments (e.g., *Action, Operation)*. Présentement, le comportement d'actions et d'opérations est décrit informellement à l'aide de textes arbitraires qui ne sont pas définis dans l'univers UML. Cette situation rend difficile, voire impossible, le partage de descriptions comportementales entre analystes et entre outils. L'appel à propositions cherche donc une solution indépendante de tout langage de programmation qui permet de spécifier le sens et la structure des actions du *statechart diagram* ainsi que des opérations du *class diagram*. L'OMG suggère qu'un tel raffinement d'UML permettra éventuellement les vérifications formelles et la réutilisation de haut niveau.

La seule réponse [UASC_2000] reçue par l'OMG provient du UML Action Semantics Consortium[14]. Cette proposition est présentement en cours de révision et sera formellement remise à l'OMG à la fin du mois de mars 2001. La solution propose des révisions majeures au métamodèle UML, notamment le remplacement de la hiérarchie *Action* dans le package *UML.BehavioralElements.CommonBehavior* et l'introduction d'une couche sémantique sous-jacente (i.e., *Execution Model*) qui définit les concepts primitifs pour spécifier les effets de l'exécution des actions. Le Tableau 3 résume les familles d'actions introduites dans le métamodèle révisé, et le Tableau 4 présente les métaéléments associés à l'*Execution Model*. En résumé, l'approche proposée définit l'infrastructure conceptuelle pour construire des modèles dynamiques plus précis et formels mais exige la création de modèles de plus haut niveau pour modéliser les concepts concrets de langages de programmation (e.g., opérations arithmétiques).

[14] Le UML Action Semantics Consortium est constitué principalement de I-Logix, Kabira, Rational, Kennedy Carter, Project Technology et Telelogic. WWW: <http://www.umlactionsemantics.org>, 2001-03-16.

Famille	Exemples de métaéléments	Commentaires
Action Foundation	*Action*, *ControlFlow*, *DataFlow*, *PrimitiveAction*, *Procedure*, *InputPin*, *OutputPin*	Concepts de base pour la hiérarchie *Action*. *Procedure* remplace *Method.body*.
Composite Actions	*GroupAction*, *ConditionalAction*, *LoopAction*, *Variable*, *Clause*	Actions complexes composées de sous-actions et de structures de contrôle.
Read and Write Actions	*AttributeAction*, *VariableAction*, *LinkAction*, *CreateObjectAction*, *DestroyObjectAction*, *AddVariableValueAction*, *ClearVariableAction*, *ReadVariableAction*, *WriteVariableAction*	Actions pour la lecture et l'écriture d'objets, d'attributs, de variables et de liens.
Computation Actions	*ApplyFunctionAction*, *CodeAction*, *LiteralValueAction*, *NullAction*	Actions qui transforment des valeurs d'entrée en valeurs de sortie.
Collection Actions	*FilterAction*, *IterateAction*, *MapAction*, *ReduceAction*	Actions qui appliquent une sous-action aux éléments d'une collection.
Messaging Actions	*InvocationAction*, *ReceiveAction*, *ReplyAction*	Actions qui décrivent l'échange de messages entre objets.
Exception Actions	*HandlerAction*, *RaiseAction*	Actions qui génèrent ou qui traitent les exceptions et les interruptions.

Tableau 3: Familles d'actions dans le nouveau métamodèle proposé dans UML Action Semantics ([UASC_2000])

Famille	Exemples de métaéléments	Commentaires
Foundation Execution Model	*Identity*, *Change*, *Step*, *Snapshot*, *History*, *ExecutionSnapshot*	Concepts de base du modèle d'exécution.
Instance and Link Execution	*AttributeValue*, *DataValueIdentity*, *InstanceIdentity*, *LinkIdentity*, *ObjectIdentity*, *ObjectSnapshot*	Représentation des instances et des liens dans le modèle d'exécution.
Action Foundation Execution	*ActionExecution*, *PinValue*, *PrimitiveActionExecution*, *ProcedureExecution*	Modèle d'exécution des actions en général.
Composite Action Execution	*ClauseExecution*, *VariableExecution*, *LoopExecutionSnapshot*	Modèle d'exécution des actions complexes.
Read and Write Action Execution	*CreateObjectExecution*, *DestroyObjectExecution*, *ReadSelfExecution*, *ReadAttributeExecution*	Modèle d'exécution des actions de lecture et d'écriture.
Computation Action Execution	*CodeExecution*, *LiteralValueExecution*, *ApplyPrimitiveFunctionExecution*, *NullActionExecution*	Modèle d'exécution des actions de calcul.
Collection Action Execution	*FilterExecution*, *IterateExecution*, *MapExecution*, *ReduceExecution*	Modèle d'exécution des actions de collection.
Messaging Action Execution	*InvocationPacket*, *SynchronousActionExecution*, *ReplyPacket*	Modèle d'exécution des actions de messagerie.
Exception Action Execution	à paraître dans la prochaine révision	
State Machine Execution	*ChangeOccurrence*, *SignalOccurrence*, *TimeOccurrence*, *StateMachineExecutionSnapshot*	Raffine le modèle d'exécution d'une machine à états.

Tableau 4: Familles associées à l'*Execution Model* dans le nouveau métamodèle proposé dans UML Action Semantics ([UASC_2000])

3.2 Communauté Développement

3.2.1 Ateliers de génie logiciel

Les ateliers de génie logiciel (AGL) sont conçus pour faciliter l'analyse, la conception et le développement de systèmes logiciels. Puisque la grande majorité des AGL supportent maintenant le formalisme UML, nous avons choisi de rapporter l'état de l'art de ces outils.

Outil	Origine	Référence
ArgoUML	U. California, Irvine	[ArgoUML]
Bold for Delphi	Boldsoft	[BoldSoft]
Bridgepoint	Project Technology	[ProjTech]
Elixir CASE	Elixir Technology	[Elixir]
Enterprise Architect	SparxSystems	[Sparx]
GDPro	Embarcadero	[Embarcadero]
JVision	Object Insight	[ObjectInsight]
Magicdraw UML	NoMagic	[NoMagic]
MetaEdit+	MetaCase	[MetaCase]
Objecteering	Softeam	[Softeam]
ObjectiF	MicroTool	[MicroTool]
Real-time Studio Professional	Artisan Software	[Artisan]
Rhapsody	i-Logix	[iLogix]
Rose	Rational	[Rational]
SoftModeler	Softera	[Softera]
SoftwareThroughPictures UML	Aonix	[Aonix]
StructureBuilder	WebGain	[WebGain]
System architect 2001	Popkin	[Popkin]
Tau UML Suite	Telelogic	[TelelogicUML]
Together Control Center	TogetherSoft	[Together]

Tableau 5: AGL populaires supportant le formalisme UML

Notre choix s'est arrêté sur les AGL couramment les plus populaires[15]. Le Tableau 5 présente les candidats qui ont été investigués. L'évaluation des outils s'est faite à partir des versions les plus récentes disponibles au mois de mars 2001. Pour chaque

[15] La popularité selon notre expérience et le service Google. Source: <http://directory.google.com/Top/Computers/Software/Object-Oriented/Methodologies/UML/Tools/>, 2001-03-17.

outil, les informations pertinentes ont été obtenues des sources suivantes: la version d'évaluation de l'application, les manuels techniques, les guides d'utilisation, les fiches descriptives, le site WWW.

Notre étude révèle que ces AGL atteignent aujourd'hui une maturité intéressante. La majorité incorporent tous les diagrammes de la plus récente spécification UML. Les concepts de *Stereotype*, *TaggedValue* et *Constraint* sont aussi intégrés dans la création de modèles. Concrètement, les utilisateurs peuvent créer leur propres métaéléments réutilisables de type *Stereotype*, et ils peuvent raffiner un métaélément arbitraire en lui associant des *TaggedValue* et des *Constraint* à volonté; en particulier, l'AGL Objecteering déploie une infrastructure conçue explicitement pour la création, l'exploitation et la gestion de profils. La génération de code à partir du modèle UML ainsi que l'opération en sens inverse sont possibles; certains AGL offrent la synchronisation du code source et du modèle (e.g., *roundtrip engineering*) pour assurer une correspondance exacte entre le modèle et l'implantation réelle d'un système. L'interopérabilité des outils s'effectue principalement par l'importation et l'exportation de documents XMI, mais certains exposent des interfaces (i.e., API) qui donnent accès aux objets de plusieurs niveaux: l'AGL en général, l'interface graphique de l'AGL, les diagrammes du modèle, le code source de l'implantation du modèle, etc. En termes de fonctions avancées, certains AGL offrent une ou plusieurs des possibilités suivantes:

- génération automatique de documentation
- génération automatique de scénarios de tests
- aide au contrôle de qualité (cohérence, métriques, styles, etc.)
- visualisation des comportements à partir des diagrammes
- offre de conseils et critique active lors de la modélisation
- association d'hyperliens arbitraires aux diagrammes ou leur contenu
- intégration avec des outils complémentaires (e.g., environnement de développement, éditeur, compilateur, gestionnaire de configuration, etc.)
- modélisation avancée avec patrons de conception, librairies pré-configurées
- et autres…

Parmi ces outils, seuls les AGL Together et GDPro se rapprochent conceptuellement des objectifs de notre travail. Ces deux AGL basent la création de modèles sur un métamodèle accessible et manipulable qui intègre les détails du code source avec les autres éléments du modèle avec des métaéléments explicites comme *CodeBlock*, *Statement* et *FunctionCallExpression*. Dans chaque cas cependant, le métamodèle est propre à l'outil et n'est en aucun cas l'enrichissement du métamodèle UML officiel.

3.2.2 Recherche

La recherche axée sur l'utilisation d'UML dans la conception de logiciels est encore jeune. Cette section présente le projet FUJABA qui a retenu notre attention en raison de son approche intéressante pour marier UML et la programmation. Ensuite, nous donnons un aperçu des autres travaux de recherche qui ont contribué indirectement à nos résultats.

3.2.2.1 FUJABA

Le projet FUJABA (From UML to Java And Back Again) [Köhler_2000] propose de créer un langage de programmation visuel à partir des *class diagram* et des diagrammes de comportement (i.e., *statechart diagram*, *collaboration diagram, activity diagram*). FUJABA s'appuie sur le Story Driven Modeling [Jahnke_1998], une technique de spécification pour les comportements complexes qui capte l'évolution des objets avec une série de graphes qui représentent les configurations instantanées (*snapshot*) d'un scénario donné. Concrètement, le projet FUJABA comprend un éditeur de diagrammes qui permet de spécifier les composantes du système avec le *class diagram* et de définir le comportement en combinant selon des règles précises les *statechart*, *activity* et *collaboration diagrams*.

L'approche FUJABA s'intéresse avant tout à la spécification par graphes de modèles dynamiques et à la génération automatique du code qui met en oeuvre ces modèles. Dans son état actuel, la rétroconception de code source arbitraire n'est pas possible

puisque FUJABA impose certaines conventions de nommage d'entités ainsi que des styles de programmation pré-définis.

3.2.2.2 Autres travaux de recherche

Notre travail s'inspire indirectement des travaux de recherche provenenant de domaines plus ou moins connexes. La liste suivante suggère l'étendue des champs d'activité qui ont été sondés:

- [Belaunde_1999]
  Une approche pour l'implantation d'un dépôt de données UML convivial et flexible.
- [Rasmussen_2000]
  La mise en oeuvre d'un AGL pour la modélisation UML qui impose le respect de la sémantique du métamodèle.
- [Abdurazik_2000]
  Une évaluation de UML en tant que langage de description d'architecture (Architecture Description Language, ADL). UML est discuté dans le contexte des ADL Rapide, Darwin, Wright, Aesop, ACME, UniCon, MetaH, ControlH, C2, ROOM et SADL.
- [Badros_2000]
  La création du langage JavaML pour baliser en XML le code source Java.
- [Knuth_1984]
  La programmation littéraire (*literate programming*) de Knuth: le mariage de la documentation et du code d'une application.
- [Aigner_2000]
  La description du Stanford University Intermediate Format (SUIF) , une représentation intermédiaire de logiciels pour les ateliers de compilation.
- [GCC_2001]
  Une description de la représentation interne d'un programme C / C++ dans le compilateur GCC.

En résumé, la modélisation du corps d'une méthode n'est pas un problème nouveau. Cependant, cette modélisation en termes du métamodèle UML représente un défi qui n'a pas été relevé à date.

3.3 Communauté Maintenance

3.3.1 Outils

La maintenance de systèmes logiciels repose sur une composition stratégique des activités de rétroconception, de compréhension et de réingénierie. Il s'ensuit que le coffre d'outils de cette communauté renferme une panoplie d'applications plus ou moins spécialisées qui sont mises en service selon la tâche à accomplir. Une sélection typique de ces outils apparaît dans le Tableau 6 (outils commerciaux) et le Tableau 7 (outils expérimentaux).

L'étude sommaire des outils plus récents révèle l'impact négligeable de l'effort UML dans la communauté de maintenance jusqu'à maintenant, autant du point de vue modélisation de systèmes que du côté notation graphique. En effet, chaque outil utilise son propre métamodèle pour la construction d'une représentation intermédiaire du logiciel à partir du code source. De plus, la présentation graphique de l'information (e.g., hiérarchie de classes, dépendances fonctionnelles, etc.) s'effectue selon une notation qui varie d'un outil à l'autre.

Outil	Origine	Référence
CodeSurfer	GrammaTech	[GrammaTech]
Code Integrity Management System	Programming Research Ltd.	[ProgResearch]
Concerto 2 / AUDIT	Sema	[Sema]
Discover	Upspring Software	[Upspring]
Genitor Object Construction Suite	Starbase Corp.	[Starbase]
HindSight	Integrisoft	[IntegriSoft]
Imagix 4D	Imagix	[Imagix]
Reengineer	McCabe & Associates	[McCabe]
ASG-Insight	ASG Software Solutions	[ASG]
InstantQA, Reasoning5, Refine	Reasoning Systems	[Reasoning]
SNiFF+	Wind River Systems	[WindRiver]
Telelogic Tau Logiscope	Telelogic	[Telelogic]
TraceIT, e-TraceIT	Data Integrity	[DataIntegrity]

Tableau 6: Sélection d'outils commerciaux dans la communauté de maintenance

Outil	Origine	Référence
COBOL/SRE	Andersen Consulting	[Ning_1994]
Acacia	AT&T	[Chen_1998]
Ajax	Carnegie Mellon University	[Ajax]
Arcadia	Arcadia Consortium	[Kadia_1992]
Arch	Siemens	[Schwanke_1991]
Aria	AT&T	[Devanbu_1996]
CIA	AT&T	[Chen_1990]
CodeCrawler	Universität Bern	[CodeCrawler]
Dali	Software Engineerig Institute, Carnegie Mellon University	[Kazman_1999]
Datrix	Bell Canada	[Mayrand_1996]
DECODE	University of Hawaii at Manoa	[Quilici_1995]
DESIRE	Microelectronics and Computer Technology Corporation	[Biggerstaff_1989]
FIELD	Brown University	[Reiss_1995]
GENOA	AT&T	[Devanbu_1992]
GRASPR	MIT	[Wills_1992]
Lackwit	Carnegie Mellon University	[OCallahan_1997]
Lemma	IBM	[Mays_1996]
ManSART	MITRE	[Chase_1996]
Microscope	Hewlett-Packard	[Ambras_1988]
OMEGA	Stanford University	[Linton_1984]
PAT	n/d	[Harandi_1990]
Pattern-Lint	University of Illinois at Urbana Champaign	[Sefika_1996]
ProDAG	Arcadia Consortium	[Richardson_1992]
Program Explorer	IBM	[Lange_1995]
RevEngE	University of Toronto, McGill University, University of Victoria, IBM Center for Advanced Studies	[Whitney_1995]
Rigi	University of Victoria	[Müller_1993]
RMTool	University of British Colombia	[Murphy_1995]
SCAN	University of Michigan	[Al-Zoubi_1991]
SCRUPLE	University of Michigan	[Paul_1994]
SCED	Tampere University of Technology	[Koskimies_1998]
Scene	Tampere University of Technology	[Koskimies_1996]
Shimba (Rigi + SCED)	Tampere University of Technology, University of Victoria	[Systä_2000]
Software Bookshelf	IBM Center for Advanced Studies, Consortium for Software Engineering Research	[Finnigan_1997]

Tableau 7: Sélection d'outils expérimentaux dans la communauté de maintenance

3.3.2 Recherche

3.3.2.1 Modélisation de logiciel

La majorité des outils du Tableau 6 et du Tableau 7 effectuent leur travail sur un modèle du logiciel qui est construit à partir des informations extraites du code source. Ce modèle est généralement conservé (e.g., fichiers, dépôt de données, etc.) afin qu'il puisse être réutilisé à volonté et partagé parmi les membres de l'équipe d'analyse. La stratégie d'abstraire le logiciel dans un modèle comporte deux avantages techniques importants: l'uniformité et l'efficacité.

L'uniformité découle du fait que les concepts sémantiques exprimés dans le code source sont modélisés de la même façon peu importe la syntaxe particulière d'un langage de programmation: par exemple, le *PERFORM UNTIL* de COBOL et le *do...while* de C représentent le même concept dans le modèle. L'uniformité facilite grandement la réutilisation technologique car elle permet la conception d'outils génériques.

Le deuxième avantage, l'efficacité, est attribuable d'une part à la compacité de la représentation d'informations et d'autre part aux pré-traitements qui sont encapsulés dans le modèle. Pour mieux comprendre la compacité, il suffit de comparer les ressources nécessaires pour encoder l'information: par exemple, le modèle peut utiliser un vecteur de seulement deux bits pour encoder les trois niveaux d'accès représentés par les chaînes de caractères "public", "protected" et "private" dans le fichier source. En ce qui concerne les pré-traitements, considérons l'effort requis pour déterminer la hiérarchie de classes d'un système C++. Puisqu'il n'existe aucune consigne précise pour la disposition des déclarations de classes, tous les fichiers sources doivent être lus pour découvrir les classes et leurs positions dans la hiérarchie. Par contre, un modèle peut encapsuler très efficacement cette information: avec l'aide de pointeurs, par exemple, permettant ainsi la navigation directe parmi les classes d'une hiérarchie.

Les formalismes d'abstraction de logiciels qui ont été explorés jusqu'à présent reposent fondamentalement sur le concept de graphes. Les formalismes diffèrent entre eux principalement au niveau de l'information représentée par les noeuds et les arcs du graphe. Le Tableau 8 présente divers types de formalismes en décrivant le genre d'information qui s'y trouve et en donnant quelques exemples représentatifs. De façon générale, les formalismes fournissant un niveau d'abstraction plus élevé apparaissent en premier dans le tableau.

Graphe	Noeud	Arc	Exemples
graphe d'architecture	composante de haut niveau du système	connexion	Wright [Allen_1997]
graphe d'entités-relations	entité du logiciel	relation	Acacia [Chen_1998] Famix [Demeyer_1999]
graphe de flot de données	producteur, consommateur, dépôt, traitement	donnée	Flow graph [Wills_1992] Plan [Rich_1990]
graphe de flot de contrôle	pas d'exécution	transfert de contrôle	
graphe d'appels	fonction	invocation de fonction	
graphe sémantique abstrait (abstract semantic graph)	concept du langage de programmation	composition, type, opérande, etc.	Reprise [Rosenblum_1991] Gen++ [Devanbu_1994] Alf [Murray_1992]
arbre syntaxique abstrait (abstract syntax tree)			

Tableau 8: Quelques formalismes de représentation de logiciel

La grande variété de représentations témoigne de la diversité des besoins d'information des outils de maintenance. Malheureusement, l'incompatibilité de ces outils spécialisés constitue une entrave majeure au partage et à la réutilisation de résultats obtenus antérieurement. Reconnaissant les bénéfices de l'intégration des divers formalismes, plusieurs chercheurs ont tenté de définir un modèle unifié: Telos [Mylopoulos_1990], Large Software System Information Environment (LaSSIE) [Devanbu_1991], Common Object-based Re-engineering Unified Model (CORUM) [Woods_1998, Kazman_1998] et Integrated Intermediate Representation (IIR) [Koschke_1998] en sont des exemples. Malgré ces efforts, aucun consensus important n'a été atteint à date, et la fragmentation des outils en familles incompatibles perdure.

Le faible intérêt pour UML manifesté par la communauté de maintenance malgré son succès incontestable dans la communauté de développement s'explique en partie par l'article de Demeyer *et al.* [Demeyer_1999]. Après avoir considéré le métamodèle UML pour le projet de réingénierie FAMOOS, les auteurs estiment que la spécification courante d'UML (i.e., UML 1.1 lors de la rédaction de l'article) n'est pas une solution adéquate pour essentiellement trois raisons. Premièrement, la rétroconception du code source vers un modèle UML n'est que partiellement définie, augmentant ainsi le risque que les modèles créés soient ambigus et donc plus difficilement partageables et compréhensibles. Deuxièmement, les concepts concrets du langage de programmation n'ont pas toujours d'équivalents directs dans le métamodèle; par exemple, l'invocation de méthode et l'accès à une variable ne sont pas modélisés explicitement par le métamodèle UML. Et troisièmement, les mécanismes d'extensions du langage menacent l'interopérabilité des outils en permettant les modifications arbitraires au métamodèle.

Toutefois, ces conclusions ne sont pas unanimes. L'équipe du projet SPOOL admet l'existence de ces difficultés dans [Schauer_2001] mais soutient qu'elles ne sont pas insurmontables et qu'elles représentent plutôt un défi à relever par la communauté de rétroconception. De son côté, le chercheur Pinali propose un environnement de rétroconception reposant sur le métamodèle UML [Pinali_1999] pour la récupération de patrons de conception. Le besoin de modéliser certains détails d'implantation des méthodes pour effectuer la récupération motive Pinali à enrichir le métamodèle UML avec des métaéléments comme *Code*, *Literal*, *LocalVar*, *TypeCast*, *TypeRef*, *Ref*, *Assign*, *UnOp* et *BinOp*. Cependant, les structures de contrôle (e.g., *if*, *for*, *while*, *switch*, etc.) ne font pas partie des extensions car elles ne sont pas utilisées dans ce projet.

3.3.2.2 Échange de données

L'incompatibilité des métamodèles utilisés par les familles d'outils de maintenance complique leur interopérabilité mais ne l'exclut pas complètement. L'utilisation d'un mécanisme d'échange de données peut, en principe, permettre la collaboration d'outils

provenant de familles différentes. Techniquement, ce mécanisme d'échange doit spécifier d'une part la forme des échanges (e.g., la grammaire et l'encodage d'un document) et d'autre part la sémantique des données échangées (e.g., le métamodèle). Plusieurs tentatives de solutions d'échange ont été proposées au cours de la dernière décennie: CDIF [CDIF_1994], RSF [Wong_1998], Famix [Demeyer_1999], GraX [Ebert_1999] et TAXForm [Bowman_2000] en sont des instances plus populaires parmi d'autres. Malgré (ou à cause de) ce choix, aucune de ces technologies n'a réussie à récolter l'élusive approbation majoritaire jusqu'à présent.

Frustrés de leur isolement et désireux de mettre un terme à l'incompatibilité de leurs outils, une vingtaine d'individus de l'industrie et du milieu académique se sont réunis lors de l'atelier WoSEF[16] [Sim_2000] en juin 2000 pour esquisser la définition d'un format d'échange qui serait utile à tous. Après avoir considéré les technologies d'échange existantes, incluant la famille UML (i.e., MOF, UML, XMI), la majorité des participants se sont entendus pour définir le nouveau Graph Exchange Language (GXL) [Holt_2000], un langage XML conçu explicitement pour la description de graphes. Ayant réussi à formaliser la syntaxe et l'encodage de GXL [Schürr_2001] lors d'un atelier en janvier 2001 à Dagstuhl[17], le groupe a lancé les travaux consistant à élaborer un (ou plusieurs) métamodèles pour décrire les systèmes logiciels à plusieurs niveaux d'abstraction. Un premier métamodèle embryonnaire du nom provisoire de *Dagstuhl Middle Model* [Lethbridge_2001] est couramment à l'étude.

---

[16] L'atelier WoSEF (Workshop on Standard Exchange Format) a été tenu lors de la conférence ICSE 2000 en Irlande.

[17] L'atelier *Interoperability of Reengineering Tools (Seminar No. 01041, Report No. 296)* a été tenu du 21 au 26 janvier 2001, au centre Schloss Dagstuhl, dans la ville de Wadern en Allemagne.
Source: <http://www.dagstuhl.de/DATA/Reports/01041/>, 2001-03-17.

Chapitre 4: Énoncé du problème

Ce chapitre a deux objectifs principaux. Le premier consiste à préciser la nature exacte du problème à résoudre (i.e., la modélisation du corps d'une méthode). Le second consiste à proposer les caractéristiques de la solution recherchée (i.e., le profil UML). La présentation est divisée en quatres sections majeures: les choix préliminiaires, les attentes techniques, les contraintes et les besoins.

La section 4.1, Choix préliminaires, définit l'étendue du problème en spécifiant quels styles et langages de programmation sont considérés, comment les pré-traitements et les raccourcis syntaxiques sont gérés et quels éléments sont hors de portée.

La section suivante rappelle les deux attentes techniques qui ne sont pas définies explicitement dans la spécification UML mais qui sont utiles à la solution proposée: la modélisation des types dérivés et l'attribution d'identificateurs uniques aux éléments d'un modèle.

La section 4.3, Contraintes, discute des facteurs qui influencent les caractéristiques de la solution proposée. En particulier, les contraintes associées au langage UML et à l'équilibre de fonctionnalité sont présentées.

La dernière section spécifie les besoins que doit satisfaire la solution proposée. Ces besoins sont répartis selon deux thèmes majeurs qui caractérisent le profil recherché: son niveau d'abstraction et son contenu.

4.1 Choix préliminaires

Cette section présente les décisions d'ordre technique qui précisent la portée de la solution proposée. On y retrouve des discussions sur les styles et langages de

programmation visés, l'exclusion des artefacts de pré-traitement (*preprocessing*), la modélisation des raccourcis syntaxiques (*syntactic sugar*) et finalement les éléments exclus en raison de leur nature trop générale.

4.1.1 Styles et langages de programmation

Un système logiciel peut être implanté de nombreuses façons. Une décision importante consiste à déterminer le style de programmation (e.g., impératif, orienté objet, fonctionnel, logique, etc.) qui est approprié pour le projet. Vient ensuite le choix du langage de programmation proprement dit: C, Java, C++, Smalltalk, Scheme, Prolog, et ainsi de suite. De plus, on retrouve parfois dans un même système des éléments qui sont implantés dans des styles et langages différents: par exemple, certains programmes C++ incorporent parfois des fragments en assembleur pour des raisons de performance.

La solution présentée dans ce mémoire est applicable aux styles de programmation impératifs et orientés objet, avec une pertinence particulière pour les langages C, C++ et Java. La décision de restreindre la solution à un sous-ensemble de styles et de langages de programmation est motivée premièrement par les besoins du projet SPOOL, et deuxièmement par la contrainte d'équilibre discutée dans la section 4.3.2.

4.1.2 Pré-traitements

Les langages C et C++ font appel à la notion de pré-traitement (i.e., *preprocessing*) lors de la compilation des fichiers sources. Le *préprocesseur* est un outil qui est automatiquement invoqué avant le compilateur pour construire à partir de fichiers sources un *translation unit*, c'est-à-dire un fichier complet prêt à être compilé. Les transformations à effectuer sont spécifiées par des directives spéciales insérées à même les fichiers sources et comprennent l'inclusion de fichiers (e.g., #include), l'exclusion de sections du code (e.g., #ifdef, #ifndef) et la substitution de symboles (communément appelés *macros*) par leur définition textuelle (e.g., #define). Puisque ce mécanisme d'aide à la compilation n'affecte d'aucune façon la définition ultime du

corps d'une méthode, notre solution n'en tient pas compte. Le profil proposé ne modélise donc pas les directives de compilation et les macros qui peuvent se retrouver dans un fichier source qui n'a pas été pré-traité.

4.1.3 Raccourcis syntaxiques

Les raccourcis syntaxiques (i.e., *syntactic sugar*) facilitent l'utilisation d'un langage en permettant l'abrégement d'expressions qui sont textuellement encombrantes dans le code source. L'Exemple 2 illustre l'utilisation d'un raccourci syntaxique pour alléger l'application répétée de l'opérateur *ostream::operator<<()* dans le langage C++.

expression originale

```
(cout.operator<<("x = ")).operator<<(x);
```

raccourci syntaxique

```
cout << "x = " << x;
```

Exemple 2: Raccourci syntaxique pour *operator<<()* en C++

La solution proposée traite les expressions abrégées comme si elles étaient non-abrégées puisque l'usage de raccourcis syntaxiques n'affecte pas l'implantation effective de la méthode, c'est-à-dire les instructions machines qui sont exécutées lorsque la méthode est activée.

4.1.4 Considérations d'ordre global

Les commentaires, l'espacement (e.g., lignes vides, tabulations, espaces), les coordonnées (e.g., numéro de ligne et de colonne) ainsi que nombreuses autres particules d'information (e.g., version, auteur, dates de création ou de modification, etc.) sont d'un intérêt indéniable pour la rétroconception, la compréhension et la réingénierie. Toutefois, l'intégration de ces éléments aux modèles de systèmes constitue une extension du métamodèle UML d'ordre global qui n'est pas exclusive

aux corps de méthodes. Pour cette raison, la solution proposée ne vise pas à inclure ces détails.

4.2 Attentes techniques

La solution que nous proposons s'appuie sur certaines pré-conditions d'ordre technique concernant la modélisation UML de systèmes logiciels. Les sous-sections suivantes présentent ces attentes: la modélisation des types dérivés et l'emploi d'identificateurs uniques.

4.2.1 Types dérivés

Les langages de programmation comme C et C++ permettent la définition de types *dérivés*. Ces types sont dits dérivés parce qu'ils sont construits en conjuguant un constructeur de type avec un type existant. Par exemple, *pointeur* n'est pas un type proprement dit; on le combine avec un type T afin de spécifier le type *pointeur au type T*. Le *tableau* et la *référence* sont d'autres exemples de constructeurs de types dérivés. Le métamodèle UML ne possède pas de métaéléments prédéfinis pour modéliser directement les types dérivés mais il existe toutefois des stratégies (e.g., celle présentée à l'annexe A) pour les représenter dans un modèle. La solution que nous proposons présume l'utilisation d'une stratégie pour modéliser les types dérivés.

4.2.2 Identificateurs uniques

Le concept de portée dans les langages de programmation permet la définition d'entités identiques à condition qu'elles ne soient pas visibles au même moment lors de l'exécution du logiciel. Par exemple, les variables de service *i*, *j* et *k* de type *integer* utilisées dans les boucles apparaissent maintes fois dans le code source d'un système. L'identification non ambiguë d'une de ces variables nécessite la mention de son nom, de son type, ainsi que de la portée dans laquelle elle est définie. Malheureusement, l'identification d'une portée n'est pas directement possible et nécessite l'usage

d'éléments supplémentaires comme le nom d'un fichier, le numéro de ligne, et ainsi de suite. Pour simplifier cette situation, une stratégie qui attribue un identificateur unique à tous les éléments du système est utilisée. De plus, les éléments du modèle qui n'apparaissent pas explicitement dans le code source mais qui sont néanmoins nécessaires pour sa modélisation (e.g., la définition des types primitifs d'un langage) sont aussi identifiés par un identificateur unique. La solution que nous proposons présume que tous les éléments du modèle possèdent un identificateur unique.

4.3 Contraintes

L'extension proposée du métamodèle UML peut prendre plusieurs formes qui sont a priori aussi valides les unes que les autres mais qui en pratique posent divers problèmes. Cette section présente les contraintes retenues dans ce travail qui permettent de guider l'élaboration de la solution.

4.3.1 UML

4.3.1.1 Familiarité

La grande popularité d'UML signifie que sa notation, sa terminologie et son approche en général deviennent de plus en plus familières auprès d'un nombre croissant d'adeptes, facilitant ainsi la communication efficace d'idées et de descriptions de logiciels. Il s'avère donc important de respecter dans la mesure du possible la philosophie et le style UML (e.g., nomenclature de métaéléments) lors de la définition d'un profil afin de minimiser l'écart "culturel" qui pourrait exister vis-à-vis le métamodèle UML déjà bien établi.

4.3.1.2 Stabilité

L'initiative UML Action Semantics (décrite à la section 3.1.2) laisse présager des remaniements majeurs de la famille *Action* lors des prochaines révisions du métamodèle. À toutes fins pratiques, une éventuelle restructuration de cette hiérarchie

du métamodèle exigerait au minimum la mise-à-jour, sinon la redéfinition complète de tout profil qui en dépendrait. Afin de réduire les risques associés aux modifications du groupe Action, le profil proposé ne doit pas comporter de dépendances sur les membres de cette famille.

4.3.2 Équilibre de fonctionnalités

L'élaboration d'une solution pratique exige un certain équilibre entre des fonctionnalités qui se nuisent mutuellement: la généralité, l'expressivité et l'efficacité.

4.3.2.1 Généralité

Un avantage découlant de l'utilisation de modèles est l'indépendance vis-à-vis les particularités syntaxiques des langages de programmation (voir la discussion sur l'uniformité à la section 3.3.2.1). En représentant les composants fondamentaux des logiciels de manière uniforme, la modélisation permet à un seul outil générique de remplacer une famille d'outils spécialisés pour chacun des langages utilisés. Selon cette perspective, l'extension proposée du métamodèle doit être suffisamment abstraite pour représenter les divers concepts pertinents à implantation de méthodes qui se retrouvent dans les langages comme C, C++ et Java.

4.3.2.2 Expressivité

Le succès des activités de rétroconception, de compréhension et de réingénierie dépend en grande partie de la richesse en information du modèle du logiciel. Par exemple, les invocations dans l'Exemple 3 sont similaires mais elles représentent l'appel de deux méthodes complètement différentes. La décision d'invoquer telle ou telle autre méthode repose sur l'identification précise du type de l'argument.

définitions de deux fonctions de même nom

```
char  f(int   a) { return 'y';}
float f(float a) { return g(-a);}
```

invocations similaires...

```
f(1);          // f(int)
f(1.0);        // f(float)
```

Exemple 3: Invocations de fonctions basées sur les types d'arguments

Comme le démontre cet exemple, la solution proposée doit permettre une modélisation aussi détaillée que possible.

4.3.2.3 Efficacité

Par définition, le modèle d'un système logiciel se veut une synthèse de l'ensemble des composantes qui le constituent. En termes concrets, cela signifie que le modèle d'un système contemporain peut avoir à représenter le contenu de centaines ou de milliers de fichiers de code source. Malgré les progrès technologiques qui accroissent régulièrement les capacités de mémoire vive et de stockage, le défi de modéliser compactement l'information renfermée dans des millions de lignes de code source demeure imposant.

La question importante du temps d'exécution s'ajoute au problème d'espace décrit ci-haut, surtout dans le contexte de tâches interactives. À plusieurs occasions, les requêtes et les analyses de l'usager auront à parcourir (conceptuellement, du moins) le système au complet pour accomplir leur travail. Par exemple, déterminer les cas d'utilisation d'une variable globale qui doit être modifiée requiert une recherche à travers tout le système. Ici encore, le choix d'une représentation concise favorise l'efficacité des traitements et la réduction du temps d'exécution.

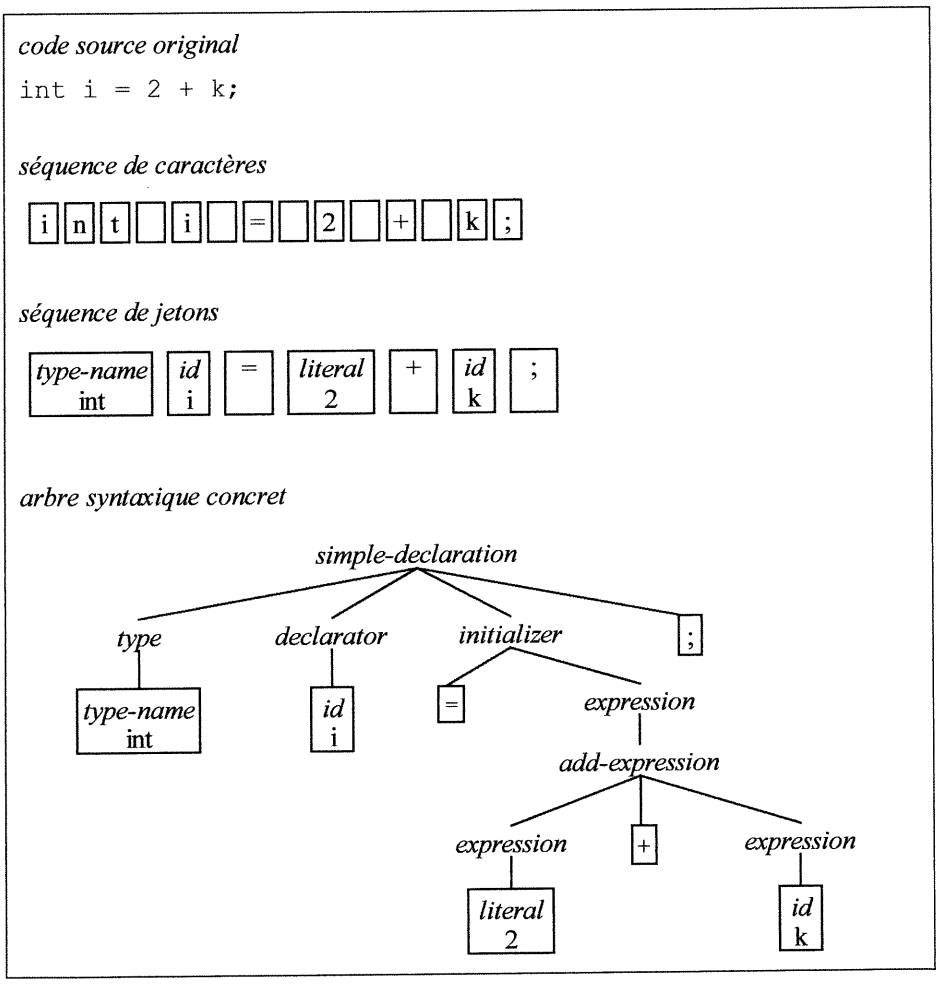
En résumé, les paragraphes précédents suggèrent que la solution proposée doit synthétiser l'information pertinente d'une façon concise pour minimiser la taille du modèle et faciliter l'accès à l'information.

4.4 Besoins

Cette section s'intéresse aux besoins concrets que doit satisfaire la solution proposée. La première sous-section situe le niveau d'abstraction approprié du modèle en rappellant les représentations possibles du corps d'une méthode et en discutant de leurs mérites respectifs. Ensuite, la deuxième sous-section précise le contenu souhaitable du profil recherché.

4.4.1 Niveau d'abstraction

Le corps d'une méthode peut être modélisé de plusieurs façons. L'Exemple 4 illustre trois représentations qui sont rencontrées lors de la compilation du code source.



Exemple 4: Quelques représentations du code source

La première représentation modélise le corps d'une méthode comme une séquence de caractères. Cette approche, utilisée par l'analyseur lexical, contient très peu d'information explicite: on n'y distingue que des caractères.

L'analyseur syntaxique fait appel à un deuxième modèle qui traite le corps de la méthode comme une suite de jetons (e.g., *type-name*, *id*, *literal*) caractérisés au besoin par leur lexème (e.g., int, i, 2). Cette représentation possède déjà plus d'informations accessibles que le modèle précédent: en particulier, les atomes sémantiques du langage (e.g., *type-name*, *id*, *literal*) sont maintenant visibles. Par contre, les relations entre les jetons ne sont pas données.

La troisième approche décrit le corps de la méthode à l'aide de l'arbre syntaxique concret (*parse tree*, *concrete syntax tree*) qui est construit par l'analyseur syntaxique à partir des jetons décrits précédemment. Cette représentation détaillée expose directement la correspondance entre les jetons et les productions grammaticales qui ont été solicitées pour bâtir l'arbre. L'intérêt de ce modèle découle du fait que l'information visible s'approche des concepts primitifs des langages de programmation (e.g., types, variables, énoncés, expressions). Cependant, l'inconvénient de cette représentation provient de la surabondance de noeuds internes (e.g., *expression* dans l'exemple) qui n'ajoutent rien à la sémantique ultime de la méthode. Pour cette raison, une version épurée de cette représentation nommée *arbre syntaxique abstrait* (*abstract syntax tree*) est utilisée. Cet arbre compact sert souvent de base pour des représentations encore plus riches: par exemple, le *graphe sémantique abstrait* (*abstract semantic graph*) décrit dans [Rosenblum_1991] contient des liens supplémentaires qui permettent entre autres la navigation à partir d'un identificateur jusqu'à son point de définition.

Il est bien sûr possible d'élever le niveau d'abstraction de ce modèle davantage. Par exemple, le corps de la méthode pourrait être décrit par trois listes: les variables lues, les variables écrites et les méthodes invoquées. Cependant, nous croyons que les activités de rétroconception, de compréhension et de réingénierie exigent un modèle

qui s'appuie plus étroitement sur les concepts fondamentaux des langages de programmation. Dans le cas de la rétroconception, un modèle trop abstrait nuirait entre autres à la détection de méthodes qui jouent des rôles importants au sein du système. Par exemple, une méthode qui contient un énoncé *switch* à plusieurs branches signale la présence d'un centre de décision potentiellement important pour le système. Pour la compréhension du logiciel, une panoplie d'analyses dépendent sur les détails d'implantation de la méthode. Une liste partielle de ces analyses comprend le slicing, les métriques de complexité algorithmique et la détection de clones. Et finalement, l'activité de réingénierie requiert l'utilisation d'un modèle suffisamment élaboré qui permet d'abord l'étude des détails d'une implantation et ensuite, l'application des correctifs ou des transformations nécessaires. Par exemple, la restructuration automatisée d'un système défectueux peut nécessiter l'insertion d'un test de validation immédiatement après chaque appel d'une méthode critique lorsque cet appel est précédé de l'ouverture d'un fichier.

En résumé, le niveau d'abstraction du profil proposé doit modéliser le corps d'une méthode en termes de concepts primitifs du langage tels que l'on retrouve dans l'arbre syntaxique abstrait. Cette granularité est désirable puisque les activités de rétroconception, de compréhension et de réingénierie analysent et manipulent directement ces composantes élémentaires qui définissent un logiciel.

4.4.2 Contenu

La section précédente a décrit le niveau d'abstraction souhaitable de la solution sans toutefois expliciter les entités pertinentes à modéliser. Les sous-sections suivantes présentent maintenant le contenu important qui doit être représenté par les éléments du profil proposé: les énoncés et les expressions.

4.4.2.1 Énoncés

Malgré la variété d'allures qu'elle peut prendre, une méthode détient la responsabilité fondamentale de spécifier la création, la lecture, l'écriture et la destruction des

données d'un programme. La création d'une donnée survient par exemple lors de la définition d'une variable ou lorsque l'opérateur *new* de C++ est appelé avec un type approprié. La destruction des données s'effectue automatiquement dans beaucoup de cas mais doit parfois être spécifiée explicitement dans le code (e.g., opérateur *delete* en C++). La lecture d'une donnée a lieu chaque fois qu'elle participe à une expression tandis que son écriture est normalement signalée par une assignation.

Les styles de programmation impératif et orienté objet exigent que les actions décrites ci-haut soient structurées dans des énoncés. Les énoncés de base incluent les déclarations, les assignations et les appels de méthodes. La séquence d'exécution procède séquentiellement d'un énoncé à l'autre mais elle peut être redirigée par les énoncés de branchement inconditionnel (e.g., *goto*, *break*, etc.) ainsi que par les énoncés de sélection (e.g., *if-else*, *switch-case*) qui effectuent des branchements basés sur la valeur d'une expression de contrôle. Les énoncés d'itération (e.g., *for*, *while, do*) permettent la répétition contrôlée des énoncés en vérifiant la valeur d'une expression booléenne avant (ou après) la prochaine itération.

Puisque les énoncés définissent la structure d'une méthode et encadrent les actions fondamentales du logiciel, la solution proposée doit modéliser le concept de l'énoncé.

4.4.2.2 Expressions

Les logiciels effectuent leur travail en manipulant des données. Chaque donnée est caractérisée par son type qui définit l'ensemble des valeurs possibles qu'elle peut prendre ainsi que les propriétés de ces valeurs et les opérations qui leur sont applicables. Une valeur d'un certain type provient de l'évaluation d'une expression du même type. Toute expression appartient à l'une des deux familles suivantes: les expressions simples ou les expressions complexes. L'expression simple dénote une valeur littérale (e.g., *true*, *1*, *3.1415*, *'a'*, *"allo"*), une référence à une entité définie ailleurs dans le système (e.g., *x*, *args*, *top*, *MAX_COUNT*) ou le nom d'un type (e.g., *char*, *float*, *Node*). L'expression complexe est définie récursivement comme la composition (légale) de sous-expressions et d'opérateurs prédéfinis par le langage.

Les opérateurs n'apparaissent pas toujours explicitement dans les expressions pour des raisons stylistiques propres à chaque langage. Cependant, une expression complexe peut toujours être reformulée pour rendre explicite l'application d'un opérateur sur des sous-expressions. Voici quelques exemples d'expressions complexes accompagnées d'une réécriture qui met en évidence l'application d'opérateurs. Nous utilisons la convention $\Delta\theta(a_1, a_2, \ldots a_n)$ pour dénoter l'application de l'opérateur θ sur les arguments $a_1, a_2, \ldots a_n$.

Expression complexe	**Expression avec application explicite d'opérateurs**
`2 - (a + b)`	`Δ-(2, Δ+(a, b))`
`z > 0`	`Δ>(z, 0)`
`t[6]`	`Δ[](t, 6)`
`*p`	`Δ*(p)`
`f(x)`	`Δ()(f, x)`
`obj.v`	`Δ.(obj, v)`
`(float) i`	`Δcast(float, i)`
`sizeof(char)`	`Δsizeof(char)`
`new Node`	`Δnew(Node)`

Exemple 5: Expressions complexes et application explicite d'opérateurs

Conceptuellement, chaque application d'opérateur constitue un pas atomique dans l'évaluation d'une expression. La séquence d'évaluation des pas est déterminée par les lois de précédence définies par chaque langage.

La solution proposée doit modéliser le concept des expressions puisqu'elles révèlent exactement où et comment les données d'un programme sont utilisées.

Chapitre 5: Profil RCR

La panoplie de métamodèles vue à la section 3.3 rappelle qu'il existe plusieurs approches pour modéliser les systèmes logiciels. En particulier, la définition d'un métamodèle pour la modélisation détaillée de méthodes compte maintes solutions possibles. Ce chapitre décrit notre solution, le profil RCR (Rétroconception Compréhension Réingénierie).

Le profil RCR propose des extensions au métamodèle UML pour permettre la rétroconception, la compréhension et la réingénierie de logiciels. Dans la suite du texte, les entités appartenant à ce profil sont nommées selon la convention *RCR.Nom* pour les distinguer des entités appartenant à d'autres domaines (e.g., UML, autres profils).

La prochaine section constitue un survol du profil RCR. La première sous-section introduit les concepts principaux à l'aide d'un exemple simple. La sous-section suivante présente la hiérarchie complète des nouveaux métaéléments en les groupant par familles conceptuellement distinctes qui facilitent la compréhension.

La deuxième section explique les deux stratégies principales qui ont été utilisées pour la conception du profil RCR: la flexibilité et les dépendances. La *flexibilité* est une stratégie d'ordre plus abstrait comparativement aux *dépendances* qui exposent certains détails de la modélisation UML.

La troisième section fournit une description détaillée pour chacun des métaéléments du profil.

Finalement, une conclusion résume le contenu du chapitre.

Nous rappelons que ce chapitre décrit le métamodèle *logique* du profil RCR et non un métamodèle *physique*. Le métamodèle logique décrit les entités et les relations de manière à faciliter la compréhension. Un métamodèle physique spécifie explicitement les structures de données à utiliser (e.g., liste, ensemble) ainsi que les adaptations (e.g., renommer des entités, réorganiser des hiérarchies) ou les extensions (e.g., ajouter des entités) au métamodèle logique pour satisfaire les besoins techniques de son implantation concrète.

5.1 Survol

Cette section introduit les concepts principaux du profil RCR en modélisant le corps d'une méthode simple. Ensuite, une vue globale du profil RCR est fournie en présentant la hiérarchie complète des nouveaux métaéléments.

5.1.1 Concepts principaux

Le profil RCR repose sur quatre concepts de base: les *blocs*, les *énoncés*, les *expressions* et les *pas d'évaluation*. Ces concepts sont introduits progressivement à l'aide de la définition de la méthode *m* donnée dans l'Exemple 6.

```
float m()
{
    float x;
    x = 1.23;
    return x;
}
```

Exemple 6: Méthode simple

Le corps de cette méthode (borné par les accolades) comporte les trois énoncés suivants: une déclaration, un énoncé d'expression et un énoncé de retour. La déclaration introduit la variable de type *float* portant le nom *x* dans la portée du corps de *m*. L'énoncé d'expression assigne la valeur littérale 1.23 de type *float* à cette variable. Enfin, l'énoncé de retour retourne la valeur de la variable. La section 6.3

présente des exemples plus avancés qui modélisent entre autres un appel de méthode simple, un appel de méthode par l'intermédiaire d'un objet, ainsi que divers types d'énoncés.

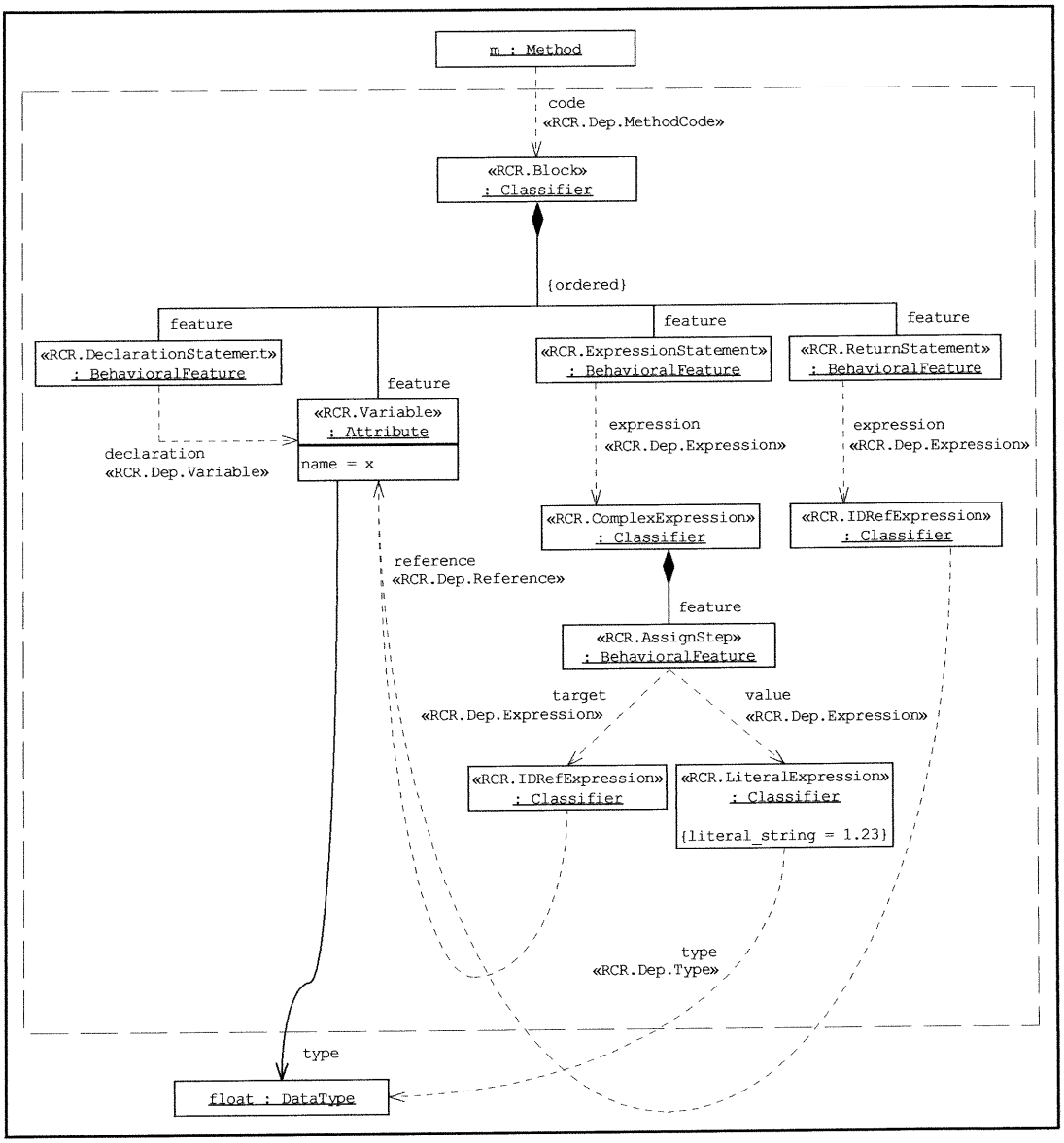
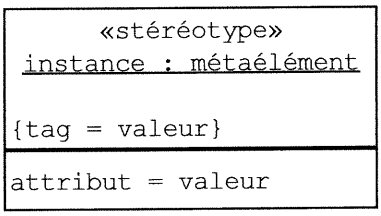


Figure 21: Diagramme d'objets illustrant l'usage du profil RCR pour modéliser le corps de la méthode simple de l'Exemple 6

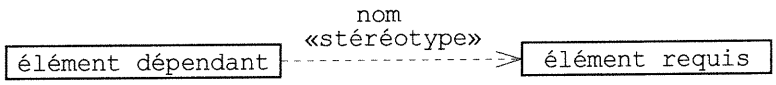
La Figure 21 est un diagramme d'objets qui illustre la modélisation du corps de la méthode *m* avec le profil RCR. En particulier, les instances à l'intérieur du rectangle délimité par les tirets sont générées à partir des métaéléments du profil RCR.

Pour bien comprendre le contenu de ce diagramme, nous rappelons brièvement la notation UML pertinente pour cet exemple.

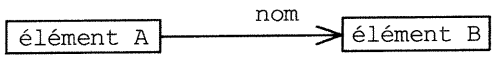
- Chaque rectangle blanc représente l'instance d'un métaélément. Le nom de l'instance (facultatif) et le nom du métaélément instancié sont soulignés et placés dans le compartiment du haut. Lorsque l'instance est un stéréotype, le nom du stéréotype est indiqué entre guillemets au-dessus de l'information soulignée. Chaque TaggedValue est placée entre accolades sous l'information précédente. Le deuxième compartiment indique les valeurs des attributs de cette instance particulière.



- La dépendance (i.e., UML.Foundation.Core.Dependency) entre deux éléments est notée par une flèche en tirets qui se dirige de l'élément dépendant (i.e., *client*) vers l'élément requis (i.e., *supplier*). Comme les autres métaéléments d'UML, une dépendance peut être spécialisée par un stéréotype et elle peut être nommée.



- Le lien[18] (i.e., UML.BehavioralElements.CommonBehavior.Link) entre deux instances apparaît comme un trait solide. L'ajout d'une tête de flèche précise la direction d'un lien unidirectionnel. Chaque extrémité du lien peut être nommée.



- La composition d'éléments est signalée par la flèche à tête de losange noir qui s'oriente d'un sous-élément vers l'élément composé. Chaque extrémité peut être nommée.

[18] Le lien correspond à l'instanciation d'une association (i.e., Foundation.Core.Association).

Regardons maintenant le modèle représenté par ce diagramme. L'interface entre le métamodèle UML standard et le métamodèle du profil RCR se situe au niveau de la dépendance «RCR.Dep.MethodCode» nommée *code* qui apparaît au sommet du diagramme. Cette dépendance stéréotypée permet de lier une instance de Method avec un bloc de code modélisé par le profil RCR. En plus de «RCR.Dep.MethodCode», le profil RCR définit et utilise plusieurs autres dépendances stéréotypées qui permettent d'unir une ou plusieurs entités du modèle entre elles.

Les blocs sont utilisés dans le profil RCR pour définir une portée et pour grouper des énoncés et des variables. Le bloc de code dans l'exemple (i.e., «RCR.Block») est constitué des trois énoncés de la méthode *m* (i.e., «RCR.DeclarationStatement», «RCR.ExpressionStatement», «RCR.ReturnStatement») ainsi que de la variable *x* (i.e., «RCR.Variable»). La suite de cette section décrit le contenu de ce bloc en le parcourant de gauche à droite et de haut en bas.

Le premier énoncé introduit la variable *x* dans le bloc. Ce fait est modélisé par la dépendance «RCR.Dep.Variable» nommée *declaration* qui relie les instances «RCR.DeclarationStatement» et «RCR.Variable».

L'énoncé suivant (i.e., «RCR.ExpressionStatement») évalue l'expression «RCR.ComplexExpression». Dans le profil RCR, les expressions complexes sont constituées de *pas d'évaluation*. De façon générale, un pas d'évaluation représente une action atomique qui est appliquée à des sous-expressions. Le pas d'évaluation de cet exemple, «RCR.AssignStep», représente l'assignation d'une valeur à une variable. Concrètement, la dépendance spécifiant la variable assignée se nomme *target*, tandis que la dépendance *value* spécifie la valeur de l'assignation. Ces deux dépendances sont des stéréotypes «RCR.Dep.Expression». La dépendance *target* aboutit sur l'expression simple «RCR.IDRefExpression». Une expression de ce genre modélise une référence à une entité définie ailleurs: ici, elle remonte par la dépendance nommée *reference* vers la variable *x* déclarée précédemment. De son côté, la dépendance *value* se dirige vers l'expression simple «RCR.LiteralExpression» qui

modélise le concept d'expression littérale. Dans ce cas-ci, cette valeur est de type *float* et la chaîne de caractères qui la spécifie (i.e., 1.23) est donnée par le TaggedValue *literal_string*.

Finalement, l'énoncé de retour modélisé par «RCR.ReturnStatement» retourne la valeur de l'expression indiquée par la dépendance nommée *expression*. Dans cet exemple, cette expression est de type «RCR.IDRefExpression» et elle fait référence à la variable *x*.

5.1.2 Hiérarchies des nouveaux métaéléments

Le profil RCR définit 52 nouveaux métaéléments sous forme de stéréotypes basés sur quatre métaéléments du package UML.Foundation.Core: Attribute, BehavioralFeature, Classifier et Dependency. Cette section présente ces métaéléments en les groupant par familles: les blocs, les énoncés, les expressions, les pas d'évaluation et les dépendances. Le métaélément RCR.Variable est unique et n'appartient pas à une famille comme telle.

Chaque hiérarchie est décrite par un diagramme de classes. La superclasse de chaque hiérarchie correspond au métaélément UML qui sert de base pour les stéréotypes de cette famille. De plus, les *tags* des nouveaux TaggedValue associés aux stéréotypes apparaissent dans un compartiment nommé *Tags*. Par contre, les contraintes des stéréotypes et les interdépendances des entités du profil ne sont pas incluses dans ces diagrammes pour éviter une surcharge d'information visuelle. Ces informations ainsi que la définition de chaque stéréotype seront données dans la prochaine section.

5.1.2.1 Blocs

Les blocs modélisent une portée et servent à regrouper les énoncés et les variables. La famille des blocs comprend trois stéréotypes basés sur le métaélément Classifier. La superclasse RCR.Block introduit le tag *code_language*.

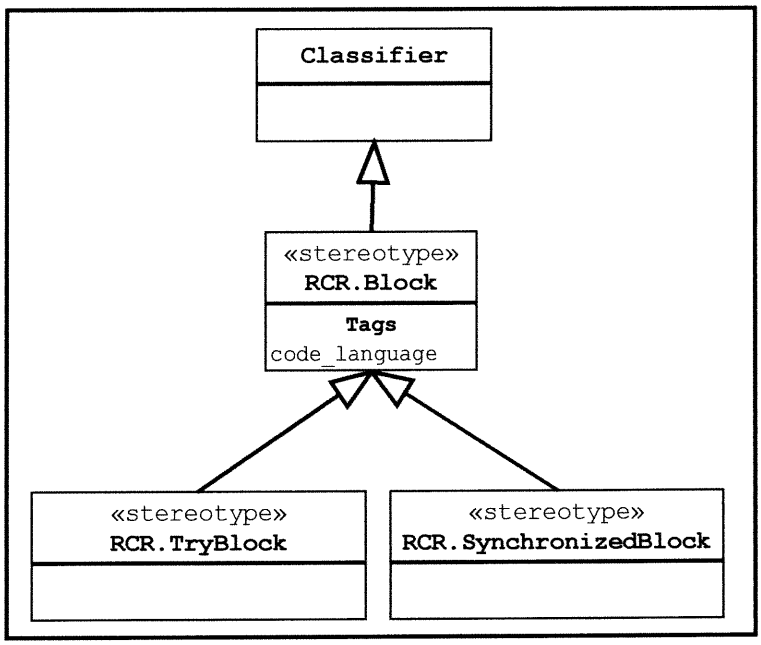


Figure 22: Hiérarchie des blocs du profil RCR

5.1.2.2 Énoncés

Cette famille de 23 stéréotypes modélise les divers types d'énoncés qui composent les corps de méthodes. La superclasse RCR.Statement spécialise le métaélément UML.Foundation.Core.BehavioralFeature et elle lègue à ses descendants le tag *statement_label*. Cette famille contient également trois hiérarchies secondaires qui précisent davantage les entités RCR.JumpStatement, RCR.BranchStatement et RCR.IterationStatement.

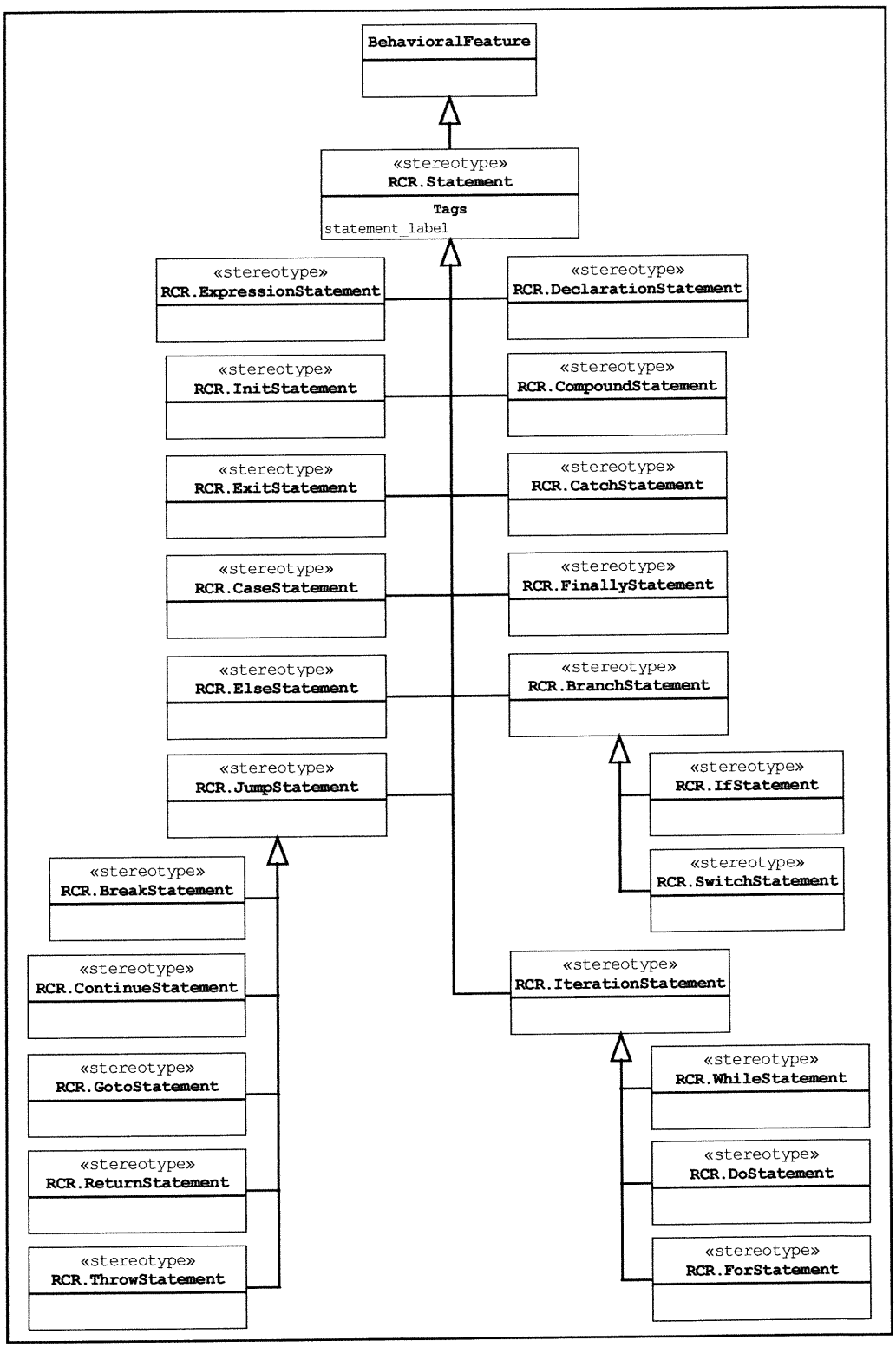


Figure 23: Hiérarchie des énoncés du profil RCR

5.1.2.3 Expressions

Cette famille modélise les expressions qui sont rencontrées dans le corps d'une méthode avec six stéréotypes basés sur le métaélément standard Classifier. La hiérarchie contient d'une part les variétés d'expressions simples et d'autre part les expressions complexes. Le métaélément stéréotypé RCR.LiteralExpression impose le tag *literal_string* à chacune de ses instances.

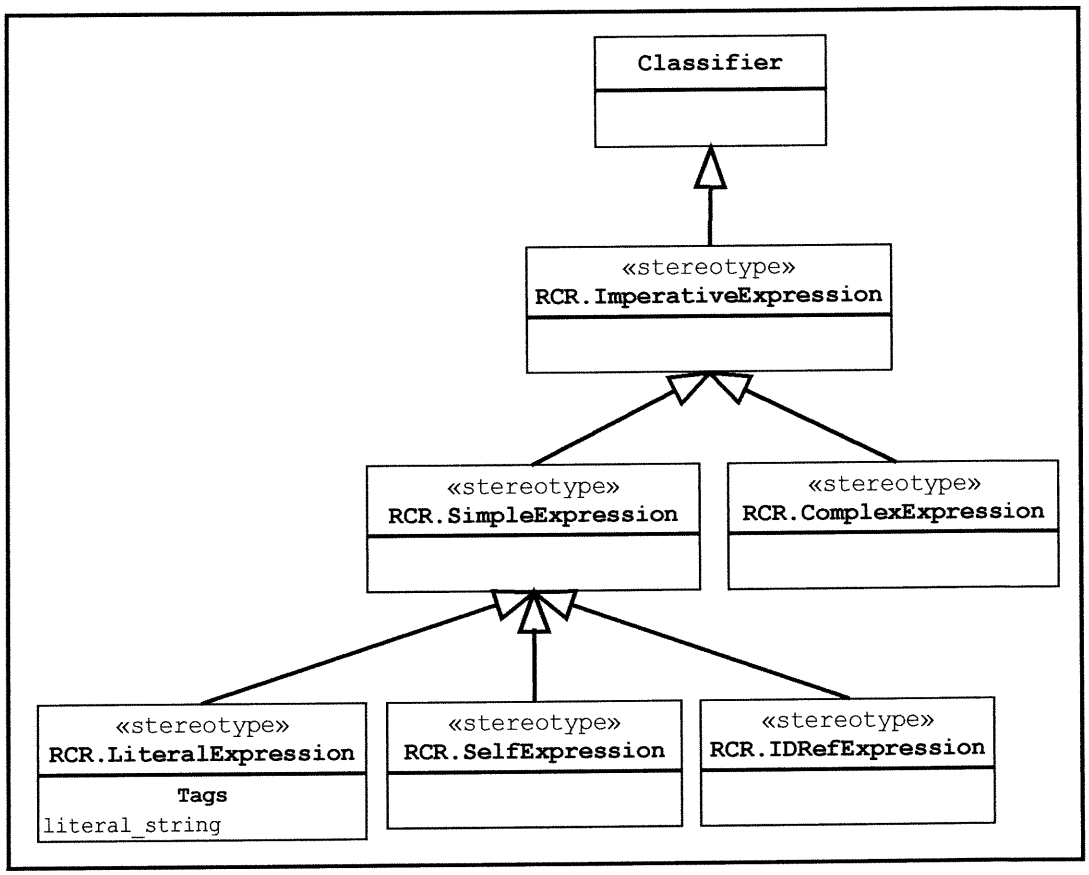


Figure 24: Hiérarchie des expressions du profil RCR

5.1.2.4 Pas d'évaluation

Les pas d'évaluation modélisent les actions atomiques d'un programme. Cette famille contient neuf types de pas d'exécution dérivés de la superclasse générique RCR.Step qui spécialise le métaélément BehavioralFeature. Chaque stéréotype de ce groupe possède un tag dédié (e.g., *createStep_kind*, *destroyStep_kind*, etc.) qui permet de préciser au besoin la nature exacte du pas d'exécution.

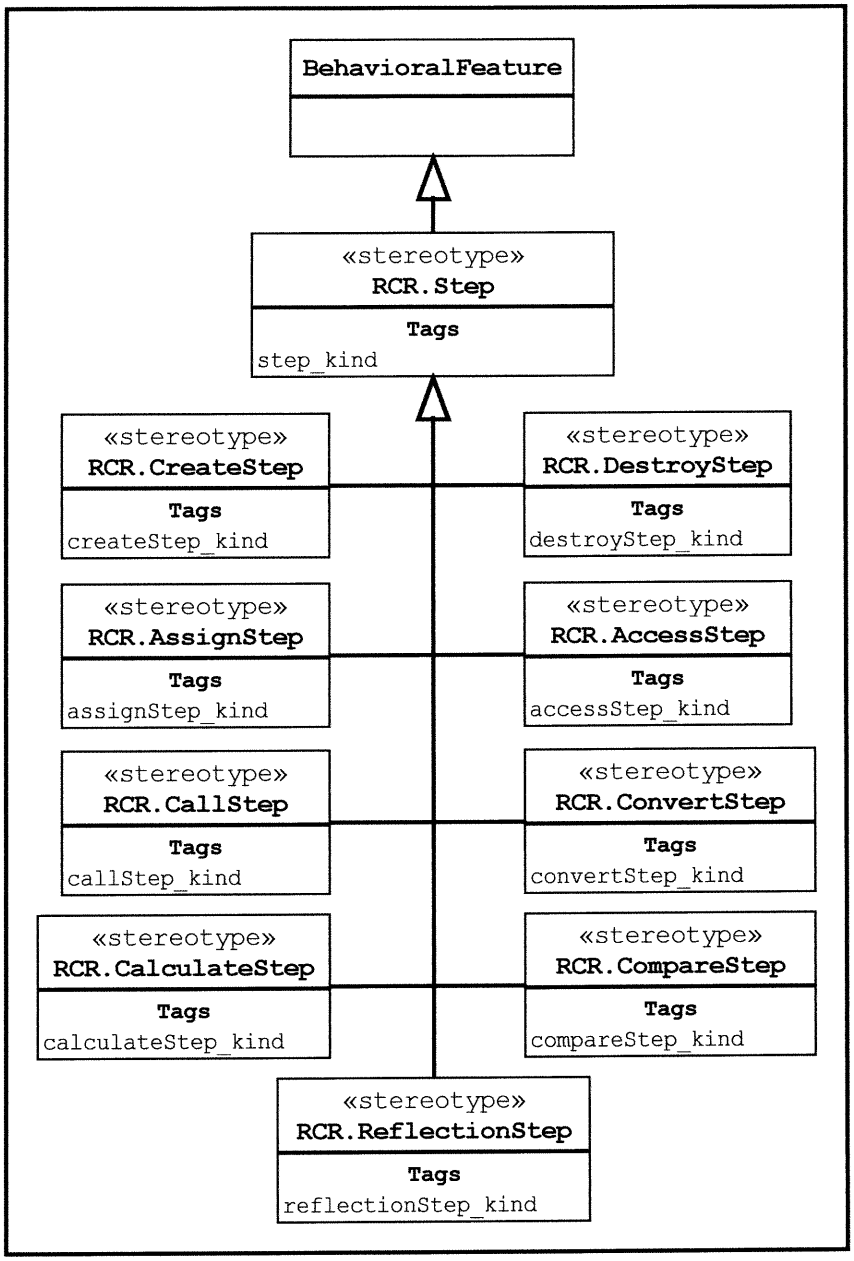


Figure 25: Hiérarchie des pas d'évaluation du profil RCR

5.1.2.5 Variable

L'entité RCR.Variable représente le concept d'une variable locale rencontrée dans le corps d'une méthode. Ce stéréotype est basé sur le métaélément Attribute.

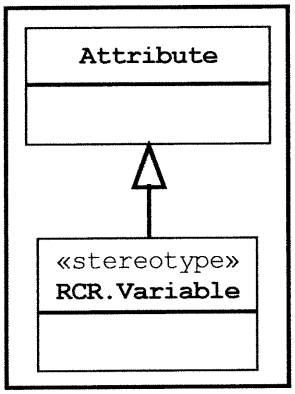


Figure 26: Métaélément unique RCR.Variable du profil RCR

5.1.2.6 Dépendances

Le profil RCR fait appel à ces neuf dépendances stéréotypées pour établir des liens précis entre les autres entités du profil et les métaéléments UML standard. Spécifiquement, ces dépendances ne modélisent pas les entités concrètes du corps d'une méthode; plutôt, elles adaptent le mécanisme de modélisation UML pour le profil RCR. Une explication plus détaillée de leur rôle est fournie dans la section suivante.

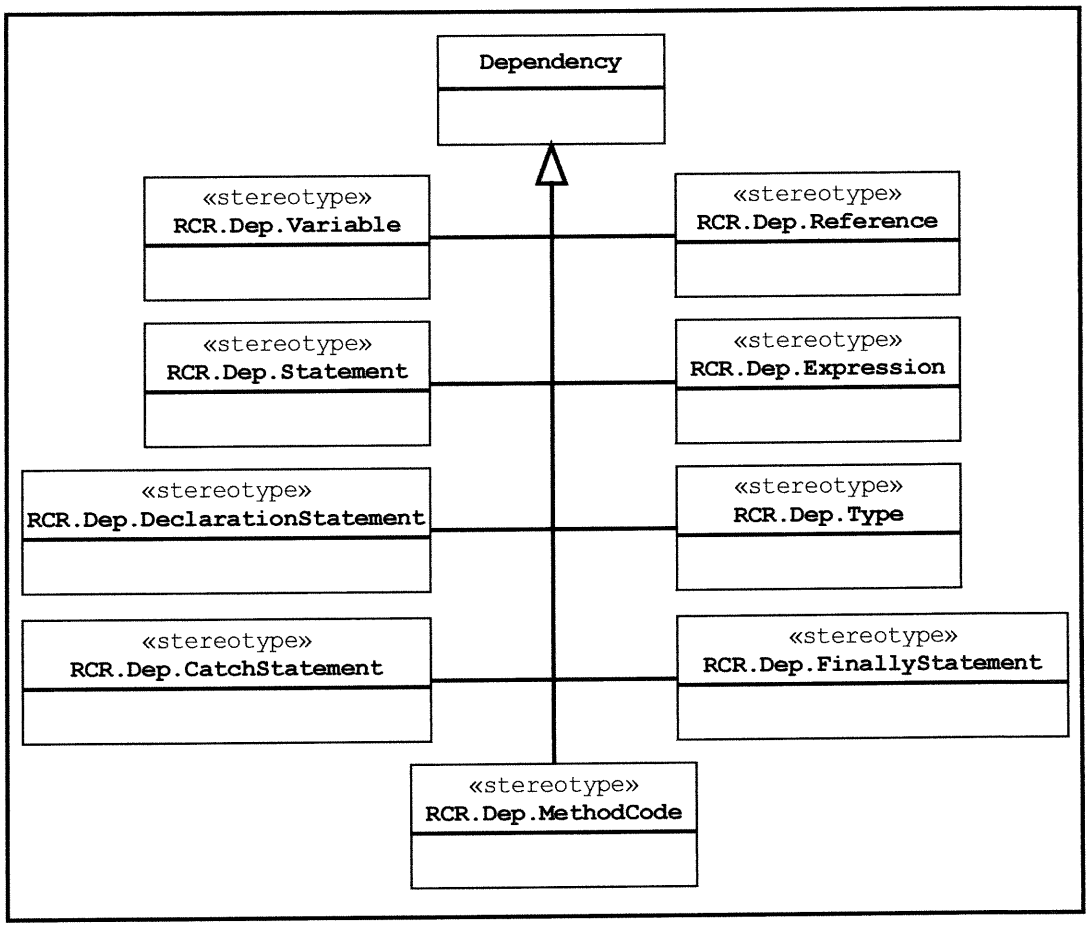


Figure 27: Hiérarchie des dépendances du profil RCR

5.2 Stratégies de conception

Nous expliquons dans cette section les deux stratégies qui sous-tendent la conception du profil RCR. La *flexibilité* est discutée d'abord, suivie ensuite d'une discussion plus élaborée sur l'utilisation des *dépendances* pour contourner certaines limites des mécanismes d'extension d'UML.

5.2.1 Flexibilité

La modélisation de systèmes logiciels écrits en langages de programmation quelconques nécessite un métamodèle expressif qui n'impose pas de restrictions arbitraires. Par conséquent, le profil RCR fournit une riche sélection de métaéléments

de base mais ne spécifie pas explicitement comment ils doivent être combinés pour créer du code syntaxiquement et sémantiquement valide. Prenons par exemple l'énoncé *if* tel qu'il est défini en C++ [ISO_1998]. Si un énoncé *if* est suivi d'une partie *else* et que cet énoncé *if* contient lui-même un énoncé *if*, alors l'énoncé *if* interne doit obligatoirement être suivi d'une partie *else*. En d'autres mots, la structure suivante est imposée:

```
if (c1)
      if (c2) a2
      else b2
else b1
```

Le profil RCR n'impose pas de telles restrictions car nous avons adopté la position que les validations sémantiques relèvent de la compétence d'outils et non du métamodèle comme tel.

5.2.2 Dépendances

Les nouveaux métaéléments du profil RCR sont obtenus par le stéréotypage de métaéléments existants. Puisque le mécanisme de stéréotypage ne permet pas l'ajout ou le retrait d'attributs, les stéréotypes sont structurellement identiques aux métaéléments qu'ils étendent. Certainement, le TaggedValue offre la possibilité d'attacher une nouvelle propriété à une entité. Cependant, cette propriété est limitée aux valeurs textuelles seulement; en particulier, elle ne peut pas faire référence à d'autres métaéléments.

Conceptuellement, plusieurs nouveaux métaéléments possèdent des attributs qui n'ont pas d'équivalents dans le métaélément étendu. Par exemple, le métaélément RCR.SynchronizedBlock, qui est un stéréotype de Classifier, modélise un bloc d'énoncés protégé par un verrou qui empêche l'exécution par plus d'un processus au même instant. Comment modéliser le verrou qui ne possède aucun analogue parmi les attributs de la classe de base Classifier?

La solution que nous avons retenue fait appel au métaélément Dependency. Selon le métamodèle UML, Dependency modélise la dépendance entre un ou plusieurs

ModelElements. L'ensemble de ModelElements dépendants se nomme *client* tandis que l'ensemble de ModelElements requis se nomme *supplier*. Un mécanisme de typage très flexible est obtenu en stéréotypant le métaélément Dependency et en lui imposant des restrictions quant aux types de ModelElements qui peuvent être *client* ou *supplier*. Ensuite, en utilisant l'attribut *name* de Dependency (hérité de ModeleElement), il devient possible de simuler un attribut quelconque pour un métaélément arbitraire. L'approche se résume ainsi: soit M.x : T, l'attribut *x* de type *T* appartenant au métaélément *M*. Pour modéliser l'attribut *x*, on crée d'abord un stéréotype de Dependency avec la contrainte que tous les *suppliers* doivent être de type T. Puis, on ajoute une dépendance de ce genre portant le nom *x* ayant le métaélément *M* comme *client*.

5.2.2.1 RCR.Dep.MethodCode

Stereotype	**RCR.Dep.MethodCode**
Base Class	UML.Foundation.Core.Dependency
Tagged Values	
Dependencies	
Constraints	1. client : UML.Foundation.Core.Method 2. supplier : RCR.Block
Notes	

Le stéréotype RCR.Dep.MethodCode permet de lier le métaélément UML Method avec un bloc de code RCR.Block du profil RCR. Cette dépendance, nommée *code*, établit le pont entre le métamodèle UML et le métamodèle étendu RCR.

Les deux contraintes assurent que cette dépendance joint les métaéléments appropriés, à savoir les Method aux RCR.Block. Notons que la multiplicité par défaut des ensembles *client* et *supplier* demeure inchangée: un ou plusieurs *clients* peuvent dépendre sur un ou plusieurs *suppliers*. En particulier, ceci permet d'avoir plusieurs implantations pour la même méthode. Cette situation est rencontrée par exemple lorsqu'une méthode est implantée en plusieurs langages ou bien que l'implantation existe en plusieurs versions.

5.2.2.2 Autres dépendances

Le reste des stéréotypes de Dependency dans le profil RCR sont utilisés pour unir un ensemble (i.e., un ou plusieurs métaéléments arbitraires à un ou plusieurs métaéléments d'un certain type). Ils diffèrent uniquement au niveau du type de métaélément de l'ensemble *supplier*. Par exemple, la dépendance RCR.Dep.Expression exige que les métaéléments requis soient de type RCR.ImperativeExpression tandis que ceux de RCR.Dep.Type sont de type Classifier. De plus, tous ces stéréotypes imposent l'ordonnancement (i.e., *ordered*) des éléments de l'ensemble *supplier*. En d'autres mots, il est plus approprié de considérer *supplier* comme une liste d'éléments pour ces dépendances. Le Tableau 9 indique le type de *supplier* exigé par chaque stéréotype.

Stéréotype	**Type de *supplier* exigé**
RCR.Dep.Expression	RCR.ImperativeExpression
RCR.Dep.Type	UML.Foundation.Core.Classifier
RCR.Dep.Variable	RCR.Variable
RCR.Dep.Statement	RCR.Statement
RCR.Dep.DeclarationStatement	RCR.DeclarationStatement
RCR.Dep.CatchStatement	RCR.CatchStatement
RCR.Dep.FinallyStatement	RCR.FinallyStatement
RCR.Dep.Reference	UML.Foundation.Core.ModelElement

Tableau 9: Stéréotypes de Dependency et le type de *supplier* exigé

5.3 Détails

Cette section détaille chaque nouveau métaélément du profil RCR à l'exception des dépendances qui sont décrites à la section 5.2.2. La présentation reprend l'ordre utilisé dans les sections précedentes, à savoir: les blocs, les énoncés, les expressions, les pas d'exécution et la variable.

5.3.1 Blocs

Les blocs modélisent une portée et servent à regrouper les énoncés et les variables locales.

Stereotype	RCR.Block
Base Class	UML.Foundation.Core.Classifier
Tagged Values	1. code_language
Constraints	1. feature : RCR.Statement or RCR.Variable
Dependencies	
Notes	

Le métaélément RCR.Block est la superclasse de tous les blocs du profil RCR. Ce stéréotype de Classifier introduit le TaggedValue *code_language* qui permet de spécifier le langage de programmation des énoncés qui s'y trouvent. La contrainte qui accompagne RCR.Block impose que les *features* du Classifier soient de type RCR.Statement ou RCR.Variable. Ce métaélément est utilisé pour modéliser tous les blocs de code qui ne sont pas représentés par les blocs RCR.TryBlock et RCR.SynchronizedBlock décrits ci-dessus.

Stereotype	RCR.TryBlock
Base Class	RCR.Block
Tagged Values	
Dependencies	1. catch : RCR.Dep.CatchStatement 2. finally : RCR.Dep.FinallyStatement
Constraints	
Notes	

Le métaélément RCR.TryBlock modélise un bloc de code muni d'un mécanisme de surveillance qui détecte l'occurrence de signaux (e.g., exception, erreur) générés lors de l'exécution d'un énoncé dans une situation anormale. Lorsque le signal est détecté, le flot de contrôle est immédiatement transféré au premier énoncé de type RCR.CatchStatement associé au bloc qui peut traiter le type de signal généré. S'il y a un ensemble d'énoncés de type RCR.FinallyStatement associés au bloc, ces énoncés sont exécutés dans tous les cas possibles. En d'autres mots, leur exécution a lieu peu importe si le bloc RCR.TryBlock complète normalement ou pas, ou qu'un énoncé RCR.CatchStatement soit trouvé et exécuté, ou pas.

La dépendance nommée *catch* associe les énoncés de type RCR.CatchStatement au bloc, tandis que celle nommée *finally* associe le bloc aux énoncés de type RCR.FinallyStatement.

Stereotype	RCR.SynchronizedBlock
Base Class	RCR.Block
Tagged Values	
Dependencies	1. lock : RCR.Dep.Expression
Constraints	
Notes	

Le métaélément RCR.SynchronizedBlock modélise un bloc de code protégé par une liste de verrous qui doivent tous être acquis par un processus avant que les énoncés inclus puissent être exécutés. La dépendance nommée *lock* associe RCR.SynchronizedBlock à la liste d'expressions identifiant les verrous à acquérir.

5.3.2 Énoncés

Les stéréotypes de ce groupe modélisent les divers types d'énoncés qui sont rencontrés dans l'implantation des méthodes. Les énoncés sont présentés selon l'ordre suivant: énoncés simples, énoncés de branchement, énoncés de saut, énoncés d'itération et autres énoncés.

5.3.2.1 Énoncés simples

Stereotype	RCR.Statement
Base Class	UML.Foundation.Core.BehavioralFeature
Tagged Values	1. statement_label
Dependencies	
Constraints	
Notes	

Le métaélément RCR.Statement est la superclasse de tous les énoncés du profil RCR. Puisque les énoncés représentent l'aspect comportemental d'un bloc de code, nous avons choisi de baser ce stéréotype sur le métaélément BehavioralFeature. Cette décision facilite également l'intégration naturelle des énoncés dans les blocs puisque la relation de composition existant entre Classifier (i.e., RCR.Block) et Feature (i.e., RCR.Statement) est maintenue. L'ajout du TaggedValue *statement_label* permet d'assigner à tous les énoncés une étiquette. Cette étiquette est utilisée par exemple avec les énoncés *goto* dans les langages C et C++.

Stereotype	**RCR.DeclarationStatement**
Base Class	RCR.Statement
Tagged Values	
Dependencies	1. declaration : RCR.Dep.Variable
Constraints	
Notes	

Le métaélément RCR.DeclarationStatement modélise un énoncé qui introduit une ou plusieurs variables dans un bloc de code. La dépendance *declaration* établit le lien avec la liste des variables introduites. Le nom, le type et la valeur d'initialisation de la nouvelle variable sont modélisés à partir du métaélément RCR.Variable lui-même.

Stereotype	**RCR.ExpressionStatement**
Base Class	RCR.Statement
Tagged Values	
Dependencies	1. expression : RCR.Dep.Expression
Constraints	
Notes	

Le métaélément RCR.ExpressionStatement modélise un énoncé dont le but est d'évaluer une ou plusieurs expressions. Souvent, ces expressions prennent la forme d'une invocation de méthode ou d'une assignation de variable. Ces cas, ainsi que tous les autres, sont modélisés au niveau du contenu d'une expression, plus particulièrement avec les métaéléments de type RCR.Step. La dépendance *expression* établit le lien avec la liste des expressions à évaluer.

5.3.2.2 Énoncés de branchement

Les énoncés de branchement introduisent des branches d'exécution qui sont empruntées selon la valeur d'une expression de contrôle.

Stereotype	**RCR.BranchStatement**
Base Class	RCR.Statement
Tagged Values	
Dependencies	1. value : RCR.Dep.Expression 2. branch : RCR.Dep.Statement
Constraints	
Notes	

Le métaélément RCR.BranchStatement est la superclasse des énoncés de branchement. Les deux dépendances associent l'énoncé à la valeur de contrôle (*value*) et à la liste d'énoncés (*branch*) représentant les branches à exécuter. L'algorithme de sélection des branches n'est pas spécifié par ce métaélément mais plutôt par ses descendants.

Stereotype	**RCR.IfStatement**
Base Class	RCR.BranchStatement
Tagged Values	
Dependencies	
Constraints	
Notes	1. voir RCR.ElseStatement

Le métaélément RCR.IfStatement modélise un énoncé qui transfert le contrôle à sa première branche si la valeur de contrôle peut être interprétée comme ayant la valeur 'vraie'. Si cette valeur n'est pas 'vraie', le contrôle passe à la deuxième branche si elle existe, sinon le contrôle quitte l'énoncé pour passer au suivant. Typiquement, la deuxième branche est un énoncé de type RCR.ElseStatement, mais le profil RCR ne l'impose pas pour permettre une flexibilité maximale.

Stereotype	**RCR.SwitchStatement**
Base Class	RCR.BranchStatement
Tagged Values	
Dependencies	
Constraints	
Notes	1. voir RCR.CaseStatement 2. voir RCR.ElseStatement

Le métaélément RCR.SwitchStatement modélise un énoncé de branchement contrôlé par une expression pouvant prendre une valeur qui n'est pas limité à 'vrai' ou 'faux'. Typiquement, les branches sont des énoncés de type RCR.CaseStatement et le flôt de contrôle passe à l'énoncé associé à la valeur égale à la valeur de contrôle. Si aucun énoncé satisfait la valeur de contrôle et qu'un énoncé de type RCR.ElseStatement fait partie des branches, cet énoncé est exécuté. Ce dernier cas représente l'énoncé portant l'étiquette 'default' dans les langages comme C, C++ et Java. Bien sûr, d'autres comportements sont possibles. Le choix du langage de programmation, spécifié au

niveau du bloc qui appartient cet énoncé, détermine les détails de la sémantique et du comportement.

5.3.2.3 Énoncés de saut

Les énoncés de saut transfèrent le flot de contrôle à un autre point d'exécution.

Stereotype	**RCR.JumpStatement**
Base Class	RCR.Statement
Tagged Values	
Dependencies	1. location : RCR.Dep.Expression
Constraints	
Notes	

Le métaélément RCR.JumpStatement est la superclasse des énoncés de saut dans le profil RCR. Il participe à une dépendance, nommée *location*, qui indique la destination du saut. Les sous-classes décritent ci-dessous raffinent la sémantique générale de RCR.JumpStatement.

Stereotype	**RCR.BreakStatement**
Base Class	RCR.JumpStatement
Tagged Values	
Dependencies	
Constraints	
Notes	

Le métaélément RCR.BreakStatement modélise un énoncé de saut typiquement utilisé pour interrompre la séquence d'exécution normale des énoncés dans un bloc. Lors de l'exécution de cet énoncé, le flot d'exécution est interrompu et passe habituellement à un énoncé périphérique au bloc courant. Bien sûr, le comportement exact varie en fonction du langage de programmation. Le RCR.BreakStatement apparaît normalement dans le contexte des énoncés d'itération ou de sélection.

Stereotype	RCR.ContinueStatement
Base Class	RCR.JumpStatement
Tagged Values	
Dependencies	
Constraints	
Notes	

Le métaélément RCR.ContinueStatement modélise un énoncé de saut typiquement utilisé à l'intérieur d'un bloc contrôlé par un énoncé d'itération. L'exécution de cet énoncé transfert le flot de contrôle à l'énoncé d'itération qui procède à préparer la prochaine itération si nécessaire.

Stereotype	RCR.GotoStatement
Base Class	RCR.JumpStatement
Tagged Values	
Dependencies	
Constraints	
Notes	

Le métaélément RCR.GotoStatement modélise un énoncé de saut qui transfert le flot de contrôle à l'énoncé identifié par la dépendance *location* de la superclasse RCR.JumpStatement. Quoique le métaélément RCR.GotoStatement soit à toutes fins pratiques identique au métaélément RCR.JumpStatement, nous avons choisi de le modélisé explicitement pour bien le situer parmi la famille des autres énoncés de saut.

Stereotype	RCR.ReturnStatement
Base Class	RCR.JumpStatement
Tagged Values	
Dependencies	1. expression : RCR.Dep.Expression
Constraints	
Notes	

Le métaélément RCR.ReturnStatement modélise un énoncé de saut qui transfert le flot de contrôle tout en associant une valeur à cette transition qui peut ensuite être réutilisée au point de transfert. Ce type d'énoncé s'emploie dans la plupart des cas à la fin du corps d'une méthode pour retourner le résultat de son travail. Généralement, la destination du saut n'est pas spécifiée explicitement car elle correspond au point d'exécution suivant l'invocation de la méthode. Puisque l'invocation peut survenir à

n'importe quel endroit, le point de retour est déterminé automatiquement lors de la compilation. La dépendance nommée *expression* associe cet énoncé à l'expression de la valeur à retourner.

Stereotype	**RCR.ThrowStatement**
Base Class	RCR.JumpStatement
Tagged Values	
Dependencies	1. exception : RCR.Dep.Expression
Constraints	
Notes	

Le métaélément RCR.ThrowStatement modélise un énoncé de saut semblable à l'énoncé RCR.ReturnStatement: le flot de contrôle est transféré ailleurs et une valeur particulière est associée à la transition. Cependant, l'énoncé RCR.ThrowStatement est normalement utilisé pour signaler l'occurrence d'une condition anormale et le point de transfert est déterminé dynamiquement en fonction du type de la valeur retournée. Ce métaélément se retrouve typiquement à l'intérieur d'un bloc de type RCR.TryBlock et les énoncés RCR.CatchStatement sont les points de transfert d'exécution habituels. La dépendance *exception* associe RCR.ThrowStatement à l'expression de la valeur à retourner.

5.3.2.4 Énoncés d'itération

Les énoncés d'itération permettent la répétition contrôlée d'un groupe d'énoncés.

Stereotype	**RCR.IterationStatement**
Base Class	RCR.Statement
Tagged Values	
Dependencies	1. condition : RCR.Dep.Expression 2. statement : RCR.Dep.Statement
Constraints	
Notes	

Le métaélément RCR.IterationStatement est la superclasse des énoncés d'itération. Il modélise le concept d'un énoncé qui permet l'exécution répétée d'une liste d'énoncés sous son contrôle en fonction d'une certaine valeur. Cette valeur est donnée par l'expression liée à RCR.IterationStatement par la dépendance nommée *condition*.

Habituellement, la condition peut être interprétée comme 'vrai' ou 'faux' et l'itération se poursuit dans le cas 'vrai'. La dépendance *statement* identifie la liste d'énoncés à répéter.

Stereotype	**RCR.WhileStatement**
Base Class	RCR.IterationStatement
Tagged Values	
Dependencies	
Constraints	
Notes	

Le métaélément RCR.WhileStatement modélise un énoncé d'itération qui évalue la condition d'itération avant d'exécuter une première fois les énoncés contrôlés. Si la condition d'itération n'est pas remplie, l'exécution passe à l'énoncé qui suit RCR.WhileStatement. Sinon, les énoncés contrôlés sont exécutés une première fois. Après l'exécution du dernier énoncé contrôlé, le cycle recommence à nouveau à partir de l'évaluation de la condition.

Stereotype	**RCR.DoStatement**
Base Class	RCR.IterationStatement
Tagged Values	
Dependencies	
Constraints	
Notes	

Le métaélément RCR.DoStatement modélise un énoncé d'itération qui exécute une première fois les énoncés contrôlés puis évalue la condition d'itération pour déterminer si une prochaine répétition doit être faite. Si oui, le cycle recommence à partir de l'exécution des énoncés contrôlés; si non, l'exécution passe à l'énoncé suivant l'énoncé RCR.DoStatement.

Stereotype	**RCR.ForStatement**
Base Class	RCR.IterationStatement
Tagged Values	
Dependencies	1. declaration : RCR.Dep.DeclarationStatement 2. update : RCR.Dep.Expression
Constraints	
Notes	

Le métaélément RCR.ForStatement modélise un énoncé d'itération qui ajoute deux éléments nouveaux à l'itération de base RCR.IterationStatement. Le premier élément, identifié par la dépendance nommée *declaration*, permet l'introduction de nouvelles variables dans le contexte du bloc d'énoncés sous contrôle. Le second élément consiste à évaluer après chaque itération une liste d'expressions associées à RCR.ForStatement par la dépendance *update*. Typiquement, ces expressions mettent à jour des variables qui influencent la condition d'itération.

5.3.2.5 Autres énoncés

Les métaéléments décrits dans cette section modélisent les divers énoncés que l'on peut rencontrer et qui n'appartiennent pas aux groupes d'énoncés vus précédemment.

Stereotype	**RCR.InitStatement**
Base Class	RCR.Statement
Tagged Values	
Dependencies	1. statement : RCR.Dep.Statement
Constraints	
Notes	

Le métaélément RCR.InitStatement modélise un énoncé utilitaire qui, d'un point de vue conceptuel, est exécuté avant tous les autres énoncés d'un bloc. Notons cependant que cet énoncé n'apparaît pas explicitement dans le code car il sert à modéliser les opérations qui sont ajoutées invisiblement par le processus de compilation. Ces opérations servent habituellement à initialiser certaines variables (e.g., les paramètres formels de la méthode) utilisées par les énoncés qui suivent. La dépendance *statement* associe à RCR.InitStatement la liste d'énoncés qui sont exécutés lorsque cet énoncé est atteint.

Stereotype	**RCR.ExitStatement**
Base Class	RCR.Statement
Tagged Values	
Dependencies	1. statement : RCR.Dep.Statement
Constraints	
Notes	

Le métaélément RCR.ExitStatement modélise un énoncé utilitaire identique à RCR.InitStatement à la différence près qu'il est exécuté après les autres énoncés d'un bloc. Cet énoncé permet par exemple de modéliser la destruction implicite d'objets locaux.

Stereotype	**RCR.CompoundStatement**
Base Class	RCR.Statement
Tagged Values	
Dependencies	1. block : RCR.Dep.Block
Constraints	
Notes	

Le métaélément RCR.CompoundStatement modélise un énoncé constitué d'un bloc de type RCR.Block. Cet énoncé est utile pour la modélisation car il permet d'introduire un bloc à l'intérieur d'un bloc. La dépendance *block* relie RCR.CompoundStatement au bloc qu'il contient.

Stereotype	**RCR.CatchStatement**
Base Class	RCR.Statement
Tagged Values	
Dependencies	1. exception : RCR.Dep.Type 2. statement : RCR.Dep.Statement
Constraints	
Notes	

Le métaélément RCR.CatchStatement modélise un énoncé dédié au traitement des exceptions générées par les énoncés RCR.ThrowStatement lors de l'exécution. La dépendance *exception* relie l'énoncé RCR.CatchStatement au type d'exception qu'il peut traiter, tandis que la dépendance *statement* mène à la liste d'énoncés qui définit le traitement fournit. Ce type d'énoncé est habituellement utilisé avec un RCR.TryBlock.

Stereotype	**RCR.FinallyStatement**
Base Class	RCR.Statement
Tagged Values	
Dependencies	1. statement : RCR.Dep.Statement
Constraints	
Notes	

Le métaélément RCR.FinallyStatement modélise un énoncé dont le contenu doit absolument être exécuté. Ce type d'énoncé accompagne normalement un bloc de type RCR.TryBlock pour assurer que certains énoncés seront exécutés même si le bloc n'a pu complété normalement. La dépendance *statement* relie l'énoncé RCR.FinallyStatement à la liste d'énoncés qui doivent être exécutés.

Stereotype	**RCR.CaseStatement**
Base Class	RCR.Statement
Tagged Values	
Dependencies	1. value : RCR.Dep.Expression 2. statement : RCR.Dep.Statement
Constraints	
Notes	

Le métaélément RCR.CaseStatement modélise un énoncé associé à une valeur précise. Normalement, ce genre d'énoncé apparaît comme une des branches d'un énoncé de branchement. Le contenu de cet énoncé est exécuté lorsque la valeur qui lui est associée par la dépendance *value* correspond à la valeur de branchement. La dépendance *statement* mène à la liste d'énoncés qui sont exécutés.

Stereotype	**RCR.ElseStatement**
Base Class	RCR.Statement
Tagged Values	
Dependencies	1. statement : RCR.Dep.Statement
Constraints	
Notes	

Le métaélément RCR.ElseStatement modélise un énoncé semblable à l'énoncé RCR.CaseStatement, mais sa valeur associée représente conceptuellement "les autres valeurs". Dans un usage typique, cet énoncé constitue une des branches d'un énoncé de branchement à l'instar du RCR.CaseStatement. Dans ce contexte, cet énoncé est exécuté lorsque la valeur de branchement ne correspond à aucune des valeurs associées aux énoncés RCR.CaseStatement. L'énoncé peut alors être considéré comme s'il portait l'étiquette 'default' utilisée par les langages C, C++, Java, etc. On retrouve également cet énoncé comme la branche alternative d'un énoncé RCR.IfStatement. Dans ce cas, "les autres valeurs" correspond à 'faux' seulement. La

dépendance *statement* relie l'énoncé RCR.ElseStatement à la liste d'énoncés qui doivent être exécutés.

5.3.3 Expressions

Les métaéléments de ce groupe sont utilisés pour modéliser les expressions.

Stereotype	**RCR.ImperativeExpression**
Base Class	UML.Foundation.Core.Classifier
Tagged Values	1. expression_kind
Dependencies	
Constraints	1. feature : RCR.Step
Notes	

Le métaélément RCR.ImperativeExpression est la superclasse des métaéléments modélisant les expressions rencontrées dans les langages impératifs et orientés objet. Ce stéréotype est dérivé du métaélément Classifier en raison de la similarité entre les relations structurelles Classifier—Feature et RCR.ImperativeExpression—RCR.Step. Les métaéléments de type RCR.Step (décrits dans la prochaine section) composent les expressions complexes. Pour assurer l'intégrité sémantique de cette composition, la contrainte associée à ce stéréotype exige que les *features* de RCR.ImperativeExpression soient de type RCR.Step exclusivement. Le TaggedValue nommé *expression_kind* permet de raffiner au besoin l'expression modélisée par une instance de ce métaélément. Le nom ImperativeExpression plutôt que Expression a été retenu afin de signaler clairement que les expressions modélisées par le profil RCR appartiennent au domaine de la programmation impérative et aussi pour éviter toute confusion avec le métaélément standard UML.Foundation.DataTypes.Expression.

Stereotype	**RCR.SimpleExpression**
Base Class	RCR.ImperativeExpression
Tagged Values	
Dependencies	
Constraints	
Notes	

Le métaélément RCR.SimpleExpression est la superclasse des trois métaéléments modélisant les expressions simples. Une expression simple se distingue d'une expression complexe du fait qu'elle ne contient pas de sous-expressions.

Stereotype	**RCR.LiteralExpression**
Base Class	RCR.SimpleExpression
Tagged Values	1. literal_string
Dependencies	1. type : RCR.Dep.Type
Constraints	
Notes	

Le métaélément RCR.LiteralExpression modélise une expression littérale (e.g., "xyz", 'a', 123, 0.456, 1.0e-12). Le TaggedValue *literal_string* qui accompagne RCR.LiteralExpression contient le texte de la valeur littérale, tandis que la dépendance *type* permet d'associer un type à l'expression.

Stereotype	**RCR.SelfExpression**
Base Class	RCR.SimpleExpression
Tagged Values	
Dependencies	
Constraints	
Notes	

Le métaélément RCR.SelfExpression modélise l'expression qui dénote l'instance dans laquelle apparaît l'expression. Cette expression correspond aux mots clés *this*, *self*, etc. utilisés par les langages orientés objet.

Stereotype	**RCR.IDRefExpression**
Base Class	RCR.SimpleExpression
Tagged Values	
Dependencies	1. reference : RCR.Dep.Reference
Constraints	
Notes	

Le métaélément RCR.IDRefExpression modélise une expression qui dénote une entité définie ailleurs dans le système. La dépendance *reference* associe RCR.IDRefExpression à l'entité référée.

Stereotype	RCR.ComplexExpression
Base Class	RCR.ImperativeExpression
Tagged Values	
Dependencies	
Constraints	
Notes	

Le métaélément RCR.ComplexExpression modélise une expression complexe composée d'opérateurs (prédéfinis par le langage de programmation) qui manipulent des sous-expressions. Ces manipulations sont modélisées par les métaéléments de type RCR.Step décrits dans la section suivante.

5.3.4 Pas d'évaluation

Les pas d'évaluation représentent l'application d'opérateurs prédéfinis à des expressions pour obtenir des valeurs ou pour effectuer des changements d'état interne du logiciel en exécution.

Stereotype	RCR.Step
Base Class	UML.Foundation.Core.BehavioralFeature
Tagged Values	1. step_kind
Dependencies	
Constraints	
Notes	

Le métaélément RCR.Step est la superclasse de tous les pas d'exécution du profil RCR. Ce stéréotype est basé sur le métaélément BehavioralFeature. Le TaggedValue *step_kind* permet de raffiner au besoin le pas d'exécution modélisé. Par exemple, certains pas d'exécution (e.g., conversion de type) n'apparaissent pas explicitement dans le code car ils sont générés automatiquement pas le processus de compilation. Le TaggedValue *step_kind* peut enregistrer ce fait avec les mots clés *implict* ou *explicit* si désiré.

Stereotype	RCR.CreateStep
Base Class	RCR.Step
Tagged Values	1. createStep_kind
Dependencies	1. type : RCR.Dep.Type 2. argument : RCR.Dep.Expression
Constraints	
Notes	

Le métaélément RCR.CreateStep modélise la création d'une instance d'un type de donnée. Le type de donnée considéré est habituellement complexe (e.g. classe, tableau) par opposition à un type primitif (e.g., int, float, char, etc.). Dans plusieurs cas, la création apparaît explicitement dans le code (e.g., le mot clé *new* en C++ et Java), mais parfois elle est implicite. Par exemple, l'exécution d'une déclaration de variable locale de type classe en C++ construit implicitement une instance de cette classe et l'assigne à la variable. En Java, la variable doit être explicitement initialisée ou assignée une instance. Les détails précisant la nature exacte (e.g., malloc, calloc, new) de la création peuvent être ajoutés à l'aide du TaggedValue *createStep_kind*. La dépendance *type* identifie le type de l'instance créée. La deuxième dépendance nommée *argument* mène à la liste d'arguments destinée à préciser les caractéristiques de l'instance; ces arguments peuvent être les arguments du constructeur, les dimensions d'un tableau, et ainsi de suite.

Stereotype	RCR.DestroyStep
Base Class	RCR.Step
Tagged Values	1. destroyStep_kind
Dependencies	1. target : RCR.Dep.Expression
Constraints	
Notes	

Le métaélément RCR.DestroyStep complémente RCR.CreateStep en modélisant la destruction d'une instance d'un type de donnée. Comme pour RCR.CreateStep, le type de donnée typiquement visé par ce métaélément est complexe. Le TaggedValue *destroyStep_kind* permet de spécifier au besoin la nature exacte de la destruction (e.g., free, delete, delete[]). La dépendance *target* associe RCR.DestroyStep à la liste d'expressions spécifiant les entités à détruire.

Stereotype	**RCR.AssignStep**
Base Class	RCR.Step
Tagged Values	1. assignStep_kind
Dependencies	1. target : RCR.Dep.Expression 2. value : RCR.Dep.Expression
Constraints	
Notes	

Le métaélément RCR.AssignStep modélise l'assignation d'une valeur à une entité assignable (e.g., variable, attribut d'objet, etc.). Le TaggedValue *assignStep_kind* aide à préciser la nature exacte de l'assignation car certains langages comme C, C++ et Java proposent plusieurs variantes qui combinent l'assignation avec une opération simple (e.g., +=, -=, *=, /=, %=). La dépendance *target* associe RCR.AssignStep à l'entité à être assignée, tandis que la dépendance *value* lie RCR.AssignStep à la valeur assignée.

Stereotype	**RCR.AccessStep**
Base Class	RCR.Step
Tagged Values	1. accessStep_kind
Dependencies	1. composite : RCR.Dep.Expression 2. part : RCR.Dep.Expression
Constraints	
Notes	

Le métaélément RCR.AccessStep modélise l'accès à une composante d'une entité composée. L'entité composée est constituée de sous-entités qui sont individuellement accessibles. Typiquement, RCR.AccessStep est utilisé pour signaler l'accès à un attribut d'objet (e.g., Objet.attribut, PointeurObjet→attribut) et l'accès à une entrée de tableau (e.g., Tableau[i]). Le TaggedValue *accessStep_kind* permet de spécifier la nature exacte de l'accès si nécessaire; trois valeurs possibles tirées de C, C++ et Java sont *dot* (.), *arrow* (->) et *index* ([]). La dépendance nommée *composite* associe l'entité composée à RCR.AccessStep, tandis que *part* spécifie la composante visée par l'accès.

Stereotype	RCR.CallStep
Base Class	RCR.Step
Tagged Values	1. callStep_kind
Dependencies	1. operation : RCR.Dep.Expression 2. argument : RCR.Dep.Expression
Constraints	
Notes	

Le métaélément RCR.CallStep modélise l'invocation d'une opération. La dépendance *operation* associe RCR.CallStep à l'expression qui dénote l'opération invoquée. La deuxième dépendance, nommée *argument*, aboutit aux expressions spécifiant les arguments actuels de l'invocation. Comme pour les autres métaéléments de ce groupe, le TaggedValue dédié *callStep_kind* permet d'enregistrer au besoin des détails additionnels sur l'invocation modélisée.

Stereotype	RCR.CalculateStep
Base Class	RCR.Step
Tagged Values	1. calculateStep_kind
Dependencies	1. operand : RCR.Dep.Expression
Constraints	
Notes	

Le métaélément RCR.CalculateStep modélise un calcul primitif. L'ensemble de calculs disponibles est prédéfini par chaque langage de programmation. La dépendance *operand* associe RCR.CalculateStep à la liste d'opérandes utilisées pour le calcul. Le TaggedValue *calculateStep_kind* permet de caractériser davantage le calcul modélisé. Par exemple, les calculs fréquemment rencontrés comme l'addition, la soustraction, la multiplication et la division peuvent être représentés par les symboles suivants: +, -, *, /.

Stereotype	RCR.CompareStep
Base Class	RCR.Step
Tagged Values	1. compareStep_kind
Dependencies	1. operand : RCR.Dep.Expression
Constraints	
Notes	

Le métaélément RCR.CompareStep modélise une comparaison d'entités. Les langages de programmation définissent plusieurs variétés de comparaisons mais elles

produisent toutes une valeur qui peut être interprétée comme 'vrai' ou 'faux'. Comme pour RCR.CalculateStep, RCR.CompareStep participe à la dépendance *operand* qui l'associe aux opérandes qui sont comparées entre elles. Le TaggedValue *compareStep_kind* permet de préciser la nature exacte de la comparaison (e.g., ==, <, >, <=, >=, !=).

Stereotype	**RCR.ConvertStep**
Base Class	RCR.Step
Tagged Values	1. convertStep_kind
Dependencies	1. value : RCR.Dep.Expression 2. type : RCR.Dep.Type
Constraints	
Notes	

Le métaélément RCR.ConvertStep modélise une conversion de type. Conceptuellement, ce mécanisme utilise une valeur d'un certain type pour produire une seconde valeur d'un deuxième type. La dépendance *value* associe RCR.ConvertStep à l'expression dont le type est à convertir. La dépendance *type* spécifie le type obtenu après la conversion. Le TaggedValue *convertStep_kind* permet de préciser davantage la conversion modélisée: par exemple le langage C++ propose quatre genres de conversions distinctes: *static_cast*, *dynamic_cast*, *reinterpret_cast* et *const_cast*.

Stereotype	**RCR.ReflectionStep**
Base Class	RCR.Step
Tagged Values	1. reflectionStep_kind
Dependencies	1. entity : RCR.Dep.Reference
Constraints	
Notes	

Le métaélément RCR.ReflectionStep permet de modéliser une variété d'opérateurs qui s'intéressent aux détails de l'implantation des données par opposition aux données elles-mêmes. Par exemple, le langage C++ fournit les opérateurs *&*, *sizeof* et *typeid* qui retournent l'adresse, la taille et le type d'une variable, respectivement. Le TaggedValue *reflectionStep_kind* permet de spécifier la nature exacte de l'opérateur modélisé. Nous proposons l'utilisation des valeurs suivantes pour identifier les opérateur les plus courants: *address_of*, *size_of*, *type_of* et *dereference* (i.e.,

l'opérateur complémentaire de *address_of*). La dépendance nommée *entity* associe RCR.ReflectionStep à l'entité visée par l'opérateur. Cette entité est typiquement une expression, mais elle peut être un type connu du système (e.g., sizeof(ClasseABC)).

5.3.5 Variable

Ce groupe contient un seul métaélément: RCR.Variable.

Stereotype	**RCR.Variable**
Base Class	UML.Foundation.Core.Attribute
Tagged Values	
Dependencies	
Constraints	1. initialValue : RCR.ImperativeExpression
Notes	

Le métaélément RCR.Variable modélise une variable. En raison de leur ressemblance sémantique, le stéréotype RCR.Variable est dérivé du métaélément Attribute. Ce choix est aussi motivé en raison de la similarité des relations structurelles Classifier—Attribute et RCR.Bloc—RCR.Variable. La seule contrainte additionnelle imposée par ce stéréotype exige que l'attribut *initialValue* soit de type RCR.ImperativeExpression.

5.4 Conclusion

Ce chapitre a présenté le profil RCR. La première section a introduit avec l'aide d'un exemple simple les quatre concepts de base sur lesquels repose le profil RCR: les *blocs*, les *énoncés*, les *expressions* et les *pas d'évaluation*. Puis, les 52 nouveaux métaéléments du profil RCR ont été présentés à l'aide de diagrammes de classes organisés en cinq familles distinctes: les blocs, les énoncés, les expressions, les pas d'évaluation et les dépendances. Le métaélément unique RCR.Variable a aussi été présenté. Tous les métaéléments du profil RCR étendent un des quatre métaéléments du package UML.Foundation.Core: Attribute, BehavioralFeature, Classifier et Dependency.

La deuxième section a expliqué les deux stratégies sur lesquelles s'appuie la conception du profil RCR: la *flexibilité* et l'utilisation des *dépendances*.

La troisième section a détaillé tous les métaéléments du profil RCR en fournissant un texte descriptif pour chacun accompagné d'un tableau résumant les informations suivantes: nom du stéréotype, nom de la classe étendue, TaggedValues, dépendances, contraintes, notes.

Chapitre 6: Discussions

Cette section propose cinq discussions au sujet du profil RCR. La première discussion évalue le profil RCR par rapport aux besoins et contraintes décrits au chapitre 4. La seconde discussion compare le profil RCR à d'autres métamodèles afin de le situer parmi ses pairs. La troisième discussion montre comment le profil RCR est utilisé pour modéliser le code source. La quatrième discussion donne un aperçu des applications qui profitent du profil RCR. Finalement, la cinquième discussion s'intéresse à la mise en oeuvre du profil RCR.

6.1 Satisfaction des besoins et respect des contraintes

Dans cette section, le profil RCR est évalué en fonction des besoins et des contraintes décrits au chapitre 4. En particulier, le prochain paragraphe explique comment le profil RCR satisfait les deux besoins essentiels évoqués précédemment: le niveau d'abstraction de l'arbre syntaxique abstrait et la modélisation des énoncés et des expressions. Ensuite, cette section conclut en discutant comment le profil RCR respecte les contraintes proposées au chapitre 4: familiarité et stabilité face au métamodèle UML d'une part, et équilibre entre généralité, expressivité et efficacité d'autre part.

Le profil RCR modélise le corps d'une méthode avec des métaéléments représentant les blocs (i.e., RCR.Block), les énoncés (i.e., RCR.Statement), les expressions (i.e., RCR.ImperativeExpression), les pas d'évaluation (i.e., RCR.Step) et les variables (i.e., RCR.Variable). Tous ces concepts, à l'exception des pas d'évaluation, correspondent directement aux entités sémantiques et syntaxiques des langages de programmation impératifs. Quant à eux, les pas d'évaluation ont été conçus dans le cadre du profil RCR pour représenter succinctement et explicitement les actions

atomiques d'un programme qui sont implicites dans les divers types d'expressions des langages impératifs. Par exemple, le pas d'évaluation RCR.AccessStep modélise le concept d'accès dans les expressions comme "`x.a`", "`x->a`" et "`x[a]`". Les pas d'évaluation appartiennent donc au même niveau d'abstraction que les expressions car ils ne font que réexprimer autrement les concepts déjà présents dans celles-ci. Il s'ensuit que le profil RCR satisfait les deux besoins essentiels décrits au chapitre 4: d'une part il modélise le corps d'une méthode au même niveau d'abstraction qu'un arbre syntaxique abstrait, et d'autre part ce modèle contient des représentations des énoncés et des expressions qui définissent la méthode.

Le respect de la contrainte de familiarité vis-à-vis le métamodèle UML apparaît principalement dans le choix des noms des nouveaux métaéléments du profil RCR. Premièrement, les noms proviennent de l'anglais puisque la spécification officielle d'UML est rédigée dans cette langue. Deuxièmement, le profil RCR respecte le style de construction de noms qui consiste à débuter chaque nom par une lettre majuscule et de joindre les noms composés en les aboutant directement: par exemple, TryBlock, ForStatement, ImperativeExpression et ReflectionStep. Finalement, certains noms de métaéléments du profil RCR reprennent en partie les noms de métaéléments UML lorsqu'il y a un rapprochement conceptuel entre eux. Par exemple, les métaéléments du profil RCR CreateStep, DestroyStep et CallStep partagent des similarités conceptuelles avec les métaéléments UML CreateAction, DestroyAction et CallAction, respectivement.

La seconde contrainte associée au métamodèle UML est la stabilité. Deux choix de conception aident le profil RCR à respecter cette contrainte. Premièrement, le profil RCR n'étend que quatre des 132 métaéléments standard UML, nommément Attribute, BehavioralFeature, Classifier et Dependency qui proviennent tous du package UML.Foundation.Core. La décision de minimiser le nombre de dépendances sur les métaéléments UML comporte deux avantages principaux: la réduction des risques d'impacts lors de changements au métamodèle UML et la simplification des éventuelles mesures correctives à apporter au profil. Le deuxième choix de conception qui favorise la stabilité consiste à n'étendre que des métaéléments du

package UML.Foundation.Core. Par définition, ce package constitue le noyau du métamodèle UML; conséquemment, ses métaéléments doivent donc être aussi, sinon plus, stables que les métaéléments des autres packages du métamodèle. En particulier, le profil RCR n'étend aucun métaélément du groupe Action malgré certains rapprochements conceptuels en raison des transformations profondes envisagées par les travaux UML Action Semantics discutés à la section 3.1.2.

La conception du profil RCR a également été guidée par la contrainte d'équilibre entre la généralité, l'expressivité et l'efficacité. La généralité du profil RCR provient en majeure partie de l'approche modulaire qui a été adoptée pour la modélisation. Cette approche consiste à représenter les composantes fondamentales des corps de méthodes mais n'impose pas de combinaisons prédéfinies ou obligatoires de composantes, permettant ainsi de créer les combinaisons appropriées pour modéliser les particularités propres à chacun des langages de programmation. Par exemple, si T est une classe, la déclaration d'une variable de type T entraîne la création d'une instance de T dans le langage C++ mais non en Java. Le profil RCR permet de modéliser cette différence en incluant pour le modèle de la méthode C++ le pas d'évaluation RCR.CreateStep dans la spécification de la valeur initiale de la variable déclarée, et en l'omettant dans le cas de Java pour indiquer qu'aucune instanciation n'a lieu.

La généralité du profil RCR ne minimise pas pour autant son expressivité. Le profil RCR offre la possibilité de raffiner un modèle afin de satisfaire les besoins d'applications particulières à l'aide du mécanisme des TaggedValues. En particulier, chacun des métaéléments de la hiérarchie des pas d'évaluation (i.e., RCR.Step) possède un tag de la forme *step_kind* qui peut contenir un texte arbitraire permettant d'expliciter au besoin la nature exacte de l'élément modélisé. Prenons par exemple le cas de RCR.AccessStep mentionné précédemment. Ce pas d'évaluation modélise le concept général d'un accès, comme dans les cas des expressions `"x.a"`,`"x->a"`, et `"x[a]"`. Si la nature exacte de l'accès doit être modélisée, il suffit d'attribuer un mot-clé approprié au tag *accessStep_kind* selon le cas (e.g., "dot", "arrow", "index" respectivement). Le profil RCR a explicitement prévu ces tags pour les pas

d'évaluation car ils présentent plus de variantes potentielles que les blocs (i.e., RCR.Block), les énoncés (i.e., RCR.Statement), les expressions (i.e., RCR.ImperativeExpression) et les variables (i.e., RCR.Variable). Toutefois, UML permet l'application d'un TaggedValue arbitraire à n'importe quel métaélément si jamais un TaggedValue répondant au besoin immédiat n'existe pas.

Un reproche [Demeyer_1999a] attribué à la modélisation des méthodes avec le métamodèle UML standard est sa lourdeur en raison des constructions nécessaires pour créer des modèles bien-formés selon la spécification UML (voir par exemple la stratégie ActivityGraph à la section 3.1.1.3). Le profil RCR évite de telles lourdeurs en modélisant explicitement les entités pertinentes (e.g., blocs, énoncés, expressions, pas d'évaluation, variables) sans utiliser de constructions superflues. Cette approche accroît l'efficacité de la solution en favorisant l'accès rapide aux données et en minimisant l'espace de stockage requis. De plus, les pas d'évaluation contribuent eux-mêmes au gain d'efficacité car ils condensent plusieurs types d'expressions similaires dans quelques métaéléments seulement (e.g., RCR.CalculateStep, RCR.CompareStep, RCR.AccessStep, RCR.ReflectionStep, etc.).

En résumé, le profil RCR modélise efficacement les éléments importants du corps d'une méthode au niveau de détail souhaité pour l'ensemble des langages visés (i.e., C, C++, Java).

6.2 Comparaison aux métamodèles existants

Cette section vise à situer le profil RCR parmi les métamodèles utilisés dans la communauté de maintenance. Plus précisément, la comparaison s'intéresse au métamodèle UML enrichi du profil RCR (ci-après nommé UML+RCR) puisque le profil RCR n'est pas un métamodèle complet en soi. Nous avons retenu comme critère principal de comparaison le contenu informationnel puisque cette dimension prédomine les autres: ultimement, c'est elle qui détermine les tâches qui peuvent être effectuées avec le modèle d'un système. Les paragraphes suivants décrivent l'échelle

de mesure utilisée pour la comparaison, fournissent des exemples de métamodèles pertinents et expliquent la position de UML+RCR parmi ces métamodèles.

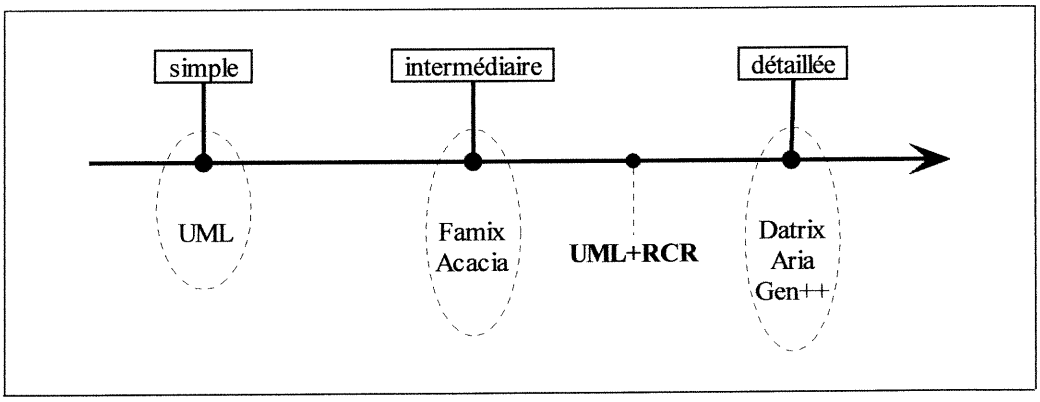


Figure 28: Situation du métamodèle UML+RCR sur l'axe de contenu informationnel

La section 4.4.1 discute de la possibilité de représenter le *corps* d'une méthode à différents niveaux d'abstraction. Cette section poursuit cette discussion en présentant les façons de représenter une *méthode* en général dans le cadre d'un modèle de système logiciel. Les divers niveaux d'abstraction sont ordonnés en termes de leur richesse informationnelle selon l'axe de la Figure 28. Cet axe représente un contenu informationnel progressivement plus riche vers son extrémité droite. L'axe possède trois jalons majeurs (i.e., *simple*, *intermédiaire*, *détaillée*) qui caractérisent les principales représentations rencontrées dans les outils populaires de la communauté de maintenance. On ne doit pas interpréter ces jalons comme étant les trois valeurs discrètes qui caractérisent tous les métamodèles; plutôt, ils doivent être considérés comme des points d'accumulation autour desquels les métamodèles s'agglomèrent. Les trois niveaux de représentation sont expliqués dans les paragraphes suivants.

La représentation *simple* ne modélise que la déclaration d'une méthode; aucun détail interne de la méthode n'est utilisé. Typiquement, cette représentation se limite au nom de la méthode, mais elle peut être enrichie progressivement jusqu'à contenir la signature complète de la méthode, c'est-à-dire son nom, son type de retour et sa liste de paramètres formels, y compris le type et la valeur par défaut de chacun d'eux. Le métamodèle UML actuel ne permet que ce type de représentation (sans l'aide des stratégies expliquées à la section 3.1.1).

La représentation *intermédiaire* complète la représentation simple en lui ajoutant de l'information sommaire dérivée à partir de la définition de la méthode. Typiquement, cette information se résume à deux listes d'éléments. La première liste identifie les méthodes qui sont appelées par la méthode modélisée tandis que la deuxième identifie les entités (e.g., attributs d'objets, variables globales, variables locales, etc.) qui sont lues ou écrites dans le corps de la méthode. Cette représentation est fréquemment rencontrée car elle peut être utilisée pour plusieurs langages de programmation et possède l'information nécessaire pour la construction des graphes de flots de contrôle interprocédural ainsi que des graphes de flots de données tout en étant relativement compacte en termes d'espace de stockage. Les métamodèles des environnements Portable Bookshelf, Acacia et FAMOOS emploient ce genre de représentation.

- L'environnement Portable Bookshelf [PBS] est une instanciation du prototype Software Bookshelf. Son but principal est de faciliter la compréhension de systèmes logiciels écrits en divers langages (e.g., C, Pascal, PL/IX) en offrant à l'usager des visualisations navigables de leurs architectures physiques.
- L'environnement Acacia [Chen_1998] vise à supporter les analyses d'atteignabilité (*reachability*) et la détection de code mort dans les systèmes C++. Ces analyses permettent l'allégement des logiciels, la sélection de tests de régression et les études de réutilisation de code.
- Le projet de réingénierie FAMOOS (Framework-based Approach for Mastering Object-Oriented Software Evolution) [FAMOOS] vise principalement à transformer des applications orientées objet de première génération en ateliers flexibles (i.e., frameworks).

La représentation *détaillée* enrichit la représentation simple avec un complément d'information beaucoup plus complet que l'information sommaire de la représentation intermédiaire. Typiquement, cette information s'apparente à l'arbre de syntaxe abstrait du corps de la méthode et peut modéliser les blocs, les énoncés, les expressions, les opérateurs primitifs ainsi que les commentaires qui définissent la méthode. Cette représentation fournit un maximum d'information mais requiert plus d'espace de stockage et s'applique plus difficilement à plusieurs langages de programmation en raison de sa spécificité. Les métamodèles associés aux outils Datrix, Gen++ et Aria utilisent la représentation détaillée.

- La suite d'outils Datrix [Mayrand_1996] est surtout utilisée pour évaluer des systèmes logiciels écrits en divers langages (e.g., C, C++, Java). Les évaluations sont principalement constituées de suites de métriques qui permettent aux évaluateurs de détecter des anomalies potentielles et de les investiguer au besoin.
- Les outils Gen++ [Devanbu_1994] et Aria [Devanbu_1996] permettent la création d'analyseurs optimisés pour la navigation de graphes. Ces deux outils sont eux-mêmes créés à l'aide de l'outil GENOA qui instancie des générateurs d'analyseurs adaptés pour une famille de graphes en particulier. Dans le cas de Gen++, les graphes à analyser correspondent aux arbres syntaxiques abstraits générés par le compilateur C++ CFront, tandis que Aria génère des analyseurs adaptés aux instances du métamodèle Reprise. Dans les deux cas, les analyseurs s'intéressent aux systèmes logiciels C++.

Comme l'indique la Figure 28, le métamodèle UML+RCR possède un pouvoir représentationnel qui se situe entre les jalons *intermédiaire* et *détaillée*. Cette observation découle du fait que le profil RCR permet de modéliser une méthode au niveau de l'arbre de syntaxe abstrait tout en étant suffisamment indépendant d'un langage de programation particulier pour être utilisable comme une représentation intermédiaire. Cette flexibilité provient surtout de l'approche modulaire décrite à la section 6.1 et du concept du pas d'évaluation qui découple le métamodèle d'un langage particulier.

6.3 Modélisation

Cette section démontre à l'aide d'exemples simples le potentiel du profil RCR pour modéliser les fragments de code retrouvés dans les langages comme C, C++ et Java. La sélection des cas présentés n'est pas exhaustive, bien sûr, mais elle permet néanmoins de comprendre les concepts fondamentaux qui sont utilisés pour l'ensemble des métaélélments du profil RCR.

Le premier exemple, la modélisation d'une méthode simple, est expliqué en détail car il introduit la notation utilisée pour les exemples suivants. De plus, cet exemple illustre comment la modélisation du corps d'une méthode avec le profil RCR est intégrée avec la modélisation UML traditionnelle.

Les exemples subséquents modélisent une déclaration de variable avec initialisation, un appel simple, un appel avec accès via un objet, une instanciation de classe assignée à une variable, une conversion de type, l'énoncé *if*, l'énoncé *switch* et l'énoncé *for*.

6.3.1 Méthode simple

```
integer f() { return 6;}
```

```
<UML.Foundation.Core.Operation uuid="%f%" name="f">
    <UML.Foundation.Core.BehavioralFeature.parameter>

        <UML.Foundation.Core.Parameter kind="return">
            <UML.Foundation.Core.Parameter.type>
                <UML.Foundation.Core.DataType idref="%integer%"/>
            </UML.Foundation.Core.Parameter.type>
        </UML.Foundation.Core.Parameter>

    </UML.Foundation.Core.BehavioralFeature.parameter>
    <UML.Foundation.Core.Operation.method>

        <UML.Foundation.Core.Method uuid="%f01%">
            <UML.Foundation.Core.Method.code>

                <RCR.Block>
                    <UML.Foundation.Core.Classifier.feature>

                        <RCR.ReturnStatement>
                            <RCR.ReturnStatement.expression>

                                <RCR.LiteralExpression literal_string="6">
                                    <RCR.LiteralExpression.type>
                                        <UML.Foundation.Core.DataType idref="%integer%"/>
                                    </RCR.LiteralExpression.type>
                                </RCR.LiteralExpression>

                            </RCR.ReturnStatement.expression>
                        </RCR.ReturnStatement>

                    </UML.Foundation.Core.Classifier.feature>
                </RCR.Block>

            </UML.Foundation.Core.Method.code>
        </UML.Foundation.Core.Method>

    </UML.Foundation.Core.Operation.method>
</UML.Foundation.Core.Operation>
```

Exemple 7: Modélisation avec le profil RCR d'une méthode simple

L'Exemple 7 illustre les concepts fondamentaux de la modélisation avec le profil RCR. La notation utilisée pour décrire ce modèle est dérivée de la notation XMI définie dans la spécification UML. Cette notation textuelle est plus concise qu'une

notation graphique et s'adapte mieux aux contraintes d'espace associées au format des pages de ce mémoire. Puisque cette notation est utilisée pour tous les exemples de cette section, une description détaillée est fournie pour le cas présent. Dans tous les exemples, le fragment de code modélisé apparaît dans le compartiment du haut, suivi du modèle dans le compartiment suivant. Les numéros de lignes dans la marge gauche ne servent qu'à faciliter l'identification des lignes d'intérêt lors des explications et ne font pas partie du modèle comme telles. Chaque entité d'intérêt dans le modèle est délimitée selon la convention XML par une balise ouvrante (i.e., <Entité>) et une balise fermante (i.e., </Entité>). La balise combinée <Entité/> combine les balises ouvrante et fermante en une seule. Les attributs textuels simples des entités sont ajoutés dans le corps de la balise ouvrante (ou combinée) comme suit: <Entité attribut="simple"/>. Les attributs à valeur plus complexe délimitent leur valeur à l'aide des balises:

```
<Entité.attributComplexe>
      <ValeurComplexe1/>
      <ValeurComplexe2/>
</Entité.attributComplexe>
```

Cette stratégie est également utilisée pour les associations ou les dépendances auxquelles participe l'entité en question:

```
<Entité.association>
      <Entité1/>
      <Entité2/>
</Entité.association>

<Entité.dépendance>
      <EntitéA/>
      <EntitéB/>
</Entité.dépendance>
```

Chaque modèle est constitué de métaéléments instanciés: les rectangles ont été rajoutés pour faciliter leur identification d'un coup d'oeil. En général, on ne doit pas nécessairement interpréter l'imbrication des rectangles comme une relation d'aggrégation ou de composition. Plutôt, l'imbrication signale la possibilité de naviguer d'un rectangle externe vers un rectangle interne. Les mécanismes de navigation ne sont pas spécifiés explicitement car les stratégies varient selon l'implantation concrète du modèle. Finalement, notons que l'intention première de

cette section est de communiquer simplement les concepts fondamentaux; nous avons donc omis les détails techniques qui auraient nuit à une présentation claire. Par exemple, tous les métaéléments étendus par un stéréotype (i.e., les métaéléments du profil RCR) apparaissent directement dans le modèle. En réalité, une instance de métaélément étendu doit être présentée comme une instance du métaélément UML de base accompagnée du stéréotype qui la modifie. De plus, le tag d'un TaggedValue est ajouté à une instance de métaélément comme un attribut textuel simple alors que techniquement une instance de TaggedValue doit être utilisée.

Examinons maintenant le modèle de l'Exemple 7. Ce modèle illustre comment la modélisation du corps d'une méthode par le profil RCR est incorporé au modèle créé par les métaéléments UML standard. La ligne 01 introduit l'instance de UML.Foundation.Core.Operation qui modélise la déclaration de la méthode. L'attribut *uuid* est un artifice notationnel qui réfère au concept d'un identificateur universel unique qui identifie cette instance parmi les autres du modèle; cet attribut n'appartient pas aux métaéléments comme tel. La valeur de *uuid* dans ce cas-ci, "%f%", illustre la convention qui est utilisée pour tous les identificateurs uniques dans les exemples de cette section: le texte de l'identificateur est borné de chaque côté par le caractère %. L'attribut *name* (hérité de UML.Foundation.Core.ModelElement) donne le nom de cette instance (i.e., "f" dans ce cas).

La ligne 02 introduit l'association *parameter* associée à la superclasse de Operation, UML.Foundation.Core.BehavioralFeature. La notation explicite de la forme *Package.Métaélément.entité* est utilisée pour préciser l'identité de l'entité (e.g., attribut, association, dépendance) qui est spécifiée. Les lignes 04 à 08 décrivent la seule instance de UML.Foundation.Core.Parameter qui est associée à cette méthode. Si cette méthode avait eu plusieurs paramètres formels, ils apparaîtraient un à la suite de l'autre selon l'ordre spécifié par la signature de la méthode. Ici, la valeur "return" de l'attribut *kind* à la ligne 04 indique que ce paramètre définit le type de retour de la méthode. Ce type est spécifié par la valeur de l'association *type* située entre les lignes 05 et 07. La ligne 06 représente une référence à une instance de UML.Foundation.Core.DataType (qui doit être définie ailleurs dans le modèle); la

valeur de l'attribut *idref* spécifie la valeur de l'attribut *uuid* (i.e., %integer%) de l'instance recherchée. Dans cet exemple, le texte "integer" utilisé comme identificateur unique est un raccourci de notation qui réfère à l'instance de DataType qui définit le type primitif "integer"; cet identificateur ne possède aucune signification particulière en soi et sa valeur aurait tout aussi bien pu être "a0000899" sans du tout affecter le modèle.

Les lignes 11 et 37 délimitent l'implantation de la méthode. Chaque implantation distincte est modélisée par une instance de UML.Foundation.Core.Method. Dans cet exemple, l'instance nécessaire est définie entre les lignes 13 et 35. La dépendance nommée *code* du profil RCR qui associe une méthode à un bloc de code est introduite à la ligne 14. Les lignes 16 et 32 délimitent l'instance de RCR.Block qui modélise le bloc de code de cette méthode. La ligne 17 introduit l'association *feature* qui donne accès au contenu du bloc. Notons que l'association *feature* est disponible ici parce que le stéréotype RCR.Block spécialise la classe de base UML.Foundation.Core.Classifier. Le seul énoncé de cette méthode, RCR.ReturnStatement, est spécifié entre les lignes 19 et 29. La valeur retournée par l'énoncé *return* est donnée par l'instance de RCR.LiteralExpression définie aux lignes 22 à 26. Le tag *literal_string* à la ligne 22 donne le texte de la valeur littérale (i.e., "6") tandis que la dépendance *type* spécifie le type de l'expression littérale. Dans ce cas, on réfère au même type primitif *integer* vu précédemment à la ligne 06.

6.3.2 Déclaration de variable avec initialisation

```
integer k = 1 + 3;
```

```
01 <RCR.DeclarationStatement>
02     <RCR.DeclarationStatement.declaration>
03
04         <RCR.Variable uuid="%k%" name="k">
05             <UML.Foundation.Core.StructuralFeature.type>
06                 <UML.Foundation.Core.DataType idref="%integer%"/>
07             </UML.Foundation.Core.StructuralFeature.type>
08             <UML.Foundation.Core.Attribute.initialValue>
09
10                 <RCR.ComplexExpression>
11                     <UML.Foundation.Core.Classifier.feature>
12
13                         <RCR.CalculateStep calculateStep_kind="+">
14                             <RCR.CalculateStep.operand>
15
16                                 <RCR.LiteralExpression literal_string="1">
17                                     <RCR.LiteralExpression.type>
18                                         <UML.Foundation.Core.DataType idref="%integer%"/>
19                                     </RCR.LiteralExpression.type>
20                                 </RCR.LiteralExpression>
21
22                                 <RCR.LiteralExpression literal_string="3">
23                                     <RCR.LiteralExpression.type>
24                                         <UML.Foundation.Core.DataType idref="%integer%"/>
25                                     </RCR.LiteralExpression.type>
26                                 </RCR.LiteralExpression>
27
28                             </RCR.CalculateStep.operand>
29                         </RCR.CalculateStep>
30
31                     </UML.Foundation.Core.Classifier.feature>
32                 </RCR.ComplexExpression>
33
34             </UML.Foundation.Core.Attribute.initialValue>
35         </RCR.Variable>
36
37     </RCR.DeclarationStatement.declaration>
38 </RCR.DeclarationStatement>
```

Exemple 8: Modélisation avec le profil RCR d'une déclaration de variable avec initialisation

L'Exemple 8 illustre la modélisation d'une déclaration de variable qui est initialisée au moment de la déclaration. Toutes les déclarations de variables locales sont modélisées par le profil RCR avec un énoncé de type RCR.DeclarationStatement. La liste de variables déclarées par cet énoncé est bornée par les balises ouvrante et fermante de la dépendance *declaration*, c'est-à-dire les lignes 02 et 37 dans ce cas-ci. Chaque variable déclarée est modélisée par une instance de RCR.Variable. Pour cet exemple,

la seule variable déclarée est représentée par l'instance de RCR.Variable décrite entre les lignes 04 et 35. L'identificateur unique et le nom de la variable sont spécifiés à la ligne 04. L'association *type* de la superclasse UML.Foundation.Core.StructuralFeature (lignes 05 à 07) indique le type de la variable. L'attribut *initialValue* de la classe de base UML.Foundation.Core.Attribute (i.e., lignes 08 et 34) contient la valeur d'initialisation de cette variable. Ici, cette valeur est définie par une expression complexe qui implique un calcul. Les pas d'évaluation de l'expression complexe sont placés entre les balises de l'association *feature* tel que décrit pour les énoncés et le métaélément RCR.Block à l'Exemple 7. Le calcul est spécifié par l'instance de RCR.CalculateStep décrit entre les lignes 13 et 29. Le tag *calculateStep_kind* à la ligne 13 précise que le calcul modélisé est une addition. Les opérandes de l'opération apparaissent entre les bornes de la dépendance *operand* (lignes 14 et 28) dans le même ordre spécifié par le code de la méthode. Dans cet exemple, chaque opérande est une expression littérale modélisée par une instance de RCR.LiteralExpression.

6.3.3 Appel simple

```
f(3);
```

```
01 <RCR.ExpressionStatement>
02     <RCR.ExpressionStatement.expression>
03
04         <RCR.ComplexExpression>
05             <UML.Foundation.Core.Classifier.feature>
06
07                 <RCR.CallStep>
08                     <RCR.CallStep.operation>
09
10                         <RCR.IDRefExpression>
11                             <UML.Foundation.Core.Operation idref="%f%"/>
12                         </RCR.IDRefExpression>
13
14                     </RCR.CallStep.operation>
15                     <RCR.CallStep.argument>
16
17                         <RCR.LiteralExpression literal_string="3">
18                             <RCR.LiteralExpression.type>
19                                 <UML.Foundation.Core.DataType idref="%integer%"/>
20                             </RCR.LiteralExpression.type>
21                         </RCR.LiteralExpression>
22
23                     </RCR.CallStep.argument>
24                 </RCR.CallStep>
25
26             </UML.Foundation.Core.Classifier.feature>
27         </RCR.ComplexExpression>
28
29     </RCR.ExpressionStatement.expression>
30 </RCR.ExpressionStatement>
```

Exemple 9: Modélisation avec le profil RCR d'un appel simple

L'Exemple 9 illustre la modélisation d'un appel de méthode avec un argument simple. Puisque le seul but de l'énoncé est l'évaluation de l'expression qu'il contient, l'énoncé est modélisé par une instance de RCR.ExpressionStatement. L'expression à évaluer est définie par l'instance de RCR.ComplexExpression entre les lignes 04 et 27. L'appel comme tel est modélisé par l'instance de RCR.CallStep.

La première dépendance de RCR.CallStep, *operation*, spécifie l'expression qui détermine quelle méthode est invoquée. Ici, une instance de RCR.IDRefExpression est utilisée puisque le symbole "f" réfère à une entité déclarée ailleurs dans le code. Cette entité est indiquée par la référence à la ligne 11, c'est-à-dire l'instance de

UML.Foundation.Core.Operation dont la valeur de l'attribut *uuid* est "%f%". La deuxième dépendance de RCR.CallStep, *argument*, spécifie les arguments actuels de l'appel. Pour cet exemple, la valeur littérale est modélisée par l'instance de RCR.LiteralExpression.

6.3.4 Appel par accès via un objet

```
obj.m("X");
```

```
01 <RCR.ExpressionStatement>
02     <RCR.ExpressionStatement.expression>
03
04         <RCR.ComplexExpression>
05             <UML.Foundation.Core.Classifier.feature>
06
07                 <RCR.CallStep>
08                     <RCR.CallStep.operation>
09
10                         <RCR.ComplexExpression>
11                             <UML.Foundation.Core.Classifier.feature>
12
13                                 <RCR.AccessStep accessStep_kind="dot">
14                                     <RCR.AccessStep.composite>
15
16                                         <RCR.IDRefExpression>
17                                             <RCR.Variable idref="%obj%"/>
18                                         </RCR.IDRefExpression>
19
20                                     </RCR.AccessStep.composite>
21                                     <RCR.AccessStep.part>
22
23                                         <RCR.IDRefExpression>
24                                             <UML.Foundation.Core.Operation idref="%m%"/>
25                                         </RCR.IDRefExpression>
26
27                                     </RCR.AccessStep.part>
28                                 </RCR.AccessStep>
29
30                             </UML.Foundation.Core.Classifier.feature>
31                         </RCR.ComplexExpression>
32
33                     </RCR.CallStep.operation>
34                     <RCR.CallStep.argument>
35
36                         <RCR.LiteralExpression literal_string="X">
37                             <RCR.LiteralExpression.type>
38                                 <UML.Foundation.Core.DataType idref="%string%"/>
39                             </RCR.LiteralExpression.type>
40                         </RCR.LiteralExpression>
41
42                     </RCR.CallStep.argument>
43                 </RCR.CallStep>
44
45             </UML.Foundation.Core.Classifier.feature>
46         </RCR.ComplexExpression>
47
48     </RCR.ExpressionStatement.expression>
49 </RCR.ExpressionStatement>
```

Exemple 10: Modélisation avec le profil RCR d'un appel par accès via un objet

L'Exemple 10 illustre la modélisation de l'appel d'une méthode par l'intermédiaire d'un objet. Comme pour l'exemple précédent, cet énoncé est modélisé par une

instance de RCR.ExpressionStatement. L'instance de RCR.ComplexExpression (lignes 04 à 46) modélise l'expression à évaluer et l'appel de méthode est modélisé par l'instance de RCR.CallStep (lignes 07 à 43). Contrairement à l'Exemple 9 cependant, l'expression définissant la méthode appelée est complexe (i.e., RCR.ComplexExpression) plutôt que simple (i.e., RCR.IDRefExpression). Cette expression complexe (lignes 10 à 31) contient l'instance de RCR.AccessStep qui modélise l'accès à l'objet. La valeur "dot" du tag *accessStep_kind* à la ligne 13 précise que cet accès s'effectue avec l'opérateur "point" (i.e., "."). La dépendance *composite* de RCR.AccessStep (lignes 14 à 20) définit l'entité composée qui est accédée; dans ce cas-ci, cette entité est l'objet représenté par la variable "obj" tel qu'indiqué par l'instance de RCR.IDRefExpression (ligne 17). La dépendance *part* spécifie la composante particulière de l'entité composée qui est accédée. Dans cet exemple, cette composante est la méthode identifiée par l'instance de RCR.IDRefExpression à la ligne 24. L'argument de l'appel est spécifié comme dans l'exemple précédent avec la dépendance *argument* (lignes 34 à 42); dans ce cas-ci, c'est la valeur littérale "X" de type "string".

6.3.5 Instanciation de classe assignée à une variable

```
z = new T(n);
```

```
01 <RCR.ExpressionStatement>
02     <RCR.ExpressionStatement.expression>
03
04         <RCR.ComplexExpression>
05             <UML.Foundation.Core.Classifier.feature>
06
07                 <RCR.AssignStep assignStep_kind="=">
08                     <RCR.AssignStep.target>
09
10                         <RCR.IDRefExpression>
11                             <RCR.Variable idref="%z%"/>
12                         </RCR.IDRefExpression>
13
14                     </RCR.AssignStep.target>
15                     <RCR.AssignStep.value>
16
17                         <RCR.ComplexExpression>
18                             <UML.Foundation.Core.Classifier.feature>
19
20                                 <RCR.CreateStep step_kind="explicit" createStep_kind="new">
21                                     <RCR.CreateStep.type>
22                                         <UML.Foundation.Core.Classifier idref="%T%"/>
23                                     </RCR.CreateStep.type>
24                                     <RCR.CreateStep.argument>
25
26                                         <RCR.IDRefExpression>
27                                             <RCR.Variable idref="%n%"/>
28                                         </RCR.IDRefExpression>
29
30                                     </RCR.CreateStep.argument>
31                                 </RCR.CreateStep>
32
33                             </UML.Foundation.Core.Classifier.feature>
34                         </RCR.ComplexExpression>
35
36                     </RCR.AssignStep.value>
37                 </RCR.AssignStep>
38
39             </UML.Foundation.Core.Classifier.feature>
40         </RCR.ComplexExpression>
41
42     </RCR.ExpressionStatement.expression>
43 </RCR.ExpressionStatement>
```

Exemple 11: Modélisation avec le profil RCR d'une instanciation de classe assignée à une variable

L'Exemple 11 illustre la modélisation d'une variable qui est assignée une instance de classe créée explicitement avec l'opérateur *new* rencontré par exemple dans les langages C++ et Java. Comme les exemples précédents, l'énoncé est modélisé par une instance de RCR.ExpressionStatement (lignes 01 à 43) et l'expression à évaluer est

spécifiée par l'instance de RCR.ComplexExpression (lignes 04 à 40). Dans cet exemple, le pas d'évaluation dominant est l'assignation modélisée par l'instance de RCR.AssignStep (lignes 07 à 37). Le tag *accessStep_kind* à la ligne 07 précise que cette assignation est simple (i.e., "="). La dépendance *target* (lignes 08 à 14) indique la variable qui reçoit la valeur de l'assignation; l'instance de RCR.IDRefExpression identifie cette variable exactement (i.e., la variable "z"). La valeur assignée est déterminée par l'expression bornée par la dépendance *value* (lignes 15 à 36). Ici, cette expression est donnée par l'instance de RCR.ComplexExpression (lignes 17 à 34) qui contient une instance du pas d'évaluation RCR.CreateStep (lignes 20 à 31) qui modélise l'instanciation dans cet exemple. Le tag *step_kind* à la ligne 20 indique par la valeur "explicit" que l'instanciation a été spécifiée explicitement dans le code et n'est pas générée par le processus de compilation. De plus, l'instanciation est obtenue à l'aide de l'opérateur "new" tel que l'indique le tag *createStep_kind* sur la même ligne. La dépendance *type* entre les lignes 21 et 23 spécifie le type qui est instancié, c'est-à-dire le type T indiqué à la ligne 22. L'instanciation est dirigée par les arguments identifiés par la dépendance *argument* aux lignes 24 à 30. Ici, le seul argument est modélisé par l'instance de RCR.IDRefExpression qui réfère à la variable "n" (ligne 27).

6.3.6 Conversion de type

```
x *= (double) 5;
```

```
01 <RCR.ExpressionStatement>
02     <RCR.ExpressionStatement.expression>
03
04         <RCR.ComplexExpression>
05             <UML.Foundation.Core.Classifier.feature>
06
07                 <RCR.AssignStep assignStep_kind="*=">
08                     <RCR.AssignStep.target>
09
10                         <RCR.IDRefExpression>
11                             <RCR.Variable idref="%x%"/>
12                         </RCR.IDRefExpression>
13
14                     </RCR.AssignStep.target>
15                     <RCR.AssignStep.value>
16
17                         <RCR.ComplexExpression>
18                             <UML.Foundation.Core.Classifier.feature>
19
20                                 <RCR.ConvertStep step_kind="explicit">
21                                     <RCR.ConvertStep.type>
22                                         <UML.Foundation.Core.DataType idref="%double%"/>
23                                     </RCR.ConvertStep.type>
24                                     <RCR.ConvertStep.value>
25
26                                         <RCR.LiteralExpression literal_string="5">
27                                             <RCR.LiteralExpression.type>
28                                                 <UML.Foundation.Core.DataType idref="%integer%"/>
29                                             </RCR.LiteralExpression.type>
30                                         </RCR.LiteralExpression>
31
32                                     </RCR.ConvertStep.value>
33                                 </RCR.ConvertStep>
34
35                             </UML.Foundation.Core.Classifier.feature>
36                         </RCR.ComplexExpression>
37
38                     </RCR.AssignStep.value>
39                 </RCR.AssignStep>
40
41             </UML.Foundation.Core.Classifier.feature>
42         </RCR.ComplexExpression>
43
44     </RCR.ExpressionStatement.expression>
45 </RCR.ExpressionStatement>
```

Exemple 12: Modélisation avec le profil RCR d'une conversion de type

L'Exemple 12 illustre la modélisation d'une variable qui est assignée une valeur dont le type est explicitement converti. Puisque cet exemple ressemble au précédent, seuls les détails nouveaux sont discutés. Le premier détail à noter est la valeur du tag *assignStep_kind* à la ligne 07: dans ce cas-ci, la valeur "*=" dénote l'assignation composée avec l'opération '*' que l'on retrouve dans les langages comme C, C++ et Java. L'autre détail important est l'utilisation du pas d'évaluation RCR.ConvertStep

(lignes 20 à 33). Les deux dépendances de RCR.ConvertStep, *type* et *value*, définissent le nouveau type désiré et la valeur à convertir, respectivement. Concrètement, la dépendance *type* est représentée aux lignes 21 à 23 et le type instancié est spécifié à la ligne 22 (i.e., le type primitif "double"). La dépendance *value* (lignes 24 à 32) spécifie la valeur à convertir, en l'occurrence la valeur littérale définie par l'instance de RCR.LiteralExpression aux lignes 26 à 30.

6.3.7 Énoncé de branchement: *if*

```
if (m > n) max = m; else max = n;
```

```
01 <RCR.IfStatement>
02     <RCR.BranchStatement.value>
03
04         <RCR.ComplexExpression>
05             <UML.Foundation.Core.Classifier.feature>
06
07                 <RCR.CompareStep compareStep_kind=">">
08                     <RCR.CompareStep.operand>
09
10                         <RCR.IDRefExpression>
11                             <RCR.Variable idref="%m%"/>
12                         </RCR.IDRefExpression>
13
14                         <RCR.IDRefExpression>
15                             <RCR.Variable idref="%n%"/>
16                         </RCR.IDRefExpression>
17
18                     </RCR.CompareStep.operand>
19                 </RCR.CompareStep>
20
21             </UML.Foundation.Core.Classifier.feature>
22         </RCR.ComplexExpression>
23
24     </RCR.BranchStatement.value>
25     <RCR.BranchStatement.branch>
26
27         <RCR.ExpressionStatement>
28             <RCR.ExpressionStatement.expression>
29
30                 <RCR.ComplexExpression>
31                     <UML.Foundation.Core.Classifier.feature>
32
33                         <RCR.AssignStep assignStep_kind="=">
34                             <RCR.AssignStep.target>
35
36                                 <RCR.IDRefExpression>
37                                     <RCR.Variable idref="%max%"/>
38                                 </RCR.IDRefExpression>
39
40                             </RCR.AssignStep.target>
41                             <RCR.AssignStep.value>
42
43                                 <RCR.IDRefExpression>
44                                     <RCR.Variable idref="%m%"/>
45                                 </RCR.IDRefExpression>
46
47                             </RCR.AssignStep.value>
48                         </RCR.AssignStep>
49
50                     </UML.Foundation.Core.Classifier.feature>
51                 </RCR.ComplexExpression>
52
53             </RCR.ExpressionStatement.expression>
54         </RCR.ExpressionStatement>
55
56         <RCR.ElseStatement>
57             <RCR.ElseStatement.statement>
58                 <!-- répéter les lignes 27 à 54 et remplacer n pour m à la ligne 44 -->
59             </RCR.ElseStatement.statement>
60         </RCR.ElseStatement>
61
62     </RCR.BranchStatement.branch>
63 </RCR.IfStatement>
```

Exemple 13: Modélisation avec le profil RCR d'un énoncé *if*

L'Exemple 13 illustre la modélisation d'un énoncé *if*. L'énoncé est modélisé par une instance de RCR.IfStatement (lignes 01 à 63). Comme tous les énoncés descendants de l'énoncé RCR.BranchStatement, l'énoncé *if* participe à deux dépendances. La première nommée *value* spécifie la valeur de branchement tandis que la seconde nommée *branch* détermine la liste d'énoncés candidats pour le branchement. Ici, la valeur de branchement est spécifiée par l'instance de RCR.ComplexExpression (lignes 04 à 22). L'instance de RCR.CompareStep (lignes 07 à 19) indique que l'expression complexe est en fait une comparaison. Plus exactement, la comparaison est "plus grand que" (i.e., ">") comme l'indique le tag *compareStep_kind* à la ligne 07. Les opérandes de cette comparaison sont spécifiées dans l'ordre approprié entre les bornes de la dépendance *operand* (lignes 08 à 18). Pour cet exemple, les deux instances de RCR.IDRefExpression réfèrent aux variables "m" et "n" définies ailleurs. Puisque la valeur de branchement d'un énoncé *if* est booléenne, deux branches seulement sont possibles. Ces deux énoncés sont donnés par la dépendance *branch* bornée par les lignes 25 et 62. Puisque l'énoncé pour le choix *true* apparaît en premier dans le code, cette convention est respectée dans le modèle également. Le premier énoncé (lignes 27 à 54) comporte une assignation modélisée par une instance de RCR.ExpressionStatement similiaire à l'Exemple 11 vu précédemment. Le second énoncé est une instanciation de RCR.ElseStatement (lignes 56 à 60). L'énoncé qui lui est associé est donné par la dépendance *statement* (lignes 57 à 59). Dans ce cas-ci, l'énoncé est identique à l'énoncé de la branche *true* à l'exception de la référence à la variable "m" qui doit être remplacée par une référence à la variable "n".

6.3.8 Énoncé de branchement: *switch*

```
switch (v) case 1:  m1(); break;
           case 2:  m2(); break;
           default: m3();
```

```
01 <RCR.SwitchStatement>
02     <RCR.BranchStatement.value>
03
04         <RCR.IDRefExpression>
05             <RCR.Variable idref="%v%"/>
06         </RCR.IDRefExpression>
07
08     </RCR.BranchStatement.value>
09     <RCR.BranchStatement.branch>
10
11         <RCR.CaseStatement>
12             <RCR.CaseStatement.value>
13
14                 <RCR.LiteralExpression literal_string="1">
15                     <RCR.LiteralExpression.type>
16                         <UML.Foundation.Core.DataType idref="%integer%"/>
17                     </RCR.LiteralExpression.type>
18                 </RCR.LiteralExpression>
19
20             </RCR.CaseStatement.value>
21             <RCR.CaseStatement.statement>
22
23                 <!-- m1() -->
24
25                 <RCR.BreakStatement/>
26
27             </RCR.CaseStatement.statement>
28         </RCR.CaseStatement>
29
30         <RCR.CaseStatement>
31             <RCR.CaseStatement.value>
32
33                 <!-- 2 -->
34
35             </RCR.CaseStatement.value>
36             <RCR.CaseStatement.statement>
37
38                 <!-- m2() -->
39
40                 <RCR.BreakStatement/>
41
42             </RCR.CaseStatement.statement>
43         </RCR.CaseStatement>
44
45         <RCR.ElseStatement>
46             <RCR.ElseStatement.statement>
47
48                 <!-- m3() -->
49
50             </RCR.ElseStatement.statement>
51         </RCR.ElseStatement>
52
53     </RCR.BranchStatement.branch>
54 </RCR.SwitchStatement>
```

Exemple 14: Modélisation avec le profil RCR d'un énoncé *switch*

L'Exemple 14 illustre la modélisation d'un énoncé *switch*. L'énoncé est modélisé par une instance de RCR.SwitchStatement. La dépendance *value* bornée par les lignes 02 et 08 donne la valeur de branchement pour l'énoncé qui en l'occurrence est une référence à la variable "v", telle que spécifiée par l'instance de RCR.IDRefExpression (lignes 04 à 06). La dépendance *branch* (lignes 09 à 53) contient la liste des énoncés candidats du branchement comme expliqué à l'Exemple 13. Les énoncés *case* sont modélisés par des instances de RCR.CaseStatement tandis que l'énoncé *default* est modélisé par une instance de RCR.ElseStatement. Puisque les deux énoncés *case* sont similaires, le premier seulement est décrit (lignes 11 à 28). Cet énoncé est associé à la valeur de branchement (i.e., 1) définie par l'instance de RCR.LiteralExpression (lignes 14 à 18). La liste d'énoncés associés à cet énoncé *case* est accessible à partir de la dépendance *statement* (lignes 21 à 27). Le premier énoncé de cette liste est un appel de méthode (ligne 23) modélisé avec une instance de RCR.ExpressionStatement tel que décrit dans l'Exemple 9. L'énoncé *break* suivant est modélisé par une instance de RCR.BreakStatement (ligne 25). Finalement, les lignes 45 à 51 décrivent l'énoncé *default* qui est modélisé comme celui de l'énoncé if de l'Exemple 13.

6.3.9 Énoncé d'itération: *for*

```
for (integer i = 0; i < 10; i++) x[i] = 0;
```

```
01 <RCR.ForStatement>
02     <RCR.ForStatement.declaration>
03
04         <RCR.DeclarationStatement>
05             <!-- integer i = 0 -->
06         </RCR.DeclarationStatement>
07
08     </RCR.ForStatement.declaration>
09     <RCR.IterationStatement.condition>
10
11         <RCR.ComplexExpression>
12             <!-- i < 10 -->
13         </RCR.ComplexExpression>
14
15     </RCR.IterationStatement.condition>
16     <RCR.ForStatement.update>
17
18         <RCR.ComplexExpression>
19             <!-- i++ -->
20         </RCR.ComplexExpression>
21
22     </RCR.ForStatement.update>
23     <RCR.IterationStatement.statement>
24
25         <RCR.ExpressionStatement>
26             <!-- x[i] = 0 -->
27         </RCR.ExpressionStatement>
28
29     </RCR.IterationStatement.statement>
30 </RCR.ForStatement>
```

Exemple 15: Modélisation avec le profil RCR d'un énoncé *for*

L'Exemple 15 illustre la modélisation d'un énoncé *for*. L'énoncé est modélisé par une instance de RCR.ForStatement (lignes 01 à 30). L'énoncé RCR.ForStatement est le plus complexe des énoncés d'itération dérivés de RCR.IterationStatement. En plus de participer aux dépendances *condition* et *statement*, l'énoncé RCR.ForStatement permet de spécifier des déclarations de variables avec la dépendance *declaration* et des expressions de mise-à-jour avec la dépendance *update*. Dans cet exemple, la dépendance *déclaration* est bornée par les lignes 02 à 08 et la déclaration elle-même est modélisée par une instance de RCR.DeclarationStatement (lignes 04 à 06) similaire à celle de l'Exemple 8 détaillée précédemment. La dépendance *condition*

(lignes 09 à 15) contient la condition d'itération spécifiée par l'instance de RCR.ComplexExpression (lignes 11 à 13) semblable à celle de l'Exemple 13. L'expression de mise-à-jour apparaît entre les ligne 16 et 22 qui bornent la dépendance *update*. Ici, cette expression est modélisée par une instance de RCR.ComplexExpression semblable à celle de l'Exemple 8. Finalement, l'énoncé contrôlé dans cette itération est donné par la dépendance *statement* (lignes 23 à 29). Dans cet exemple, l'énoncé est une instance de RCR.ExpressionStatement similaire à celle de l'Exemple 11.

6.4 Applications

Les modèles de systèmes logiciels créés à partir du métamodèle UML+RCR offrent un niveau de détail difficilement atteignable avec le métamodèle UML standard. Ces modèles plus détaillés élargissent la gamme d'analyses possibles en rendant disponible les entités que le métamodèle UML standard ne peut pas modéliser. Par exemple, la construction de graphes de flot de contrôle et de graphes de flot de données devient possible parce que le métamodèle UML+RCR modélise les appels de méthodes et l'utilisation de variables.

L'atelier SPOOL dans son état actuel permet plusieurs activités dont la rétroconception [Schauer_1998], la navigation [Robitaille_2000], la visualisation [Schauer_1998] et l'analyse d'impact [Kabaili_1999, Chaumun_2000, Kabaili_2000, Kabaili_2000a]. En se basant d'une part sur ces résultats et d'autre part sur le fait que le métamodèle UML+RCR est beaucoup plus riche que le métamodèle SPOOL courant, nous offrons dans les paragraphes suivants notre appréciation du métamodèle UML+RCR. Nous rappelons toutefois qu'une validation en bonne et dûe forme de l'extension RCR nécessitera sa mise en oeuvre dans des environnements UML réels ainsi que son utilisation pour des tâches concrètes de rétroconception, de compréhension et de réingénierie.

Pour l'activité de rétroconception, les modèles UML+RCR offrent beaucoup plus d'information sur les systèmes étudiés. Cette information permet l'identification plus précise des composantes du système et expose plus clairement leurs collaborations facilitant ainsi la création de représentations alternatives ou à d'autres niveaux d'abstraction. Par exemple, en modélisant la nature et l'organisation des énoncés, les modèles UML+RCR facilitent entre autres la détection des centres de contrôle du système (e.g., présence de l'énoncé *switch*) autour desquels se rassemblent les composantes auxiliaires. De plus, divers types d'interdépendances entre classes et entre modules peuvent maintenant être calculées parce que les modèles UML+RCR modélisent les appels de méthodes ainsi que l'utilisation d'attributs de classe et de variables locales.

L'activité de compréhension profite également des modèles détaillés UML+RCR. En plus des énoncés, des appels de méthode et des accès aux attributs et aux variables mentionnés ci-haut, le profil RCR modélise les expressions simples et les expressions complexes qui comprennent la création et la destruction d'objets, les calculs, les comparaisons, les conversions de type et les opérations réflexives (i.e., RCR.ReflectionStep). Toutes ces informations peuvent être analysées (voir section 2.2.1), mesurées (voir section 2.2.2) et visualisées afin de caractériser davantage le système logiciel, contribuant indéniablement à l'enrichissement du modèle mental du système à comprendre.

Les modèles détaillés UML+RCR s'avèrent aussi très utiles pour l'activité de réingénierie. Non seulement facilitent-ils la détection d'anomalies (e.g., code mort ou inapproprié, constructions indésirables, clones), ils permettent également l'application de mesures correctives (e.g., remplacement ou élimination de code) par manipulation directe du modèle ainsi que l'élaboration de tests de régression plus précis. De plus, le code généré automatiquement à partir d'un modèle UML+RCR peut être beaucoup plus complet que celui généré à partir d'un modèle UML standard.

Finalement, l'utilisation du protocole XMI pour échanger les modèles UML+RCR offre une solution potentielle à la communauté de maintenance qui recherche toujours un format d'échange de modèles.

6.5 Mise en oeuvre

Cette section discute des besoins et des stratégies de la mise en oeuvre du profil RCR. Deux cas sont présentés: la mise en oeuvre du profil RCR dans un contexte général et dans le contexte du projet SPOOL.

6.5.1 Cas général

Le profil RCR a été créé pour étendre le métamodèle UML avec les mécanismes d'extension standard décrits dans la spécification UML, c'est-à-dire les métaéléments Stereotype, TaggedValue et Constraint. La mise en oeuvre du profil RCR nécessite donc au minimum les deux items suivants:

- le métamodéle UML complet
  ou
  un métamodéle UML partiel
  (avec les métaéléments ModelElement, Method, Classifier, BehavioralFeature, Attribute, Dependency, Stereotype, TaggedValue et Constraint, au minimum)
- un mécanisme de gestion de profil

Le mécanisme de gestion de profil assure l'intégration des métaéléments étendus au sein du métamodèle déjà en place. Il doit être en mesure de reconnaître un métaélément étendu parmi les métaéléments standard et de le traiter d'une façon qui est appropriée selon le contexte. Idéalement, ce mécanisme facilite la création de nouveaux profils ou la modification de profils existants. L'atelier de génie logiciel commercial Objecteering [Softeam], entre autres, propose un tel gestionnaire de profil.

6.5.2 Cas SPOOL

L'atelier de rétroconception SPOOL s'appuie sur un dépôt dont le schéma est inspiré du métamodèle UML. Cette section discute de la mise en oeuvre du profil RCR dans cet environnement en présentant d'abord l'état actuel de l'atelier et ensuite en suggérant deux stratégies potentielles.

6.5.2.1 État actuel

L'état actuel de l'atelier de rétroconception SPOOL est détaillé dans [Schauer_2001]. Nous rappelons ici les points pertinents de cette description pour notre discussion.

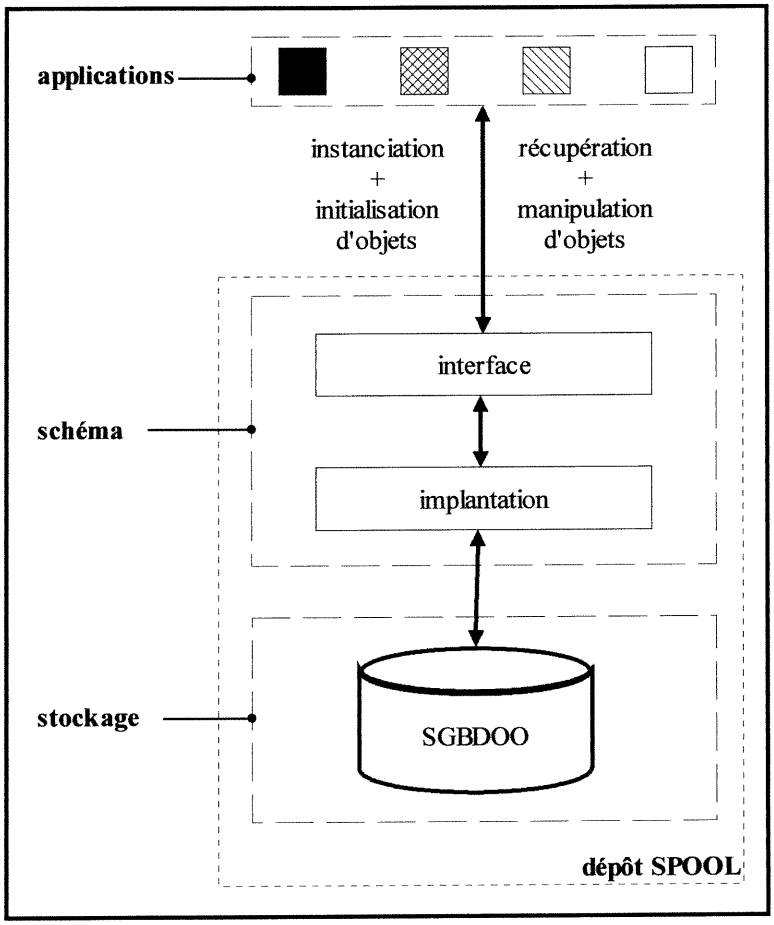


Figure 29: Architecture générale de l'environnement SPOOL (adapté de [Schauer_2001])

La Figure 29 illustre l'architecture générale de l'atelier SPOOL. L'architecture est organisée en trois couches conceptuelles: les applications, le schéma et le stockage. Le schéma définit la hiérarchie de classes qui assurent les fonctionnalités requises du dépôt: modélisation des systèmes logiciels à analyser, modélisation des entités (e.g., design patterns) recherchées et trouvées, mécanismes de navigation et de recherche (e.g., allContainedElements()), mécanismes de gestion des modifications (e.g., design pattern *Observer*), et ainsi de suite. De plus, le schéma définit les interfaces publiques qui permettent aux applications de la couche supérieure d'accéder et de modifier l'instance du schéma. La couche de stockage est utilisée pour assurer la persistence de l'instance du schéma et pour gérer les modifications qui y sont apportées. Dans le cas de SPOOL, le schéma est implanté à l'aide de classes Java. Par conséquent, le stockage est assuré par un système de gestion de base de données orienté objet (SGBDOO).

Le schéma de SPOOL (ci-après nommé SPOOL-UML) est majoritairement basé sur les métaéléments des packages UML.Foundation.Core, UML.Foundation.DataTypes, UML.Foundation.ExtensionMechanisms et UML.GeneralMechanisms.PackageManagement du métamodèle UML 1.1. En raison des besoins des activités de rétroconception et de compréhension, le schéma SPOOL-UML a été étendu pour modéliser les appels de méthodes, la création d'objets et l'utilisation de variables. L'appel de méthode et la création d'objet ont été modélisées par les métaéléments CallAction et CreateAction, respectivement. La liste des actions déclenchées par une méthode a été ajoutée au métaélément BehavioralFeature. L'utilisation de variable (i.e., lecture ou écriture) a été modélisée par l'ajout d'une liste au métaélément Method qui emmagasine les variables utilisées dans le corps de la méthode.

Quoique le schéma du dépôt de SPOOL soit inspiré du métamodèle UML, il ne peut pas être considéré comme un schéma UML *conforme* selon la spécification UML pour les raisons suivantes:

- SPOOL-UML n'utilise qu'un sous-ensemble du métamodèle UML

  En particulier, Stereotype et Constraint du package UML.Foundation.ExtensionMechanisms sont absents.

- les interfaces de SPOOL-UML et UML sont différentes

  Par exemple, l'interface SPOOL-UML pour Method contient 18 opérations tandis que celle de UML en contient 4. De plus, les opérations d'une interface UML ne sont pas toujours reproduites dans la version SPOOL-UML. Dans le cas de Method, l'interface SPOOL-UML n'a aucun équivalent pour les opérations body() et setBody() présentes dans la version UML.

- les métaéléments de SPOOL-UML et UML ont des fonctionnalités différentes

  Par exemple, la valeur d'un TaggedValue dans SPOOL-UML peut être un objet quelconque tandis qu'elle doit être textuelle dans UML.

- les hiérarchies de métaéléments de SPOOL-UML et UML sont différentes

  Par exemple, GeneralizableElement est une sous-classe de Namespace dans SPOOL-UML mais non dans UML.

- SPOOL-UML contient des métaéléments qui n'appartiennent pas au métamodèle UML

  Par exemple, les métaéléments Connection et DesignConstituent ont été rajoutés pour satisfaire les besoins particuliers de l'atelier SPOOL. De plus, de nombreux stéréotypes UML ont été "promus" au niveau de métaéléments à part entière dans SPOOL-UML tels que File et Utility pour des raisons de performance.

Dans le cadre de la présente discussion, la non-conformité[19] de SPOOL-UML signifie en particulier que les mécanismes d'extension prévus par UML (i.e., TaggedValue, Stereotype, Constraint) ne sont pas disponibles pour réaliser l'implantation du profil RCR tel qu'il est défini dans ce mémoire. La section suivante propose des stratégies alternatives pour intégrer les métaéléments du profil RCR dans l'atelier SPOOL.

[19] Notons que la conformité parfaite au métamodèle UML n'était pas le but visé par les concepteurs de SPOOL-UML. Ils cherchaient plutôt à concevoir un schéma adapté spécifiquement aux besoins concrets du dépôt d'un atelier de rétroconception.

6.5.2.2 Stratégies de mise en oeuvre

Nous présentons maintenant deux stratégies potentielles pour la mise en oeuvre du profil RCR dans l'atelier SPOOL tel qu'il est décrit à la section précédente. Ce choix de stratégies n'est pas exhaustif mais il offre néanmoins un aperçu général des options disponibles et du travail requis pour chacune.

La première stratégie est plus *propre* mais elle est aussi la plus difficile à réaliser. De plus, elle exige la modification de toutes les applications qui utilisent le dépôt. Elle consiste à concevoir à partir de zéro le schéma SPOOL-UML afin de lui intégrer un mécanisme de gestion de profil tel que décrit à la section 6.5.1. Ainsi, il devient possible d'étendre le schéma avec des profils arbitraires à n'importe quel moment et sans effort majeur. Une variante possible sur ce thème consiste à redéfinir le schéma en le basant cette fois-ci non pas sur le métamodèle UML mais plutôt sur le métamodèle UML+RCR. Naturellement, la possibilité d'intégrer des profils arbitraires est alors abandonnée.

La seconde stratégie est moins radicale que la première mais requiert tout de même des corrections majeures à l'implantation du schéma et aux applications puisque la nouvelle modélisation du corps de méthode est complètement différente et malheureusement incompatible avec l'approche en place. Les étapes majeures de cette approche sont:

- *IMPLANTATION*
  Implanter les métaéléments du profil RCR de la même façon que les métaéléments déjà en place, en prenant soin de respecter leur position dans la hiérarchie de classes (e.g., RCR.Block est une sous-classe de Classifier). Notons qu'il n'est pas nécessaire d'implanter les dépendances (i.e., RCR.Dep.*) avec cette stratégie.

  Pour chaque TaggedValue associé à un métaélément du profil, ajouter un attribut de même nom et de type String à la classe qui implante le métaélément. Par exemple, ajouter l'attribut *createStep_kind* à la classe qui implante RCR.CreateStep.

  Pour chaque dépendance (i.e., Dependency) dont le *client* est un métaélément du profil, ajouter un attribut du même nom et de type *liste* à la classe qui implante le métaélément. Le type exact de liste est déterminé par le type du *supplier* de la

dépendance. Par exemple, RCR.Dep.Statement implique une liste de métaéléments RCR.Statement.

Ajouter à ces nouvelles classes les méthodes utilitaires nécessaires (e.g., get, set) en se basant sur les implantations de métaéléments existants.

Créer les interfaces pour manipuler ces métaéléments en s'inspirant des interfaces existantes.

- *AJOUTS*
  Ajouter à l'implantation du métaélément Method un attribut qui permet de sauvegarder des objets de type RCR.Block.

  Ajouter à l'implantation et à l'interface du métaélément Method une méthode d'accès nommée *code* qui retourne la liste de RCR.Block contenue par cette méthode.

  Ajouter le code nécessaire pour instancier et initialiser les nouveaux métaéléments RCR en se basant sur l'approche existante pour les métaéléments SPOOL-UML. Par exemple, mettre à jour les fabriques d'objets au besoin (e.g., AbstractFactory, ConcreteFactory).

- *CORRECTIONS*
  Éliminer si désiré les métaéléments Action, CallAction, CreateAction ainsi que leurs interfaces.

  Corriger au besoin tout le code qui touche aux actions (e.g., Action, CallAction, CreateAction), en particulier BehavioralFeature.

  Éliminer si désiré les métaéléments Variable et LocalAttribute ainsi que leurs interfaces.

  Corriger au besoin tout le code qui touche à Variable et LocalAttribute, en particulier Method.

  Corriger au besoin les applications qui sont affectées par les changements précédents.

Après l'une ou l'autre des stratégies présentées précédemment, les étapes d'optimisation et de validation sont fortement recommandées. L'optimisation vise à maximiser ou à minimiser une ou plusieurs caractéristiques de l'exécution (e.g., temps, espace, précision) en apportant les modifications nécessaires à l'implantation. Par exemple, le schéma SPOOL-UML actuel utilise un mécanisme (i.e., AccumulatedDependency) qui calcule à l'avance le nombre de dépendances afin

d'accélérer le temps de réponse des requêtes. Pour sa part, l'étape de validation cherche à confirmer l'utilité et l'exactitude ainsi qu'à révéler les failles de l'implantation en l'appliquant à des cas de tests ou réels. Par exemple, dans le cas de SPOOL, plusieurs systèmes logiciels de complexité et de tailles variables ont été analysés[20].

En conclusion, nous croyons qu'il n'est pas possible d'intégrer et d'utiliser les métaéléments du profil RCR dans l'atelier SPOOL actuel sans apporter des transformations relativement majeures au schéma en place.

[20] C'est d'ailleurs lors de ces validations que le besoin du profil RCR est apparu.

Chapitre 7: Conclusion

7.1 Synthèse

L'intégration des activités de maintenance (e.g., rétroconception, compréhension, réingénierie) et des activités d'analyse, de conception et de développement de systèmes logiciels doit inévitablement passer par UML en raison de son omniprésence. Un obstacle majeur à cette intégration est l'incapacité actuelle d'UML de modéliser l'implantation détaillée des systèmes logiciels existants, et en particulier le corps des méthodes. Toutefois, la spécification UML prévoit des mécanismes d'extension du métamodèle qui étendent ses capacités de modélisation. Nous avons exploité cette possibilité en créant une extension du métamodèle UML qui permet la modélisation détaillée du corps des méthodes. Ce mémoire a présenté le résultat de nos efforts: le profil RCR.

La décision de créer le profil RCR a été prise à la suite d'une étude de l'état de l'art qui a révélé qu'aucune solution acceptable existait déjà. Cette étude a sondé la communauté UML, la communauté de développement de logiciels ainsi que la communauté de maintenance de logiciels. Il s'avère qu'il existe à ce jour peu de solutions qui étendent le métamodèle UML pour la modélisation détaillée de systèmes logiciels. Les solutions potentielles ont été jugées inadéquates pour diverses raisons. Les raisons les plus fréquentes étaient l'insuffisance de détail (e.g., l'atelier de rétroconception de Pinali), la granularité trop fine (e.g., UML Action Semantics) et la lourdeur excessive (e.g., stratégie ActivityGraph).

Afin de réduire la taille de l'espace de solutions potentielles, des précisions sur la nature du problème à résoudre ont été apportées. Ces précisions étaient groupées

selon quatre thèmes: les choix préliminaires, les attentes techniques, les contraintes et les besoins.

- Les choix préliminaires ont restreint le problème à résoudre comme suit: les systèmes logiciels modélisés doivent être programmés selon un style impératif ou orienté objet, le code source doit être pré-traité (i.e., preprocessed), les raccourcis syntaxiques ne sont pas retenus ainsi que les informations qui ne sont pas exclusives aux corps de méthodes (e.g., commentaires, espacement, numéros de ligne).
- Les précisions au niveau des attentes techniques ont décrit deux mécanismes qui ne font pas partie de la solution envisagée comme tels mais dont l'utilisation dans la solution est permise. Ces mécanismes sont la modélisation des types dérivés (e.g, T*, T&, T[]) et l'attribution d'un identificateur unique à chaque élément d'un modèle.
- Les contraintes à respecter ont été réparties en deux sous-groupes. Le premier sous-groupe exige que la solution respecte dans la mesure du possible le style et la philosophie de la spécification UML, et qu'elle soit relativement stable en sachant que la spécification UML évolue toujours et que certains changements au niveau de la famille Action sont envisagés dans un avenir rapproché. Le deuxième sous-groupe de contraintes exige que la solution atteigne un certain équilibre entre les propriétés antagonistes de généralité, d'expressivité et d'efficacité.
- Les besoins devant être satisfaits par la solution sont les suivants. Premièrement, le niveau d'abstraction de la modélisation doit se rapprocher du niveau d'abstraction de l'arbre syntaxique abstrait. Deuxièmement, la solution doit absolument modéliser les énoncés et les expressions qui définissent le corps d'une méthode.

La solution que nous avons proposée est le profil RCR. Le profil RCR étend le métamodèle UML selon les règles de l'art à l'aide des mécanismes d'extension standard (i.e., les métaéléments Stereotype, TaggedValue et Constraint). Ces métaéléments sont combinés pour créer un nouveau métamodèle virtuel qui est intégré au métamodèle UML traditionnel. Le métamodèle du profil RCR est composé de 52 nouveaux métaéléments qui sont basés sur seulement quatre métaéléments UML standard provenant du package UML.Foundation.Core, à savoir Attribute, BehavioralFeature, Classifier et Dependency. Les neuf métaéléments dérivés de Dependency servent à établir les liens sémantiques entre les métaéléments du profil RCR et du métamodèle UML. Les autres métaéléments sont utilisés pour modéliser les entités suivantes: blocs, énoncés, expressions, variables et pas d'évaluation.

- Les blocs sont modélisés avec RCR.Block et ses descendants RCR.TryBlock et RCR.SynchronizedBlock.

- Les énoncés sont modélisés avec RCR.Statement et ses descendants:
  - énoncés simples: RCR.DeclarationStatement, RCR.ExpressionStatement
  - énoncés de branchement: RCR.BranchStatement, RCR.IfStatement, RCR.SwitchStatement
  - énoncés de saut: RCR.JumpStatement, RCR.BreakStatement, RCR.ContinueStatement, RCR.GotoStatement, RCR.ReturnStatement, RCR.ThrowStatement
  - énoncés d'itération: RCR.IterationStatement, RCR.WhileStatement, RCR.DoStatement, RCR.ForStatement
  - autre énoncés: RCR.InitStatement, RCR.ExitStatement, RCR.CaseStatement, RCR.ElseStatement, RCR.CompoundStatement, RCR.CatchStatement, RCR.FinallyStatement

- Les expressions sont modélisées avec RCR.ImperativeExpression et ses descendants RCR.SimpleExpression, RCR.LiteralExpression, RCR.IDRefExpression, RCR.SelfExpression et RCR.ComplexExpression.

- Les variables sont modélisées avec RCR.Variable.

- Les pas d'évaluation représentent l'application d'opérateurs prédéfinis et sont modélisés avec RCR.Step, RCR.CreateStep, RCR.DestroyStep, RCR.AssignStep, RCR.AccessStep, RCR.CallStep, RCR.ConvertStep, RCR.CalculateStep, RCR.CompareStep et RCR.ReflectionStep.

Ensuite, cinq discussions sur le profil RCR ont été présentées. La première a démontré comment le profil RCR a satisfait les besoins et a respecté les contraintes du problème.

- Les deux besoins ont été satisfaits puisque le profil RCR modélise le corps d'une méthode avec des métaéléments qui représentent des énoncés (i.e., RCR.Statement, etc.) et des expressions (i.e., RCR.ImperativeExpression), ainsi que des blocs (i.e., RCR.Block, etc.), des variables (i.e., RCR.Variable) et des opérateurs (i.e., RCR.Step, etc.). Toutes ces entités appartiennent au niveau d'abstraction de l'arbre syntaxique abstrait.
- La contrainte de familiarité vis-à-vis la spécification UML a été respectée par le choix des noms des nouveaux métaéléments. Les noms sont en anglais, ils sont construits de la même façon que les noms des autres métaéléments du métamodèle UML et ils réutilisent certaines parties de noms de métaéléments

UML lorsqu'il y a un rapprochement sémantique entre un métaélément du profil RCR et un autre du métamodèle UML: par exemple, RCR.CreateStep et CreateAction.

- La contrainte de stabilité a été largement respectée puisque le profil RCR ne dépend que de quatre métaéléments du métamodèle UML: Attribute, BehavioralFeature, Classifier et Dependency. De plus, ces quatre métaéléments proviennent du noyau du métamodèle UML (i.e., UML.Foundation.Core), ce qui minimise davantage les risques de modifications majeures à ces métaéléments.
- La contrainte d'équilibre a aussi été respectée puisque le profil RCR offre à la fois généralité, expressivité et efficacité.
  - La généralité provient de l'approche modulaire qu'offre le profil RCR pour la modélisation. Cette modularité est possible puisque le profil RCR offre une riche sélection d'entités primitives et n'impose aucune contrainte quant à la façon de les combiner, permettant ainsi une modélisation qui tolère les particularités de plusieurs langages.
  - L'expressivité intéressante du profil RCR provient de l'usage des TaggedValue associés à ses métaéléments, et en particulier à la hiérarchie RCR.Step. Les TaggedValue permettent de raffiner au besoin la nature exacte de l'entité modélisée à l'aide d'un mot-clé ou un texte arbitraire.
  - L'efficacité du profil RCR est obtenue par la modélisation explicite et directe des concepts pertinents (e.g., blocs, énoncés, expressions, pas d'évaluation, variables) et par la nature synthétique des pas d'évaluation.

La deuxième discussion a comparé le métamodèle UML étendu par le profil RCR (UML+RCR) à d'autres métamodèles rencontrés dans la communauté de maintenance. Un axe représentant le contenu en information d'un métamodèle a été utilisé pour illustrer graphiquement la position du métamodèle UML+RCR par rapport au métamodèle UML standard et aux métamodèles associés aux technologies Famix, Acacia, Datrix, Gen++ et Aria. Les métamodèles ont été regroupés sur l'axe dans trois zones associées à des représentations dites *simple*, *intermédiaire* ou *détaillée*. Le métamodèle UML+RCR a été placé à mi-chemin entre les zones intermédiaire et détaillée en raison de son contenu à la fois détaillé et indépendant d'un langage de programmation spécifique.

La troisième discussion a illustré comment le profil RCR est utilisé pour modéliser des fragments de code souurce. Les neuf exemples ont été choisis pour mettre en évidence les techniques de modélisation envisagées lors de la conception du profil

RCR. Les fragments de code comprennent une méthode simple, une déclaration de variable avec initialisation, un appel simple, un appel par accès via un objet, une instanciation de classe assignée à une variable, une conversion de type, les énoncés de branchement *if* et *switch* et l'énoncé d'itération *for*.

La quatrième discussion a expliqué pourquoi les modèles détaillés créés avec le métamodèle UML+RCR sont intéressants pour diverses applications. En particulier, l'activité de rétroconception dispose de beaucoup plus d'information pour identifier les composantes d'un système et leurs interrelations. L'activité de compréhension profite également des modèles détaillés car ils permettent d'effectuer les analyses, les mesures et les visualisations qui enrichissent le modèle mental. L'activité de réingénierie peut utiliser ces modèles pour détecter les anomalies, pour effectuer les corrections à même le modèle, pour élaborer des tests de régression et pour générer automatiquement du code plus complet.

La cinquième discussion a exploré la mise en oeuvre du profil RCR dans le cas général et dans le cas particulier du projet SPOOL. Le cas général requiert un métamodèle UML minimal comprenant les métaéléments ModelElement, Method, Attribute, BehavioralFeature, Classifier, Dependency, Stereotype, TaggedValue et Constraint ainsi qu'un mécanisme de gestion de profil. Le cas SPOOL pose un problème car son schéma n'est pas conforme au métamodèle UML standard et il n'a pas été conçu pour permettre son extension avec les profils. Deux stratégies alternatives sont proposées, mais elles impliquent invariablement des bouleversements majeurs du schéma existant.

7.2 Travaux futurs

7.2.1 Catalogues de modélisation

Le métamodèle UML+RCR est suffisamment général et expressif pour modéliser le code source C, C++ et Java. Cependant, à cause de sa flexibilité inhérente, il est

possible de modéliser certaines entités de code source de plusieurs façons équivalentes. De plus, certaines entités doivent être modélisées différemment selon le langage utilisé. Afin de minimiser les incompatibilités et les ambiguités, il serait utile d'établir un catalogue de modélisation pour chaque langage modélisé qui spécifie exactement la correspondance entre une entité du code source et son modèle correspondant.

7.2.2 Validations concrètes

En raison de contraintes techniques et temporelles, le profil RCR n'a pu être validé par des cas d'études concrets impliquant la modélisation de systèmes logiciels réels. Nous estimons que de telles validations seraient très intéressantes à réaliser parce qu'en plus de révéler les forces et les faiblesses du profil RCR, elles génèreraient des modèles détaillés de systèmes qui pourraient stimuler le développement d'outils (rétroconception, analyses, métriques, visualisations, réingénierie, etc.) qui exploitent les données additionnelles. De plus, la mise en oeuvre du profil RCR dans un atelier de conception UML permettrait d'explorer les mécanismes de gestion de profil ainsi que le comportement des mécanismes d'échange tels que XMI avec les modèles étendus.

7.2.3 Extensions

Puisque les activités de maintenance sont pertinentes pour tous les systèmes logiciels, il serait utile d'étendre le profil RCR afin qu'il puisse supporter d'autres langages (e.g., COBOL) et d'autres styles de programmation (e.g,, langages de scriptage, langages fonctionnels). Ces extensions exigeraient probablement l'introduction de nouveaux métaéléments.

Une extension supplémentaire qui enrichit l'ensemble du modèle d'un logiciel (i.e., elle ne s'applique pas uniquement à la représentation des corps de méthodes) est la modélisation des commentaires et l'inclusion des attributs comme les numéros de ligne, les versions, et ainsi de suite. Dans le même ordre d'idées, la modélisation des

fichiers non pré-traités (dans le cas de C et C++) serait aussi très utile puisque ces fichiers représentent en effet le système aux yeux d'un observateur humain. Le modèle pourrait alors permettre, par exemple, de visualiser le logiciel avant et après la substitution textuelle des *macros*[21] qui parfois sont très complexes et cachent beaucoup d'information utile.

7.3 Réflexion finale

L'expérience de l'auteur au sein du projet SPOOL a démontré que la création de modèles de systèmes logiciels existants est à la fois la tâche la plus importante et la plus difficile à réaliser correctement. Les ateliers de conception et de développement UML offrent déjà d'impressionnantes fonctionnalités qui facilitent certaines activités de rétroconception, de compréhension et de réingénierie. Nous espérons que le profil RCR stimulera un rapprochement supplémentaire qui mènera vers l'unification éventuelle des activités de maintenance aux autres activités du cycle de vie des systèmes logiciels.

[21] Le terme *macro* désigne, dans la programmation C et C++, une chaîne de caractères qui sera substituée par une deuxième chaîne de caractères lors du pré-traitement.

Bibliographie

[Abdurazik_2000]
Abdurazik A. *Suitability of the UML as an Architecture Description Language with Applications to Testing.* Technical report ISE-TR-00-01, Information and Software Engineering, George Mason University, Fairfax, VA, USA, 2000.

[ADTF_1999]
Analysis and Design Platform Task Force. *White Paper on the Profile Mechanism (version 1.0).* OMG Document ad/99-04-07, Object Management Group, Framingham, USA, April 1999.
Source: <ftp://ftp.omg.org/pub/docs/ad/99-04-07.pdf>, 2001-03-10.

[Aigner_2000]
Aigner G, Diwan A, Heine DL, Lam MS, Moore DL, Murphy BR, Sapuntzakis C. *The SUIF Program Representation.* Computer Systems Laboratory, Stanford University, USA, August 2000.
Source: <http://suif.stanford.edu/suif/suif2/doc-2.2.0-4/suifguide.ps>, 2001-03-20.

[Ajax]
Ajax. Carnegie Mellon University. USA.
Source: <http://www.cs.cmu.edu/~roc/Ajax.html>, 2001-03-17.

[Allen_1997]
Allen RJ. *A Formal Approach to Software Architecture.* PhD Thesis, Report CMU-CS-97-144. School of Computer Science, Carnegie Mellon University, Pittsburgh, USA, May 1997.

[Al-Zoubi_1991]
Al-Zoubi R, Prakash A. *Software Change Analysis via Attributed Dependency Graphs.* Technical Report CSE-TR-95-91, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, USA, May 1991.

[Ambras_1988]
Ambras J, O'Day V. MicroScope: A Knowledge-Based Programming Environment. *IEEE Software*, 5(3):50–58, May 1988.

[Antoniol_1998]
Antoniol G, Fiutem R, Cristoforetti L. Using Metrics to Identify Design Patterns in Object-Oriented Software. In *Proceedings of the Fifth. International Symposium on Software Metrics (METRICS98)*, pp 23–34. Bethesda, USA, November 1998.

[Antoniol_1999]
Antoniol G, Calzolari F, Tonella P. Impact of Function Pointers on the Call Graph. In *Proceedings of the Third EuropeanConference on Software Maintenance and Reengineering*, pp 51–59. Amsterdam, Netherlands, March 1999.

[Aonix]
Aonix. USA.
Source: <http://www.aonix.com/content/products/stp/uml.html>, 2001-03-19.

[ArgoUML]
ArgoUML. USA.
Source: <http://argouml.tigris.org/>, 2001-03-19.

[Armstrong_1998]
Armstrong MN, Trudeau C. Evaluating Architectural Extractors. In *Proceedings of the Fifth Working Conference on Reverse Engineering (WCRE'98)*, pp 30–39. Honolulu, USA, October 1998.

[Artisan]
Artisan Software Tools, Inc. USA.
Source: <http://www.artisansw.com/products/professional_overview.asp>, 2001-03-19.

[ASG]
Allen Systems Group, Inc. USA.
Source: <http://www.asg.com/>, 2001-03-18.

[Badros_2000]
Badros GJ. JavaML: A Markup Language for Java Source Code. In *Proceedings of the Ninth International Conference on the World Wide Web*, Amsterdam, Netherlands, May 2000. Elsevier Science B. V.

[Baker_1988]
Baker DA, Fisher DA, Shultis JC. *The Gardens of IRIS*. Arcadia Document IncSys-88-03, Incremental Systems Corporation, August 1988. Draft.

[Baker_1995]
Baker BS. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proceedings of the Second Working Conference on Reverse Engineering*, pp 86–95. Toronto, Canada, July 1995.

[Balazinska_1999]
Balazinska M, Merlo E, Dagenais M, Lagüe B, Kontogiannis K. Partial Redesign of Java Software Systems Based on Clone Analysis. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pp 326–336. Atlanta, USA, October 1999.

[Balazinska_1999a]
Balazinska M, Merlo E, Dagenais M, Lagüe B, Kontogiannis K. Measuring clone based reengineering opportunities. In *Proceedings of the Sixth IEEE International Symposium on Software Metrics*, pp 292–303. Boca Raton, USA, November 1999.

[Belaunde_1999]
Belaunde M. A Pragmatic Approach for Building a User-friendly and Flexible UML Model Repository. In France R, Rumpe B (eds), *UML'99 – The Unified Modeling Language; Beyond the Standard*. Lecture Notes in Computer Science, volume 1723, Springer, 1999.

[Bellay_1997]
Bellay B, Gall H. A Comparison of Four Reverse Engineering Tools. In *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE'97)*, pp 2–11. Amsterdam, Netherlands, October 1997.

[Biggerstaff_1989]
Biggerstaff TJ. Design Recovery for Maintenance and Reuse. *IEEE Computer*, 22(7):36–49, July 1989.

[Biggerstaff_1994]
Biggerstaff TJ, Mitbander BG, Webster DE. Program Understanding and the Concept Assignment Problem. *Communications of the ACM*, 37(5):72–82, May 1994.

[Blaha_1998]
Blaha M, LaPlant D, Marvak E. Requirements for Repository Software. In Proceedings of the Working Conference on Reverse Engineering (WCRE'98), pp 164-173. Honolulu, USA, October 1998.

[BoldSoft]
BoldSoft. Sweden.
Source: <http://www.boldsoft.com/products/boldfordelphi/>, 2001-03-19.

[Booch_1994]
Booch G. *Object-Oriented Analysis and Design with Applications* (2nd edition). Redwood City, USA: Benjamin/Cummings, 1994.

[Booch_1999]
Booch G, Rumbaugh J, Jacobson I. *The Unified Modeling Language User Guide*. Reading, USA: Addison-Wesley, 1999.

[Bowman_2000]
Bowman IT, Holt RC, Godfrey MW. Connecting Architecture Reconstruction Frameworks. *Information and Software Technology*. 42(2):91–102, January 2000. Elsevier Science.

[CDIF_1994]
CDIF Technical Committee. *CDIF Framework for Modeling and Extensibility*. Technical Report EIA/IS-107, Electronic Industries Association, Arlington, USA, January 1994.
Source (abandonnée): <http://www.eigroup.org/cdif/how-to-obtain-standards.html>, 2001-03-23.

[Chase_1996]
Chase MP, Harris DR, Roberts SN, Yeh AS. Analysis and Presentation of Recovered Software Architectures. In *Procedings of the Third Working Conference on Reverse Engineering (WCRE '96)*, pp 153–162. Monterey, USA, November 1996.

[Chaumun_1998]
Chaumun MA. *Change Impact Analysis in Object-Oriented Systems: Conceptual Model and Application to C++*. Master's thesis, Département d'informatique et de recherche opérationnelle, Université de Montréal, Canada, November 1998.

[Chaumun_1999]
Chaumun MA, Kabaili H, Keller RK, Lustman F. A Change Impact Model for Changeability Assessment in Object-Oriented Software Systems. In *Proceedings of the Third Euromicro Working Conference on Software Maintenance and Reengineering*, pp 130–138, Amsterdam, Netherlands, March 1999.

[Chaumun_2000]
Chaumun MA, Kabaili H, Keller RK, Lustman F, St-Denis G. Design Properties and Object-Oriented Software Changeability. In *Proceedings of the Fourth Euromicro Working Conference on Software Maintenance and Reengineering*, pp 45–54. Zurich, Switzerland, February 2000.

[Chen_1990]
Chen Y-F, Nishimoto M, Ramamoorthy C. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.

[Chen_1998]
Chen Y-F, Gansner ER, Koutsofios E. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. *IEEE Transactions on Software Engineering*, 24(9):682–694, September 1998.

[Chikofsky_1990]
Chikofsky EJ, Cross II JH. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*; 7(1):13–17, January 1990.

[Cifuentes_1998]
Cifuentes C. Binary Reengineering of Distributed Object Technology. In *Proceedings of the Fifth Working Conference on Reverse Engineering (WCRE'98)*, p 253. Honolulu, USA, October 1998.

[CodeCrawler]
CodeCrawler. Institut für Informatik und angewandte Mathematik (IAM), Universität Bern.
Source: <http://www.iam.unibe.ch/~lanza/codecrawler/codecrawler.html>, 2001-03-18.

[DataIntegrity]
Data Integrity, Inc. USA.
Source: <http://www.dii2000.com/Products_Page.html>, 2001-03-18.

[Demeyer_1999]
Demeyer S, Tichelaar S, Steyaert P. *FAMIX 2.0 — The FAMOOS Information Exchange Model*. Technical Report, Software Composition Group, University of Berne, Switzerland, August 1999.
Source: <http://www.iam.unibe.ch/~famoos/FAMIX/Famix20/famix20.pdf>, 2001-03-18.

[Demeyer_1999a]
Demeyer S, Ducasse S, Tichelaar S. Why Unified is not Universal: UML Shortcomings for Coping with Round-trip Engineering. In France R, Rumpe B (editors), *UML'99 – The Unified Modeling Language; Beyond the Standard*. Lecture Notes in Computer Science, volume 1723, Springer, 1999.

[Devanbu_1991]
Devanbu PT, Brachman RJ, Selfridge PG, Ballard BW. LaSSIE: a Knowledge-Based Software Information System. *Communications of the ACM*, 34(5):35–49, May 1991.

[Devanbu_1992]
Devanbu PT. GENOA - a Language and Front-End Independent Source Code Analyzer Generator. In *Proceedings of the 14th International Conference on Software Engineering*, pp 307–319. 1992.

[Devanbu_1994]
Devanbu PT, Eaves L. *Gen++ — An Analyzer Generator for C++ Programs*. Technical report, AT&T Bell Laboratories, Murray Hill, New Jersey, 1994.

[Devanbu_1996]
Devanbu PT, Rosenblum D, Wolf A. Generating Testing and Analysis Tools with Aria. *ACM Transactions on Software Engineering and Methodology*, 5(1):42–62, January 1996.

[Devanbu_2000]
Devanbu PT. Retargetability in Software Tools. ACM Applied Computing Review, 2000. (To appear).
Source: <http://castle.cs.ucdavis.edu/~devanbu/files/acr2k.pdf>, 2001-03-20.

[Deveaux_2000]
Deveaux D, Saint-Denis G, Keller RK. XML Support to Design for Testability. In *Proceedings of the Workshop on XML and Object Technology (XOT'2000)*. Sophia Antipolis and Cannes, France, June 2000. Held in conjunction with the 14th European Conference on Object-Oriented Programming (ECOOP'2000).
Source: <ftp://www.disi.unige.it/person/CataniaB/xot2000/paper4.pdf>, 2001-05-02.

[Ebert_1997]
Ebert J, Kullbach B, Panse A. The Extract-Transform-Rewrite Cycle – A Step towards MetaCARE. In Nesi P, Lehner F (editors), *Proceedings of the 2nd Euromicro Conference on Software Maintenance & Reengineering*, pp 165–170. Los Alamitos, USA, 1998.

[Ebert_1999]
Ebert J, Kullbach B, Winter A. GraX — An Interchange Format for Reengineering Tools. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pp 89–98. Atlanta, USA, October 1999.

[Elixir]
Elixir Technology. Singapore.
Source: <http://www.elixirtech.com/ElixirCASE/>, 2001-03-19.

[Embarcadero]
Embarcadero Technologies. USA.
Source: <http://www.advancedsw.com/products/gdpro.html>, 2001-03-19.

[FAMOOS]
FAMOOS. Software Composition Group, University of Berne. Switzerland.
Source: <http://www.iam.unibe.ch/~famoos/>, 2001-04-26.

[Fenton_1998]
Fenton NE, Pfleeger SL. *Software Metrics: A Rigorous and Practical Approach* (2nd Edition), PWS, 1998.

[Fenton_2000]
Fenton NE, Neil M. Software Metrics: Roadmap. In *Proceedings of the 22nd International Conference on The Future of Software Engineering 2000*, pp 357–370. Limerick, Ireland, June 2000.

[Finnigan_1997]
Finnigan P, Holt RC, Kalas I, Kerr S, Kontogiannis K, Müller HA, Mylopoulos J, Perelgut S, Stanley M, Wong K. The Software Bookshelf. *IBM Systems Journal*, 36(4):564–593, November 1997.

[Fowler_1999]
Fowler M. *Refactoring: Improving the Design of Existing Code*. Reading, USA: Addison-Wesley, 1999.

[Gannod_1999]
Gannod GC, Cheng BHC. A Framework for Classifying and Comparing Software Reverse Engineering and Design Recovery Techniques. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pp 77–88. Atlanta, USA, October 1999.

[GCC_2001]
GCC Team. C/C++ Internal Representation. GNU.
Source: <http://gcc.gnu.org/onlinedocs/c-tree.html>, 2001-03-20.

[GrammaTech]
GrammaTech, Inc. USA.
Source: <http://www.grammatech.com/products/codesurfer/codesurfer.html>, 2001-03-17.

[Griswold_1996]
Griswold WG, Atkinson DC, McCurdy C. Fast, Flexible Syntactic Pattern Matching and Processing. In *Proceedings of the IEEE Fourth Workshop on Program Comprehension(WPC-96)*, Berlin, Germany, March 1996.

[Harandi_1990]
Harandi MT, Ning JQ. Knowledge-Based Program Analysis. *IEEE Software*, 7(1):74–81, January 1990.

[Hofmeister_1999]
Hofmeister C, Nord RL, Soni D. Describing software architecture with UML. In Donohoe P, editor. *Software Architecture (aka Proceedings of the First Working IFIP Conference on Software Architecture)* pp 145–160. San Antonio, USA, February 1999. Boston, USA: Kluwer Academic Publishers.

[Holt_2000]
Holt RC, Winter A, Schürr A. GXL: Towards a Standard Exchange Format. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE 2000)*. Brisbane, Australia, November 2000.

[Horwitz_1992]
Horwitz S, Reps T. The use of program dependence graphs in software engineering. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pp 392–411. Melbourne, Australia, May 1992.

[iLogix]
I-Logix. USA.
Source: <http://www.ilogix.com/products/rhapsody/>, 2001-03-19.

[Imagix]
Imagix Corp. USA.
Source: <http://www.imagix.com/products/imagix4d.html>, 2001-03-17.

[IntegriSoft]
IntegriSoft, Inc. USA.
Source: <http://www.integrisoft.com/hindsight.htm>, 2001-03-18.

[ISO_1998]
ISO/IEC JTC 1, Subcommittee SC 22. *International Standard: Programming Languages—C++*, ISO/IEC 14882:1998(E). International Organization for Standardization, Geneva, Switzerland, September 1998.

[Jackson_2001]
Jackson D, Waingold A. Lightweight Extraction of Object Models from Bytecode. In *Proceedings of the 1999 International Conference on Software Engineering*, pp 194–202. Los Angeles, USA, May 1999.

[Jacobson_1992]
Jacobson I, Christerson M, Jonsson P, Övergaard G. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Wokingham, England: Addison-Wesley, 1992.

[Jahnke_1998]
Jahnke JH, Zuendorf A. Specification and Implementation of a Distributed Planning and Information System for Courses Based on Story Driven Modeling. In *Proceedings of the 9th International Workshop on Software Specification & Design*, pp 77–86. Ise-Shima (Isobe), Japan, April 1998.

[Kabaili_1999]
Kabaili H, Keller RK, Lustman F, Saint-Denis G. Design Properties for Reliability Assessment in Object-Oriented Systems. 6 page poster presentation & 6 page technical paper presented at the Tenth International Symposium on Software Reliability Engineering (ISSRE'99). Boca Raton, FL, November 1999.

[Kabaili_2000]
Kabaili H, Keller RK, Lustman F, Saint-Denis G. Class Cohesion Revisited: An Empirical Study on Industrial Systems. In *Proceedings of the Fourth International Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, pp 29–38. Sophia Antipolis and Cannes, France, June 2000. Held in conjunction with the 14th European Conference on Object-Oriented Programming (ECOOP'2000).

[Kabaili_2000a]
Kabaili H, Keller RK, Lustman F, Saint-Denis G. *Cohesion Revisited: An Empirical Study on Industrial Systems*. Technical Report GELO-135, Université de Montréal, Canada, September 2000. Revised and extended version of workshop paper at ECOOP-2000; invited submission to the journal *l'Objet*.

[Kadia_1992]
Kadia R. Issues Encountered in Building a Flexible Software Development Environment: Lessons from the Arcadia Project. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, pp 169–180. Tyson's Corner, USA, December 1992.

[Kazman_1998]
Kazman R, Woods S, Carrière SJ. Requirements for Integrating Software Architecture and Reengineering Models: CORUM II. In *Proceedings of the Working Conference on Reverse Engineering(WCRE'98)*, pp 154–163. Honolulu, USA, October 1998.

[Kazman_1999]
Kazman R, Carrière SJ. Playing Detective: Reconstructing Software Architecture from Available Evidence. *Journal of Automated Software Engineering*, 6(2):107–138, April 1999.

[Keller_2001]
Keller RK, Bédard J-F, Saint-Denis G. Design and Implementation of a UML-based Design Repository. In *Proceedings of the Thirteenth Conference on Advanced Information Systems Engineering (CAiSE'01).* Interlaken, Switzerland, June 2001. To appear.

[Kienle_2000]
Kienle HM, Czeranski J, Eisenbarth T. Exchange Format Bibliography. *Workshop on Standard Exchange Format (WOSEF) at the 22nd International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, June 2000.

[Klösch_1996]
Klösch RR. Reverse Engineering: Why and How to Reverse Engineer Software. In *Proceedings of the California Software Symposium (CSS '96)*, pp 92–99. University of Southern California, USA, April 1996.

[Knuth_1984]
Knuth, D. E. Literate Programming. *The Computer Journal*, 27(2):97–111, May 1984.

[Kobryn_1999]
Kobryn C. UML 2001: A Standardization Odyssey. *Communication of the ACM*, 42(10):29–37, October 1999.

[Köhler_2000]
Köhler HJ, Nickel U, Niere J, Zündorf A. Integrating UML diagrams for production control systems. In *Proceedings of the 22nd International Conference on Software Engineering*, pp 241–251. Limerick, Ireland, June 2000.

[Kontogiannis_1998]
Kontogiannis K. Distributed Objects and Software Application Wrappers: A Vehicle for Software Re-engineering. In *Proceedings of the Fifth Working Conference on Reverse Engineering (WCRE'98)*, p 254. Honolulu, USA, October 1998.

[Koschke_1998]
Koschke R, Girard J-F, Würthner M. An Intermediate Representation for Reverse Engineering Analyses. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, pp 241–251. Honolulu, USA, October 1998.

[Koskimies_1996]
Koskimies K, Mössenböck H. Scene: Using Scenario Diagrams and Active Text for Illustrating Object-Oriented Programs. In *Proceedings of the 18th International Conference on Software Engineering*, pp 366–375. Berlin, Germany, March 1996.

[Koskimies_1998]
Koskimies K, Systä T, Tuomi J, Männistö T. Automated Support for Modeling OO Software. *IEEE Software*, 15(1):87–94, January-February 1998.

[Lange_1995]
Lange DB, Nakamura Y. Interactive Visualization of Design Patterns Can Help in Framework Understanding. In *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95)*, pp 342–357. Austin, USA, October 1995.

[Lejter_1992]
Lejter M, Meyers S, Reiss SP. Support for Maintaining Object-Oriented Programs. *IEEE Transactions on Software Engineering*, 18(12):1045–1052, December 1992.

[Lethbridge_2001]
Lethbridge TC, Plödereder E, Tichelaar S, Riva C, Linos P. *The Dagstuhl Middle Model (DMM) (Version 0.001)*. January 2001.
Source:
<http://titan.cnds.unibe.ch:8080/Exchange/uploads/DMM/dmmdescriptionv0001.pdf>, 2001-03-19.

[Linton_1984]
Linton MA. Implementing Relational Views of Programs. *SIGPLAN Notices*, 19(5):132–140, May 1984.

[Marchionini_1995]
Marchionini G. *Information Seeking in Electronic Environments*. Cambridge, England: Cambridge University Press, 1995.

[Mayrand_1996]
Mayrand J, Coallier F. System Acquisition Based on Software Product Assessment. In *Proceedings of the 18th International Conference on Software Engineering*, pp 210–219. Berlin, Germany, March 1996.

[Mayrand_1996]
Mayrand J, Leblanc C, Merlo E. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In *Proceedings of the International Conference on Software Maintenance*, pp 244–253. Monterey, USA, November 1996.

[Mays_1996]
Mays RG. *Power Programming with the Lemma Code Viewer*. IBM Technical Report, TRP Networking Laboratory, Research Triangle Park, NC, February 1996.

[McCabe]
McCabe & Associates, Inc. USA.
Source: <http://www.mccabe.com/products/reengineer.htm>, 2001-03-18.

[Mendonça_1996]
Mendonça NC, Kramer J. Requirements for an Effective Architecture Recovery Framework. In *Proceedings of the Second ACM SIGSOFT International Software Architecture Workshop (ISAW-2)*, pp 101–105. San Francisco, USA, October 1996.

[Mendonça_1997]
Mendonça NC, Kramer J. A Quality-Based Analysis of Architecture Recovery Environments. In *Proceedings of the First Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'97)*, pp 54–59. Berlin, Germany, March 1997.

[Mendonça_1999]
Mendonça NC. *Software Architecture Recovery for Distributed Systems*, PhD dissertation, Department of Computing, Imperial College of Science, Technology and Medicine, University of London, England, UK, November 1999.

[MetaCase]
MetaCase Consulting. Finland.
Source: <http://www.metacase.com/>, 2001-03-19.

[MicroTool]
microTOOL GmbH. Germany.
Source: <http://www.microtool.de/objectiF/en/>, 2001-03-19.

[Müller_1993]
Müller HA, Orgun MA, Tilley SR, Uhl JS. A Reverse Engineering Approach To Subsystem Structure Identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, December 1993.

[Müller_1994]
Müller HA, Wong K, Tilley SR. Understanding Software Systems Using Reverse Engineering Technology. In *The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS)*, Montreal, Canada, May 1994.

[Müller_1998]
Müller HA, Wong K, Storey M-A. Reverse Engineering Research Should Target Cooperative Information System Requirements. In *Proceedings of the Fifth Working Conference on Reverse Engineering (WCRE'98)*, p 255. Honolulu, USA, October 1998.

[Müller_2000]
Müller HA, Jahnke J, Smith D, Storey M-A, Tilley S, Wong K. Reverse Engineering: A Roadmap. In Finkelstein A (editor), *Proceedings of the Conference on The Future of Software Engineering*, from the 22nd International Conference on Software Engineering (ICSE2000), pp 47–60. Limerick, Ireland, June 2000.

[Murphy_1995]
Murphy GC, Notkin D, Sullivan K. Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp 18–28. New York, USA, October 1995.

[Murphy_1996]
Murphy GC, Notkin D. Lightweight Lexical Source Model Extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3):262–292, July 1996.

[Murphy_1998]
Murphy GC, Notkin D, Griswold WG, Lan ES. An Empirical Study of Static Call Graph Extractors. *ACM Transactions on Software Engineering and Methodology*, 7:158–191, April 1998.

[Murray_1992]
Murray RB. A Statically Typed Abstract Representation for C++ Programs. In *Proceedings of the USENIX C++ Conference*, pp 83–97. Portland, USA, August 1992.

[Mylopoulos_1990]
Mylopoulos J, Borgida A, Jarke M, Koubarakis M. Telos: Representing Knowledge about Information Systems. *ACM Transactions on Information Systems*, 8(4):325–362, October 1990.

[Ning_1994]
Ning JQ, Engberts A, Kozaczynski WV. Automated support for legacy code understanding. *Communications of the ACM,* 37(5):50–57, May 1994.

[NoMagic]
NoMagic, Inc. USA.
Source: <http://www.magicdraw.com/>, 2001-03-19.

[ObjectInsight]
Object Insight, Inc. USA.
Source: <http://www.object-insight.com/html/product_info.html>, 2001-03-19.

[OCallahan_1997]
O'Callahan R, Jackson D. Lackwit: A Program Understanding Tool Based on Type Inference. In *Proceedings of the 1997 International Conference on Software Engineering*, pp 338–348. Boston, USA, May 1997.

[OMG_1996]
OMG. *Object Analysis and Design — RFP-1*. OMG TC Document ad/96-05-01 – rev. 1, Object Management Group, Framingham, USA, June 1996.
Source: <ftp://ftp.omg.org/pub/docs/ad/96-05-01.pdf>, 2001-03-06.

[OMG_1997]
OMG. *A Discussion of the Object Management Architecture*. Object Management Group, January 1997.
Source: <ftp://ftp.omg.org/pub/docs/formal/00-06-41.pdf>, 2001-03-06.

[OMG_1998]
OMG. *Action Semantics for the UML—Request For Proposal*. OMG Document ad/98-11-01, Object Management Group, Framingham, USA, November 1998.
Source: <ftp://ftp.omg.org/pub/docs/ad/98-11-01.pdf>, 2001-03-16.

[OMG_2000]
OMG. *The OMG Unified Modeling Language Specification* (version 1.3). Object Management Group, March 2000.
Source: <ftp://ftp.omg.org/pub/docs/formal/00-03-01.pdf>, 2001-03-03.

[OMG_2000a]
OMG. *Meta Object Facility(MOF) Specification* (version 1.3). Object Management Group, March 2000.
Source: <ftp://ftp.omg.org/pub/docs/formal/00-04-03.pdf>, 2001-03-08.

[OMG_2000b]
OMG. *OMG XML Metadata Interchange (XMI) Specification* (version 1.1). Object Management Group, November 2000.
Source: <ftp://ftp.omg.org/pub/docs/formal/00-11-02.pdf>, 2001-03-10.

[Opdyke_1992]
Opdyke WF. *Refactoring Object-Oriented Frameworks*. PhD dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, USA, 1992.

[Paul_1994]
Paul S, Prakash A. A Framework for Source Code Search using Program Patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, June 1994.

[PBS]
Portable Bookshelf. University of Waterloo. Canada.
Source: <http://swag.uwaterloo.ca/pbs/>, 2001-04-26.

[Penteado_1996]
Penteado RD, Germano F, Masiero PC. An Overall Process Based on Fusion to Reverse Engineer Legacy Code. In *Procedings of the Third Working Conference on Reverse Engineering(WCRE '96)*, pp 179–188. Monterey, USA, November 1996.

[Penteado_1998]
Penteado RD, Prado AF, Masiero PC, Braga RTV. Reengineering of Legacy Systems Based on Transformation Using the Object Oriented Paradigm. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, pp 144–153. Honolulu, USA, October 1998.

[Pinali_1999]
Pinali Doederlein O. *Design Pattern Extraction for Software Documentation.* Master's thesis, Computer Science Department,Vrije Universiteit Brussel, Belgium & Département informatique, École des Mines de Nantes, France, 1999.

[Popkin]
Popkin Software. USA.
Source: <http://www.popkin.com/products/sa2001/systemarchitect.htm>, 2001-03-19.

[ProgResearch]
Programming Research Ltd. England.
Source: <http://www.programmingresearch.com/cims.html>, 2001-03-18.

[ProjTech]
Project Technology Inc. USA.
Source: <http://www.projtech.com/prods/bp/>, 2001-03-19.

[Quilici_1995]
Quilici A, Chin D. DECODE: A Cooperative Program Understanding Environment. Journal of Software Maintenance, 8(1):3–34, 1996.

[Rasmussen_2000]
Rasmussen RW.*A Framework for the UML Metamodel.* Master's thesis, Institute for informatics, University of Bergen, Norway, April 2000.

[Rational]
Rational Software Corporation. USA.
Source: <http://www.rational.com/products/rose/>, 2001-03-19.

[Reasoning]
Reasoning, Inc. USA.
Source: <http://www.reasoning.com/>, 2001-03-18.

[Reiss_1995]
Reiss SP. *The FIELD Programming Environment: A Friendly Integrated Environment for Learning and Development.* The Kluwer International Series in Engineering and Computer Science. Norwell, USA: Kluwer Academic Publishers, 1995.

[Rich_1990]
Rich C, Waters RC. *The Programmer's Apprentice.* Reading, USA: Addison-Wesley & New York, USA: ACM Press, 1990.

[Richardson_1992]
Richardson DJ, O'Malley TO, Moore CT, Aha SL. Developing and Integrating ProDAG in the Arcadia Environment. In *Proceedings of ACM SIGSOFT '92: Fifth Symposium on Software Development Environments*, pp 109–119. Washington, USA, December 1992.

[Richner_1999]
Richner T, Ducasse S. Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information. *IEEE International Conference on Software Maintenance (ICSM '99)*, pp 13–22. Oxford, England, August-September 1999.

[Robitaille_2000]
Robitaille S. *Support informatique à la compréhension des logiciels orientés objet de taille industrielle.* Mémoire de maîtrise, Département d'informatique et de recherche opérationnelle, Université de Montréal, Canada, Avril 2000.

[Rosenblum_1991]
Rosenblum DS, Wolf AL. Representing Semantically Analyzed C++ Code with Reprise. 1991.

[Rumbaugh_1991]
Rumbaugh J, Blaha M, Premerlani W, Eddy F, Lorensen W. *Object-Oriented Modeling and Design.* Englewood Cliffs, USA: Prentice Hall, 1991.

[Rumbaugh_1999]
Rumbaugh J, Jacobson I, Booch G. *The Unified Modeling Language Reference Manual.* Reading, USA: Addison-Wesley, 1999.

[Saint-Denis_2000]
Saint-Denis G, Schauer R, Keller RK. Selecting a Model Interchange Format. The SPOOL Case Study. In *Proceedings of the Thirty-Third Annual Hawaii International Conference on System Sciences*, Maui, HI, January 2000. Provided on CD-ROM, 10 pages.

[Schauer_1998]
Schauer R, Keller RK. Pattern Visualization for Software Comprehension. In *Proceedings of the Sixth International Workshop on Program Comprehension (IWPC'98)*, pp 4–12. Ischia, Italy, June 1998.

[Schauer_2001]
Schauer R, Keller RK, Laguë B, Knapen G, Robitaille S, St-Denis G. The SPOOL Design Repository: Architecture, Schema, and Mechanisms. In Erdogmus H, Tanir O (editors), *Advances in Software Engineering. Topics in Evolution, Comprehension, and Evaluation.* Springer-Verlag, 2001. To appear.

[Schürr_2001]
Schürr A, Sim SE, Holt RC, Winter A. *GXL (1.0) Document Type Definition.* February 2001.
Source: <http://www.gupro.de/GXL/gxl.dtd>, 2001-03-19.

[Schwanke_1991]
Schwanke RW. An Intelligent Tool for Re-engineering Software Modularity. In *Proceedings of the 13th International Conference on Software Engineering*, pp 83–92. Austin, USA, May 1991.

[Seemann_1998]
Seemann J, Wolff von Gudenberg J. Pattern-Based Design Recovery of Java Software. In *Proceedings of the ACM SIGSOFT Sixth International Symposium on Foundations of Software Engineering*, pp 10–16. Orlando, USA, November 1998.

[Sefika_1996]
Sefika M, Sane A, Campbell RH. Monitoring Compliance of a Software System With Its High-Level Design Models. In *Proceedings of the 18th International Conference on Software Engineering (ICSE-18)*, pp 387–396. Berlin, Germany, March 1996.

[Sema]
Sema Group. Espagne.
Source: <http://dis.sema.es/products/Concerto/AUDIT/>, 2001-03-20.

[Sim_1999]
Sim SE, Clarke CLA, Holt RC, Cox AM. Browsing and Searching Software Architectures. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'99)*, pp 381–390. Oxford, England. August 1999.

[Sim_2000]
Sim SE, Koschke R. WoSEF: Workshop on Standard Exchange Format. *ACM Software Engineering Notes*. To appear.
Source: <http://www.cs.toronto.edu/~simsuz/wosef/sen-wosef.pdf>, 2001-03-23.

[Softeam]
Softeam. France.
Source: <http://www.objecteering.com/fr/produits.htm>, 2001-03-19.

[Softera]
Softera Ltd. Israel.
Source: <http://www.softera.com/products.htm>, 2001-03-19.

[Sparx]
Sparx Systems. Australia.
Source: <http://www.sparxsystems.com.au/ea.htm>, 2001-03-19.

[Starbase]
Starbase Corp. USA.
Source: <http://genitor.com/genitor/ocs.htm>, 2001-03-18.

[Systä_1999]
Systä T. On the Relationships between Static and Dynamic Models in Reverse Engineering Java Software. 1999.

[Systä_2000]
Systä T, Yu P, Müller H. Analyzing Java Software by Combining Metrics and Program Visualization. In *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering (CSMR 2000)*, pp 199–208. Zurich, Switzerland, February-March 2000.

[Telelogic]
Telelogic AB. Sweden.
Source: <http://www.telelogic.com/logiscope/>, 2001-03-18.

[TelelogicUML]
Telelogic AB. Sweden.
Source: <http://www.telelogic.com/UML/>, 2001-03-19.

[Theodoros_1998]
Theodoros L, Edwards HM, Bryant A, Willis N. ROMEO: Reverse Engineering from OO Source Code to OMT Design. In *Proceedings of the Fifth Working Conference on Reverse Engineering (WCRE'98)*, pp 30–39. Honolulu, USA, October 1998.

[Tichelaar_2000]
Tichelaar S, Ducasse S, Demeyer S, Nierstrasz O. A Meta-model for Language-Independent Refactoring. In *Proceedings of the 2000 International Symposium on Principles of Software Evolution (ISPSE 2000)*. Kanazawa, Japan, November 2000.

[Together]
TogetherSoft Corporation. USA.
Source: <http://www.togethersoft.com/together/togetherCC.html>, 2001-03-19.

[UASC_2000]
UML Action Semantics Consortium. Response to OMG RFP ad/98-11-01— Action Semantics for the UML, (version 18). February 2001.
Source: <http://www.kabira.com/as/download/ActionSemantics.zip>, 2001-03-16.

[Upspring]
Upspring Software. USA.
Source: <http://www.upspringsoftware.com/products/discover/index.html>, 2001-03-17.

[vonMayrhauser_1994]
von Mayrhauser A, Vans AM. *Program Understanding – A Survey*. Technical Report CS-94-120, Colorado State University, USA, August 1994.

[W3C_2000]
W3C. *Extensible Markup Language (XML) 1.0* (Second Edition). World Wide Web Consortium, October 2000.
Source: <http://www.w3.org/TR/2000/REC-xml-20001006.pdf>, 2001-03-10.

[Waters_1999]
Waters R, Abowd GD. Architectural Synthesis: Integrating Multiple Architectural Perspectives. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pp 2–12. Atlanta, USA, October 1999.

[WebGain]
WebGain, Inc. USA.
Source: <http://www.webgain.com/products/structure_builder/>, 2001-03-19.

[Weinberg_1971]
Weinberg GM. *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold, 1971.

[Weiser_1981]
Weiser M. Program Slicing. In *Proceedings of the Fifth International Conference on Software Engineering*, pp 439–449. San Diego, USA, March 1981.

[Whitney_1995]
Whitney M, Kontogiannis K, Johnson JH, Bernstein M, Corrie B, Merlo E, McDaniel J, De Mori R, Müller HA, Mylopoulos J, Stanley M, Tilley S, Wong K. Using an Integrated Toolset for Program Understanding. In *Proceedings of the 1995 IBM CAS Conference (CASCON'95)*, pp 262–274. Toronto, Canada, November 1995.

[Wills_1992]
Wills LM. *Automated Program Recognition by Graph Parsing*. A.I. Technical Report No. 1358, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, USA, July 1992.

[WindRiver]
Wind River Systems, Inc. USA.
Source: <http://www.wrs.com/products/html/sniff.html>, 2001-03-18.

[Wong_1998]
Wong K. *Rigi User's Manual—Version 5.4.4*. Department of Computer Science, University of Victoria, Canada, June 1998.
Source: <ftp://ftp.rigi.csc.uvic.ca/pub/rigi/doc/rigi-5.4.4-manual.pdf>, 2001-03-23.

[Woods_1998]
Woods, S., O'Brien L, Lin T, Gallagher K, Quilici A. An Architecture for Interoperable Program Understanding Tools. In *Proceedings of the Sixth International Workshop on Program Comprehension (IWPC '98)*, pp 54–63. Ischia, Italy, June 1998.

[Yong_1999]

Yong SH, Horwitz S, Reps T. Pointer analysis for programs with structures and casting. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pp 91–103. Atlanta, USA, May 1999.

Annexe A

Modélisation de types dérivés avec le métamodèle UML

De façon générale, les langages de programmation comme C, C++ et Java définissent des types primitifs (e.g., char, float) qui peuvent être utilisés directement. Ces langages offrent également la possibilité au programmeur de construire ses propres types (e.g., union, struct, class) en définissant un regroupement de champs (et des méthodes si orienté objet). Il existe une troisième variété de type qui est fréquemment utilisée par les programmeurs: le type dérivé. Le type dérivé est construit en appliquant un constructeur de type à un type déjà connu (qui peut être primitif, usager ou dérivé). Ces constructeurs de types sont par exemple '*' (i.e., pointeur), '&' (i.e., référence) et '[]' (i.e., tableau). Les types dérivés obtenus sont par exemple T* (i.e., pointeur à T), T& (i.e., référence à T) et T[] (i.e., tableau de T).

Le métamodèle UML prévoit deux métaéléments pour la modélisation des types primitifs et usagers, soient DataType et Class, respectivement. Cependant, il ne possède pas de métaélément explicite pour la modélisation d'un type dérivé. Dans le contexte de la modélisation de systèmes logiciels pour la rétroconception, la compréhension et la réingénierie, il est vital de modéliser précisément les types dérivés pour ne pas compromettre la fidélité du modèle.

La stratégie suivante permet de modéliser les types dérivés avec le métamodèle UML. Cette approche repose sur le concept d'un type paramétré, aussi connu sous le nom de "template". Le type paramétré est un type partiellement défini qui varie en fonction de la valeur d'un ou plusieurs paramètres formels. Dans le cas du type dérivé, le paramètre formel représente le type sur lequel le type dérivé repose. Par exemple, soit Pointer<T>, le type paramétré qui représente le constructeur de type *. Le type X* est modélisé par l'instanciation de Pointer<T> avec la valeur X liée à T. Une notation

pratique pour cette instanciation est Pointer<X>. De la même manière, les constructeurs de types &, [] et *typedef* peuvent être représentés par les types paramétrés suivants: Reference<T>, Array<T> et Alias<T>, respectivement.

Le métamodèle UML permet la paramétrisation de n'importe quel métaélément dérivé de ModelElement grâce à l'association TemplateParameter. Il suffit donc de modéliser Pointer<T>, Reference<T>, Array<T> et Alias<T> comme des instances paramétrées de Classifier. Ces constructeurs de types peuvent être conservés dans un package utilitaire contenant la définition des types primitifs disponibles dans un langage de programmation particulier. Puis, lorsqu'un type dérivé est rencontré dans le système modélisé, il suffira d'instancier un des constructeurs de type avec une référence au type de base.