N° d'ordre: 3785

Thèse

présentée

devant l'Université de Rennes 1

pour obtenir

le grade de Docteur de l'Université de Rennes 1
Mention Informatique

par

Martin Monperrus

Équipe d'accueil : INRIA/Triskell - ENSIETA/Dtn
École Doctorale : Matisse
Composante universitaire : IFSIC

Titre de la thèse :

La mesure des modèles par les modèles : une approche générative

Soutenue le 6 octobre 2008 devant la commission d'examen.
Composition du jury :

Présidente			
Françoise	André	Professeur à l'Université de Rennes 1	
Rapporteurs			
Stéphane	Ducasse	Directeur de Recherche à l'INRIA	
Houari	Sahraoui	Professeur à l'Université de Montréal	
Examinateurs			
Dominique	Luzeaux	Directeur à la DGA	
Joël	Champeau	Enseignant-chercheur à l'ENSIETA	
Jean-Marc	Jézéquel	Professeur à l'Université de Rennes 1	(Directeur de thèse)
Invités			
Brigitte	Hoeltzener	Enseignant-chercheur à l'ENSIETA	
Gabriel	Marchalot	Ingénieur à Thales Airborne Systems	

Résumé

La mesure des modèles par les modèles: une approche générative

L'ingénierie dirigée par les modèles est une approche du génie logiciel qui utilise des modèles comme artefacts de première importance, à partir desquels la validation, le code, les tests et la documentation sont dérivés. Les modèles peuvent être généralistes (e.g.; UML), propres à un domaine (e.g.; les systèmes temps réels), ou même spécifiques au métier d'une compagnie.

La mesure est une activité d'ingénierie qui permet d'obtenir une information quantitative sur les processus d'ingénierie ou les systèmes en cours de développement. La mesure des modèles tôt dans le cycle de développement permet aux architectes et aux managers d'estimer les coûts, d'identifier les risques et les défauts, de valider des propriétés et de suivre une démarche d'assurance qualité dès le début du développement. Malheureusement, il est coûteux de développer un outil de mesure ad hoc pour chaque type de modèles manipulés.

Nous proposons une approche indépendante du métamodèle pour définir des métriques de modèles. Les métriques sont spécifiées à un haut niveau d'abstraction, de manière plus rigoureuse qu'avec le langage naturel, de manière plus concise qu'avec un langage de programmation et débarrassées des préoccupations d'implémentation. Ensuite, à partir de cette spécification déclarative des métriques, un outil peut générer le composant de mesure, directement intégré dans un environnement de modélisation. La contribution globale de cette approche est de donner une implémentation des métriques de modèles, intégrée, fondée sur des modèles, et à un coût moindre.

Abstract

Model driven model measurement: a generative approach

Model-Driven Engineering (MDE) is an approach to software development that uses models as primary artifacts, from which validation, code, test and documentation are derived. Several metamodels are used in the same process. They range from general purpose ones (e.g.; UML), to domain (e.g.; for real-time systems) and company metamodels.

Measurement is an engineering activity that enables to obtain quantitative information on the engineering process or the systems being developed. Measurement of models at an early phase of the development life cycle allows architects and managers to estimate costs, to identify risks and flaws, to validate some properties and to perform early quality assurance. Unfortunately, it is costly to develop an ad hoc measurement tool for each manipulated metamodel.

We propose a metamodel-independent framework to define model metrics. Metrics are specified at a high level of abstraction, thus more rigorously than with natural language, more concisely than with a programming language and free of implementation concerns. Then, from this declarative specification of metrics, a toolchain is able to generate the measurement software seamlessly integrated into a modeling environment. The overall contribution of this approach is to give a model-driven and integrated implementation of model metrics at a reasonable cost.

Remerciements

Je tiens tout d'abord à remercier mon directeur de thèse Jean-Marc Jézéquel, de qui j'aurais encore beaucoup à apprendre. Merci à Joël Champeau et Brigitte Hoeltzener, mes encadrants, de m'avoir apporté un soutien sans faille dans la conduite de mes travaux. Même si Benoit Baudry n'apparaît pas officiellement dans cette thèse, ses conseils et discussions furent bien plus que précieux.

Merci à Houari Sahraoui et Stéphane Ducasse d'avoir accepté d'évaluer ce mémoire, ainsi qu'à Françoise André d'avoir présidé le jury. Je suis aussi redevable à Gabriel Marchalot d'une collaboration industrielle fructueuse.

Merci à tous mes collègues de l'ENSIETA à Brest et de l'équipe Triskell à l'INRIA Rennes pour avoir fait ce bout de chemin ensemble. Merci aux étudiants dont la curiosité et la soif de comprendre sont à la fois une satisfaction pour l'enseignant et un modèle pour le chercheur.

Merci à la DGA d'avoir financé cette thèse et la crèche de Kerigonan à Brest d'avoir gardé mes filles pour que je la mène à bien.

Bérénice, tu as sans doute aucun, à la fois encouragé, permis et supporté ces travaux. J'espère un jour te le rendre.

"I can play with chessmen according to certain rules. But I can also invent a game in which I play with the rules themselves. The pieces in my game are now the rules of chess, and the rules of the game are, say, the laws of logic. In that case I have yet another game and not a metagame."
Ludwig Wittgenstein, *Philosophical Remarks*, [Wittgenstein, 1975, p. 319]

Table des matières

1 Introduction

Pour reprendre l'expression de Granger [Granger, 1993], nous vivons à l'âge de la science. Les systèmes techniques issus des découvertes scientifiques sont partout. Ils sont intimement mêlés à notre quotidien: nos moyens de transport, nos moyens de production d'énergie, nos moyens de production d'alimentation, nos moyens de communication. Le développement d'un train, d'une centrale nucléaire, d'un plant de blé génétiquement modifié ou d'un ordinateur est une tâche longue, de l'ordre de plusieurs années, et qui fait intervenir un grand nombre de personnes.

Le développement de tels systèmes nécessite des supports pour communiquer entre les nombreux acteurs du projet, et ce, de l'idée originelle à la pose du dernier boulon. Il est d'usage d'utiliser des documents écrits en langage naturel pour expliquer le produit que l'on veut obtenir (un cahier des charges) ou comment celui-ci doit être construit (une méthode de réalisation).

Ces systèmes techniques sont porteurs de risques. D'une part, un risque financier, car les sommes mises en jeu dans leur développement sont considérables. Il est donc préférable d'aller au bout des développements commencés et de produire un système fini, utilisable, et utilisé. D'autre part, un certain nombre de systèmes techniques comportent des risques pour la vie humaine. Il est ainsi souhaitable d'éviter la chute d'un pont ou d'un avion, ou l'explosion d'une centrale nucléaire.

Pour réduire ces risques au maximum, les ingénieurs qui conçoivent ces systèmes sont confrontés à la nécessité d'estimer, et si possible de prouver l'impossibilité d'un accident avant la construction du produit final. Pour ce faire, les ingénieurs utilisent des documents qui sont plutôt des modèles, comme par exemple, des formules mathématiques qui prédisent la résistance d'un pont.

En bref, le développement des systèmes techniques nécessite un grand nombre de documents de différents types, pour communiquer, mais aussi pour vérifier le plus tôt possible des propriétés des systèmes en cours de construction.

Notre recherche réside dans l'étude d'un type de documents particuliers utilisé dans le développement des systèmes techniques. Ce type de document est appelé, dans la littérature correspondante, un modèle, et nous gardons cette terminologie dans nos travaux. Toutefois, c'est un abus de langage, et il s'agit plutôt un type de modèle particulier. Les modèles que nous étudions dans cette thèse consistent à lister les éléments du système étudié et leurs relations[1].

Ces modèles sont apparus en mathématiques dans les années 1960, dans une théorie appelée théorie des graphes, puis ont été de plus en plus utilisés dans les développements des systèmes logiciels. Dans le milieu des années 1990, ces modèles ont peu à peu constitué le squelette d'une méthode de développement des logiciels nommée

[1] Afin d'éviter la confusion, nous choisissons d'utiliser dans cette introduction, et de manière interchangeable, *ce type de modèles*, *ces modèles*, et même *nos modèles* en référence non pas à cette thèse, mais à ce domaine de recherche.

ingénierie dirigée par les modèles (IDM) qui est très étudiée dans les laboratoires de recherche et de plus en plus utilisée dans l'industrie[2].

Pourquoi ces modèles sont-ils utilisés et pourquoi sont-ils en phase d'expansion ? Ces modèles possèdent plusieurs caractéristiques intéressantes. Premièrement, il est apparu que ces modèles pouvaient servir non seulement à décrire le logiciel en cours de développement, mais aussi à décrire le domaine d'application du logiciel. Par exemple, les modèles en question peuvent servir à décrire la structure d'une entreprise, ou à décrire les règles qui guident le processus à suivre pour recevoir une commande et la traiter jusqu'à la fin. Deuxièmement, il semble que ces modèles peuvent remplacer d'autres documents utilisés dans le développement des systèmes qui ne sont pas exclusivement logiciels. Pour reprendre les exemples précédents, il est envisageable d'utiliser ces modèles pour décrire une partie d'un pont ou d'un train, ou pour remplacer un cahier des charges écrit en langage naturel par des modèles. Troisièmement, ils ont l'avantage de pouvoir automatiser certaines parties du développement des systèmes. Dans le cas extrême, là où une personne produisait un document en langage naturel, et qu'une autre produisait le document suivant ou posait le boulon final, le modèle remplaçant permet une automatisation complète de la deuxième étape. Quatrièmement, ils permettent des analyses sur les propriétés des systèmes. Par exemple, là où le cahier des charges donnait lieu à une estimation vague du coût de développement par des experts, un modèle remplaçant permet une analyse précise dont résulte un coût de développement précis et sans erreur d'estimation. Enfin, il est notable que les mêmes modèles sont utilisés pour automatiser le développement ainsi que pour vérifier les propriétés. Il n'est plus possible d'introduire des erreurs accidentelles. Au contraire, la formule utilisée pour calculer la résistance d'un pont est totalement distincte des plans dessinés pour sa construction. Le plan étant différent du modèle, la résistance peut être valable sur le modèle mathématique, mais pas sur le plan.

Une méthode de vérification des propriétés des systèmes en cours de développement consiste à mesurer les documents utilisés. À titre d'exemple, pour deux systèmes comparables, celui dont le cahier des charges est plus grand sera potentiellement le plus cher à développer. Nos travaux reposent donc sur l'hypothèse que la mesure des modèles est intéressante car elle permet de vérifier des propriétés sur les systèmes en cours de développement en vue de réduire leur coût, d'améliorer leur qualité, et d'augmenter en conséquence la satisfaction des utilisateurs finaux.

Notre thèse est que 1) il est possible de spécifier des mesures sur les modèles comme des modèles eux-mêmes et 2) la production de l'outil de mesure, i.e. son implémentation, peut être entièrement automatisée à partir de ce modèle de mesure.

Le premier point est une instance du leitmotiv de l'IDM *tout est modèle*. Nous considérons les mesures de modèles comme des modèles. Le second point est une mise en oeuvre du principe d'automatisation lié à l'utilisation des modèles.

Pour résumer, nous avons donc respectivement deux buts et deux types de modèles. D'une part, il y a la volonté d'analyser des modèles du système pour augmenter leur qualité. Ces modèles sont appelés par la suite modèles de domaine. Notre deuxième but est d'automatiser le développement des outils de mesure par l'utilisation de

[2] L'ingénierie dirigée par les modèles est aussi appelée développement dirigée par les modèles ou architecture dirigée par les modèles. Les expressions d'origine anglaise sont respectivement, et dans l'ordre d'apparition chronologique, *Model-Driven Development (MDD)*, *Model-Driven Architecture (MDA)* et *Model-Driven Engineering (MDE)*

modèles de mesure des modèles.

Notre thèse est divisée en trois principaux chapitres. Le premier chapitre est consacré à l'état de l'art. Nous commencerons par préciser le contexte de nos travaux en donnant un aperçu de la question de la mesure et de l'ingénierie des modèles. Ensuite nous donnerons quelques exemples de métriques, dans le double but d'illustrer la problématique de la mesure, et de montrer l'influence des modèles sur celle ci. Une accointance qui trouve ses racines dans l'étymologie car les deux mots partagent la même racine [Favre, 2006]. Enfin, nous donnerons une liste exhaustive des moyens proposés pour spécifier des métriques.

Le deuxième chapitre présente notre contribution. Nous commencerons par préciser notre problème, qui s'articule en trois points. Puis nous présenterons une approche de la mesure des modèles par les modèles, du point de vue des artefacts mis en oeuvre, puis du point de vue du processus. L'artefact central de notre proposition est un métamodèle de spécification de métriques que nous présenterons en détail. Ensuite, nous décrirons l'architecture logicielle, i.e. les algorithmes et les principes de génération de code. Nous conclurons cette section par une analyse de notre contribution au regard d'un ensemble de critères issus de la littérature.

Le dernier chapitre constitue la validation de notre contribution. Nous appliquons notre approche dans quatre études de cas. Nous verrons comment mesurer le code source des logiciels, des modèles d'architecture logicielle, des systèmes de surveillance maritime, et des cahiers des charges. Ce faisant, nous montrerons que notre approche peut s'appliquer à une grande variété de domaines d'application. Ce dernier point ouvrira d'ailleurs le dernier chapitre, dédié aux perspectives de nos travaux.

2 État de l'art

> "It is unrealistic to expect that general complexity measures (of either the code or system design) will be good predictors of many différent attributes [...] Rather than seek a single measure, we should identify specific attributes of interest and obtain accurate measures of them" [1]
> Norman E. Fenton [Fenton, 1991]

Ce chapitre présente un état de l'art sur les différentes manières de spécifier et d'implémenter des métriques sur des artefacts logiciels et particulièrement sur des modèles du logiciel.

Nous commencerons toutefois par poser le contexte de la mesure et de l'ingénierie dirigée par les modèles. Ainsi, nous présenterons quelques arguments qui montrent que le besoin de mesurer est indépendant du domaine, de l'artefact ou du niveau d'abstraction considéré. Nous donnerons un aperçu de l'ingénierie dirigée par les modèles et l'intuition qui la fonde.

Ensuite, nous préciserons les trois concepts centraux de notre thèse, à savoir la notion de mesure, de modèle, et de métamodèle et ce que nous pouvons en faire.

Nous enchaînerons par l'énumération de quelques exemples de métriques de la littérature. Seulement quelques exemples car la littérature sur le sujet est considérable. Ces quelques exemples ont pour but d'illustrer l'immense variété des métriques, tout en montrant les derniers développements de la question mesure dans les approches récentes à base de modèles (e.g.; UML, MDD, composants logiciels)

Enfin, nous visiterons toutes les approches de la littérature que nous connaissons et qui permettent de définir des métriques de manière exécutable. C'est à la lumière de cette dernière partie que notre contribution pourra être estimée.

2.1 Le contexte

2.1.1 Pourquoi mesurer ?

L'activité de mesure est au coeur du processus de découverte scientifique [Henderson-Sellers, 1996] et du processus d'ingénierie [Fenton, 1991]. Henderson-Sellers [Henderson-Sellers, 1996] prend pour exemple les découvertes astronomiques du siècle des Lumières. Tycho Brahe (1546-1601) compila un grand nombre de données astronomiques. À partir de ces données, Johannes Kepler (1571-1630) en tira un modèle qui explique la trajectoire des planètes, et Isaac Newton en tira les lois de la gravitation. La mesure avait donné naissance au modèle d'une part, et l'avait validé d'autre

[1] Il n'est pas réaliste d'attendre que des mesures générales de complexité (au niveau du code comme au niveau du design) donnent de bonnes prédictions de plusieurs attributs différents. Plutôt que de chercher une mesure unique, nous devrions identifier des attributs spécifiques intéressants et obtenir des mesures précises de ceux-ci.

part. La définition d'un modèle validé constitue alors une étape dans l'exploration de nouveaux modèles plus complets.

La mesure des trajectoires des planètes est une mesure descriptive. Une mesure descriptive est obtenu à partir de l'observation d'une réalité. Généralement, les sciences expérimentales [Zuse, 1991] sont basées sur des mesures descriptives. Du point de vue de l'ingénieur, la mesure n'est pas seulement descriptive, elle est aussi prédictive. Une mesure prédictive est une mesure faite sur un système en cours de développement, sur un produit non fini. Par exemple, dans le cas de la construction d'un pont, l'ingénieur en génie civil doit établir une mesure prédictive de la charge maximale supportée par le pont en question [Spector et Gifford, 1986].

Les activités de mesure sont menées à la fois pour améliorer notre connaissance de la nature et des systèmes, mais aussi pour prédire une ou plusieurs caractéristiques finales du système développé.

Le besoin de mesurer les artefacts d'ingénierie est indépendant du domaine considéré. Les résultats du Programme d'Étude Amont *Mesure de la complexité* [Chretienne et al., 2004] mené par la Direction Générale de l'Armement illustre particulièrement bien ce point. La mesure de la complexité doit se faire dans tous les domaines, de l'informatique à la mécanique, en passant par l'analyse des organisations, et à tous les moments du cycle de vie, de la spécification des besoins au code source exécutable.

Nous étudions maintenant le but de la mesure dans le cadre d'un processus d'ingénierie logicielle. La discussion est une synthèse des arguments présentés dans [Zuse, 1991, Fenton, 1991, Henderson-Sellers, 1996]. D'après ces trois auteurs, la mesure du logiciel est essentiellement prédictive.

Il semble que le but principal de la mesure en phase amont du développement logiciel soit la prédiction de la qualité finale du logiciel produit. Dans ce but, la mesure est directement liée à un ou plusieurs modèles de qualité (e.g.; [ISO/IEC, 2001]). Notons d'ailleurs qu'un ouvrage entier est dédié à la mesure et aux modèles de qualité pour le logiciel [Kan, 1995]. Pour illustrer ce point, nous considérons le modèle de qualité logiciel ISO [ISO/IEC, 2001], qui est un standard ISO. Celui-ci définit six attributs principaux de qualité qui sont:

Capacité fonctionnelle en quoi le logiciel satisfait les besoins exprimés ?

Fiabilité à quel fréquence le logiciel tombe-t-il en panne ?

Ergonomie est-il facile d'utiliser le logiciel ?

Efficacité quels sont les performances du logiciel ?

Maintenabilité le logiciel est-il facile à modifier ?

Portabilité le logiciel peut-il être transféré facilement dans un autre environnement (e.g.; un autre système d'exploitation, une autre architecture matérielle) ?

Au vu de ce modèle de qualité, la mesure sert à estimer les caractéristiques de qualité du produit final en phase amont du développement des systèmes. Par exemple, si l'on considère une architecture en paquetage, disponible en phase amont, avec ses dépendances externes, une mesure de la maintenabilité est le nombre de cycle dans le graphe de dépendances. Plus ce nombre est grand, plus la maintenance est difficile [Martin, 2000]. Cette mesure permet d'estimer la maintenabilité finale, et peut servir de fonction à minimiser afin d'obtenir une architecture optimale. En d'autres termes, la mesure en phase amont permet de valider l'architecture du logiciel, sachant que

les propriétés de cette architecture influent sur les caractéristiques finales de qualité. Nous verrons plus loin les approches et validations nécessaires à la définition des mesures.

Le deuxième but de la mesure est d'aider la gestion des projets de développement logiciel. Le gestionnaire est directement intéressé par les points suivants:

Estimation de coût quel sera le coût du développement étant donnés les besoins ? Quels impacts financiers ont certains choix d'architecture ?

Productivité quelle est la productivité des équipes de développement ? Comment la caractériser ?

Finalement, toutes les mesures prédictives deviennent mesures descriptives dans une analyse des projets terminés. La mesure est alors associée à un processus de collection de données [Fenton, 1991, p. 16]. Ces données servent alors de base pour améliorer les techniques, les outils, les processus et les mesures elles-mêmes.

Comme la mesure permet d'obtenir de meilleurs produits et processus de développement logiciel, d'un point de vue économique, la mesure est créatrice de valeur. De ce point de vue, et pas seulement logiciel, [Metrotrade Project et Regmet Project, 2003] annonce que la mesure coûte 1% du PIB de l'Europe, avec un retour sur investissement de 2 à 7% du PIB.

Nos travaux sont entièrement ancrés sur l'hypothèse que la mesure est créatrice de valeur. Soit la mesure de productivité des activités de mesure P :

$$P(x, y, z) = \frac{x}{y + z} \tag{2.1}$$

où x est la valeur ajoutée due à la mesure, y le coût de développement de l'outil de mesure et z le coût de mesure lui-même. Nos travaux visent à réduire le paramètre y par la génération de l'outil de mesure, et à réduire le paramètre z en automatisant le processus de mesure (comme boucle de *feedback* durant la modélisation). La diminution de y et z font mathématiquement augmenter P.

2.1.2 L'ingénierie dirigée par les modèles

> "The time for MDD has come" [2], Selic [Selic, 2003]

L'ingénierie dirigée par les modèles (IDM) est une des approches modernes du développement des logiciels. Nous commencerons par présenter les problèmes classiques du développement logiciel, historiquement rassemblés sous le terme de *crise du logiciel*. Ensuite, nous replacerons l'IDM dans une perspective historique pour enfin lister les principaux arguments des défenseurs de l'IDM.

La crise du logiciel

> "We found ourselves up to our necks in the software crisis!" [3], Dijkstra [Dijkstra, 1972].

[2] Le temps de l'IDM est arrivé.
[3] Nous sommes dans la crise du logiciel jusqu'au cou !

L'expression *crise du logiciel* a été créée à la Conférence sur le Génie Logiciel de l'OTAN en 1968 [Aspray et al., 1996]. Il exprime l'inaptitude des méthodes de développement à produire des logiciels fonctionnels et robustes dans des coûts et délais estimables. Les principaux symptômes de la crise du logiciel sont:

- les coûts des projets dépassent le budget alloué;
- le logiciel n'est pas délivré à temps;
- le logiciel ne satisfait pas toutes les exigences initiales ou est de mauvaise qualité;
- le projet est annulé.

D'après l'étude présentée dans [The Standish Group, 1995], ces problèmes sont très fréquents: 31% des projets sont annulés, 85% des projets coûtent plus de 20% que l'estimation.

D'aucuns, comme R. France[4], pensent que la crise du logiciel n'est pas résolue. R. France avance même l'hypothèse qu'elle est permanente, du fait que la complexité des applications croît de manière comparable à la maturité des méthodes et techniques du génie logiciel.

La principale cause de la crise du logiciel semble être la relative jeunesse du génie logiciel, quelques décennies seulement, comparée aux autres disciplines d'ingénierie. Spector a ainsi comparé l'ingénierie des logiciels à l'ingénierie des ponts, avec ses millénaires d'histoire et d'expérience dans [Spector et Gifford, 1986].

Dans un sens, l'IDM est une nouvelle tentative de résolution de la crise du logiciel.

Origines Il est difficile de dater la naissance de l'ingénierie des modèles. En effet, d'après Barry Boehm, dans [Boehm, 2006], l'ingénierie dirigée par les modèles n'est pas vraiment une nouveauté : *Successful MDD approaches were being developed as early as the 1950's, in which engineers would use domain models of rocket vehicles, civil engineering structures, or electrical circuits and Fortran infrastructure to enable user engineers to develop and exécute domain applications.* [5]

De même, les efforts de recherche des années 1980 autour du *computer aided software engineering* (CASE) traitaient des mêmes problèmes avec le même type de solution. Pourtant, d'après Schmidt [Schmidt, 2006], les outils et techniques du mouvement CASE n'ont pas été adoptés par l'industrie, et ce malgré l'importante littérature académique et la profusion d'outils commerciaux. Les raisons proposées par Schmidt sont les suivantes : le fossé entre les abstractions proposées et les plateformes alors disponibles était trop grand. De ce fait, le générateur de code était très difficile à concevoir, à maintenir, et à faire évoluer. Le code généré était trop volumineux et trop complexe pour compenser la rusticité des plateformes. D'autre part, les environnements de développement CASE passaient difficilement à l'échelle, ne supportant peu ou pas l'ingénierie concurrente, c'est à dire la production parallèle du logiciel par plusieurs personnes et équipes. Enfin, la prolifération d'environnements et de librairies fermés rendaient l'interopérabilité et l'intégration avec le logiciel existant coûteuse et peu fiable.

[4] Présentation (*keynote*) à la conférence IDM 2007.

[5] Des approches réussies d'ingénierie dirigée par les modèles ont été développées dès les années 1950. Des ingénieurs ont alors utilisé des modèles de domaine pour des fusées, des structures de génie civil ou des circuits électriques, avec une infrastructure Fortran qui leur permettait de développer et d'exécuter des applications propres au domaine.

Enfin, l'IDM d'aujourd'hui a pour fondation les travaux des années 1990 autour des *Domain-Specific Software Architecture (DSSA)* [Mettala et Graham, 1992] du *Model-Integrated Computing* [Sztipanovits et Karsai, 1997] et des *Domain-Specific Languages* [van Deursen et al., 2000].

Pour définir l'IDM, nous choisissons la définition de Mellor et al. qui est suffisamment large tout en étant claire sur l'intuition qui la fonde. Dans cette thèse, nous considérons MDD, MDA et MDE (IDM en anglais) comme une classe d'équivalence, car les subtiles différences entre chacune de ces expressions ne font pas consensus dans la littérature.

Définition 2.1 Model-driven development is simply the notion that we can construct a model of a system that we can then transform into the real thing. [6] *[Mellor et al., 2003]*

Principaux arguments en faveur de l'IDM Richard Soley et l'*Object Management Group* (OMG) ont publié un article fondateur pour l'IDM intitulé *Model Driven Architecture* [Soley, 2000]. Le but de cette nouvelle approche, symbolisée par la création et le dépôt légal de l'expression, est de résoudre les problèmes d'intégration et d'interopérabilité (*"It's about integration. It's about interoperability"* [7] [Soley, 2000, Préface]). Ce papier présentait une solution au problème de la prolifération des intergiciels (*middleware*). L'intuition du papier est de dériver du code à partir d'une description abstraite du logiciel. Avec une telle approche, un changement d'intergiciel, ou plus généralement de plateforme, n'impacte pas le modèle – stable – de l'application mais seulement le générateur de code. La figure 2.1, reproduite de [Soley, 2000] fait apparaître les principaux aspects du MDA. D'une part, le MDA est indépendant du domaine, et peut s'appliquer dans tous les contextes. Cet aspect est symbolisé par les flèches partant dans toutes les directions (e.g.; espace, finance, télécommunication). D'autre part, le MDA tel que présenté dans [Soley, 2000] s'appuie sur un ensemble de standards et de technologies. Il y a la couche des modèles stables, indépendants de la plateforme (exprimé dans UML, MOF et CWM) et la couche des applications finales (Corba, Java, .Net). De cet article séminal, nous retenons surtout l'idée d'une approche indépendante du domaine, et de la génération de code à partir d'un modèle abstrait de l'application.

Même si l'on peut trouver des occurrences plus anciennes de l'expression *Model Driven Engineering* (MDE) [Costa, 2001], Stuart Kent introduit le MDE en 2002 dans l'article éponyme [Kent, 2002]. Kent s'appuie sur la proposition du MDA, mais propose une approche qui se veut plus large (*"MDE is wider in scope than MDA"*[8]) et s'attaque à d'autres problèmes que l'intégration et l'interopérabilité. Kent s'exprime en termes de dimension de modélisation. Il réaffirme la pertinence de la dimension indépendance de la plateforme (PIM/PSM), seul axe de Soley, auquel il ajoute la dimension métier (*subject area*), la dimension concurrence et applications distribuées, la dimension gestion des modèles. En reprenant les arguments de Kent, MDE com-

[6] L'IDM est simplement la notion que nous pouvons construire un modèle d'un système puis le transformer en la chose réelle.

[7] C'est une question d'intégration. C'est une question d'interopérabilité.

[8] Le MDE a une vocation plus large que le MDA.

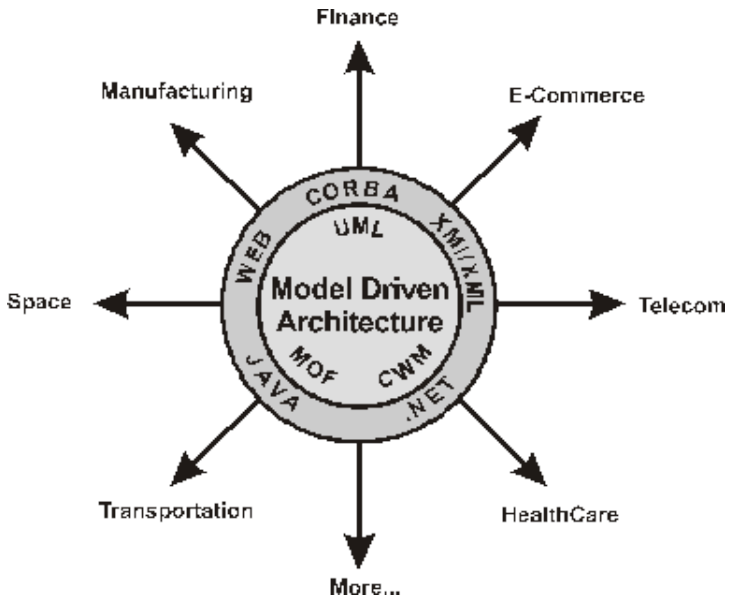

FIG. 2.1 : La figure originale de présentation du MDA

bine dans une même approche du développement logiciel les processus, l'analyse et l'architecture.

En 2003, le journal IEEE Software a publié un numéro spécial sur le *Model-Driven Development*, autre nom de l'IDM. Dans ce numéro, et dans le cadre de cette section d'introduction à l'IDM, nous considérons l'article de Bran Selic *The Pragmatics of Model-Driven Development* [Selic, 2003]. L'article de Selic s'appuie dès l'introduction et de manière filée, sur une comparaison avec les autres disciplines d'ingénierie. Les modèles sont essentiels pour construire par exemple un pont ou une automobile. Pour Selic, les modèles aident à comprendre un problème complexe et ses solutions potentielles. Là est l'argument central de Selic: l'IDM permet d'explorer plusieurs solutions car la création d'un modèle coûte moins cher que la création manuelle du code source correspondant. Ce qui est d'ailleurs présenté comme une des conditions requise pour un modèle: *a model must be inexpensive*[9]. Enfin, dans le cas du logiciel, il est possible de générer des programmes complets à partir des modèles. Selic conclut alors que l'IDM, par l'exploration des possibilités et la génération de code, a le potentiel d'un bénéfice majeur de productivité et de fiabilité du logiciel.

Plus récemment, en 2006, le journal IEEE Computer a consacré une section spéciale sur le sujet, cette fois-ci intitulé *Model-Driven Engineering*. Nous avons exposé ci-dessus le regard de Schmidt [Schmidt, 2006] sur les origines de l'IDM par rapport aux outils *CASE*. Par ailleurs, le but de Schmidt est de remplacer le battage publicitaire lié à l'IDM [Schmidt, 2006]) par des considérations techniques et des retours d'expérience. Comparé aux auteurs précédents, Schmidt est plus radical sur les apports de l'IDM. Pour lui, les approches précédentes basées sur des langages de programmation généralistes, dits de troisième génération, sont incapables de soulager la complexité des plateformes d'aujourd'hui et de détecter et d'éviter les erreurs tôt

[9] Un modèle ne doit pas être cher.

dans le cycle de vie. Schmidt articule l'IDM autour des langages de modélisation spécifiques au domaine (*Domain-specific modeling languages (DSML)*), et les moteurs de transformation et de génération de code. Les DSML intègrent l'utilisateur final dans la modélisation et aident à l'adéquation fonctionnelle du logiciel livré. Les moteurs de transformation assurent la cohérence par construction entre les propriétés des modèles et celles de l'implémentation finale.

Enfin, Jézéquel explicite [Jézéquel, 2008] le lien entre l'IDM et les approches fondées sur les aspects et les lignes de produits (*product lines*) et avance l'argument que la modélisation et le tissage d'aspects sont des activités symétriques. Jézéquel conclut que la force de l'IDM est de rendre automatique le tissage entre modèle d'analyse pour produire le logiciel final. Batory quant à lui, souligne [Batory, 2006] que l'approche IDM est proche des travaux liés à la métaprogrammation: *MDE should be a next-generation OO technology that focuses on programming at the meta-application level (and higher levels) [...]. At this level, program design and development is a computation, historically called metaprogramming.*[10]

Nous avons présenté dans cette section les arguments des principaux défenseurs de l'IDM en faveur de son adoption. Pour conclure la section et se placer cette fois-ci dans le futur, voici les trois grands défis à relever pour réaliser la vision de l'IDM, d'après R. France et B. Rumpe [France et Rumpe, 2007].

Langages de modélisation Comment créer et utiliser des langages de modélisation du niveau du problème ? Comment analyser les modèles ?

Séparation des préoccupations Comment gérer des points de vue multiples et se recoupant au travers de plusieurs langages ?

Manipulation et gestion des modèles Comment définir, analyser, utiliser les transformations de modèles ? Comment gérer la traçabilité, les différentes versions ? Comment utiliser les modèles à l'exécution ?

Nos travaux se situent très clairement dans le cadre de l'IDM. D'une part, l'ingénierie dirigée par les modèles promeut l'usage de modèles pour vérifier les propriétés du système dont certaines s'expriment à l'aide de métriques (nous verrons en 2.3 des exemples de métriques). D'autre part, l'IDM permet d'obtenir des logiciels de mesure des modèles à moindre coût par génération de code (argument de Selic appliqué à la mesure des modèles).

2.2 Précisions sur les concepts utilisés

> "The four english words Model, Mold, Module, and Modification are all derived from the Indo-European root MED, which led to the Latin word Modus, that is Measure as a noun. [...] while the notion of measure is one of the roots of all scientific sciences, in particular of so-called exact sciences, this notion leads also to the notion of Model which [...] is consi-

[10] L'IDM devrait être la prochaine génération des technologies orientées objet, qui mettra l'accent sur la programmation au niveau méta-application (et plus haut). À ce niveau, l'architecture et le développement des programmes résultent de l'exécution d'un autre programme, appelée historiquement méta-programmation.

dered to be the confluent of sciences." [11]
Favre, [Favre, 2006]

2.2.1 Des mesures

Le mot français *mesure* est un mot polysémique, comme en témoigne la longueur de l'entrée *mesure* dans le Petit Robert. Il recouvre à la fois une action, une définition d'action et une valeur numérique. Une mesure comme action est une suite de procédures effectuée à un moment donné, par exemple, la mesure du Mont-Blanc par satellite en 1983. La mesure comme définition d'action est une spécification d'un ensemble de règles permettant d'attribuer un nombre ou un symbole à une chose. Par exemple, la mesure de la longueur d'une voiture est la distance maximale entre le pare-choc arrière et le pare-choc avant. La mesure comme valeur numérique est le résultat d'une mesure action de mesure, par exemple *La mesure du Mont-Blanc est 4808 mètres*. On préfère usuellement la formulation *Le Mont-Blanc mesure 4808 mètres*, où le verbe mesurer est un verbe d'état, ce qui montre le caractère interne d'une mesure par rapport à la chose mesurée. Fenton [Fenton, 1991] insiste sur le fait que la mesure est toujours liée à un attribut et que l'on ne doit pas l'omettre. Dans cette perspective, il aurait fallu dire dans l'exemple précédent *La mesure de hauteur du Mont-Blanc est de 4808 mètres*. Notons aussi que le polymorphisme suscité disparaît partiellement en anglais. Le mot *measurement* est la mesure comme action, alors que *measure* reste polysémique en tant que mesure comme procédure et mesure comme valeur.

Enfin, plusieurs auteurs font la distinction entre mesure et métrique. Une métrique est une fonction de deux éléments satisfaisant 1) $f(x, y) = 0 \Leftrightarrow x = y$; 2) $f(x, y) = f(y, x)$; 3) $f(x, z) \leq f(x, y) + f(y, z)$. C'est donc un concept de distance (l'étymologie lie le mot métrique au mètre) alors qu'une mesure est une fonction d'un seul élément. Pour une approche mathématique de la mesure, nous nous reportons à l'ouvrage de Krantz [Krantz et al., 1971]. En étant strict, il est donc préférable d'utiliser le terme mesure. Toutefois, nous partageons le point de vue de [Henderson-Sellers, 1996] et constatons que l'usage commun permet l'utilisation interchangeable des deux termes.

Afin de surmonter les pièges sémantiques dus à la terminologie utilisée (cf. [Garcìa et al., 2006]), nous adoptons les définitions suivantes.

Définition 2.2 *Une métrique est un homomorphisme du monde empirique vers le monde des nombres. En tant qu'homomorphisme, une métrique préserve les relations empiriques (inspirée de [Zuse, 1991, p.16]).*

Définition 2.3 *Une mesure est une suite d'actions effectuée afin d'assigner un nombre aux attributs des entités du monde empirique (inspirée de [Fenton, 1991, p.5]).*

Définition 2.4 *Une valeur de mesure, ou valeur de métrique, est le résultat d'une mesure pour une métrique identifiée et un objet donné de la réalité empirique (inspiré de* measurement result *[CEI et ISO, 1993, p.15]).*

[11] Les quatre mots anglais *Model, Mold, Module*, et *Modification* sont tous dérivés de la racine indo-européenne MED, qui a donne le mot latin Modus, qui veut dire "Mesure" [...] tandis que la notion de mesure est une des racines de toutes les sciences, en particulier des sciences dites exactes, cette notion mène aussi à la notion de modèle qui [...] est considéré comme étant au confluent des sciences.

Nous essaierons au maximum d'utiliser dans la suite de cette thèse les mots *métrique*, *mesure* et l'expression *valeur de mesure* conformément à ces définitions.

2.2.2 Des modèles

L'ingénierie dirigée par les modèles place les modèles au coeur de la conception et du développement des systèmes. À ce titre, il est important de définir ce qu'est un modèle. Un tour d'horizon des articles d'encyclopédie (e.g; [Mouloud, 2006]) montre la variété des domaines dans lesquels est utilisé le mot *modèle*. De même, l'usage des modèles varie grandement. On trouve dans la littérature des dizaines d'articles et d'ouvrages sur les modèles. Dans cet état de l'art, nous nous appuyons sur des articles d'encyclopédie [White, 2000, Mouloud, 2006, Frigg et Hartmann, 2006] pour la vision globale de la modélisation. L'article de Jeff Rothenberg [Rothenberg, 1989] est un pont entre modèle et informatique. Enfin, nous présentons quelques travaux sur le sujet effectués par des personnes appartenant au monde du génie logiciel. Il est cependant hors du cadre de cet état de l'art d'aborder les questions épistémologiques associées à la notion de modèle.

On trouve dans l'article *Modèle* de l'*Encyclopédie Universalis* [Mouloud, 2006] une classification des modèles par domaine d'application. Ils sont: mathématique; physique; sciences de la Terre; biologie; sciences sociales; psychologie; linguistique; art. Dans chaque domaine, Mouloud étudie les modèles utilisés et leurs types. Prenons l'exemple des mathématiques: un modèle en mathématique est un concept formalisé dans la *théorie des modèles* [Andler et al., 2006], une branche de la logique formelle. Un modèle en mathématique est donc fondamentalement différent d'un modèle mathématique.

De manière comparable, White, Frigg et Rothenberg [White, 2000, Frigg et Hartmann, 2006, Rothenberg, 1989] propose une classification des modèles. Nous proposons donc ici une synthèse des différents types de modèles:

modèles physiques C'est un objet physique. Il peut être modèle réduit ou prototype[12] [White, 2000, Frigg et Hartmann, 2006, Rothenberg, 1989];

modèles mathématiques Un ensemble d'équations ou d'expressions logiques [White, 2000];

modèles fictionnels (proche des métaphores conceptuels de [Rothenberg, 1989] et des modèles mentaux de [White, 2000]) C'est une vue idéalisée d'un système, par exemple une pendule sans friction, la vue d'un atome comme un système solaire miniature, ou le cerveau comme une machine [Frigg et Hartmann, 2006];

modèles iconiques Ce sont des images ou des dessins [White, 2000];

modèles textuels descriptifs C'est une description en langage naturel d'un système et/ou de sa dynamique. On peut donc avoir un modèle descriptif d'un modèle fictionnel [White, 2000, Frigg et Hartmann, 2006, Rothenberg, 1989].

Contrairement au découpage par domaine de Mouloud, où n'apparaissait pas les modèles d'ingénierie, on les trouve dans cette taxonomie. Par exemple, le modèle mathématique des équations de résistance à la charge d'un pont est un modèle d'ingénierie. Par contre, il est difficile d'y classer les modèles des logiciels, qui sont parfois des descriptions textuelles mais pas en langage naturel.

[12] I.e.; un exemplaire d'un système avant sa production industrielle

Outre la classification par domaine et la classification par type, il nous semble important de citer la classification de Seidewitz [Seidewitz, 2003]. Pour Seidewitz, on peut classer les modèles en deux catégories. Les modèles descriptifs et les modèles prescriptifs. Les modèles descriptifs réfèrent à une réalité existante. Ainsi tous les modèles explicatifs liés à l'observation sont descriptifs. On comprend alors mieux l'absence des modèles d'ingénierie dans [Mouloud, 2006], qui ne considère implicitement que les modèles descriptifs. Les modèles prescriptifs considèrent un système qui n'existe pas encore. En ce sens, ils peuvent être vus comme une spécification ou comme une base d'étude des propriétés du système à construire.

Une approche de la définition d'un modèle est de lister ces attributs et ces propriétés. Sur ce point, on peut constater un consensus entre les auteurs étudiés ([Rothenberg, 1989, White, 2000, Selic, 2003]). Un modèle permet de raisonner sur un système à moindre coût que sur le système lui-même. Par exemple, il est moins coûteux de tester différentes tailles de câble pour un pont suspendu avec un modèle mathématique que de construire N ponts! Dans certains cas, un modèle permet même d'étudier des systèmes impossibles à construire ou à tester (par exemple, les modèles de courants atmosphériques et océaniques considérant des durées de plusieurs siècles).

Outre l'aspect économique, Selic propose trois autres attributs d'un modèle:

l'abstraction Il faut comprendre ici abstraction au sens granularité. Un modèle doit enlever ou cacher des détails;

la compréhensibilité La compréhension du système par l'étude du modèle doit être plus facile que par l'étude du système lui-même. On retrouve cet argument chez [Mouloud, 2006] sous le terme *clarté*;

la prédictibilité C'est la faculté du modèle à prédire les propriétés réelles du système auquel il réfère (ou les propriétés du système du futur dans le cas d'un modèle prescriptif).

On voit dans l'approche de Selic que la définition d'un modèle doit être cherchée dans sa relation avec le système auquel il réfère. Kuehne a étudié la relation entre le modèle et l'objet auquel il réfère [Kuehne, 2006]. Ces relations sont principalement:

X modélise Y par fragment (*token model of*) À chaque élément du modèle X correspond un élément de l'objet modélisé Y (relation un pour un). Certains éléments de l'objet modélisé n'apparaissent pas dans le modèle. Cette relation est proche de l'abstraction de Selic (enlever ou cacher des détails).

X modélise Y ontologiquement (*ontological model of*) À chaque élément du modèle X correspondent plusieurs éléments de l'objet modélisé Y (relation un pour plusieurs). Le rapport entre les éléments du modèle et les éléments de l'objet modélisé est de type ontologique.

X modélise linguistiquement Y (*linguistic model of*) À chaque élément du modèle X correspondent plusieurs éléments de l'objet modélisé Y (relation un pour plusieurs). Le rapport entre les éléments du modèle et les éléments de l'objet modélisé est de type linguistique.

Ces trois relations sont illustrées figure 2.2. Le papier de Kuehne présente d'autre relations (abstraction, classification, modèle par type). Certaines se recoupent et semblent équivalentes. Notons aussi que Favre [Favre, 2006] définit quatre relations entre systèmes et cinq relations *modèle-de*. Finalement, nous retiendrons l'analyse de

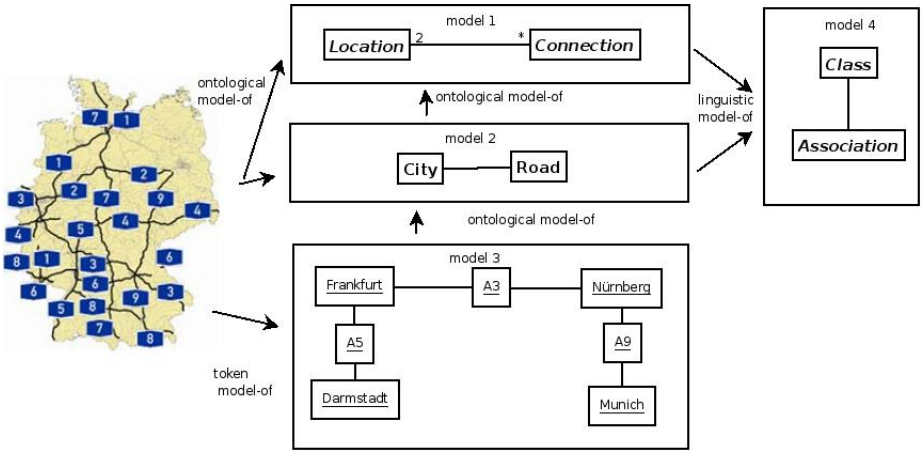

FIG. 2.2 : Illustration des relations *est un modèle de*

[Rothenberg, 1989], la plus pragmatique et la plus utilisable. Rothenberg définit trois conditions pour dire qu'un objet X est un modèle de Y:

Définition 2.5 – *un modèle X réfère à un objet Y (une réalité observée, un système à construire);*
– *un modèle X a un but précisément identifié;*
– *il est plus rentable d'utiliser le modèle X que l'objet Y auquel il réfère pour atteindre le but recherché.*

Dans la suite de cette thèse, nous vérifierons au besoin ces trois conditions pour valider qu'un objet est un modèle d'un autre objet.

Qui plus est, cette définition est suffisamment ouverte pour pouvoir dire que tout est modèle [Bézivin, 2005]. Ce principe unificateur est au coeur de l'IDM. Bézivin l'explicite avec une liste de ce qui peut être perçu comme modèle:

Les programmes Un programme peut être vu comme le modèle de ses traces d'exécution. Il peut aussi être vu comme un modèle de l'activité auquel il réfère. On trouve cette deuxième interprétation dés 1994 dans [Lehman, 1994].

Les traces Une trace est un modèle de l'exécution d'un programme. La relation à la Kuehne entre le modèle et l'exécution est alors de type fragment.

Les plateformes Les plateformes peuvent être considérées comme le modèle d'un système d'exploitation, d'un *framework*, ou d'une architecture matérielle. Cela comprend les plateformes du passé, et donc le code existant/historique (*legacy*).

Les composants logiciels Un composant peut être perçu comme modèle des instances déployées.

Les artefacts de l'IDM En adoptant un point de vue réflexif, tous les artefacts de l'IDM sont des modèles: métamodèles, transformations, vérifications.

Le lien entre modèle et mesure est abordé par Zuse [Zuse, 1991, p. 28]. Zuse déclare que la modélisation des logiciels est une étape essentielle pour établir des critères de qualité du logiciel. Il renforce l'importance de la modélisation en énonçant que beaucoup de problèmes des mesures viennent en fait des modèles sous-jacents. Ces

considérations sont au coeur de nos travaux. D'une part, les modèles sont nécessaires pour la mesure, d'autre part la qualité des mesures dépend, entre autres choses, de la qualité des modèles eux-mêmes.

2.2.3 Des métamodèles

> "Somehow we need a way to go meta" [13]
> Kent, [Kent, 2002].

Nous avons exploré dans la section précédente la nature des modèles et nous avons choisi une définition dite par intension[14] issue de la littérature. Mais qu'en est-il des métamodèles ? Le but de cette section est de choisir une définition pour le concept de métamodèle.

En effet, la définition du concept de métamodèle n'est pas triviale. La question a été récemment visitée dans [Seidewitz, 2003, Favre, 2006, Kuehne, 2006]. Kuehne commence par explorer la nature du préfixe méta. Le préfixe méta veut habituellement dire que l'on applique un concept à lui-même. Par exemple, une méta-discussion est une discussion sur les discussions, le méta-apprentissage consiste à apprendre à apprendre. Ainsi, la définition la plus simple d'un métamodèle est:

Définition 2.6 *Un métamodèle est un modèle de modèles [Miller et Mukerji, 2003].*

Dans la section précédente, nous avons vu que la relation *modèle-de* s'applique à deux objets. Dans les exemples du préfixe méta, nous avons cité une méta-discussion comme discussion sur les discussions. Le complément *sur les discussions* est au pluriel. Il en est de même dans la définition précédente. Ainsi, Seidewitz note qu'un métamodèle est un modèle pour une famille de systèmes. Ainsi, soit $Z_1 ... Z_n$ une famille de systèmes. Alors Y est un métamodèle si et seulement si $Y = modele - de(\{modele - de(Z_1), ..., modele - de(Z_n)\})$.

Le problème n'est que reporté. En effet, Kuehne et Favre ont montré l'existence de plusieurs relations *modèle-de*. Si bien qu'il faut décider si l'équivalence précédente est valable pour n'importe quelle relation *modèle-de*. C'est-à-dire vérifier si :

$$\forall f, \forall g_i \in E_{modele-de} \implies Y = f(\{g_1(Z_1), ..., g_n(Z_n)\}) \tag{2.2}$$

À cette question, Favre ne répond pas: *Is a metamodel a model of a model? [...] the answer is in the middle, or better said, one can say either YES or NO, depending on the interpretation of the word model* [15]. Kuehne énonce qu'il suffit que f soit une relation *modèle-de* de type ontologique ou linguistique [Atkinson et Kühne, 2003]. Ainsi, dans l'exemple de la figure 2.2, le modèle 1 est un métamodèle vis à vis du modèle 2, le modèle 2 est un métamodèle vis à vis du modèle 3, le modèle 4 est un métamodèle vis à vis des modèles 1,2. Le modèle 4 illustre bien que la notion de métamodèle est au dessus d'un ensemble de modèles.

Ceci étant, il apparaît que le travail de métamodélisation est un travail de modélisation comme un autre [Bézivin, 2005]. La question de la *métaïté* (*metaness* de

[13] D'une manière ou d'une autre, nous avons besoin d'un moyen pour faire du *meta*.
[14] au contraire des définitions par extension
[15] Un métamodèle est-il un modèle de modèles ? [...] la réponse est au milieu, ou encore mieux, peut être parfois OUI parfois NON, en fonction de l'interprétation du mot modèle.

[Kuehne, 2006]) en IDM peut alors être considérée comme une instance nouvelle de la question des méta-mathématiques. En somme, Bézivin dans son domaine énonce la même chose que Wittgenstein en mathématique : *Hilbert's metamathematics must turn out to be mathematics in disguise* [16] [Waismann, 1983, p.136].

Expression d'un métamodèle On peut trouver dans la littérature un grand nombre de formalismes pour exprimer des métamodèles. Nous invitons le lecteur à se reporter au chapitre correspondant de [Tolvanen, 1998] pour plus d'informations.

Aujourd'hui une grande majorité des outils [Ledeczi et al., 2001, Budinsky et al., 2004] se placent dans le cadre du MOF[17] [OMG, 2004] pour exprimer des métamodèles. Notons que les travaux autour d'*Atom3* [de Lara et Vangheluwe, 2002] utilisent des métamodèles définis avec le paradigme ER (Entité-Relation) [Chen, 1976] et que les travaux de Ducasse et al. utilisent le *Famix* [Demeyer et al., 2001] dans un environnement *Smalltalk* [Ducasse et Gîrba, 2006, Ducasse et al., 2008].

La sémantique exécutable d'un métamodèle (génération de code, transformation de modèles, interprétation) peut s'exprimer de plusieurs manières [Harel et Rumpe, 2004]. Dans le cadre de cette thèse, nous avons utilisé l'approche *Kermeta* [Muller et al., 2005] basée sur EMOF. L'approche *Kermeta* met en mouvement les modèles avec une sémantique dite opérationnelle. Cela consiste à ajouter des opérations aux classes des métamodèles manipulés. Ce point est illustré dans le listing 2.1.

Listing 2.1 : Exemple d'un métamodèle de machines à état et sa sémantique exécutable en *Kermeta*

```
1
2  // représente la notion d'état
3  class State {
4      attribute name : String
5      // définition d'une association binaviguable
6      attribute outgoingTransition : set Transition[0..*]#source
7
8      /* une sémantique exécutable de l'automate est codée
9      *  dans cette opération
10     */
11     operation step(c : String) : String
12     is do
13        result := outgoingTransition.select { t |
14          t.inputStimulus.equals(c) }.one.outputMessage
15     end
16 }
17
18 // représente la notion de transition
19 class Transition {
20     reference source : State[1..1]#outgoingTransition
21     reference target : State[1..1]
22     attribute outputMessage : String
23     attribute inputStimulus : String
24 }
```

[16] Les méta-mathématiques de Hilbert se révéleront être des mathématiques sous un déguisement
[17] ou d'une de ses variantes e.g.; EMOF, CMOF, KM3 [Jouault et Bézivin, 2006]

Classe	Exemples
Processus	Nombre de demandes de maintenance évolutive Nombre de défauts par classe d'origine
Produits	Nombre de lignes de code Quantité de code réutilisable Nombre de défauts par classe d'origine
Ressources	Coût du développement Temps alloué à la maintenance

TAB. 2.1: Quelques exemples pour chacune des trois classes de métriques

Génération d'outil à partir d'un métamodèle L'idée de générer un outil à partir d'un métamodèle est très ancienne. Park et al. listent en 1993 [Park et Kim, 1993] de telles tentatives dans le domaine du génie logiciel (outils *MetaCASE*), des modèles de données et l'ingénierie des connaissances. Là où Park et al. sont innovants, c'est qu'ils proposent de générer l'outil de modélisation lui-même. De même, la thèse de N. Revault en 1996 expose l'équivalent de ce que les outils GMF et *Topcased* proposent aujourd'hui [Revault, 1996].

Cette idée a depuis été explorée dans le cadre du *Model-Integrated Computing* [Sztipanovits et Karsai, 1997], de la modélisation à points de vue multiples [de Lara et Vangheluwe, 2002], jusqu'à l'implémentation en Java, dans *Eclipse*, et soutenue par IBM de l'*Eclipse Modeling Framework* [Budinsky et al., 2004], et des projets associés (GMF[18], *Topcased*[19]).

2.3 Quelques exemples de métriques

Maintenant que nous avons décrit le contexte de nos travaux et précisé les concepts utilisés, nous présentons quelques exemples de métriques.

Une classification largement partagée des métriques consiste en trois classes [Fenton, 1991, p. 74]:

- les métriques de processus sont liées aux activités;
- les métriques de produits concernent les artefacts créés et échangés;
- les métriques de ressources mesurent les ressources nécessaires à chacune des activités du processus de développement.

Le tableau 2.3, inspiré de [Fenton, 1991], donne quelques exemples de métriques pour chacune de ces classes. Notons que cette classification n'est pas exclusive. Certaines métriques, comme par exemple le nombre de défauts par classe d'origine, se situent dans deux classes à la fois.

La littérature sur la mesure des logiciels étant très vaste, nous avons donc dû sélectionner quelques éléments seulement. Le choix qui a guidé la sélection est basé sur deux principes: d'une part, la volonté d'illustrer les trois classes par des papiers fructueux (*seminal*), et d'autre part, la volonté de montrer l'évolution des métriques conjointement à l'évolution des processus et des produits (e.g.; du développement

[18] http://www.eclipse.org/gmf
[19] http://www.topcased.org

procédural avec un langage procédural à l'ingénierie dirigée par les modèles avec UML).

2.3.1 Métriques sur les processus de développement

Nous présentons dans cette section deux métriques:
– le nombre de défauts trouvés après la livraison;
– la couverture de test des modèles.

.

Le choix de la première métrique repose sur deux raisons. D'une part, Zuse présente dans son livre [Zuse, 1991, p. 491] le résultat d'une étude sur les métriques les plus utilisées. La mesure du nombre de défauts trouvés après la livraison est utilisée par 61% des entreprises interrogées. D'autre part, c'est une métrique de processus qui arrive en fin de cycle de vie, qui est largement utilisée pour calibrer et valider les modèles de prédiction des défauts à venir, sujet central de la qualité et de la fiabilité des logiciels [Fenton et Neil, 1999]. Notons que le nombre de défauts trouvés après la livraison est une métrique de processus qui est collectée manuellement durant l'utilisation du logiciel.

La couverture de test des modèles est une métrique de processus présentée dans le document *MDD Engineering Metrics Definition* issu du projet européen Modelware [Modelware Project, 2006]. La couverture de test des modèles est le pourcentage d'éléments de modèles complètement testés durant les n derniers mois, où n est propre à chaque compagnie en fonction de la durée des projets (traduit de [Modelware Project, 2006, p. 46])

Nous avons illustré dans cette section deux métriques de processus, tout en illustrant l'évolution de leur définition par rapport aux pratiques de modélisation.

2.3.2 Métriques sur les ressources

Une métrique de ressource est le coût des logiciels [Boehm et Sullivan, 2000]. À ce sujet, l'approche Cocomo est séminale [Boehm et al., 1995]. Cocomo est un modèle d'estimation de coût basé sur l'estimation de la taille du logiciel et plusieurs paramètres liés au processus de développement et au domaine d'application. Le modèle Cocomo a donné naissance à une abondante littérature sur l'estimation du coût des logiciels. La formule de Cocomo est présentée en 2.3. Le résultat est un coût en personne/mois (PM). A est une constante calibrée suivant un historique de coût. B représente les économies ou surcoût d'échelle et est calculé suivant des paramètres du processus de développement (e.g.; la taille de l'équipe). $Size$ est la taille estimée du logiciel en KLOC.

$$PM_{nominal} = A * (Size)^B \tag{2.3}$$

Face aux nouvelles pratiques de développement liées à la modélisation, le projet ModelWare propose une évolution des métriques de coût [Modelware Project, 2006]. La métrique correspondante est l'effort de modélisation. Elle est définie comme la somme de tous les efforts manuels dépensés pour produire les éléments des modèles du système. À cette définition est associée la formule suivante:

$$ModEff = \text{Création des modèles} \quad \begin{array}{l} + \text{ coût d'écriture des transformations} \\ + \text{ coût d'exécution des transformations} \end{array} \tag{2.4}$$

Notons que l'ingénierie dirigée par les modèles étant générative, il est donc nécessaire d'expliciter dans le coût l'aspect manuel du développement. De même, le coût se divise en trois quantités dont les deux dernières sont directement liées à la génération d'artefacts. Nous ne connaissons pas d'études quantifiant la part moyenne de chacun de ces coûts dans le coût final.

2.3.3 Métriques sur les produits

Métriques sur les logiciels procéduraux

Dans cette section, nous présentons brièvement trois métriques *traditionelles*: le nombre de lignes de code, la métrique de McCabe, et les métriques de flots d'information de Henry et Kafura. Nous les avons sélectionnées pour la richesse de la littérature associée et du fait de l'existence d'évaluations empiriques.

La mesure du nombre de lignes de code est une mesure très utilisée [Fenton, 1991, p. 246]. Sa définition est intuitive bien que certains auteurs ont montré que la définition unique et non-ambiguë est difficile [Fenton, 1991, p. 249], ce qui a conduit Kan [Kan, 1995] à énoncer que *the lines of code (LOC) metric is anything but simple*[20]. Les ambiguïtés sont par exemple: doit-on compter les retours à la ligne ? les commentaires ? les appels de fonction imbriqués ?

En terme de validation, Kitchenham et al. [Kitchenham et al., 1990] ont mis à jour un lien empirique entre le nombre de lignes de code et la densité de défauts et de changements. De même, Withrow [Withrow, 1990] donne les résultats d'une expérience qui montre qu'un module Ada doit avoir une taille ni trop petite ni trop grande, qui correspond à un minimum en terme de densité de faute. C'est le principe de Boucle d'Or en référence au conte éponyme. Nous rappelons aussi le lien fort qui existe entre la mesure du nombre de lignes de code et les modèles d'estimation de coût des logiciels (e.g.; [Boehm et al., 1995]). Notons l'existence de points de vue critiques dans la littérature sur la pertinence de cette métrique [Khoshgoftaar et Munson, 1990] pour prédire la densité de fautes.

La mesure de McCabe [McCabe, 1976] est une mesure très connue [Henderson-Sellers, 1996, p. 92]. Elle est basée sur la théorie des graphes et est présentée dans l'article original comme le nombre de chemins linéairement indépendants. Plusieurs auteurs ont discuté de la formule, mais reste l'idée principale de compter les points de branchements dans le code (e.g.; le instructions conditionnelles ou les boucles). Kitchenham et al. [Kitchenham et al., 1990] ont trouvé une corrélation empirique (coefficient de Spearman) entre la mesure de McCabe et trois attributs de qualité (le nombre de changements, le nombre de fautes, et la complexité subjective).

Enfin, il faut noter les métriques *fan-in* et *fan-out* de Henry et Kafura [Henry et Kafura, 1981]. Elles sont définies par le nombre de dépendances entrantes et sortantes pour un artefact donné. La précision de cette définition est donc directement liée à la précision des frontières de l'artefact et de la notion de dépendance. Ferneley

[20] La mesure du nombre de lignes de code est tout sauf facile.

[Ferneley, 1997] a trouvé une relation empirique entre ces mesures et le temps de développement, la densité de faute et la qualité du design (évaluée subjectivement). Notons la présence de résultats bien moins probants dans [Kitchenham et al., 1990] (coefficient de corrélation bien plus faible).

Métriques sur les logiciels orientés objet

Le papier de Chidamber et Kemerer sur les métriques orientées-objet [Chidamber et Kemerer, 1994] est un des plus cités en génie logiciel [Wohlin, 2007] (deuxième sur la période 1986-2005). Dans ce papier, Chidamber et al. définissent six métriques pour les classes des logiciels orientés-objet:

- WMC: le nombre de méthodes pondéré par leur complexité;
- DIT: la profondeur maximum d'héritage pour une classe;
- NOC: le nombre de classes directement descendantes;
- CBO: la mesure de couplage correspondant au nombre de classes dont la classe considérée est dépendante;
- RFC: le nombre de méthodes qui peuvent être potentiellement exécutées en réponse à un message;
- LCOM: la mesure du manque de cohésion, c'est le nombre de paires de méthodes qui n'utilisent aucune variable d'instance en commun.

Notons que ces définitions ont fait par la suite l'objet de précisions ou d'adaptation pour tel ou tel langage orienté objet. Basili et al. [Basili et al., 1996] ont montré empiriquement que: 1) ces métriques sont de bon indicateurs pour prédire la probabilité de faute et 2) que ces indicateurs sont meilleurs que les métriques traditionnelles. Notons aussi les études de [Hitz et Montazeri, 1996, Demeyer et Ducasse, 1999] sur le sujet.

Métriques sur UML

Nous présentons dans cette section les travaux concernant les métriques sur UML. Ces travaux sont importants pour nous car, d'une part, ils définissent des métriques sur des modèles, et d'autre part, l'objet mesuré – un modèle UML – est explicitement instance d'un métamodèle défini comme tel.

Des métriques OO aux métriques sur UML il n'y a qu'un pas. Ce pas est franchi en 1998 par Marchesi [Marchesi, 1998] deux ans après la publication des premières versions du standard UML. Marchesi définit 5 métriques sur les classes UML, 5 métriques globales sur les diagrammes de classes (e.g.; le nombre pondéré des dépendances d'une classe), 3 métriques sur les paquetages (e.g.; le nombre de hiérarchie de classes) et 3 métriques sur les cas d'utilisation (e.g.; le nombre de communications entre cas d'utilisation et acteur).

L'exploration des métriques sur les diagrammes de classes est poursuivie par Genero et al. [Genero et al., 2000]. Ils montrent en 2002 [Genero et al., 2002b] un lien empirique de 11 de ces métriques avec la complexité subjective puis publient en 2005 une étude bibliographique sur le sujet [Genero et al., 2005]. Notons aussi le papier de Tang et al. [Tang et Chen, 2002] qui n'a pas pour but de définir des nouvelles métriques mais de présenter une méthode outillée de mesure des modèles UML. Tang et al. mesurent des modèles issus de Rational Rose.

Les métriques sur les diagrammes comportementaux consistent en des métriques sur les diagrammes d'état. Genero et al. [Genero et al., 2002a] définissent cinq métriques sur ces diagrammes, qui sont ensuite explorées dans le cadre théorique *DISTANCE* [Poels et Dedene, 2000] puis validées comme étant liées à la difficulté de compréhension avec une petite expérience empirique. Baroni a présenté dans un symposium doctoral [Baroni, 2005] ses travaux en cours sur le sujet, qui, à notre connaissance, n'ont pas fait encore fait l'objet de publications.

La littérature sur la mesure des cas d'utilisation sera présentée plus tard dans cette thèse, dans le chapitre sur la mesure des exigences. La mesure des modèles de composants logiciels, comprenant les composants UML, est présentée dans la section qui suit.

Pour autant que nous en sachions, les autres diagrammes d'UML n'ont pas fait l'objet de travaux dédiés à leurs mesures. Enfin, notons deux publications sur la mesure des expressions OCL[21], de Reynoso et al. [Reynoso et al., 2003] et de Cabot et al. [Cabot et Teniente, 2006]. La dernière définit explicitement la métrique sur OCL en fonction du métamodèle OCL, et donc raisonnant sur une expression OCL comme un ensemble d'objets liés.

Métriques sur les composants logiciels

Historiquement, les modèles des logiciels sont de plus en plus abstraits [Boehm, 2006]. Cette dernière section considère l'ingénierie basée sur les composants [Szyperski, 1998] qui représente une étape importante dans l'abstraction et est étudiée par une communauté de recherche active. Nous présentons dans cette section quelques travaux sur la mesure des composants logiciels dont Schmietendorf [Schmietendorf et al., 2000] puis Gill ont montré l'importance [Gill et Grover, 2003]. Le but de cette section est de prouver que des métriques peuvent être définies sur un domaine très ciblé. Dans une certaine mesure, nous considérons les composants logiciels comme des modèles d'un langage de modélisation spécifique (DSML) i.e.; comme des modèles de domaine (au sens large).

Cho et al. ont proposé 13 métriques sur les composants [Cho et al., 2001], regroupées en 3 catégories: complexité, adaptabilité, réutilisabilité. Par exemple, une métrique de complexité statique d'un composant est le nombre de dépendances (composition, agrégation, généralisation) entre les classes du composant. Ils appliquent ensuite leurs métriques sur des exemples jouets. Les métriques ne sont pas explicitement liées à un type de composant mais l'étude de cas considère des composants de type EJB. Notons aussi qu'ils n'explicitent pas du tout le lien avec un métamodèle quelconque.

Washizaki et al. [Washizaki et al., 2003] définissent 5 métriques pour mesurer les composants logiciels. Par exemple, RCO (*Rate of Component Observability*) est le pourcentage de propriétés accessibles en lecture par la façade du composant. L'aspect novateur de leur approche est de mesurer les composants en boite noire, avec les informations extérieures et les méta-informations disponibles. Ils considèrent les composants EJB sans métamodèle associé.

Narasimhan et al. [Narasimhan et Hendradjaya, 2004] définissent une quinzaine de métriques sur les composants tel l'ACD (*Active Component Density*), définie comme

[21] *Object Constraint Language* [Warmer et Kleppe, 2003]

Critère d'analyse	Lien avec l'IDM
Métrique de processus	Modèle de processus (e.g.; SPEM) ?
Métrique de ressources	Information structurée de gestion de projets ?
Métrique de produits	Utilisation totale de l'IDM ?
Métrique syntactique	Possibilité de monter au niveau modèle ?
Métrique sémantique	Artefact structuré par une métamodèle ?

TAB. 2.2: Grille d'analyse des métriques logicielles

le ratio entre le nombre de composants actifs sur le nombre total de composants. Leur contribution principale est de définir des métriques dynamiques, i.e.; obtenues à l'exécution d'un composant donné. Narasimhan et al. considèrent les composants Corba en particulier.

Mahmood et al. [Mahmood et Lai, 2005] se placent dans le cadre des composants UML pour définir une métrique de complexité inspirée des points de fonctions. Bien que Mahmood et al. se situent dans le cadre d'UML, aucune référence explicite au métamodèle UML n'est faite.

Conclusion

Nous avons présenté dans cette section quelques exemples de métriques dans différents contextes. Nous avons voulu montrer qu'à chaque nouvelle technique, qu'à chaque nouvelle abstraction proposée, la mesure est nécessaire pour quantifier les analyses et augmenter la qualité. Nous pouvons donc penser que les modèles de domaine de l'IDM auront autant besoin d'être mesurés. Pour illustrer ce point, la table 2.2 propose une grille d'analyse des métriques logicielles. Elle illustre le lien entre les principales caractéristiques des métriques et l'ingénierie dirigée par les modèles.

2.4 Spécification et implémentation des métriques

> "It is clear that we must improve metric definitions if metrics are to be properly validated, and that we need data collection and analysis tools if metrics are to be used in practical software production environments." [22]
> Kitchenham, [Kitchenham et al., 1990]

Nous partageons les vues de Kitchenham, et explorons donc les moyens d'améliorer la définition des métriques, et les moyens d'obtenir des valeurs de mesure. Nous passons donc en revue les différents moyens existants, que nous analyserons de façon synthétique à la toute fin de cet état de l'art. Ainsi, nous verrons qu'il est possible de définir des métriques avec le langage naturel (2.4.1), ou directement avec un langage de programmation généraliste (2.4.2). Ensuite, nous montrerons que l'utilisation d'un métamodèle facilite la définition et la précision des métriques (2.4.4). Nous verrons alors que plusieurs auteurs ont proposé d'utiliser un langage dédié existant pour la définition des métriques (2.4.5), comme par exemple SQL. Une autre approche

[22] Il est clair que nous devons améliorer la définition des métriques afin de les valider correctement, et que nous avons besoin d'outils de mesure et d'analyse afin de les utiliser dans des environnements industriels de production de logiciels.

consiste à définir un langage dédié à la spécification des métriques (2.4.6). Enfin, nous présenterons les travaux visant à définir une approche de la mesure à la fois générique et générative (2.4.7).

2.4.1 Avec le langage naturel

Bien entendu, il est possible de définir une métrique avec le langage naturel. Dans ce cas, la métrique réfère ou inclut la description de l'artefact mesuré. Une grande partie des métriques de la littérature est de ce type. Voici un exemple d'une nouvelle métrique pour Java, proposée en anglais dans [Dufour et al., 2003] :

> "size.appLoad.value: The number of bytecode instructions loaded in application specific classes. Whenever a class is loaded, its size in bytecode instructions is added to a running total. Including the standard libraries in size.load.value again distorts this metric, rendering it insufficiently discriminative. size.appLoad.value is the closest equivalent to the static size of an executable. It is less ambiguous than size.appLoadedClasses.value, which is illustrated by the difference between JAVAC (175 loaded classes, 44664 loaded instructions) and SOOT (531 classes, 45111 loaded instructions). size.appLoad.value is less sensitive to the different programming styles in these benchmarks." [23]

Les métriques écrites en langage naturel sont parfaites pour communiquer. Mais elles ont deux inconvénients majeurs. Elles peuvent être ambiguës, et ne sont pas exécutables. La traduction de la métrique en outil est donc sujet 1) à l'interprétation et 2) à l'introduction de fautes. Ces deux arguments s'appliquent aussi bien à la définition des métriques et à la définition des artefacts mesurés.

2.4.2 Avec un langage de programmation généraliste

Les métriques peuvent aussi être définies directement dans un langage de programmation. Dans cette perspective, l'implémentation *est* la définition. Dans ce cas, le problème majeur est que la définition de la métrique devient vite incompréhensible, totalement noyée dans un océan de préoccupation d'implémentation.

Par exemple, l'outil JNCSS qui donne essentiellement une mesure précise de nombre de lignes de code (LOC) d'un programme Java fait environ 37 KLOC. La spécification de la métrique dans la documentation fait l'objet d'un paragraphe et d'un tableau [24]. La plus grande partie du code source de l'outil consiste donc à effectuer d'autres tâches, comme *parser* un programme Java ou afficher les résultats.

Le tableau 2.3 donne la taille de certains outils de mesure des programmes Java. Il a pour but de donner une idée de l'ordre de grandeur des prix et des tailles des

[23] *size.appLoad.value* : le nombre d'instructions *bytecode* chargées par des classes applicatives. À chaque fois qu'une classe est chargée, sa taille en nombre d'instructions *bytecode* est ajouté au total. L'ajout de la librairie standard dans *size.appLoad.value* bruite cette métrique et la rend trop peu discriminante. *size.appLoad.value* est l'équivalent le plus proche de la taille statique d'un exécutable. Elle est moins ambiguë que *size.appLoadedClasses.value*, ce qui est illustré par la différence entre JAVAC (175 classes chargées, 44664 instructions chargées) et SOOT (531 classes, 45111 instructions chargées). *size.appLoad.value* est moins sensible aux différents styles de programmation dans ces bancs d'essai.

[24] http://www.kclee.de/clemens/java/javancss/

Nom	Prix ($)	Taille	Langage	Coût
CodeReports	2000	?	?	
Ndepend	415	?	?	
JHawk	60	?	?	
Understand for Java	495	?	Perl	19 P/M
Resource Standard Metrics	200	?	?	
Imagix 4D	2000	?	?	
Semantic Designs: Java Source Code Metrics	250	?	?	
Metrics	Free	13.51 KLOC (JNCSS)	Java	35 P/M
SLOCCount	Free	3.5 KLOC	Perl/C	8 P/M
CodeCount	Free	3 KLOC	C	8 P/M
LOCC	Free	24.45 KLOC (JNCSS)	Java	67 P/M
JavaNCSS	Free	37.49 KLOC (JNCSS)	Java	106 P/M
DependencyFinder	Free	17.41 KLOC (JNCSS)	Java	47 P/M
JDepend	Free	1.8 KLOC (NCSS)	Java	4.5 P/M
CKJM	Free	0.33 KLOC (JNCSS)	Java	1 P/M
Cyvis	Free	2.5 KLOC (JNCSS)	Java	6.3 P/M

TAB. 2.3: Nom, prix, taille et coût estimé de certains outils de mesure des programmes Java

logiciels. Quand le code source est disponible, les coûts présentés sont calculés avec la formule Cocomo basique ($C = 2.4 * KLOC^{1.05}$). Ce tableau sera utilisé comme argument dans la suite de cette thèse.

2.4.3 En utilisant l'introspection ou un MOP

L'implémentation d'un outil de mesure avec un langage de programmation généraliste peut être considérablement allégée en utilisant des fonctionnalités méta du langage utilisé. Ce faisant, il n'y plus lieu d'avoir une phase de *parsing* et de reconstruction sémantique, ce qui simplifie considérablement l'implémentation de l'outil. Par exemple, Lalonde et Pugh utilisent l'introspection dans *Smalltalk* pour collecter des métriques [LaLonde et Pugh, 1994]. De manière similaire, l'utilisation d'un MOP (*Metobject Protocol*) [Kiczales et al., 1991] permet d'obtenir des valeurs de métriques dynamiques (e.g. [Yu-ying et al., 2007]).

2.4.4 En utilisant un métamodèle

Dans cette section, nous présentons les travaux de la littérature qui appuient la définition de métriques au dessus d'un métamodèle explicite, sans introduire une approche particulière quant à l'implémentation. Les travaux qui utilisent à la fois un métamodèle et une spécification exécutable sont présentés dans les sections suivantes.

En 1997, Misic et al. [Misic et Moser, 1997] décrivent un métamodèle générique pour les logiciels orientés objet en utilisant la notation Z. La description du métamodèle est faite de manière textuelle en Z, et de manière graphique avec des diagrammes E-R. Ils définissent ensuite deux métriques de manière textuelle et mathématique (point de fonctions et *system meter*) en fonction des concepts du métamodèle.

La même année, Abounader et al. [Abounader et Lamb, 1997] publient un rapport technique dans lequel ils proposent un métamodèle. L'approche est novatrice: ils listent les principales métriques OO de la littérature, puis font l'union des concepts nécessaires à leur spécification. Ils produisent donc à la fois un métamodèle et un tableau de dépendances entre métriques et concepts du métamodèle. Abounader et al. n'abordent toutefois pas la question de la spécification et de l'implémentation des métriques.

Tichelaar et al. [Tichelaar et Demeyer, 1998] définissent aussi un métamodèle OO générique dans le but de spécifier une interface d'échange entre outils de rétro-ingénierie. Le métamodèle comprend une douzaine de classes. Tichelaar et al. explicitent dès le résumé le lien avec les activités de mesure, considérées comme partie intégrante de la rétro-ingénierie. Notons que ce métamodèle, nommé ultérieurement FAMIX, est à la base de l'outil de rétro-ingénierie *Moose*, lequel comprend un grand nombre de métriques [Lanza et Ducasse, 2002]. Les mesures, comme le métamodèle, sont exprimées en *Smalltalk*.

Reissing [Reissing, 2001] a une approche similaire à Misic. Son but est de formaliser la spécification des métriques. Il définit un métamodèle OO inspiré du métamodèle UML. Ensuite, il spécifie des métriques de la littérature au dessus de ce métamodèle, avec un mélange de langage naturel et de formules mathématiques. La formalisation ne se situe donc qu'au niveau des concepts du langage, et non au niveau des concepts de la mesure. Lanza et Ducasse [Lanza et Ducasse, 2002, Mens et Lanza, 2002] définissent un métamodèle des logiciels orientés objet. Cette approche par graphe met donc l'accent sur les relations entre les éléments du métamodèle ce qui est illustré par la figure 2.3. À partir de ce métamodèle, Lanza et al. définissent trois métriques génériques: *NodeCount* compte les noeuds satisfaisant un prédicat, *EdgeCount* compte les arcs satisfaisant deux prédicats sur la source et la cible, *PathLength* calcule la longueur d'une chaîne d'arcs. Lanza et al. définissent trois métriques dérivées *Ratio*, *Sum* et *Average*. Leur approche est dédiée aux métriques du logiciel OO, l'implémentation de ces métriques génériques est faite en *Smalltalk* dans l'outil *CodeCrawler*. L'aspect fondamental de ces travaux réside dans la généricité. On peut considérer que les métriques génériques de Lanza et al. sont un proto-métamodèle de métriques.

2.4.5 En détournant un langage dédié

Plusieurs auteurs ont proposé de détourner un langage dédié afin de mesurer les logiciels. Les principales propositions sont d'utiliser : SQL, conçu pour faire des re-

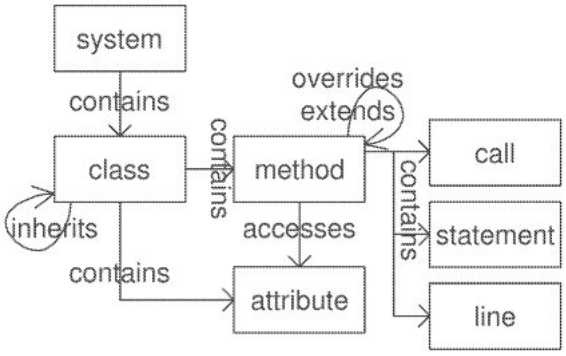

FIG. 2.3 : Métamodèle de logiciel OO de [Lanza et Ducasse, 2002]

Métrique de C&K	LOC
WMC	62
DIT	76
NOC	21
CBO	59
RFC	96
LCOM	82

TAB. 2.4: Taille approximative de l'implémentation des métriques de Chidamber et al. par [Harmer et Wilkie, 2002]

quêtes sur une base de données; *XQuery*, conçu pour faire des requêtes sur des fichiers XML, et OCL conçu pour exprimer des contraintes sur des modèles UML. D'autres approches sont plus *exotiques*, i.e.; proposent des langages à petite diffusion.

Avec SQL (*Structured Query Language*)

Le langage *Structured Query Language* (SQL) a été proposé pour spécifier des métriques. L'idée est de métamodéliser le logiciel à l'aide d'un schéma de base de données, puis de transformer le logiciel en données dans la base, et d'exprimer les métriques comme des requêtes SQL.

Suivant cette approche, Harmer et al. [Harmer et Wilkie, 2002] commencent par définir un métamodèle de langage sous forme de schéma ER. Ce schéma comprend 5 entités (*File, Method, Data, Class, Package*) et 22 relations. Ils modifient le compilateur GNU pour alimenter la base de données à partir de code C++ et Java. Ils expriment alors les métriques de Chidamber [Chidamber et Kemerer, 1991] sous forme de requêtes SQL, et d'un mélange de SQL et de code C pour certaines. Ils soulèvent la question du volume de code nécessaire à l'implémentation des métriques. À ce titre, nous présentons leur valeurs (tableau 2.4) en guise d'ordre de grandeur. L'approche de Harmer est pragmatique et n'hésite pas à mixer deux langages dans le but d'obtenir à faible coût les valeurs de mesure voulues.

Lavazza et al. [Lavazza et Agostini, 2005] suivent la même approche mais pour mesurer des modèles UML. Par rapport aux travaux de Harmer et al., ils sont ainsi

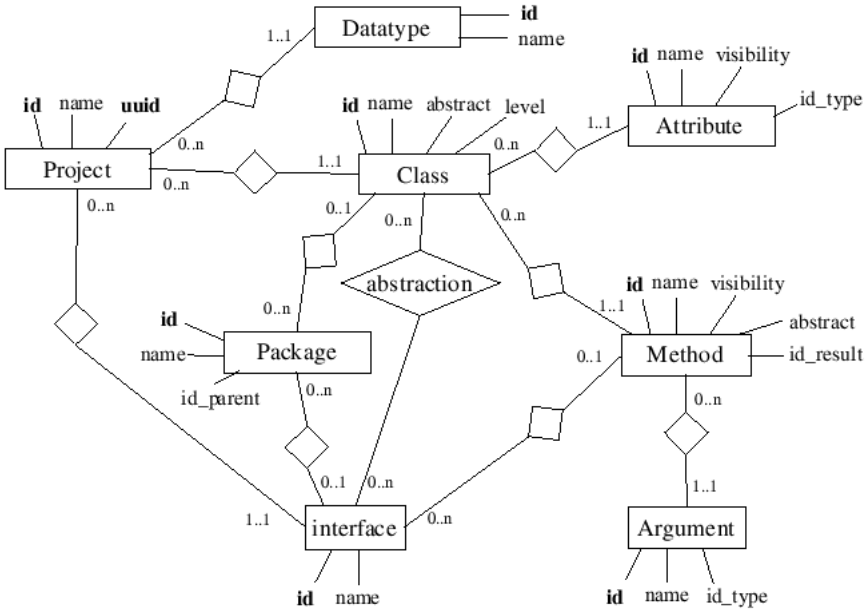

FIG. 2.4 : Un extrait du schéma ER pour UML de [Lavazza et Agostini, 2005]

capables d'obtenir des valeurs de mesure plus tôt dans le cycle de vie. Afin de prendre en compte des modèles UML, ils définissent un schéma E-R qui reprend les principaux concepts du métamodèle UML. Celui-ci est présenté figure 2.4. Un outil écrit en Java lit les modèles UML au format XMI et remplit la base de données. Les métriques sont toutes écrites en SQL et sont effectuées via un outil de requête standard (*MySQL Control Center*). Comme toutes les métriques sont en SQL, elles sont forcément de la forme *nombre d'éléments satisfaisant une requête*. Ce point est illustré dans les exemples de métriques ci-dessous:

Listing 2.2 : Exemple de métrique exprimée en SQL

```
# Number of classes
SELECT count(*)
FROM Class
WHERE id_project=@prj

# Nombre de classes qui n'appartiennent pas
# à une hierarchie de classe
SELECT count(*)
FROM Class
WHERE id_project=@prj
  AND level=0
  AND id NOT IN
    (SELECT parent
     FROM generalization
     WHERE id_project=@prj)
```

Avec OCL (*Object Constraint Language*)

Baroni et al. [Baroni et al., 2002, Baroni et Abreu, 2002] proposent d'utiliser le langage *Object Constraint Language* (OCL) pour spécifier des métriques de design orienté-objet. S'appuyant sur la littérature, ils font le constat que, d'une part, certaines métriques sont ambiguës, en particulier quand elles sont définies avec le langage naturel, et que, d'autre part, des formalisations trop formelles (i.e.; mathématiques) sont difficiles à implémenter. Dans ce dernier cas, l'erreur peut donc venir de la formule elle même ou de la traduction de la formule en code exécutable.

L'approche de Baroni et al. est d'abord instanciée sur le métamodèle GOODLY [Baroni et al., 2002], puis sur le métamodèle UML [Baroni et Abreu, 2002]. Baroni et al. énoncent que les métriques exprimées en OCL sont précises, sans ambiguïté, tout en restant facilement compréhensibles. Un exemple est donné ci-dessous. Comme le montre l'exemple, certaines expressions OCL sont le corps d'une méthode qui renvoie la valeur d'une métrique, d'autres sont des fonctions de type *helper* qui sont appelées par des méthodes de type métriques.

Listing 2.3 : Exemple de métriques exprimées en OCL

```
1  // Total number of Classes in the Package.
2  Package:: TC(): Integer
3  post: result = allClasses()->size()
4
5  // Set of all Classes belonging to the current Package.
6  Package:: allClasses(): Set(Class)
7  post: result = self.contents()->
8      iterate (elem: ModelElement;
9               acc: Set (Class) = oclEmpty(Set(Class))
10      | elem.oclIsTypeOf(Class) implies
11 acc->union (acc->including(elem.oclAsType (Class))))
```

Suivant la même approche, McQuillan et al. [McQuillan et Power, 2006b] expriment des métriques en OCL au dessus du *Dagstuhl Middle Metamodel* (DMM) [Lethbridge et al., 2004] dans le but de mesurer des programmes java. Pour ce faire l'architecture suivante est mise en place: compilation des sources Java en *bytecode*, transformation du *bytecode* en instance du DMM, compilation des expressions OCL en Java, mesure. On retrouve dans Baroni et al. les deux familles d'expressions OCL: métrique et *helpers*. McQuillan et al. ont poursuivi dans cette voie dans [McQuillan et Power, 2006a], en remplaçant le métamodèle DMM par le métamodèle UML. Notons qu'ils fournissent le code OCL complet des métriques. Il est donc possible de comparer finement leur approche avec une autre.

XQuery

On trouve dans la littérature deux contributions qui proposent d'exprimer des métriques en utilisant le langage *XQuery* [Boag et al., 2007, Eichberg et al., 2006]. *XQuery* est un langage fonctionnel et déclaratif ayant pour but de manipuler des documents XML. EL Wakil et al. [El Wakil et al., 2005] justifient ce choix par les arguments suivants: *XQuery* est précis grâce à un modèle de données standardisé et une syntaxe formelle; son pouvoir d'expression permet de spécifier des métriques

complexes; il est exécutable (interprété ou compilé); il est standardisé par le W3C; il a un large spectre et peut s'appliquer à tous les fichiers XML.

EL Wakil et al. [El Wakil et al., 2005] proposent d'utiliser *XQuery* sur des fichiers XMI. Ils sont ainsi à même de mesurer des modèles UML issus de différents outils, et donc, c'est là leur but, d'obtenir des valeurs de métriques de design orienté-objet. EL Wakil et al. annoncent avoir spécifié 65 métriques en *XQuery*.

L'approche d'Eichberg et al. [Eichberg et al., 2006] s'appelle *QScope*. Elle est construite au dessus du *framework Magellan* qui a pour but de raisonner sur tous les artefacts d'un logiciel (fichiers sources, fichiers de déploiement eg.; XML pour *JavaBeans*, etc.). Ainsi l'approche *QScope* permet d'exprimer des métriques sur l'ensemble des artefacts, avec le même langage, *XQuery*. Eichberg et al. ont implémenté 18 métriques dont la liste est donnée dans l'article. En guise d'exemple, la métrique LCOM est donnée et nous la reproduisons dans le listing 2.4

Listing 2.4 : La métrique LCOM en XQuery

```
1  for $A in $db:prj-files/bat:class[@name=$param]
2  let $methods := java:all-methods($A)
3  let $n := fn:count($methods)
4  let $fields := java:all-declared-fields($A)
5  let $k := fn:count($fields)
6  return
7    if ($k = 0) then 0
8    else if ($n = 0) then 100
9    else (
10     (1-
11       fn:sum(
12         for $v in $fields
13         return fn:count(
14             java:methods-accessing-field($v, $methods)
15           ) div $n
16       ) div $k
17     ) * 100
18   )
```

Pour notre propos, l'approche *XQuery* soulève deux points importants. EL Wakil et al. soulignent que leur approche, étant basée sur XMI, peut s'appliquer à n'importe quel modèle, outre UML, basé sur MOF. Le point de vue d'Eichberg et al. est de mesurer tous les artefacts d'un logiciel, et non plus seulement le code source. En cela, ces deux contributions élargissent l'application des métriques tout en gardant un langage de spécification commun.

Avec des langages exotiques

Le langage de script intégré à Rational Rose Kim et Boldyreff [Kim et Boldyreff, 2002] proposent d'utiliser le langage de script intégré à l'outil de modélisation UML Rational Rose. Ils définissent et implémentent un ensemble de 27 métriques sur les modèles UML qui se répartissent en 5 familles: les métriques sur le modèle dans sa globalité (e.g.; le nombre de paquetages), les métriques sur les classes (e.g.; le nombre d'attributs par classe); les métriques sur les messages et les métriques sur les cas d'utilisation (e.g.; le nombre d'acteurs associés à un cas d'utilisation). Les

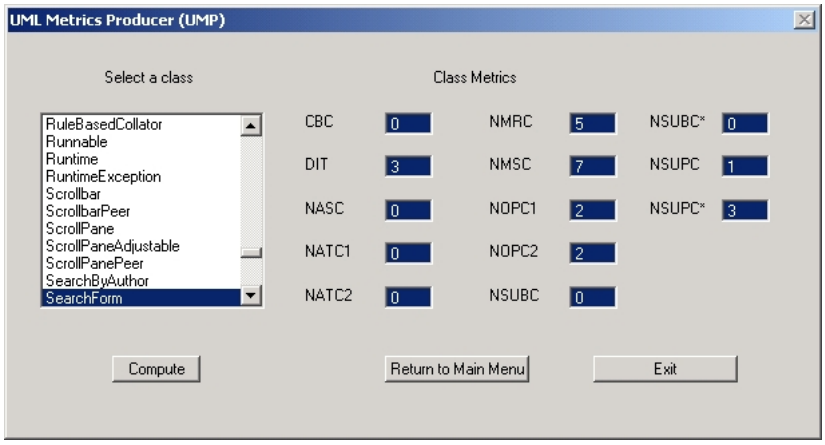

FIG. 2.5 : Une vue de l'interface graphique de UMP [Kim et Boldyreff, 2002]

auteurs annoncent un outil écrit en 1152 lignes de code. L'outil intègre une interface graphique, dont une capture est présentée figure 2.5. Les auteurs ne précisent pas si cette interface graphique est aussi écrite en *BasicScript*.

L'approche de Kim et al. est comparable à [Lavazza et Agostini, 2005] dans la mesure où elle vise à mesurer des modèles UML. De même, on retrouve le leitmotiv de l'approche Crocodile [Lewerentz et Simon, 1998] pour ce qui est de l'intégration avec l'outil d'ingénierie (ici un environnement de modélisation UML). Notons que l'utilisation de *BasicScript*, et donc la réutilisation direct d'une partie du code de Rational Rose, permet de ne pas se soucier de la lecture des modèles et de l'accès à leur sémantique. En effet, les primitives de *BasicScript* permettent de naviguer sémantiquement dans les modèles i.e.; suivant les concepts du métamodèle UML.

MEL (*Method Engineering Language*) Saeki [Saeki, 2003] propose un langage de description de processus, le *Method Engineering Language* (MEL, [Brinkkemper et al., 2001]) pour décrire des métriques pour les systèmes d'information. Saeki commence par argumenter de l'importance de la mesure dans les processus de développement. Son but est donc d'enrichir les méthodes de développement par des activités de mesure. L'intuition est d'utiliser le même langage, MEL, pour décrire à la fois les produits du processus, les étapes du processus et les métriques sur les produits.

Pour exemple, nous présentons dans le listing 2.5 son extrait de métamodèle de cas d'utilisation décrit en MEL. L'équivalent graphique, aussi présenté dans le papier considéré, est présenté sur la figure 2.6.

Saeki définit à partir du métamodèle de cas d'utilisation 4 métriques fondamentales, et une métrique dérivée. Celles-ci sont présentées dans la table 2.5. La spécification de l'une d'elle dans le langage MEL, NO_EXTENDS, est présentée dans le listing 2.6.

Vis-à-vis de la problématique de cette thèse, l'approche de Saeki souligne l'importance de la mesure dans la spécification des processus de développement logiciel et la possibilité de définir des métriques en fonction des concepts d'un métamodèle (l'approche de Saeki se sert explicitement d'un métamodèle). Notons aussi que Saeki propose un couplage fort entre le métamodèle et les mesures, car les valeurs de me-

Nom	Explication
NO_EXTENDS	Nombre de relations *extends* entre cas d'utilisation
NO_USES	Nombre de relations *uses* entre cas d'utilisation
NO_CD	Nombre de relations *Control Dependency* entre cas d'utilisation
NO_DD	Nombre de relations *Date Dependency* entre cas d'utilisation
NUCT	Nombre de types de cas d'utilisation
Modifiability	$w1 * NO\_EXTENDS + w2 * NO\_USES + w3 * NO\_CD + w4 * NUCT$

TAB. 2.5: Métriques de cas d'utilisation dans [Saeki, 2003]

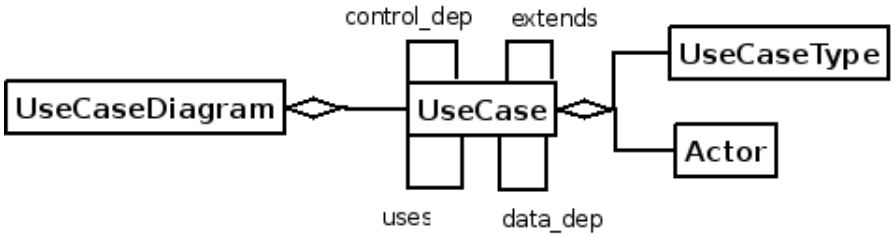

FIG. 2.6 : Métamodèle de cas d'utilisation de [Saeki, 2003]

sures sont considérées comme des attributs des classes du métamodèle. Enfin, le fait d'utiliser le même langage pour définir le processus et les métriques est un avantage. Il permet d'éviter la phase d'apprentissage d'un nouveau langage pour spécifier des métriques.

Listing 2.5 : Extrait d'un métamodèle de cas d'utilisation décrit en MEL

```
1  PRODUCT Use Case Model;
2   ID Use Case Model;
3   IS_A Diagram, Structured Text;
4   LAYER Diagram;
5   PART OF Analysis Model;
6   NAME TEXT;
7  PRODUCT Use Case:
8   LAYER Concept;
9   PART OF Use Case Model;
10  SYMBOL Oval;
11  NAME TEXT;
12  ASSOCIATED W ITH {(uses, ), (extends,),
13    (hasUseCase,), (communicates,)}.
14 ASSOCIATION hasUseCase:
15  ASSOCIATES (Use Case Diagram, Use Case);
16  CARDINALITY (1..n; 1..1).
```

Listing 2.6 : La métrique NO_EXTENDS spécifiée en MEL

```
1  PRODUCT NO_EXTENDS;
2   LAYER Concept;
3   PART OF Metrics Model;
4   NAME TEXT;
5   ATTRIBUTES
6       value : REAL
```

```
ASSOCIATED WITH {(has_No_extends, ),
          (refer_to_No_extends, )}.
(RULE :
 value
  = (AllDependencies - #Instances(extends)) / AllDependencies'
 AllDependecies
= (#Instances(Use Case) * (#Instances(Use Case)-1)/2;
).
```

Logique de description Une approche originale de la mesure a été proposée par Kastenberg [Kastenberg, 2004]. Son approche consiste à transformer un programme en un graphe de classes (au sens logique du terme) et les métriques logicielles en des formules de logique de description [Baader et al., 2003]. Les graphes de classes sont un type de modèle introduit par Kastenberg, qui consiste à ajouter des informations à un modèle de graphe classique. La logique de description est un formalisme de représentation des connaissances de la même famille que l'approche programmation logique (*Prolog*).

Le but de Kastenberg est, dans ses termes, d'avoir une description formelle des métriques, sans avoir à faire de calculs mathématiques. En des mots plus orientés logiciels, nous le traduisons par avoir une description déclarative et exécutable des métriques.

L'outillage mis en oeuvre dans cet approche est conséquent, de la transformation d'un programme en un graphe de classes, à la définition d'une grammaire pour la logique de description, en passant par l'exécution dans l'outil académique de transformation de graphes *Groove*. Finalement, Kastenberg exprime les trois métriques suivantes: fan-in/fan-out; manque de cohésion (LCOM); complexité cyclomatique. Un exemple de spécification de métriques en logique de description est donnée dans le listing 2.7.

Listing 2.7 : Métriques spécifiées en logique de description

```
# l'ensemble des procedures ayant un fan-in supérieur à 2
(countFirst(call : calls), mapToConstant(Procs, 2))

# formule finale de la métrique LCOM
classdef2NrOfProcsAcc =
      (((((CLASS_DEF COMPOSE
        first_child) COMPOSE
            REF_TR_CLOS next sibling ) COMPOSE
                OBJBLOCK) COMPOSE
                    first_child) COMPOSE
                        REF_TR_CLOS next_sibling )
                            VARIABLE_DEF)
                                InstVars2NrOfProcsAcc)
averageAccInstVars = average(CLASS DEF  Procs2NrOfNotAccInstVars)
```

Dans cet état de l'art, cette approche illustre bien les trois pieds de la mesure, du point de vue conceptuel aussi bien que du point de vue des outils: l'expression des modèles, l'expression des métriques et l'expression de la sémantique d'exécution des métriques.

Avec un langage de transformation de modèles Vépa et al. [Vépa et al., 2006] proposent d'utiliser le langage de transformation de modèles ATL [Jouault et Kurtev, 2005] pour la mesure des métamodèles. La problématique du papier est la mesure automatique d'un entrepôt de métamodèles. Vépa et al. montrent aussi que la présentation des résultats de mesure peut être implémentée comme une transformation de modèles, e.g.; une transformation des valeurs de mesure vers HTML. Vépa et al. ont implémenté 8 métriques en ATL mais ne donnent pas le code correspondant. Cette fonctionnalité a été déployée en service web[25] et donne maintenant, d'après la documentation[26], 35 métriques sur les métamodèles KM3 [Jouault et Bézivin, 2006]. L'approche de Vépa et al. donne une vision nouvelle de la spécification des métriques. D'une part c'est une approche explicitement et clairement orientée modèle, d'autre part elle peut potentiellement s'appliquer à n'importe quel modèle. Notons toutefois que pour un métamodèle donné, le code est écrit de manière ad hoc. De manière similaire, Mora et al. [Mora et al., 2008] proposent d'utiliser QVT pour mesurer des logiciels.

2.4.6 En définissant un langage dédié à la mesure

Plusieurs auteurs ont proposé de définir des métriques avec un langage dédié à ce problème. Par langage dédié, nous entendons la définition des concepts, d'une syntaxe (textuelle ou graphique), et la définition de l'exécutabilité associée.

L'approche *GraphLog*

Mendelzon et al. [Mendelzon et Sametinger, 1995] proposent un outil de visualisation et de requête de code nommé Hy+. Cet outil est utilisé pour mesurer le code, vérifier des contraintes de design, d'implémentation ou de style, et détecter la présence de patron de conception. Ces trois activités sont faites à partir de requêtes spécifiées avec le langage graphique *GraphLog* [Consens et Mendelzon, 1990].

Le langage *GraphLog* est basé sur une représentation par graphe aussi bien des données que des requêtes. La spécification avec une syntaxe graphique est donc directe et naturelle. Avec *GraphLog*, les métriques exprimables sont toutes de la forme *nombre de résultats d'une requête*.

La métrique donnée en exemple dans l'article est une métrique sur du code C++ : le nombre de méthodes étant surchargées dans plus de 20 sous-classes. La spécification graphique en *GraphLog* de cette métrique est montrée figure 2.7. Les auteurs annoncent que l'on pourrait aussi définir en *GraphLog* les métriques: nombre de méthodes par classe, profondeur d'héritage, nombre de sous-classes, couplage, RFC, nombre de lignes de code, nombre de variables d'instance, nombre de classes fortement couplées. Toutefois, les spécifications correspondantes ne sont pas données.

L'approche de Mendelzon et al. montre d'une part la possibilité d'utiliser un langage de requête pour spécifier des métriques du logiciel, et d'autre part une première approche de spécification des métriques du logiciel graphiquement.

[25] http://www.mdworkbench.com/measuring.php
[26] http://www.mdworkbench.com/measurement/metrics/definition.htm

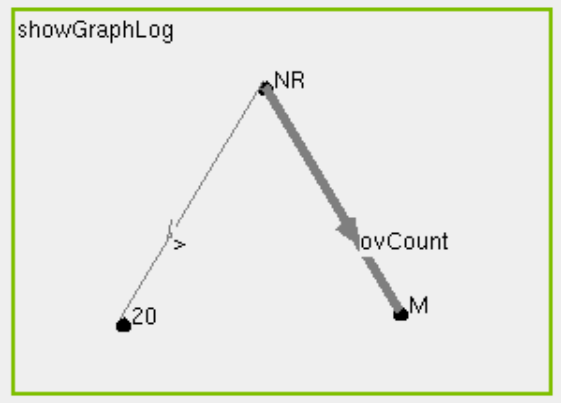

FIG. 2.7 : Exemple de métrique spécifiée en *GraphLog*

Outils commerciaux proposant un langage dédié

SDMetrics SDMetrics [SDMetrics, 2006] est un outil commercial de mesure des modèles UML. Il est indépendant des modeleurs et se base sur le format XMI pour la lecture des modèles. La plupart des métriques par défaut de l'outil sont implémentées en Java. Nous le présentons dans cet état de l'art, car il contient une fonctionnalité qui s'apparente à la définition d'un langage dédié pour les métriques. Nous nous basons entièrement sur la documentation de l'outil disponible sur le site web officiel www.sdmetrics.com. Notons que le concepteur de SDMetrics, Jürgen Wüst, a un fort bagage académique en terme de mesure et de qualité du logiciel.

Wüst fait le constat que l'utilisation d'UML varie grandement suivant les entreprises. L'outil SDMetrics doit donc pouvoir s'adapter aux pratiques d'une entreprise donnée, afin d'avoir des valeurs de mesure qui sont adaptées au processus et à la maturité des entreprises en terme de modélisation UML. Cela s'exprime par une fonctionnalité de spécification de métrique.

Dans SDMetrics, une spécification d'un ensemble de métriques sur UML se traduit par un fichier XML, que l'outil charge à la demande. Notons que certaines métriques fournies par défaut dans l'outil sont exprimées dans ce fichier XML. On peut donc considérer ce fichier XML comme un langage dédié à la spécification des métriques. Dans cette perspective la DTD correspondante est un métamodèle de spécification de métriques.

Le langage réifie le concept de métrique avec une balise <*metric*>. Un objet métrique comprend un certain nombre d'attributs et d'éléments contenus. L'élément contenu <projection> est la spécification de la métrique. On retrouve aussi un élément <set>, équivalent des méthodes OCL de type *helper* dans les approches basées sur OCL présentées précédemment. Un exemple est donné dans le listing 2.8.

Listing 2.8 : Exemple de métrique spécifiée avec l'outil *SDMetrics*

```
1 <!--a single metric -->
2 <metric name="NumPubOps" domain="class" category="Size">
3  <description>
4   The number of public operations in a class.
5  </description>
```

Famille	Exemple
Requêtes	SELECT METHODS
Conditions sur le nom	*SELECT FIELDS WHERE NameLike "m_"*
Conditions sur ces métriques	*SELECT METHODS WHERE NbLinesOfCode > 20*
Conditions sur la structure	*SELECT TYPES WHERE IsUsedBy "System.Net"*
Conditions sur les éléments	*SELECT METHODS WHERE IsPropertySetter*
Conditions sur le cycle de vie	*SELECT METHODS WHERE BecameObsolete*
Conditions sur l'encapsulation	*SELECT METHODS WHERE CouldBeInternalProtected*

TAB. 2.6: Exemple de primitives CQL

```
6   <projection relset="ownedoperations"
7    condition="visibility='public'"/>
8  </metric>
9
10 <!-- intermediate helper set -->
11 <set name="AssocOut" domain="interface">
12  <description> The set of associations with navigability
13   away from the interface.
14  </description>
15  <projection relset="ownedattributes" condition="association!=''"
16   element="association" />
17 </set>
```

CQL Le langage CQL (*Code Query Language*) [Smacchia S.A., 2006] fait partie de l'outil commercial *NDepend*[27]. *NDepend* est un outil d'analyse pour les applications .NET. Le langage CQL permet d'écrire des requêtes sur une application .NET quel que soit le langage utilisé (C++, C#, VB). L'aspect intéressant de CQL est qu'il s'agit d'un langage dédié à l'analyse des logiciels, et donc en partie à la mesure.

CQL est similaire à SQL. Il fournit une architecture de requête en SQL xxx FROM xxx WHERE xxx ORDER BY xxx. Les primitives du langage sont divisées en deux familles: les requêtes (10 éléments) et les conditions. Les conditions sont au nombre de 127 et sont aussi divisées en sous-familles: les conditions sur le nom, les conditions sur des valeurs de métriques, les conditions sur la structure du code, les conditions sur les éléments du code, les conditions liées au cycle de vie, les conditions liées à l'encapsulation. Des exemples de ces conditions sont présentées dans la table 2.6. On remarque que la plupart des concepts du langage sont très spécifiques. Notons que les métriques associées sont le nombre d'éléments satisfaisant une requête.

Dans la perspective de cette thèse, CQL est un exemple de DSL proche de la spécification des métriques. Il comprend les concepts, en particulier des types de condition, nécessaires à la spécification de métriques.

L'approche *SAIL*

Marinescu et al. [Marinescu et al., 2005] ont proposé en 2005 une nouvelle approche pour l'implémentation des métriques. C'est un DSL dédié aux métriques de

[27] http://www.ndepend.com

	Java	SQL	SAIL
LOC	1650	1073	1461
Taille (octets)	37580	48057	31102

TAB. 2.7: Comparatif entre approches de [Marinescu et al., 2005]

design avec un accent particulier sur les métriques de design orientées-objet. Ce langage est nommé *SAIL*. L'argumentaire pour justifier ce DSL de métriques commence par montrer l'obligation d'automatiser la mesure pour mesurer des programmes de grande taille, puis met en lumière la complexité des programmes de métriques eux-mêmes. D'où sont alors déduits les problèmes habituels du logiciel: difficulté à tester; à comprendre; à réutiliser et à maintenir.

La définition du langage est fondée sur l'expérience des auteurs dans le domaine de la définition et de l'implémentation de métriques. Les principes du langage sont au nombre de six: la navigation dans le modèle du programme; la sélection d'entités; l'arithmétique des ensembles; le filtrage d'entités; le calcul de propriétés et l'agrégation de propriétés.

La validation de l'approche se fait de deux manières. D'une part en comparant ces principes dans SAIL, Java, et SQL. D'autre part, en montrant que pour un ensemble de métriques, la taille de l'implémentation est plus courte en SAIL. Nous présentons ces résultats dans le tableau 2.7. Les auteurs annoncent avoir implémenté 40 métriques en SAIL, mais n'en donnent ni la liste, ni leur implémentation. Nous montrons donc seulement les extraits de code SAIL qu'ils présentent, afin d'avoir une aperçu de la syntaxe et du maniement du langage (listing 2.9).

Listing 2.9 : Exemple de code SAIL

```
1  // filtrage
2  Package myPack;
3  myPack = select (*) from sysPackages
4           where name="my.package";
5
6  // navigation
7  Method[] methods;
8  methods = select (methods) from myPack.classes;
9
10 // sélection
11  struct DetailedMethod{
12  string name;
13  string sign; };
14  DetailedMethod[] detailed;
15  detailed = select (name, sign) from methods;
16
17 // exemple d'opération sur les ensembles
18 Method[] globals = myPack.globalFunctions;
19 methods += globals;
```

Pour conclure, notons que l'approche SAIL est explicitement un DSL dédié aux métriques de design des logiciels orientés objet.

RML (*Relation Manipulation Language*)

Beyer et al. [Beyer et al., 2005] spécifient un nouveau langage *Relation Manipulation Language* (RML) dans le but de détecter des patrons de conception, des défauts de design, et de calculer des métriques. En cela, l'approche est tout à fait similaire à l'approche *GraphLog* présentée précédemment. Notons aussi que l'un des auteurs, Lewerentz est aussi l'auteur de l'approche Crocodile. Toutefois, la contribution de [Beyer et al., 2005] est d'insister sur les problèmes de performance dans ces activités de calcul à base de graphes. Ils montrent ainsi que l'approche RML est significativement plus rapide en temps d'exécution que les autres approches pour un ensemble de tâches.

Le langage RML est basé sur des expressions relationnelles, c'est-à-dire des expressions manipulant des ensembles de tuple. RML contient aussi des expressions numériques. Enfin, RML comprend des opérateurs d'affectation, de contrôles (e.g.; structures conditionnelles, boucles) et des opérateurs d'entrée/sortie. Beyer et al. montrent une seule métrique écrite en RML: la métrique d'instabilité de Martin [Martin, 2000]. Celle-ci est donnée dans le listing 2.10. Cet exemple illustre l'aspect impératif de RML: boucle FOR, nombreuses affectations, expressions logiques.

Listing 2.10 : La métrique d'instabilité de Martin en RML

```
1
2  Use(x,y) := Call(x,y) | Contain(x,y) | Inherit(x,y);
3  For p IN Package(x) {
4    CaClass(x) := !PackageOf(p,x)
5                  & EX(y, Use(x,y) & PackageOf(p,y));
6    Ca := #(CaClass(x));
7    CeClass(x) := PackageOf(p,x)
8                  & EX(y, Use(x,y) & !PackageOf(p,y));
9    Ce := #(CeClass(x));
10 PRINT p, " ", Ce / (Ca+Ce), ENDL;
11 }
```

Dans notre perspective, Beyer et al. soulignent à juste titre l'importance de l'efficacité d'un langage dédié. En effet, la taille des logiciels à mesurer est importante, et il n'est pas possible, comme les auteurs le montrent, d'attendre plus de cinq heures pour obtenir un résultat de métrique écrite en SQL sur un logiciel de grande taille tel *Eclipse*.

GMEF (*Generic Metric Extraction Framework)*

L'approche d'Alikacem et Sahraoui [Alikacem et Sahraoui, 2006] consiste à définir un métamodèle de logiciel OO et un langage de description de métriques. Ce langage n'étant pas nommé, nous le nommerons GMEF en référence au titre du papier. Le métamodèle d'Alikacem et al. comprend 31 classes, il est très orienté vers C++ et supporte Java.

GMEF contient des primitives pour représenter les ensembles de base issus de la représentation du code e.g.; *classes()*, *methods(c)*, *parents(c)*. GMEF supporte la navigation dans le modèle par des opérateurs faisant référence au métamodèle, e.g.; *c.visibility*. GMEF propose les opérateurs classiques sur la manipulation des nombres

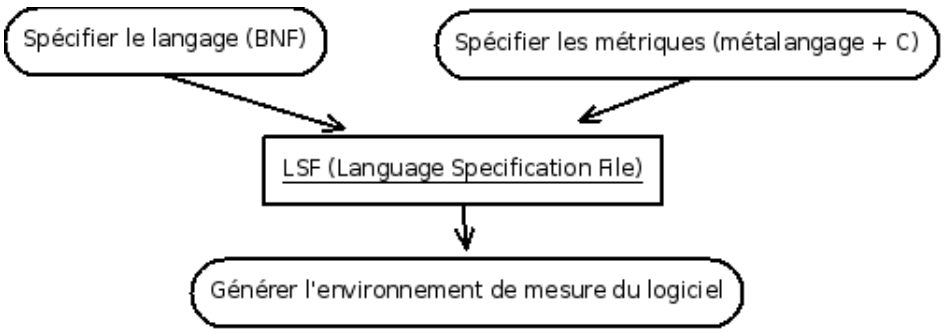

FIG. 2.8 : L'approche *Athena* [Tsalidis et al., 1992]

et des ensembles. Enfin, GMEF contient un opérateur de type quantificateur universel (*forall*).

Dans notre perspective, notons que l'approche GMEF est dédiée aux modèles OO, voire très fortement couplée à C++ et Java au vu du métamodèle. Par contre, la motivation d'une approche générique, comme le titre le met en évidence, est fondamentale pour notre propos. En effet, une approche de la mesure pour l'IDM doit pouvoir s'appliquer sur n'importe quel métamodèle.

2.4.7 Par une approche générique et générative

Dans cette section, nous présentons les travaux qui définissent une approche de la mesure générique et générative. La généricité est la faculté de pouvoir s'appliquer à des artefacts différents. En des termes IDM, cela consiste à être applicable à plusieurs métamodèles. L'aspect génératif réside dans l'utilisation de la spécification des métriques pour générer l'outil de mesure. Ces travaux forment le voisinage conceptuel le plus proche de cette thèse.

L'approche *Athena*

En 1992, Tsalidis et al. [Tsalidis et al., 1992] proposent une nouvelle approche pour la mesure des logiciels. Leur approche consiste à générer automatiquement l'environnement de mesure pour un langage donné. Pour ce faire, ils proposent un métalangage.

L'approche est présentée figure 2.8. Pour obtenir un environnement de mesure, il faut spécifier le langage cible dans une syntaxe proche d'une BNF. Ensuite, il faut spécifier les métriques dans un métalangage dédié aux mesures. En 1992, Tsalidis et al. n'utilisent pas l'expression *domain specific language* mais *design specification language*. Ce langage dédié est un mélange de commandes déclaratives et de code impératif écrit en C. La spécification du langage cible et des métriques est contenue dans un seul fichier, nommé *Language Specification File* (LSF). La génération de l'environnement de mesure se fait en deux phases successives. Premièrement, à partir du fichier LSF, un compilateur génère un fichier utilisable par l'outil YACC. Ensuite l'outil YACC génère le code C compilable par un compilateur C classique.

Le métalangage comprend des primitives à la fois génériques et déclaratives. Elles indiquent quand déclencher une action de mesure. Ces quatre primitives se dé-

clenchent respectivement pour chaque caractère, pour chaque signe, pour chaque symbole de la grammaire ou pour chaque opérateur. L'action de mesure est une fonction C ad hoc. Certaines fonctions sont factorisées dans une librairie.

Le métalangage comprend aussi un sous-langage, nommé *Graph Specification Language* (GSL) dédié aux métriques basées sur les graphes. GSL comprend quatre opérateurs binaires sur les graphes. Ainsi, par un enchaînement impératif de ces opérateurs, on peut construire un graphe représentant un aspect d'un programme. L'exemple donné dans l'article est celui de la construction du graphe de flot de données à partir d'un programme Pascal.

Ci-dessous un exemple de la partie mesure d'un LSF. Il montre la spécification de la mesure du nombre de ligne de code (LOC) et de la mesure de complexité de McCabe. Nous pouvons voir le mélange de déclaratif (e.g.; *%AllCharacters*) et d'impératif écrit en C (fonction *computeLOC*). On voit aussi un exemple de fonctions de la librairie: la fonction McCabe fait partie du noyau, elle est seulement paramétrée par quelques informations supplémentaires (`PROGRAM IF WHILE [ "mccabe++;" ]`).

Listing 2.11 : Exemple de spécification *Athena*

```
1
2  %AllCharacters [ computeLOC ]
3
4  %{
5  int LOC=0;
6  int mccabe=0;
7  }%
8
9  %TokenClass [ McCabe ] PROGRAM IF WHILE [ "mccabe++;" ]
10
11 void computeLOC(char c) {
12   static int empty=1;
13   if(c=='\n'&&!empty){
14     LOC++;
15     empty=1;
16   } lese if (!isspace(c)) empty = 0;
17 }
```

À notre connaissance, l'approche *Athena* est la première à proposer l'idée de générer un environnement de mesure à partir d'une spécification de mesure. Pour Tsalidis et al. le but est d'avoir une approche de la mesure indépendante du langage de programmation, personnalisable et permettant d'avoir des mécanismes communs d'enregistrement et de présentation des résultats de mesure. Notons que l'approche est basée sur la syntaxe textuelle des langages, qu'elle nécessite d'écrire du code C, et qu'enfin que le sous-langage dédié aux métriques basées sur les graphes paraît assez complexe pour pouvoir le donner à un large éventail d'utilisateurs.

Slammer

À notre connaissance, la problématique de la mesure des DSL est un sujet très peu abordé dans la littérature. Seuls Guerra et al. ont fait des publications sur ce sujet [Guerra et al., 2006, Guerra et al., 2007].

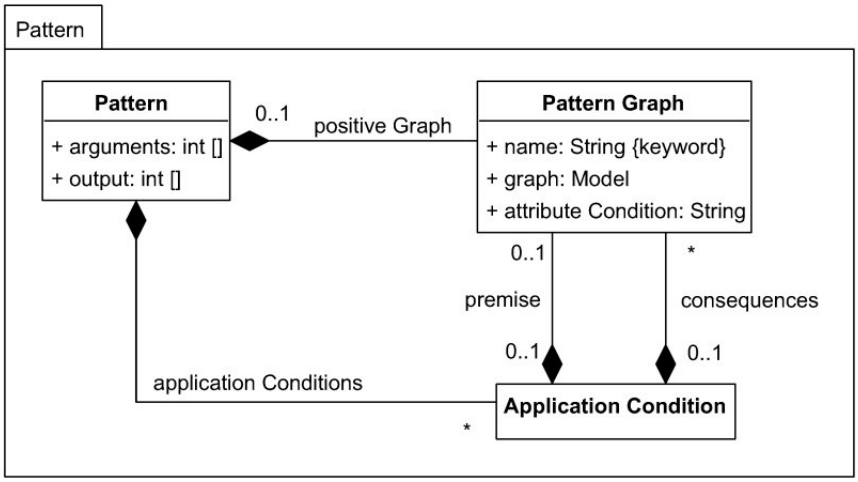

FIG. 2.9 : Le métamodèle de pattern de *Slammer*

L'approche de Guerra et al. est construite sur trois principes: le développement par DSL; la mesure des DSL; la transformation des modèles d'un DSL. L'accent est mis tout particulièrement sur la spécification visuelle des métriques et des transformations (*redesign*), d'où l'utilisation du terme DSVL pour *Domain Specific Visual Languages*. L'approche, nommée *Slammer*, consiste donc à spécifier visuellement des métriques et des transformations. En cela, elle est similaire à l'approche HY+ présentée section 2.4.6.

L'approche *Slammer* est orientée modèle: en conséquence Guerra et al. présentent un métamodèle de métriques, reproduit figure 2.10. Le concept de métrique, central, est nommé *Measurement*[28]. Celui-ci se décompose en 5 classes abstraites représentant des familles de métriques. Les métriques concrètes sont au nombre de neuf (principalement sur la troisième rangée de classes de la figure 2.10). Les attributs et associations du métamodèle sont peu nombreux. Le concept de Pattern, reproduit figure 2.9, est uniquement à base de graphes, donc totalement syntaxique. La syntaxe graphique de ce métamodèle n'est pas décrite dans le papier.

L'approche *Slammer* est la plus proche de la contribution de cette thèse. Nous partageons le même but: clarifier et simplifier la spécification des métriques et automatiser leurs développements. Notons que la partie transformation de *Slammer* est totalement en dehors du domaine de cette thèse.

2.5 Synthèse et conclusion

Nous avons présenté dans cet état de l'art l'ensemble des travaux de la littérature liés à notre thèse. Nous avons posé le contexte, puis précisé les concepts manipulés. Ensuite, nous avons montré que la problématique de la mesure évolue en même temps que les manières de créer des logiciels. Ainsi, la mesure s'adapte aux artefacts créés,

[28] C'est à notre avis un abus de langage. En effet, ce concept ne représente pas une action de mesure mais sa spécification. *Measure* ou *MeasureSpecification* aurait été plus adapté.

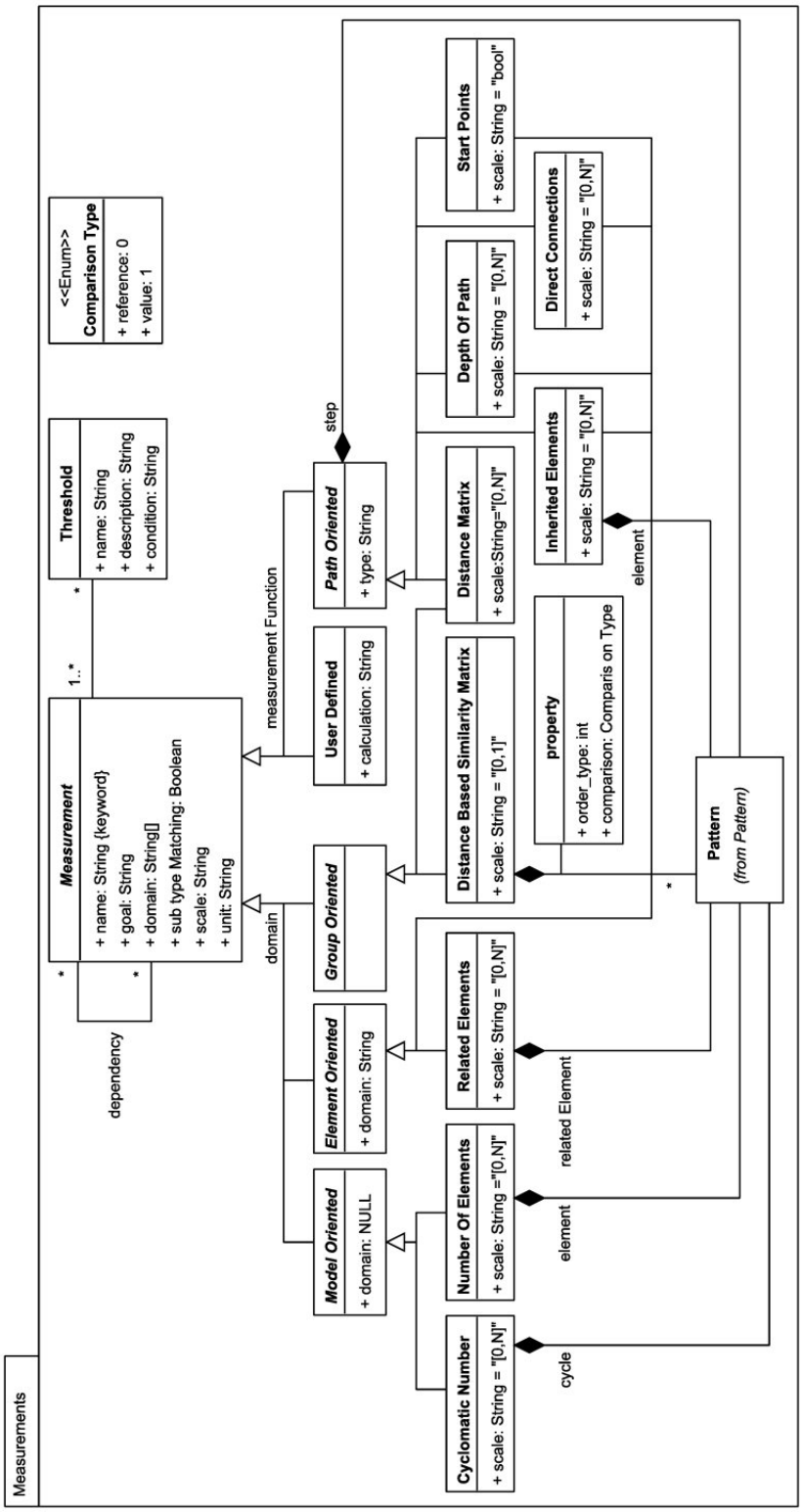

FIG. 2.10 : Le métamodèle de métriques de *Slammer*

du code source aux modèles. Enfin, nous avons étudié en profondeur les travaux liés à l'expression des métriques et aux modalités de la mesure. Nous essayons maintenant de faire une synthèse des différents points de vue. Tout d'abord, plusieurs auteurs insistent sur des points relatifs à l'environnement de mesure.

Il est important que la mesure soit intégrée dans un outil d'analyse du logiciel (e.g; [Mendelzon et Sametinger, 1995]) afin de centraliser les différentes informations issues du logiciel analysé. Certains auteurs (e.g.; [Lewerentz et Simon, 1998]) mettent l'accent sur la mesure comme un outil d'aide au développement et considèrent qu'elle doit être intégrée dans un environnement de développement. Beyer et al. [Beyer et al., 2005] posent le problème de l'efficacité et du passage à l'échelle. L'automatisation de la mesure est présentée en particulier dans [Lavazza et Agostini, 2005]. Elle permet la constitution fiable et automatique de bases de connaissance à des fins d'analyse et d'amélioration du processus et des pratiques de développement. La génération de l'outil de mesure a pour but de satisfaire automatiquement un ensemble d'exigences d'implémentation à partir d'une spécification abstraite [Tsalidis et al., 1992]. Guerra et al. insistent sur le fait que la génération réduit les coûts de développement et de maintenance des outils de mesure [Guerra et al., 2007] .

Un autre point important consiste en la manière d'exprimer des métriques. Le langage utilisé pour spécifier les mesures est-il dédié à cette problématique ? (cf. section 2.4.5) L'artefact mesuré est-il spécifié par un métamodèle (cf. section 2.4.4)? Le langage utilisé pour spécifier les mesures est-il explicitement métamodélisé (e.g.; [Guerra et al., 2007]) ?

Enfin, il est notable que les différentes approches de la mesure dépendent des artefacts mesurés syntaxiquement et/ou sémantiquement. Certaines approches s'attachent à mesurer un langage de programmation donné (e.g.; 2.3). Certaines approches sont plus ou moins génériques et peuvent mesurer un ensemble de langages de programmation (e.g.; plusieurs langages OO) (e.g.; [Lanza et Ducasse, 2002]). D'autres approches visent les phases amont du cycle de vie des systèmes et permettent de mesurer des modèles UML (e.g.; [Lavazza et Agostini, 2005]). Enfin, une approche de la mesure est totalement générique si elle permet de mesurer n'importe quel modèle quel que soit le métamodèle considéré, autrement dit n'importe quel DSL ([Guerra et al., 2007]). Dans ce cas, l'approche est dite totalement indépendante du langage / métamodèle considéré, à la fois au niveau syntaxique et au niveau sémantique.

La table 2.8 permet une comparaison complète des approches de la littérature. Pour récapituler, les points d'analyse sont les suivants:

- intégration de la mesure dans un outil d'analyse;
- intégration de la mesure dans un outil de développement;
- automatisation de la mesure;
- génération de l'outil de mesure;
- spécification par un métamodèle des artefacts mesurés;
- définition d'un DSL pour la mesure;
- spécification par un métamodèle d'un DSL de métriques;
- approche applicable à un langage de programmation;
- approche applicable à une famille de langages de programmation;
- approche applicable à UML;
- approche applicable à n'importe quel DS(M)L.

Aucune des approches de la littérature ne satisfait tous ces points. Une première façon de problématiser notre thèse est de définir une approche de la mesure qui satisferait tous les points listés ci-dessus: relatifs à l'environnement de mesure, relatifs à la spécification des mesures et enfin, relatifs à la diversité des artefacts mesurables. Nous compléterons dans le chapitre suivant cette problématisation plutôt montante (*bottom-up*) par une problématisation plus cartésienne.

	Ath	Hy+	Cro	SQ1	SQ2	BS	OCL	MEL	SD	DL	Sa.	RML	Xq1	Xq2	CQL	ATL	GMEF	Slam
Environnement																		
intégration dans un outil d'analyse		x							x					x	x			
intégration dans un IDE			x		x	x												x
efficacité de la mesure												x						
automatisation de la mesure				x	x											x		
génération de l'outil	x																	x
Domaine																		
un seul langage de prog.	x	x	x	x			x			x		x		x	x		x	
une famille de langages de prog.	x	x					x			x		x		x	x			
UML					x	x	x	x	x				x			x		
DSL								-								x		x
Orienté modèle																		
DSL pour la mesure	x								x		x				x		x	x
langage mesuré avec MM				x	x	x	x	x						x				x
DSL basé sur un MM	x							x										x
syntaxe graphique		x																x

TAB. 2.8: Synthèse des points de vue de la littérature

3 La mesure des modèles dirigée par les modèles

Nous présentons dans ce chapitre l'intégralité de notre approche de la mesure des modèles. L'approche est dirigée par les modèles car elle est fondée sur la spécification abstraite des mesures à l'aide d'un modèle complètement spécifié par un métamodèle.

Nous commencerons par définir précisément le problème auquel nous proposons une solution. Ensuite, nous présenterons l'approche dans ses grandes lignes, à la fois en terme de produit manipulé que de processus temporel. Nous pourrons alors décrire en détail, d'abord le métamodèle de spécification de métriques, puis l'architecture logicielle supportant l'approche. Enfin, nous passerons l'approche au crible d'un ensemble de points d'analyse issus de la littérature.

3.1 Définition du problème

Comme cette thèse n'est pas une nouvelle solution à un problème ancien et bien défini, nous consacrons une section à définir le problème auquel nos travaux s'attellent. Une synthèse graphique est présentée dans la figure 3.1

3.1.1 Mesurer les modèles de l'IDM

Problème 3.1 *Notre problème réside dans la spécification et l'implémentation des métriques sur les modèles.*

Les arguments habituels de la mesure (cf. 2.1.1) nous amènent à vouloir mesurer les modèles de l'IDM. Par exemple, l'assurance de la qualité des logiciels produits

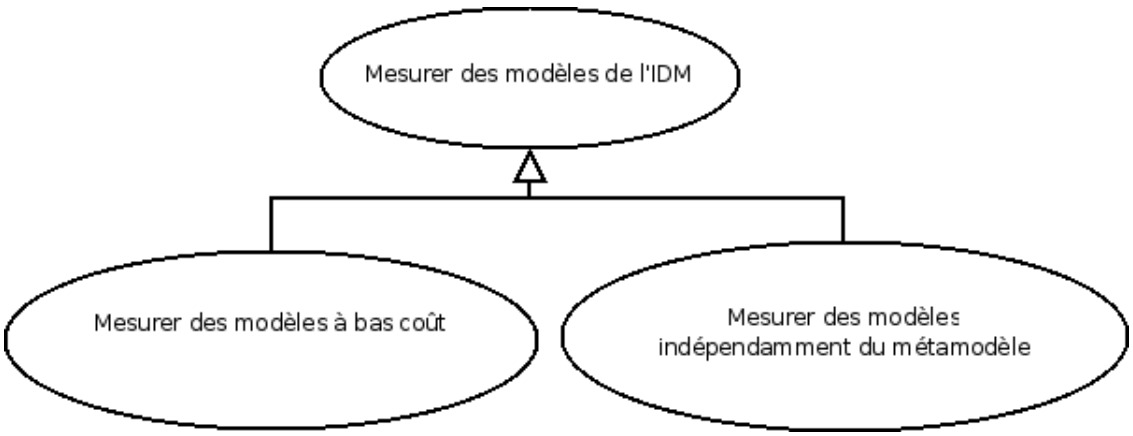

FIG. 3.1 : Synthèse de la problématique de cette thèse

avec l'IDM passe par des vérifications sur les modèles eux-mêmes, qui contiennent une information explicite et structurée qui n'est plus accessible dans le code source généré.

Dans un monde idéal, une qualité totale peut être obtenue par un processus qui interdit toute introduction de fautes, ou par des mécanismes de contrôle capables de détecter tous les défauts du logiciel. Une vision pragmatique consiste à utiliser les deux techniques, et à accepter l'impossibilité du zéro-défaut face à la complexité des artefacts produits.

Cette approche pragmatique est aussi la nôtre. Dans le cas des logiciels produits avec l'IDM, nous considérons la mesure des modèles comme l'un des outils de l'assurance qualité. Associée à des valeurs de seuil, la mesure des modèles peut devenir un outil de validation. Par exemple, une règle de validation pour les systèmes temps-réel pourrait énoncer qu'un processeur ne peut recevoir plus de 30 *threads* en même temps.

La mesure des modèles permet aussi d'obtenir des données pour mesurer quantitativement la productivité. Elle peut servir d'entrée à des modèles d'estimation des coûts de développement des systèmes (e.g.; [Boehm et al., 1995]).

3.1.2 Indépendamment du domaine d'application

Notre problème réside dans la mesure des modèles. Une partie des modèles consiste en des modèles des domaines, instances d'un DSML (*Domain Specific Modeling Language*). De manière similaire, certaines métriques de modèles sont des métriques de domaine. À notre connaissance, le concept de métrique de domaine est très peu étudié dans la littérature. Une métrique de domaine est une métrique qui réfère uniquement à des concepts du domaine. Elle est exempte des préoccupations d'implémentation de la métrique ou de codage des données mesurées.

La mesure des modèles doit être effectuée sur des modèles, et ce quel que soit le domaine d'application. Une solution à la mesure des modèles doit donc être indépendante du domaine d'application [Monperrus et al., 2008b]. De même, la mesure des modèles peut s'appliquer sur tous les artefacts d'un processus de développement IDM, du tout début à la toute fin du cycle de vie. Par exemple, une solution à la mesure des modèles doit pouvoir s'appliquer à un modèle d'exigence jusqu'à un modèle de programme complètement exécutable. Cela nous conduit à raffiner notre problème comme suit.

Problème 3.2 *Notre problème réside dans la définition d'une approche de la mesure des modèles indépendante du domaine et du cycle de vie, c'est à dire indépendante du métamodèle.*

3.1.3 À un coût acceptable

La mesure des modèles a inévitablement un coût. Nous étudions maintenant qualitativement les éléments qui le déterminent.

Le nombre des types de modèles i.e.; de métamodèles Tout d'abord, l'approche IDM fait intervenir plusieurs modèles. Par exemple, l'approche MDA décrite dans le

papier original [Soley, 2000] fait intervenir un modèle indépendant de la plateforme (PIM), un modèle de plateforme (PM), et un modèle spécifique à la plateforme (PSM). En plus de ces modèles, on peut trouver parmi les approches de type IDM des modèles d'exigences, des modèles fonctionnels, des modèles de qualité de service, etc. Ainsi, le nombre de familles de modèles, i.e.; de métamodèles est grand. L'éventail va de métamodèles standards et utilisés par de nombreuses entreprises, comme UML, à des métamodèles métier propres à une entreprise, ou à un département d'une entreprise, en passant par des métamodèles de plateforme à petite diffusion.

Pour mesurer tous les modèles, il faut être capable de mesurer en fonction de chacun de ces métamodèles. Si N est le nombre de métamodèles mis en oeuvre dans un processus de développement IDM, le coût de développement ad hoc de tous les outils de mesure est proportionnel à N.

Exigences d'implémentation Le deuxième facteur de coût d'un logiciel de mesure réside dans les exigences qu'il doit satisfaire. Celles-ci peuvent s'énoncer comme suit:

- le logiciel de mesure des modèles doit permettre une mesure automatique;
- le logiciel de mesure des modèles doit être intégré dans un environnement de modélisation;
- le logiciel de mesure doit être ouvert et adaptable;
- le logiciel de mesure doit passer à l'échelle, et mesurer efficacement des modèles de taille industrielle.

Chacune de ces exigences cachent une complexité accidentelle. L'automatisation passe par la définition d'appels programmatiques par ligne de commandes, API, ou fichiers d'automatisation (de type *ant*) par exemple. L'intégration dans un environnement de modélisation nécessite d'avoir un environnement ouvert, documenté et ayant des passerelles disponibles entre son ou ses langages de programmation et le langage choisi pour réaliser le logiciel de mesure. L'adaptabilité consiste à laisser à l'utilisateur final la possibilité d'exprimer, par exemple, des modèles de coût au dessus des mesures définies dans l'outil.

Par analogie avec les logiciels de mesure de code source, l'ordre de grandeur du coût d'un outil de mesure des modèles est de plusieurs hommes-mois en suivant la formule basique de Cocomo (à partir de l'observation empirique de la taille des logiciels de mesure de code source, cf. table 2.3).

Les taille des marchés Le troisième facteur de coût de la mesure des modèles est lié à la présence et au prix de logiciels commerciaux de mesure des modèles.

Nous faisons l'hypothèse économique que le coût des logiciels de mesure est inversement proportionnel à la taille du marché en question. Dans le cas extrême où le marché se réduit à une seule entreprise, l'entreprise en question doit supporter entièrement le coût de développement du logiciel de mesure. À l'inverse, quand le marché est très grand, i.e.; qu'un grand nombre d'entreprises utilisent un logiciel de mesure, son coût devient négligeable.

Cette hypothèse se vérifie empiriquement. Par exemple, on peut trouver des logiciels de mesure pour Java à 40$ (cf. le tableau 2.3). Nombre d'entreprises utilisent UML, mais beaucoup moins que Java et il y a très peu d'outils dédiés à la mesure

des modèles UML[1]. Il n'y pas d'outil dédié à la mesure des modèles AADL (*Architecture Analysis & Design Language*), car c'est un métamodèle utilisé par un trop petit nombre d'entreprise.

Nous avons vu dans en 3.1.3 que le champ d'application des métamodèles, i.e.; le nombre d'entreprises les utilisant, peut être très vaste comme UML, ou particulièrement restreint. Le principe exposé dans cette section montre que le coût de la mesure des modèles est inversement proportionnel à la spécificité des modèles utilisés.

La distance entre l'expert du domaine et l'outil de mesure Le quatrième facteur de coût est lié au développement de l'outil de mesure. Il repose sur l'hypothèse que dans une entreprise, la personne qui sait ce qu'il faut mesurer est l'expert du domaine. Cet expert n'est pas informaticien (sauf exception). Il faut donc faire un cycle de développement classique (spécification / architecture / développement) pour obtenir un logiciel de mesure des modèles. Autrement dit, la distance entre l'idée de la mesure par l'expert du domaine et la réalisation de l'outil de mesure par l'informaticien est grande donc coûteuse.

La mesure des modèles est une activité coûteuse. Ce n'est pourtant pas la seule activité de l'IDM, et le coût de la mesure doit constituer une proportion raisonnable du coût total des développements liés à l'IDM. Un guide de la NASA recommande que cette proportion soit inférieur à 2% [NASA Software Program, 1995, p. 32]. Or cette question du rapport est particulièrement pertinente dans le cadre de l'IDM. Émettons l'hypothèse que le logiciel de mesure des modèles soit développé manuellement. Par contre, les autres artefacts du développement seraient générés (e.g.; environnement de modélisation, application finale). En conséquence, et sous cette hypothèse, la proportion de la mesure dans le coût total des développements liés à l'IDM serait plus élevée que si le logiciel de mesure était lui aussi généré. L'ensemble de ces arguments, majoritairement qualitatifs, nous amène à affiner notre problème comme suit.

Problème 3.3 *Notre problème réside dans l'obtention des outils de mesure des modèles à un coût maintenant un rapport raisonnable par rapport au coût total.*

En d'autres termes, le coût de l'outil de mesure doit être proportionnel à sa complexité intrinsèque, i.e.; à la complexité des métriques elles-mêmes, et indépendant des complexités accidentelles d'implémentation ou de taille des marchés.

3.1.4 Exemple d'une instance du problème

La présentation détaillée de notre contribution s'appuiera sur un exemple jouet. Le domaine considéré est l'architecture logicielle (cf. section 2.3.3 de l'état de l'art). Le métamodèle de domaine est représenté figure 3.2. Celui-ci comprend les concepts de composant (*Component*), d'interface (*Interface*) et leurs relations. Un composant

[1]Nous connaissons deux outils de mesures pour les modèles UML. Les deux sont commerciaux: SDMetrics (http://www.sdmetrics.com) et *UMLQuality* http://www.qualixo.com. Notons que certains modeleurs commerciaux intègrent des fonctionnalités de mesures, mais la documentation disponible sur le sujet est quasiment inexistante.

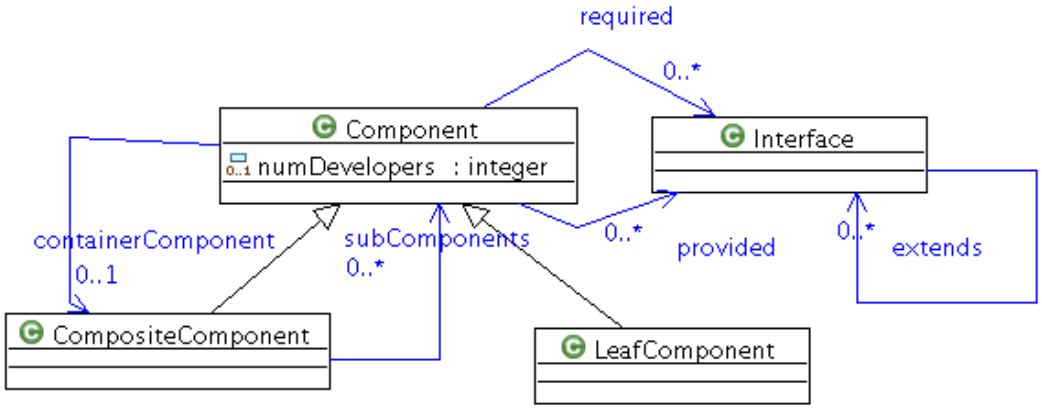

FIG. 3.2 : Le métamodèle de domaine que nous utiliserons pour illustrer notre contribution.

fournit certaines interfaces et en nécessite d'autres. Le patron de conception *composition* est appliqué pour imbriquer les composants entre eux. Des approches telles EMF ou GMF permettent de générer l'environnement de modélisation. Afin de mesurer ces modèles d'architecture logicielle, devrions nous coder manuellement l'outil de mesure ? Les exemples de métriques illustrant notre approche référeront tous à ce métamodèle d'architecture logicielle, dans le but de générer complètement l'outil de mesure des modèles.

3.2 L'approche MDM : produits et processus

Dans cette section, nous présentons l'architecture générale de notre solution au problème présenté dans la section précédente. Notre solution est une approche complètement orientée modèle de la mesure des modèles. C'est une instance de la vision IDM appliquée à la problématique de la mesure des modèles. L'IDM prône une approche générative, c'est-à-dire la génération de code source à partir de modèles. Dans le cas de la mesure des modèles, la vision IDM consiste à générer l'outil de mesure des modèles à partir d'un modèle de métriques. Ce modèle de métriques, que nous appellerons spécification de métriques, est une instance d'un métamodèle de métriques. Cette spécification contient toute l'information nécessaire pour générer le code exécutable de l'outil de mesure, satisfaisant les exigences présentées en 3.1.3. Cette approche est nommée approche MDM en référence à l'acronyme anglais de *Model-driven Measurement*. L'article *A Model-driven Measurement Approach* [Monperrus et al., 2008d] est la principale publication présentant l'approche MDM.

Définition 3.1 *L'approche MDM consiste à spécifier des métriques de modèles comme un modèle dans le but de générer entièrement l'outil de mesure associé.*

La présentation de la solution sera effectuée en trois partie. Nous commencerons par présenter les artefacts manipulés et leurs relations, puis nous décrirons le processus temporel de mesure par les modèles, pour enfin mettre l'accent sur l'aspect génératif de notre contribution.

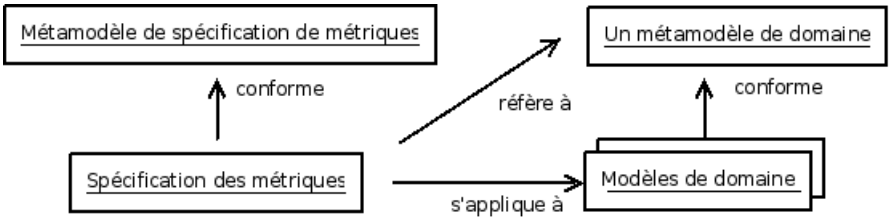

FIG. 3.3 : Artefacts de type modèles impliqués dans notre approche

Domaine	Ex. de métamodèle
Patrouille maritime	MM de patrouille maritime
Logiciel à composants	MM *Fractal*
Modélisation	UML
Métamodélisation	*Ecore*

TAB. 3.1: Exemple de domaine et métamodèle associé

3.2.1 Une vue produit

La figure 3.3 présente les principaux artefacts de notre contribution. Le coeur de notre contribution est un métamodèle de spécification de métriques. Il consiste en quelques dizaines de classes, qui représentent les concepts nécessaires à la mesure des modèles. Il sera présenté en détail en 3.3.

Les métriques de modèles sont exprimées comme une instance du métamodèle de spécification de métriques. Comme elles sont codés sous forme de modèles, il s'agit donc d'un graphe d'objets inter-connectés. Ces spécifications de métriques font référence à un métamodèle de domaine, et sont appliquées sur des modèles de domaine. Les références au métamodèle de domaine sont nécessaires, car elles permettent d'exprimer la spécificité de la métrique. Sans ces références, les métriques restent strictement syntaxiques, ce sont des métriques de graphes.

La mesure effective passe par une étape de génération de code. Un générateur de code prend en entrée le modèle de métriques afin de générer l'implémentation qui va effectivement mesurer les modèles de domaine. Les principes de génération seront présentés un peu plus loin.

Le métamodèle de domaine est la spécification des concepts nécessaires à la création des modèles de domaine. Il contient les classes et références structurant les modèles. Notons que le métamodèle de domaine n'est pas là pour définir une structure de données, il a pour but d'être le plus fidèle possible au modèle mental de l'expert du domaine. En cela, il semble que la notion de métamodèle de domaine soit quasiment identique à la notion d'ontologie de domaine. Nous verrons plusieurs métamodèles de domaine dans le chapitre validation de cette thèse. Par exemple, nous présenterons un métamodèle de domaine pour les systèmes de surveillance maritime. Notons que le domaine considéré peut être le logiciel lui-même, par exemple avec un métamodèle de logiciel à base de composants, ou même l'ingénierie dirigée par les modèles, par exemple avec le métamodèle UML.

Enfin, le quatrième élément de l'approche MDM est l'ensemble des modèles de domaine. Ceux-ci décrivent précisément une réalité, et sont la base essentielle des

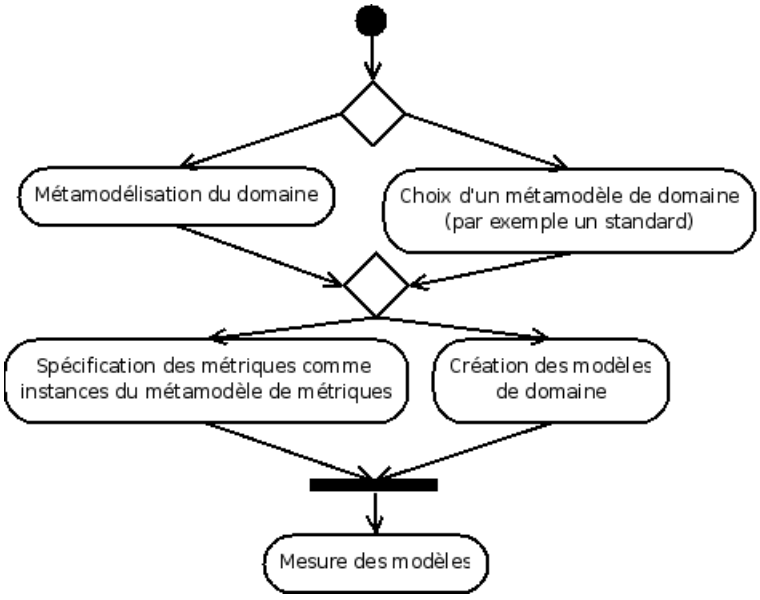

FIG. 3.4 : Un diagramme d'activité de l'approche MDM

activités de l'IDM, e.g.; transformation de modèle, génération de code, validation, génération de test, etc.

Pour conclure, remarquons que les métamodèles et modèles en question satisfont aux propriétés présentées en 2.2. L'argumentaire pour la partie métrique est le suivant:

- le métamodèle de métriques est un modèle de modèles de métriques;
- un modèle de métriques X est un modèle par rapport à l'implémentation générée Y;
- un modèle de métriques a pour but de spécifier complètement, de manière concise et non ambiguë, un outil de mesure de modèle, et sert d'entrée à un générateur de code;
- il est plus rentable d'utiliser le modèle de métriques que de coder directement un outil de mesure des modèles.

3.2.2 Une vue processus

Nous avons présenté les artefacts manipulés dans notre approche. Dans cette section nous montrons l'orchestration de tous ces éléments. Cette orchestration est résumée graphiquement par un diagramme d'activité UML dans la figure 3.4.

La figure 3.4 est découpée en trois étapes séquentielles. La première représente la mise en place de l'approche IDM. Celle-ci consiste à choisir le métamodèle considéré. Dans certains cas, il est possible de choisir un métamodèle existant, et parfois standardisé, par exemple le métamodèle UML. Il est aussi important de préciser la version des métamodèles choisis. Dans d'autres cas, quand les métamodèles existants ne sont pas dédiés à notre domaine, ou de manière pas assez explicite et pertinente, il est nécessaire de créer un métamodèle de domaine. Nous montrerons dans le chapitre validation les deux cas de figure. Notons que notre contribution n'est pas une

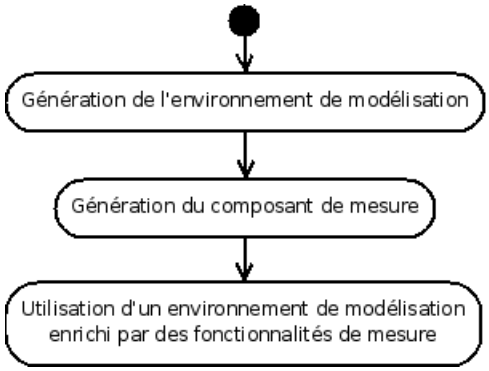

FIG. 3.5 : Les aspects génératifs de l'approche MDM

méthode de choix entre réutilisation d'un métamodèle et métamodélisation, ni une méthode de création de métamodèle.

La deuxième étape consiste à créer les modèles mis en oeuvre. La création de modèles de métriques et de modèles de domaine peut être faite en parallèle par des personnes ou équipes différentes. Les métriques sont spécifiées comme instance du métamodèle de métriques. Cela peut être fait avec une syntaxe arborescente, ou une proto-syntaxe textuelle. Les deux syntaxes sont présentées plus loin dans cette thèse. Les métriques peuvent être issues de la littérature, d'outils existants, de guides méthodologiques, ou de la connaissance empirique des experts du domaine.

La création des modèles de domaine est indépendante de la mesure et des modèles de métriques. Les modèles créés doivent être conformes au métamodèle de domaine afin de permettre à l'outil de mesure généré de fonctionner. Cette étape de mise en conformité est assuré par l'outil de modélisation par construction ou par un mécanisme de validation interne à l'environnement de modélisation.

Une fois toutes ces étapes effectuées, il est alors possible de mesurer les modèles suivant les métriques spécifiées. L'approche proposée dans cette thèse est dirigée par les modèles suivant deux points de vue: 1) elle sert à mesurer des modèles et 2) elle est fondée sur des modèles de métriques.

3.2.3 L'aspect génératif du processus

Le processus présenté en figure 3.4 est génératif sur trois points qui sont illustrés en figure 3.5. Premièrement, la métamodélisation du domaine permet de générer l'environnement de modélisation. Ce n'est pas notre contribution, nous avons vu que des outils comme *Atom3* ou EMF permettent de le faire (cf. 2.2.3). Dans notre cas, nous nous plaçons dans le monde *Eclipse/EMF*, et nous reposons sur le générateur EMF et les outils associés comme les approches GMF ou *Topcased* pour générer un éditeur graphique.

L'aspect génératif de notre contribution est que nous enrichissons automatiquement l'éditeur généré avec des fonctionnalités de mesure. Un générateur de code prend en entrée une spécification de métriques sous la forme d'un modèle instance du métamodèle et génère l'outil de mesure lui-même. Plus pratiquement, notre générateur de code produit un composant *Eclipse* (*plugin*) directement exécutable. Ce

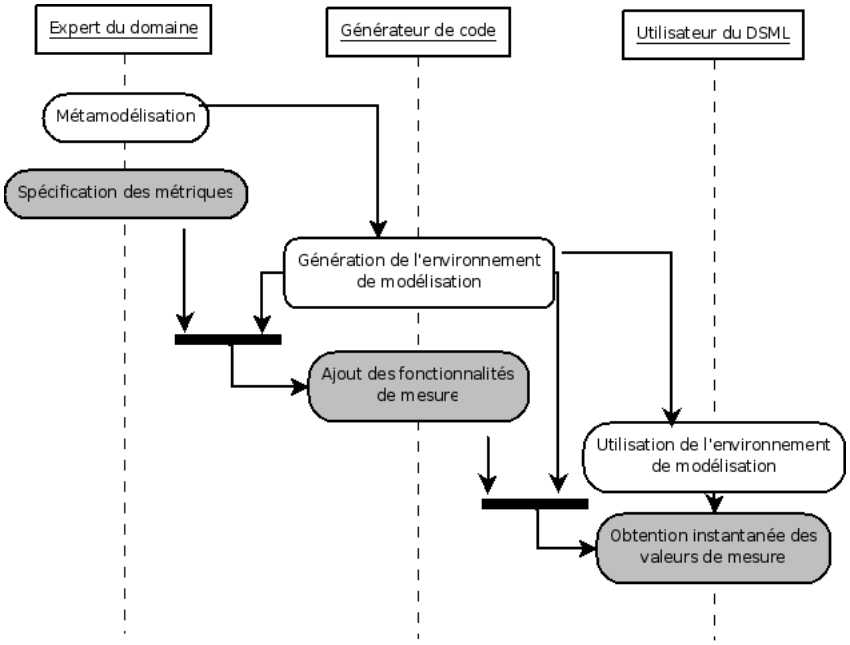

FIG. 3.6 : Vue globale de notre contribution: processus

point est présenté en détail dans 3.4.

L'outil de modélisation utilisé est donc totalement généré à la fois pour le squelette et pour les fonctionnalités de mesure. Notre contribution enrichit l'outil et a donc pour but d'enrichir les processus de création de modèles par un retour (*feedback*) constant des valeurs de mesure.

Notons qu'un nombre important des modèles de domaine sont appelés à servir de point d'entrée à un générateur de code propre au domaine. Par exemple, un modèle de composants logiciels peut être transformé en code source Java. Cette aspect génératif propre au domaine est en dehors du périmètre de cette thèse.

Conclusion La figure 3.6 résume les trois points précédents, tout en montrant la nouveauté par rapport à l'existant (les activités grisées sont celle de l'approche MDM). L'expert du domaine définit les métriques comme modèle, le générateur de code ajoute les fonctionnalités de mesure à l'environnement de modélisation, et l'ingénieur qui manipule les modèles de domaine reçoit un retour constant des valeurs de mesure.

Ce que l'approche MDM n'est pas Afin de compléter l'architecture globale de notre contribution, nous la présentons tel un négatif photographique, en énumérant ce que l'approche MDM n'est pas:

- l'approche MDM ne définit pas des nouvelles mesures;
- l'approche MDM ne valide pas statistiquement des mesures par une étude empirique;
- l'approche MDM ne définit pas des contextes d'interprétation pour les valeurs de mesures dans un domaine donné.

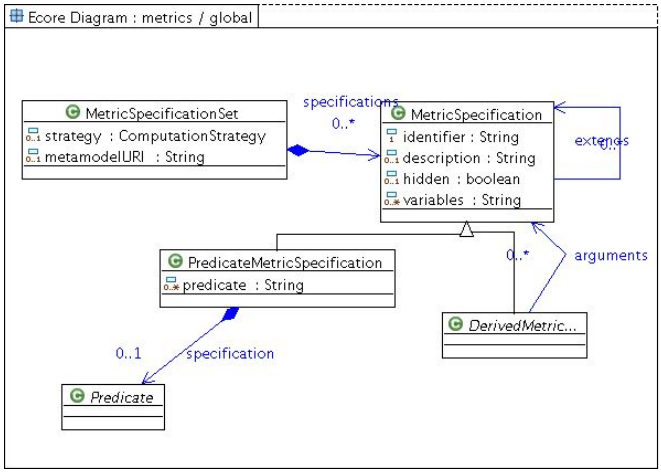

FIG. 3.7 : Le coeur du métamodèle de spécification de métriques

3.3 Le métamodèle de spécification de métriques

Dans cette section, nous présentons le métamodèle de spécification de métrique. Le but de ce métamodèle est de définir les concepts nécessaires à la spécification de métriques de modèles. D'une part, nous voulons qu'une spécification de métrique fondée sur un métamodèle soit non ambiguë et manipulable par un logiciel, afin d'avoir une plus-value par rapport au langage naturel. D'autre part, nous souhaitons qu'elle soit plus concise qu'avec un langage de programmation, et accessible à un expert de domaine sans connaissances en programmation ou en architecture logicielle. Le métamodèle sera présenté de façon structurée, classe par classe, attribut par attribut.

La figure 3.7 illustre les principes directeurs du métamodèle. Le métamodèle permet de spécifier des métriques (classe *MetricSpecification*). Certaines de ces métriques sont dérivées et prennent d'autres métriques en paramètre (par exemple une addition). Toutes les métriques d'importance sont liées à au moins un prédicat. Un prédicat est une sorte de filtre sur les éléments de modèles. Toutes ces notions sont présentées en détail plus avant, en 3.3.3.

La classe *MetricSpecificationSet* est une entité qui contient un ensemble de spécifications de métriques. Elle réfère à un métamodèle de domaine avec l'attribut *metamodelURI*. Pour un ensemble de spécifications de métriques, on peut spécifier une stratégie de parcours des modèles. Ce dernier point est présenté en détail en 3.4.1.

Une spécification de métrique contient un identifiant et une description. Notons qu'elle ne contient pas d'attribut de valeur, car la valeur est issue d'une mesure particulière, c'est donc un attribut d'implémentation qui ne doit pas apparaître dans le métamodèle.

Cette classe est abstraite et sous-classée par plusieurs classes présentées en 3.3.1.

3.3.1 Les types de métriques

Nous avons identifié sept types de métriques. Le processus de classification a suivi une décomposition orientée-objet du problème. Nous avons donc regroupé par classe

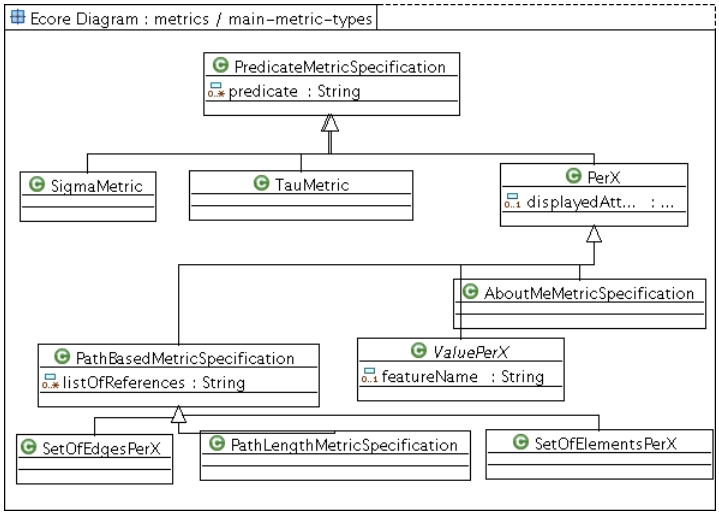

FIG. 3.8 : Les différents types de métriques identifiés

abstraite les métriques partageant les mêmes attributs ou méthodes. Les classes correspondantes sont structurées par les classes abstraites *PredicateMetricSpecification, PerXMetricSpecification, PathBasedMetricSpecification.*

La classe *PredicateMetricSpecification* est une classe abstraite, avec une variable d'instance représentant le prédicat associé. Nous expliquerons pour chacune des sous-classes en quoi un prédicat est nécessaire.

Le tye de métrique Sigma

Définition 3.2 *Le type de métrique Sigma spécifie un nombre d'éléments de modèle satisfaisant un prédicat. Une métrique Sigma est une métrique de comptage.*

C'est une métrique de modèle qui se retrouve régulièrement. En effet, l'activité de modélisation – au sens de l'IDM – est fondamentalement discrète. L'élément atomique, l'équivalent du quantum physique, est l'élément de modèle. La métrique Sigma est le pendant en mesure de cet aspect discret de la modélisation IDM . C'est pour cette raison qu'on les retrouve très souvent lors de l'application de l'approche MDM [Monperrus et al., 2007] :

1. Les métriques de comptages sont très présentes en génie logiciel. Par exemple, quels que soient la définition et le langage considéré, la métrique du nombre de lignes de code est une métrique de comptage. Des études empiriques fondées sur des données historiques ont prouvé que le nombre de lignes de code est corrélé avec le coût du logiciel [Boehm et al., 1995] ou avec sa densité de fautes [Withrow, 1990]. Nous présenterons cette métrique définie avec l'approche MDM dans la partie validation.
2. L'application du rasoir d'Occam connu aussi sous le nom de loi de parcimonie [Baker, 2004], ou encore comme règle d'ingénierie (*Keep It Short & Simple*, K.I.S.S) est un principe qui s'énonce le plus souvent comme: *Les entités ne doivent pas être multipliées plus que nécessaire.* Ce principe, appliqué à la mesure des modèles, est un argument en faveur de la métrique Sigma comparée

à des métriques plus complexes. En cela, nous partageons le point de vue de Zuse:

> *Using more sophisticated measures do not necessarily guarantee better results.*[2] Zuse [Zuse, 1991, p. 410]

3. Les métriques de comptage sont simples mais sont à la base de métriques dérivées plus complexes, par exemple avec une somme pondérée. L'obtention de la valeur finale dépend de la facilité et de l'acuité des métriques atomiques.

4. Les métriques de comptage sont utilisées dans l'industrie. Comme le souligne Henderson-Sellers [Henderson-Sellers, 1996, p. 37], l'industrie utilise de préférence des métriques compréhensibles et faciles à collecter. En cela, les métriques de comptages sont parfaitement adéquates. D'ailleurs, Cheng et al. [Cheng et al., 2005] ont étudié la question de l'analyse automatique de modèles à base de métriques, dont un nombre important sont des métriques de comptage (e.g.; le nombre de violations d'un certain patron). De même, *Motorola* [Weil et al., 2006] utilise une liste de 150 métriques de comptage sur les éléments de modèles UML pour contrôler leurs développements IDM.

5. Compter est un but de modélisation à part entière. Comme nous l'avons vu plus haut, l'activité de modélisation au sens IDM est clairement discrète. La quantification d'un produit peut donc passer par la modélisation. Par exemple, la métamodélisation des exigences, qui fait l'objet d'une section de cette thèse, permet d'obtenir des métriques de comptage, quasiment impossibles à obtenir si l'on considère le langage naturel.

La difficulté des métriques de comptage réside dans la spécification concise et non-ambiguë des éléments à compter. Dans notre approche, cette spécification est faite par des prédicats dont le métamodèle est présenté plus avant dans cette thèse.

Dans le cas de la mesure des modèles d'architecture logicielle (cf. 3.1.4), les métriques Sigma suivantes peuvent être définies.

Listing 3.1 : Exemples de métriques Sigma

```
1
2
3  metric SigmaMetric NOI is
4    description "number of interfaces"
5    elements satisfy "this.isInstance(Interface)"
6  endmetric
7
8  metric SigmaMetric NOCompComp is
9    description "too complex components"
10   elements satisfy "this.isInstance(Component)
11           and this.required > 10
12           and this.provided > 10
13           and this.subComponents > 5"
14 endmetric
```

[2] L'utilisation de mesures plus compliquées ne garantit pas nécessairement de meilleurs résultats.

Le type de métrique *PerX*

Définition 3.3 *Le type de métrique abstrait* PerX *regroupe les métriques qui s'appliquent à un élément de modèle précis. Les éléments de modèle considérés sont sélectionnés par un prédicat.*

La métrique *PerX* est locale (la localité étant l'élément de modèle) au contraire des métriques *Sigma* et *Tau*, qui sont elles, globales au modèle mesuré. Par exemple, une métrique *PerX* sur le métamodèle d'architecture logicielle est le nombre d'interfaces fournies par composant. Dans ce cas, le X (du *PerX*) correspond à une instance de le classe *Component* du métamodèle de domaine. C'est pour cette raison que la métrique *PerX* hérite de *PrédicateMetricSpecification*, afin d'avoir un prédicat qui spécifie quels sont les objets à considérer.

Le type de métrique *ValuePerX*

Définition 3.4 *Le type de métrique* ValuePerX *spécifie que la valeur de mesure correspondante peut être obtenue directement à partir d'une caractéristique (*feature*) de l'objet satisfaisant le prédicat.*

Ce type de métrique contient donc le nom de la caractéristique permettant d'obtenir une valeur. Elle est sous-classée en trois classes.

MultiplicityPerX renvoie la taille d'une collection, la caractéristique doit donc avoir une multiplicité limite supérieure à 1. *ValuePerX* renvoie la valeur d'une caractéristique qui doit être obligatoirement de type *Real*, *Integer* ou similaire. *MethodCallPerX* est utilisée dans le cas de la métamodélisation exécutable, elle permet d'appeler une méthode qui renvoie la valeur correspondante.

Le type de métrique *AboutMe*

Définition 3.5 *Le type de métrique* AboutMe *spécifie un environnement de calcul pour une autre métrique. C'est une métrique d'un ordre supérieur aux précédentes, car elle prend en paramètre une autre métrique.*

En pratique, la métrique *AboutMe* déclare une variable *__me__*, qui correspond à l'objet sélectionné par le prédicat principal. Cette variable *__me__*, qui constitue le nouvel environnement de calcul peut alors être utilisée par la métrique donnée en paramètre. Ce point est illustré dans le listing 3.2.

Pour connaître le nombre des composants couplés à une interface, il faut commencer par déclarer une spécification de métrique *AboutMe*, qui sélectionne les composants un par un, et pour chacun d'entre eux, recalcule sur le modèle entier la métrique interne, ici une métrique Sigma.

Listing 3.2 : Exemples de métriques PerX

```
1
2 metric AttributeValuePerX NODev is
3   description "The number of developers per component"
4   elements satisfy "this.isInstance(Component)"
5   value is "this.numDevelopers"
```

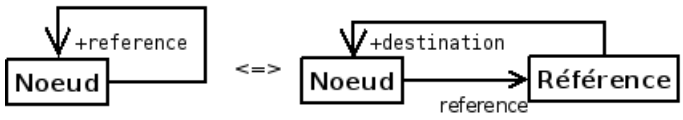

FIG. 3.9 : Dualité Noeud / Référence

```
6  endmetric
7
8  metric MultiplicityPerX NoOp is
9    description "number of provided interfaces per components "
10   size of "provided"
11 endmetric
12
13 metric AboutMeMetricSpecification COUPLING is
14   description "coupling to this interface"
15   elements satisfy "Interface.isInstance(self)"
16   internal metric spec is metric SigmaMetric COUPLING_ is
17   description "used for COUPLING"
18   // LE VARIABLE __ME__ EST UTILISÉE ICI
19   elements satisfy "this.provided == __me__
20                  or this.required == __me__ "
21 endmetric
```

Le type de métrique Tau

Définition 3.6 *Le tye de métrique Tau spécifie un nombre de références*[3] *de modèles satisfaisant une condition.*

La métrique Tau est en quelque sorte le complémentaire de la métrique Sigma. En effet, dans un métamodèle de domaine, il est toujours possible de transformer une référence en classe, comme l'illustre la figure 3.9, et dans certains cas, de suivre le chemin inverse. Une référence peut toujours être représentée par une classe suivant le patron de la figure. Cette dualité a un impact direct sur la mesure des modèles. En fonction des choix faits lors de la définition du métamodèle, une métrique peut être un nombre d'éléments de modèle ou un nombre de références dans le modèle.

La classe *TauMetric* est raffiné en trois sous-classes. Les trois sous-classes partagent le même principe mais sont sensiblement différentes (figure 3.11). Pour *NameBasedTau*, on compte le nombre de références simplement suivant le nom. Pour *PredicateBasedTau*, on compte le nombre de références dont la source et la destination satisfont un prédicat. Pour *ReferenceBasedTau*, on compte le nombre de références qui sont instances d'une référence précise du métamodèle. On peut remarquer que si le métamodèle de domaine ne contient que des noms de références uniques, alors *NameBasedTau* et *ReferenceBasedTau* sont équivalentes.

[3] Conformément à l'usage, nous utilisons le terme référence pour désigner à la fois un élément du métamodèle, et un lien entre deux éléments d'un modèle. En effet, les termes candidats pour l'une ou pour l'autre acception (*lien*, *pointeur*, *association*, *attribut*) peuvent tous constituer un piège terminologique.

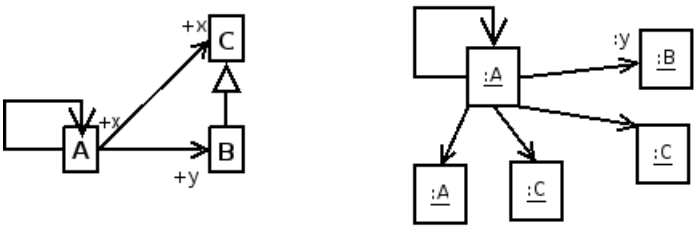

FIG. 3.10 : Un métamodèle fictif et un modèle conforme

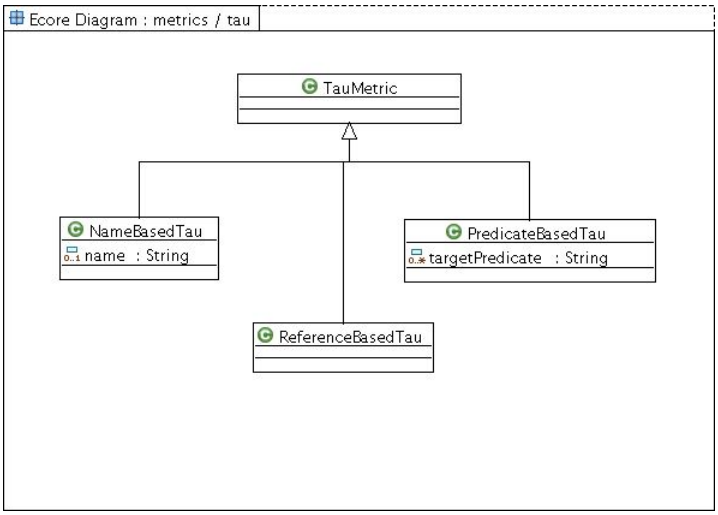

FIG. 3.11 : Les trois raffinements de la métrique Tau

L'exemple de la figure 3.10 illustre cette différence sémantique. Sur l'exemple de cette figure, soient les trois métriques Tau suivantes:

- la condition est que le nom de la référence soit x. Pour le modèle de la figure 3.10, la valeur de mesure correspondante est 4 (*NameBasedTau*):
- la condition est que la référence soit instance de l'association x entre A et C (*ReferenceBasedTau*). Pour le modèle de la figure 3.10, la valeur de mesure correspondante est 2;
- avec $\phi_{source}(t) = \text{isInstance}(A, t)$ et $\phi_{dest}(t) = \text{isInstance}(C, t)$ (*PredicateBasedTau*). Pour le modèle de la figure 3.10, la valeur de mesure correspondante est est 3.

Le listing 3.3 donne un exemple de métriques Tau pour le métamodèle d'architecture logicielle.

Listing 3.3 : Exemples de métriques Tau

```
1
2 metric NameBasedTau NODeComp is
3   description "The number of decomposition"
4   link is "CompositeComponent:subComponent"
5 endmetric
6
```

```
7
8  metric PredicateBasedTau ExceptionIsDataType is
9    description "number of dependy links between
10                components and interfaces (provided+required)"
11   source satisfies "Component.isInstance(self)"
12   target satisfies "Interface.isInstance(self)"
13 endmetric
```

La famille de métriques à base de chemin

Les trois derniers types de métriques sont tous des sous-classes de la classe abstraite *PathBasedMetricSpecification*. Ils ont en commun d'être fondés sur une liste de noms de référence, c'est à dire un type de chemin à suivre durant le parcours des modèles mesurés.

Le type de métrique *SetOfElementsPerX*

Définition 3.7 *Le type de métrique* SetOfElementsPerX *spécifie un nombre d'éléments satisfaisant un prédicat et liés à un élément racine par un certain type de chemin.*

Le chemin est défini par une liste de références. Un deuxième prédicat sert à compter les éléments pris en compte.

Le type de métrique *SetOfEdgesPerX*

Définition 3.8 *Le type de* SetOfElementsPerX *spécifie un nombre de références satisfaisant une condition et suivies à partir d'un élément racine.*

Le type de métrique *PathLength*

Définition 3.9 *Le type de métrique* PathLength *spécifie la taille maximale d'un chemin partant d'un objet racine et satisfaisant un type de chemin.*

Cette taille est calculée par un algorithme de marcheur, présenté en 3.4.1. Afin d'illustrer ces trois types de métriques à base de chemin, en voici des instances pour mesurer des modèles d'architecture logicielle.

Listing 3.4 : Exemples de métriques PathBased

```
1
2  metric SetOfElementsPerX NOIT is
3    description "The number of interfaces
4                 involved per component"
5    x satisfy "this.isInstance(Component)"
6    elements satisfy "this.isInstance(Interface)"
7    references followed "required,provided,extends"
8  endmetric
9
10 metric SetOfEdgesPerX NoPpC is
11   description "number of subcomponents per components"
```

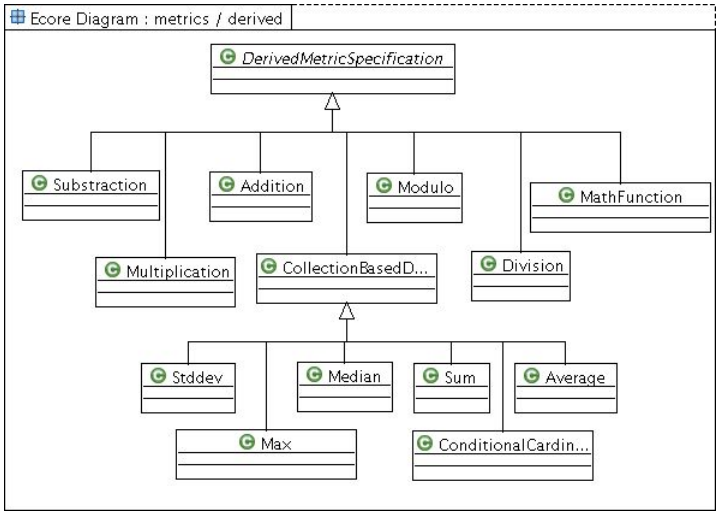

FIG. 3.12 : Opérateurs pour les métriques dérivées

```
12   x satisfy "this.isInstance(CompositeComponent)"
13   references followed "subComponents"
14 endmetric
15
16
17 metric PathLengthMetricSpecification DD is
18   description "Depth in decomposition"
19   x satisfy "this.isInstance(LeafComponent)"
20   references followed "container"
21 endmetric
```

3.3.2 Les métriques dérivées

Les métriques dérivées

Les métriques dérivées sont les métriques dont le calcul dépend seulement d'autres métriques et non du modèle directement. On y retrouve les opérateurs arithmétiques (addition, soustraction, multiplication, division, modulo), les opérateurs statistiques (moyenne, somme, médiane, déviation standard), et les fonctions mathématiques courantes, non représentées sur la figure (exponentielle, logarithme, puissance, etc.)

La gestion des constantes

Les constantes sont gérées par une classe *ConstantMetricSpecification*, qui encapsule une valeur invariable. Le code généré pour cette métrique utilise le polymorphisme pour renvoyer la même valeur quel que soit le contexte d'appel.

```
1 // code généré
2 private class two extends ConstantMetricSpecificationImpl {
3         public double getConstant() {
4                 return 2.0;}
5 }
```

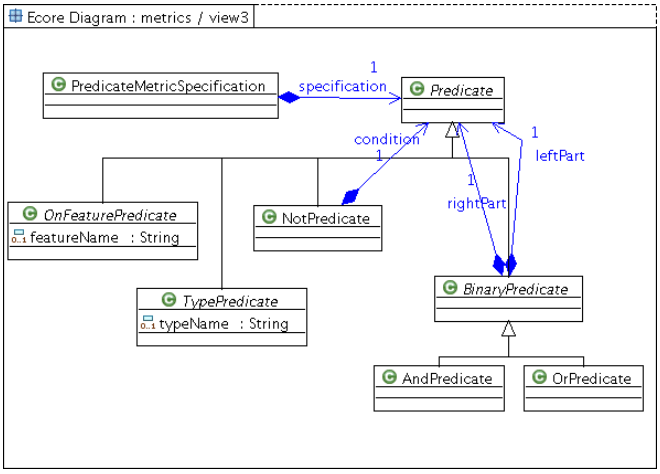

FIG. 3.13 : Les prédicats

3.3.3 Les prédicats

La partie prédicat du métamodèle de prédicat permet de spécifier des prédicats sur des éléments de modèle.

Un prédicat est une fonction qui considère un élément de modèle et renvoie *vrai* ou *faux*. En d'autres termes, c'est un morphisme de l'ensemble des éléments de modèles, noté V, vers l'ensemble des valeurs de vérité.

$$\begin{array}{rccl} \phi : & V & \rightarrow \{true; false\} \\ & x & \mapsto \phi(x) \end{array} \tag{3.1}$$

Un prédicat peut être décomposé à la manière des fonctions booléennes. Le métamodèle comprend donc les mécanismes nécessaires, à savoir les classes *NotPredicate, AndPredicate* et *OrPredicate*.

Test sur les types

Nous avons défini un mécanisme qui teste le type d'un élément de modèle, représenté par la classe *TypePredicate*. C'est une classe abstraite qui est sous classée en deux classes concrètes *IsInstance* et *IsDirectInstance*. *IsInstance* teste si un élément de modèle est une instance directe ou indirecte d'une classe du métamodèle. *IsDirectInstance* renvoie vrai uniquement si la méta-classe directe est celle voulue. En conséquence, la relation suivante est vraie par construction: $IsDirectInstance(class, x) \implies IsInstance(class, x)$.

Test sur les caractéristiques

La classe *OnFeaturePredicate* est une classe abstraite qui représente l'ensemble des tests pouvant être faits sur des caractéristiques des éléments de modèles. Ces tests sont représentés sur la figure 3.14.

Dans le cas d'un attribut, les tests sont faits sur la valeur de cet attribut. Par exemple si le nom est égal à "test", ou si l'attribut entier *num* est égal à 3. Pour

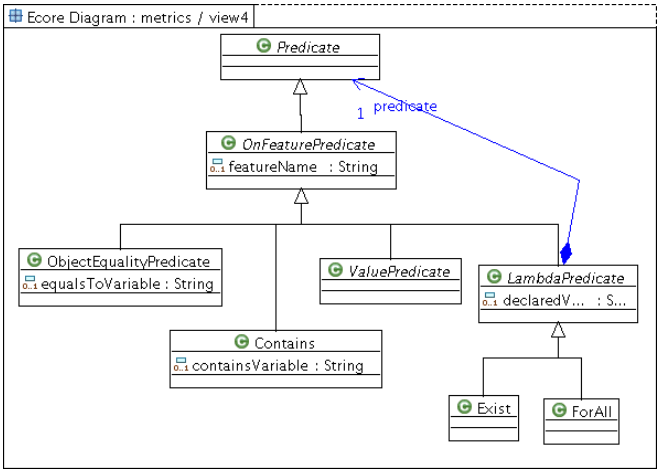

FIG. 3.14 : Test sur les caractéristiques d'un élément de modèle

une référence dont la multiplicité est égale à 1, on peut tester si l'objet référé est égal à un autre. On peut aussi tester si l'objet référé satisfait un sous-prédicat. Pour une référence dont la multiplicité est supérieur à 1, on peut tester si l'ensemble des objets référés contient un élément donné (classe *Contains*), ou contient un élément satisfaisant un prédicat (*Exists*), ou si tous les éléments satisfont un prédicat (*ForAll*).

Pour résumer, les mécanismes de composition des prédicats sont au nombre de deux: la composition de type booléenne, et la composition de type lambda.

3.3.4 Conclusion

Une métrique spécifiée comme instance du métamodèle de métriques est purement déclarative. De même, il n'y a absolument pas de préoccupations d'implémentation dans le modèle.

Nous présentons donc dans la section suivante comment le déclaratif est transformé en un artefact exécutable (les marcheurs), et comment la spécification abstraite est transformée en code Java et finalement en un composant *Eclipse* (un *plugin*).

3.4 Architecture logicielle de l'approche MDM

Nous avons présenté dans la section précédente l'ensemble des concepts du métamodèle de spécification de métriques. Ces concepts sont purement déclaratifs. Nous présentons dans cette section comment est fait le lien entre ces concepts déclaratifs et la mesure effective des modèles.

3.4.1 Le concept de marcheur

Pour mesurer un modèle, il faut définir un algorithme qui visite chacun des éléments du modèle. Pour ce faire, nous introduisons le concept de marcheur (*walker*). Un marcheur visite un modèle suivant une politique de visite encodée dans un algorithme exécutable. Le concept est réifié par la classe abstraite *Walker*. La figure 3.15 montre l'architecture d'un marcheur. Le marcheur a un élément de modèle comme point

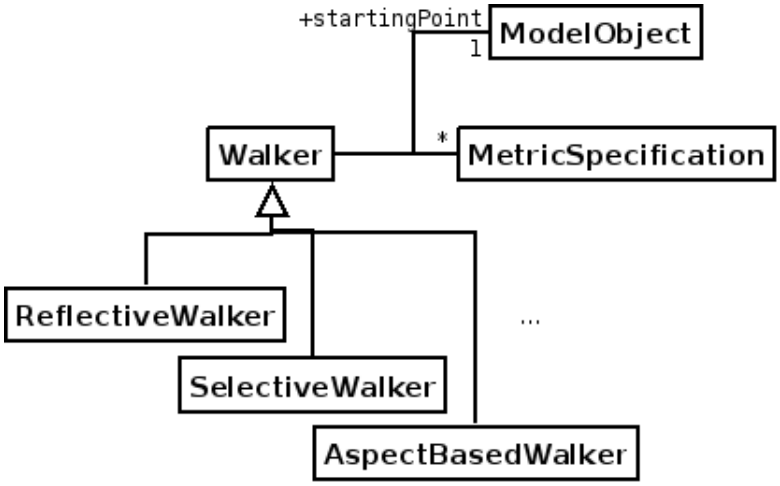

FIG. 3.15 : Marcheur: références et sous-classes

d'entrée. Il connaît un ensemble de spécifications de métriques, qu'il va prévenir à chaque visite d'un nouvel élément du modèle. Il y a plusieurs types de marcheurs, qui sont représentés par les sous-classes de la figure. Chaque classe encapsule un des algorithmes de marcheur, présentés un par un dans les paragraphes qui suivent. Notons que le marcheur n'est pas un concept du métamodèle de spécification de métriques, mais est utilisé dans l'implémentation finale.

Le diagramme de séquence de la figure 3.16 illustre le principe de la marche. Celui de la figure 3.17 illustre celui de la mesure elle même via les interactions entre le marcheur et les spécifications de métriques.

Chaque marcheur (*walker*) encapsule un algorithme de visite des modèles. L'approche est de type boite noire. L'utilisateur final de notre approche, celui qui mesure les modèles et définit des modèles de métriques conforme au métamodèle n'a pas à connaître, ni à comprendre les marcheurs. Un marcheur est choisi par défaut (le *ReflectiveWalker*). Toutefois, l'utilisateur peut au besoin choisir un marcheur particulier pour la mesure des modèles via l'attribut *computationStrategy* de la classe *MetricSpecificationSet*.

Le concept de marcheur répond aux mêmes préoccupations que le patron de conception visiteur [Gamma et al., 1995]. Nous n'avons pas utilisé ce patron pour deux raisons. Premièrement, il n'est généralement pas possible d'impacter le métamodèle de domaine (ou son implémentation) pour qu'il puisse accepter un visiteur. Deuxièmement, l'architecture par marcheur permet d'avoir un algorithme contenu dans une seule entité logique (une méthode pour le *ReflectiveWalker*, un fichier pour le *AspectBasedWalker*). Au contraire, un visiteur éclate la logique de visite dans l'ensemble des méthodes du visiteur et des classes visitées.

L'algorithme de visite des modèles par défaut: le parcours total

L'algorithme du parcours total (dans le *ReflectiveWalker*) visite tous les éléments de modèles accessibles à partir d'un élément racine. Pour ce faire, il visite chacun des éléments liés à l'élément courant, et ce, de manière transitive. Nous avons deux

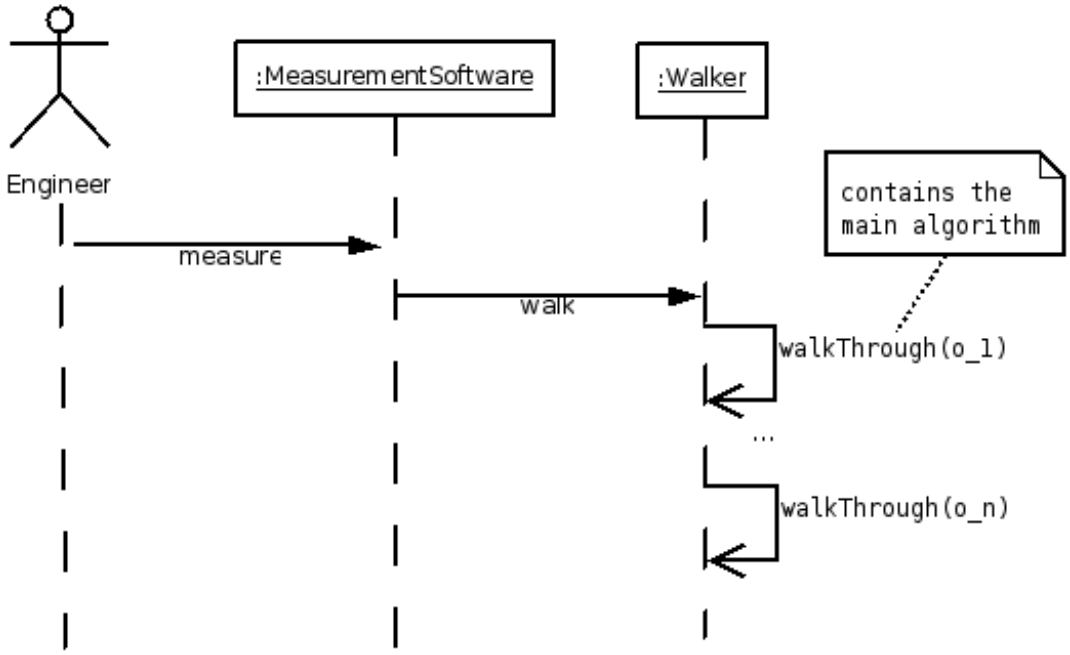

FIG. 3.16 : Principe de la marche

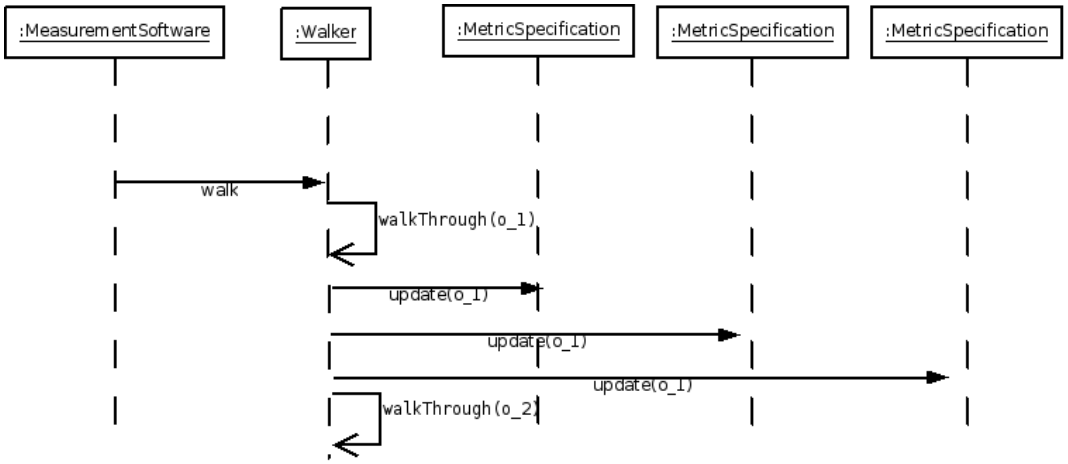

FIG. 3.17 : Interactions entre le marcheur et les spécifications de métriques

implémentations de cet algorithme, l'une par introspection, l'autre par génération de code.

Par introspection Le principe est le suivant. À partir d'un objet, on remonte à sa classe par introspection, puis on récupère la liste de tous ses attributs, que l'on visite un par un. Quand l'attribut est une collection, c'est à dire que sa multiplicité est supérieure à 1, on visite tous les objets de la collection. Le code *Kermeta* de cet algorithme est présenté dans le listing 3.5. Cet algorithme est totalement indépendant du domaine, i.e.; du métamodèle, comme le montre l'absence de type du domaine dans le code *Kermeta*.

Listing 3.5 : Le parcours des modèles par introspection en *Kermeta*

```
class ReflectiveWalker inherits Walker {
operation walkThrough(o : kermeta::standard::Object) : Void is do
  // a-t-on déjà visité l'objet
  if visitThisObject(o)
  then
    markThisObject(o) // marque l'objet comme visité
    // met à jour la valeur des métriques
    metrics.each{ x| x.update(o)}
    o.getMetaClass.typeDefinition
    .asType(ClassDefinition).allAttribute.each{ p |
      // ne visite pas les caractéristiques dérivées
      // ni les types primitifs (EDatatype en ecore)
      if not p.isDerived and ParameterizedType.isInstance(p.type)
      then
        // test sur la borne sup de la multiplicté
        if p.upper == 1
        then
          walkThrough(o.get(p))
        else
          var list : Collection<kermeta::standard::Object>
          list ?= o.get(p)
          list.each{t | walkThrough(t) }
      end
    end
  } // end foreach .allAttribute
  end
end // end operation
}// end class
```

Par génération de code et aspects L'implémentation de l'algorithme de parcours total peut se faire en utilisant la génération de code et le mécanisme d'aspect de *Kermeta*. Le but de cette implémentation est de supprimer les appels à l'introspection qui sont très coûteux en performance.

L'idée est d'implémenter le parcours du modèle par aspect sur le métamodèle. Ce point est illustré dans le listing suivant par la visite des modèles d'architecture logicielle par aspect.

Listing 3.6 : Le parcours des modèles par aspect en Kermeta

```
1  package mdm_example;
2
3  require "example-software-architecture.ecore"
4  require "../../kmt/lib/lm2.kmt"
5
6
7  class MarkableObject {
8  attribute isMarked : kermeta::standard::Boolean
9  operation visitMe() : kermeta::standard::Boolean is do
10   if self.isMarked == void or self.isMarked == false then
11       self.isMarked := true
12       result:=true
13   else
14       result:=false
15   end
16 end
17 }
18
19 @aspect "true"
20 class Component inherits MarkableObject {
21
22 operation computeMetrics(s:metrics::MetricSpecification):Void
23 is do
24   /**********************
25   CHAQUE METRIQUE DE s EST MIS A JOUR ICI
26   **********************/
27   if self.visitMe then
28     s.update(self)
29     self.containerComponent.computeMetrics(s)
30     self.provided.each{ x | x.computeMetrics(s)}
31     self.required.each{ x | x.computeMetrics(s)}
32   end
33 end
34 }
35
36 @aspect "true"
37 class CompositeComponent {
38 method computeMetrics(s:metrics::MetricSpecification):Void is do
39   if self.visitMe then
40     super(s)
41     self.subComponents.each{ x | x.computeMetrics(s)}
42   end
43 end
44 }
45
46 @aspect "true"
47 class Interface inherits MarkableObject  {
48 operation computeMetrics(s:metrics::MetricSpecification):Void
49 is do
50   if self.visitMe then
51     s.update(self)
```

```
52    end
53  end
54  }
```

On remarque que l'implémentation de ces marcheurs par aspect est hautement répétitive, nous avons donc écrit un générateur de marcheur, qui prend un métamodèle de domaine en entrée, et génère le marcheur par aspect correspondant. Notons, que l'appel à la méthode *update* de *MetricSpecification* n'est fait qu'à la tête d'unr hiérarchie de classe, ici *Component*, pour ne pas mettre à jour une métrique deux fois sur le même objet. Notons aussi qu'il est nécessaire d'appeler la méthode de la classe mère ($super(s)$) afin de parcourir les références des classes mères.

L'algorithme de parcours suivant la relation de contenance

L'algorithme de parcours suivant la relation de contenance est une variation de l'algorithme de parcours total. Il consiste à ne visiter que les éléments de modèles contenus, au sens de la composition *EMOF*, par l'élément de modèle courant. Cet algorithme peut aussi être implémenté par introspection ou par génération de code et aspect. Dans le cas de l'introspection, cela consiste à ajouter le test de contenance avant de parcourir une référence [4]. Dans le cas de la génération de code, cela consiste à ne pas considérer ces références.

Cet algorithme possède deux propriétés intéressantes. Premièrement, il permet de s'affranchir des tests assurant qu'un objet n'est pas visité deux fois, car par construction un élément de modèle n'est contenu que par un seul élément au maximum. Deuxièmement, quand on considère le fichier comme unité de modélisation, cet algorithme garantit que l'on ne mesure qu'un seul fichier, car EMF implémente la contenance comme des sous-noeuds XML dans les modèles XMI.

L'algorithme de parcours paramétré

L'algorithme de parcours paramétré consiste à suivre uniquement les références spécifiées dans une liste. C'est donc aussi une variation de l'algorithme du parcours total. Dans l'exemple du listing 3.6, cela consiste à ajouter un test entre les lignes 9 et 12:

```
1  if allowedReferences.contains(p.name) then
2  // parcours des objets liés à l'objet
3  // courant par la référence p
4  end
```

Le marcheur correspondant à cet algorithme est utilisé par et uniquement par les sous classes de *PathBasedMetricSpecification*, i.e.; *SetOfEdgesPerX, SetOfElementsPerX* et *PathLengthMetricSpecification*. En effet, la liste de références associées à ces métriques est directement traduite en une instance d'un marcheur à parcours paramétré, dont le paramètre est cette liste des références à suivre.

[4] `if p.isComposite then` ... en *Kermeta*

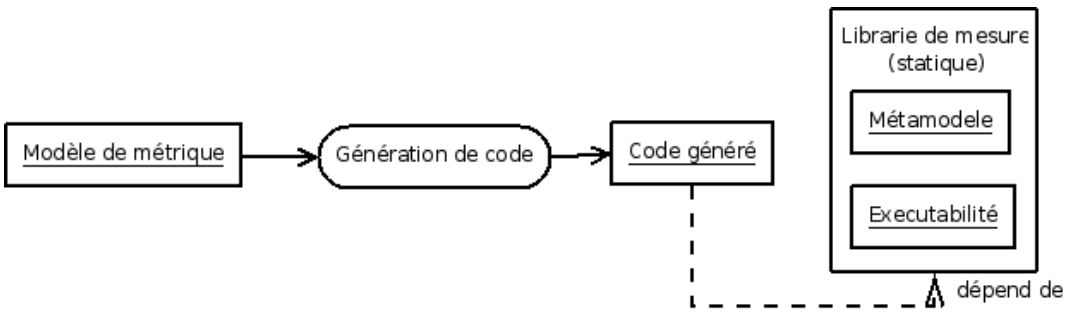

FIG. 3.18 : Principe du métamodèle comme librairie

3.4.2 La chaîne de génération de l'outil de mesure de modèles

> "The tool [...] automatically generates the measurement environment for the corresponding language" [5] Tsalidis, 1990 [Tsalidis et al., 1992]

Dans cette section, nous présentons les principes directeurs guidant la transformation d'un modèle de métriques en outil de mesure exécutable.

Principe du métamodèle comme librairie

Tout d'abord, l'exécutabilité des concepts du métamodèle est assuré par le principe du métamodèle comme librairie présentée sur la figure 3.18. Il consiste à ajouter des méthodes exécutables au métamodèle de spécification de métriques. L'ensemble métamodèle et exécutabilité forme une librairie de mesures. Le générateur de code principal prend en entrée un modèle de métriques et génère du code qui s'appuie sur la librairie. Ce principe conduit à une exécutabilité totale.

Ce principe architectural est un hybride des principes de compilation et de métamodélisation exécutable [Muller et al., 2005].

Différence par rapport au principe de compilation: Le principe de compilation consiste à prendre en entrée un modèle (historiquement nommé un AST) et à générer le code complètement exécutable. Le code généré n'a plus aucun lien explicite avec l'AST et a fortiori avec des méthodes de l'AST. La différence est que le code généré s'appuie fortement sur le métamodèle lui-même.

Différence par rapport au principe de métamodélisation exécutable: La métamodélisation exécutable consiste à exécuter directement un modèle en rapport avec la sémantique codée dans les méthodes ajoutées aux classes du métamodèle. Dans notre cas, le modèle est transformé en code, et n'existe donc plus dans l'artefact exécutable final.

Le principe hybride du métamodèle comme librairie a deux avantages: d'une part, il s'avère plus rapide à l'exécution par rapport à la métamodélisation exécutable, d'autre part, il produit du code plus clair et plus concis qu'avec une génération de code totale comme avec la compilation. Ce point est illustré dans le listing suivant 3.7.

[5] L'outil génère automatiquement l'environnement de mesure pour le langage correspondant.

FIG. 3.19 : Principe de la transformation d'une instance en sous-classe du métamodèle

Listing 3.7 : Avantage du principe de métamodèle comme librairie

```
1  // code écrit en Kermeta: métamodèle exécutable comme librairie
2
3  // cette classe hérite d'une classe du métamodèle
4  // (principe métamodèle comme librairie)
5  class NOAC inherits SigmaMetric {
6  // cette méthode est appelée par polymorphisme (librairie)
7  method phi(this: Object) : Boolean is do
8    // le prédicat est compilé
9    // il apparait explicitement (principe de compilation)
10   // il plus efficace que s'il était exécuté
11   result:= this.getMetaClass.name == "Component"
12            and (this.asType(Component).required.size > 10)
13            and (this.asType(Component).provided.size > 10)
14            and (this.asType(Component).subComponents.size > 5)
15 end // end method
16
17 } //end class
18
19 ....
20
21 // on instancie une classe du métamodèle
22 // et on appelle une méthode
23 // (principe de métamodélisation exécutable)
24 // plus clair et concis qu'avec une génération complète
25 var ms : MetricSpecification init NOAC.new
26 ms.update(o)
```

Principe de la transformation d'une instance en sous-classe

Afin d'utiliser le métamodèle comme librairie (au sens de *framework*), il faut créer des classes sous-classant celles de la librairie. Cette tâche est effectuée automatiquement par génération de code en transformant des éléments du modèle de métriques et sous-classe du métamodèle. D'un point de vue abstrait, le principe de la transformation d'une instance en sous-classe consiste à transformer une instance d'une classe du métamodèle en une sous-classe; i.e. à passer la frontière modèle-métamodèle. Ce point est illustré sur la figure 3.19. Ainsi, l'instance de *SigmaMetric* est transformé en

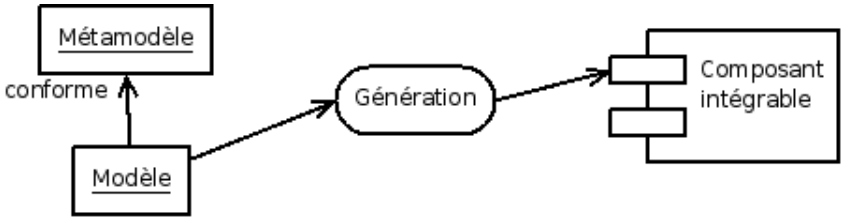

FIG. 3.20 : Principe de génération d'un composant logiciel à partir d'un modèle

sous-classe de *SigmaMetric*, tandis que le prédicat associé est transformé en méthode de cette sous-classe.

Ce principe a deux avantages significatifs. D'une part, il met bien les métriques du domaine au premier plan en les réifiant chacune comme classe. Les métriques apparaissent ainsi de manière explicite dans le code. D'autre part, il est une solution technique au principe précédent, et permet de combiner métamodélisation exécutable et génération de code.

Notons que ce principe est aussi celui d'EMF lors de la génération de code à partir d'un métamodèle. Il y a transformation d'une classe *Ecore* (instance de *EObject*) en sous-classe Java de la classe *EObject*.

Principe de la génération d'un composant logiciel à partir d'un modèle

Enfin, l'approche MDM a pour but de générer un outil de mesure directement utilisable et intégré dans un environnement de modélisation. Comme nous considérons les environnements de modélisation générés par EMF (et les outils associés), nous devons générer un composant intégrable dans *Eclipse*, vu comme une plateforme d'intégration de composants. Nous avons donc suivi le principe de la génération d'un composant logiciel à partir d'un modèle [Kieburtz et al., 1996] (par opposition à la génération d'un logiciel indépendant, dit *stand-alone*). Ce principe est celui qui dirige la génération des éditeurs de modèles dans EMF ou *Topcased*. Un composant *Eclipse*, appelé un *plugin*, est une entité physique (un fichier *Jar* ou un répertoire), regroupant le code exécutable, les méta-données d'intégration et de déploiement, et les ressources annexes (cf. [Clayberg et Rubel, 2006]).

Dans le cas de l'approche MDM, un composant *Eclipse* (de type *plugin*) est généré à partir d'un modèle de spécification de métriques. À notre connaissance, nous sommes les premiers à utiliser ce principe dans le cadre de la mesure.

Pratiquement, nous utilisons deux mécanismes pour générer le composant *Eclipse* de mesure: la génération de code Java et la copie de fichiers. Des mécanismes secondaires propres à *Eclipse* sont aussi utilisés pour créer un projet Java, des paquetages, et pour compiler le code Java généré.

La génération de code est utilisée pour traduire des informations du modèle de spécification de métrique en code exécutable afin de pouvoir mesurer effectivement les modèles de domaine. Le langage de destination est Java car il s'agit du langage natif d'*Eclipse*. Le générateur crée une classe Java par métrique et utilise abondamment le polymorphisme. Ces deux points sont illustrés dans le listing 3.8.

Listing 3.8 : Code de génération de code

```
public  String generateContents() throws Exception  {
        String content ="";

        // génération d'une classe Java par métrique
    // langage de destination dans les chaînes de caractères
        content+="\n\tprivate class " + this.getIdentifier()
              + " extends "+this.eClass().getName()+"Impl"+" {\n";

        // corps de la classe par utilisation du polymorphisme
        content += generateMethods();

    // fin de la génération de la classe
    content+="} // end class";
}
```

Le code généré étend des classes du métamodèle qui est utilisé comme une librairie. Le code généré utilise donc aussi le polymorphisme. Le listing 3.9 montre un prédicat compilé en une méthode. Cette méthode est appelée par polymorphisme à partir des classes de la librairie

Listing 3.9 : Code généré: compilation du prédicat de 3.1 en méthode Java.

```

// extension de la classe SetOfElementsPerXImpl de la librairie.
private class NIIPC extends SetOfElementsPerXImpl {

// cette méthode est générée
// et appelée par polymorphisme
public boolean phi(EObject arg) {
        EObject self = arg;
        if (self == null)
                return false;
        boolean p0 = false;
        boolean noiai = false;
        // is it this class ?
        noiai = self.eClass().getName().equals("Interface");
        // one of its superclasses ?
        for (EClass ec : self.eClass().getEAllSuperTypes()) {
                if (ec.getName().equals("Interface"))
                        noiai = true;
        }
        p0 = noiai;
        return p0;
}
```

La copie de fichiers est utilisée pour insérer des artefacts non-variables dans le composant final. Les deux classes spécifiant l'action et la vue *Eclipse* de mesure sont de ce type. De même, les méta-données du composant (les fichiers *plugin.xml* et *Manifest.MF*) sont aussi copiées.

3.4.3 Solutions aux exigences d'un outil de mesure

Nous avons présentés en 3.1.3 les exigences principales que doit satisfaire un outil de mesure. Nous donnons dans cette section ce en quoi notre solution les satisfait.

Mesure automatique Le composant *Eclipse* généré est un fichier *Jar* qui peut être appelé en ligne de commande. La mesure automatique peut donc être effectuée automatiquement par un programme de type *crontab*. Notons que l'outil de mesure produit un fichier de résultats de mesures facilement manipulable par un autre logiciel.

Intégration dans un environnement de modélisation L'outil de mesure généré s'intègre pleinement dans *Eclipse*, et avec tous les composants *Eclipse* de modélisation. Il consiste en une vue (au sens *Eclipse*) dédiée à la mesure des modèles. De plus, le composant de mesure des modèles est capable, grâce aux mécanismes *Eclipse*, de mesurer les modèles à chaque changement au niveau du disque (i.e.; à chaque enregistrement). Ce dernier mécanisme permet donc une mesure totalement transparente pour l'utilisateur final.

Modifiable L'outil de mesure étant totalement généré, il est possible de modifier le modèle de métriques autant de fois que nécessaire, par exemple lors d'un changement léger du métamodèle de domaine, ou d'une amélioration des métriques métiers.

Passage à l'échelle Notre approche a été implémentée dans deux langages différents. Toute la phase de prototypage a été faite en *Kermeta*. La deuxième implémentation est en Java et permet de mesurer des modèles de grande taille. Nous avons été capables de mesurer des modèles de plusieurs centaines de milliers d'éléments de modèles, ce que nous montrerons dans le chapitre validation de cette thèse.

Suivant l'approche de bout en bout, et notre problème-exemple, à savoir la mesure des modèles d'architecture logicielle, nous sommes en mesure de générer l'outil de mesure des modèles, dont une capture d'écran est montrée en figure 3.21.

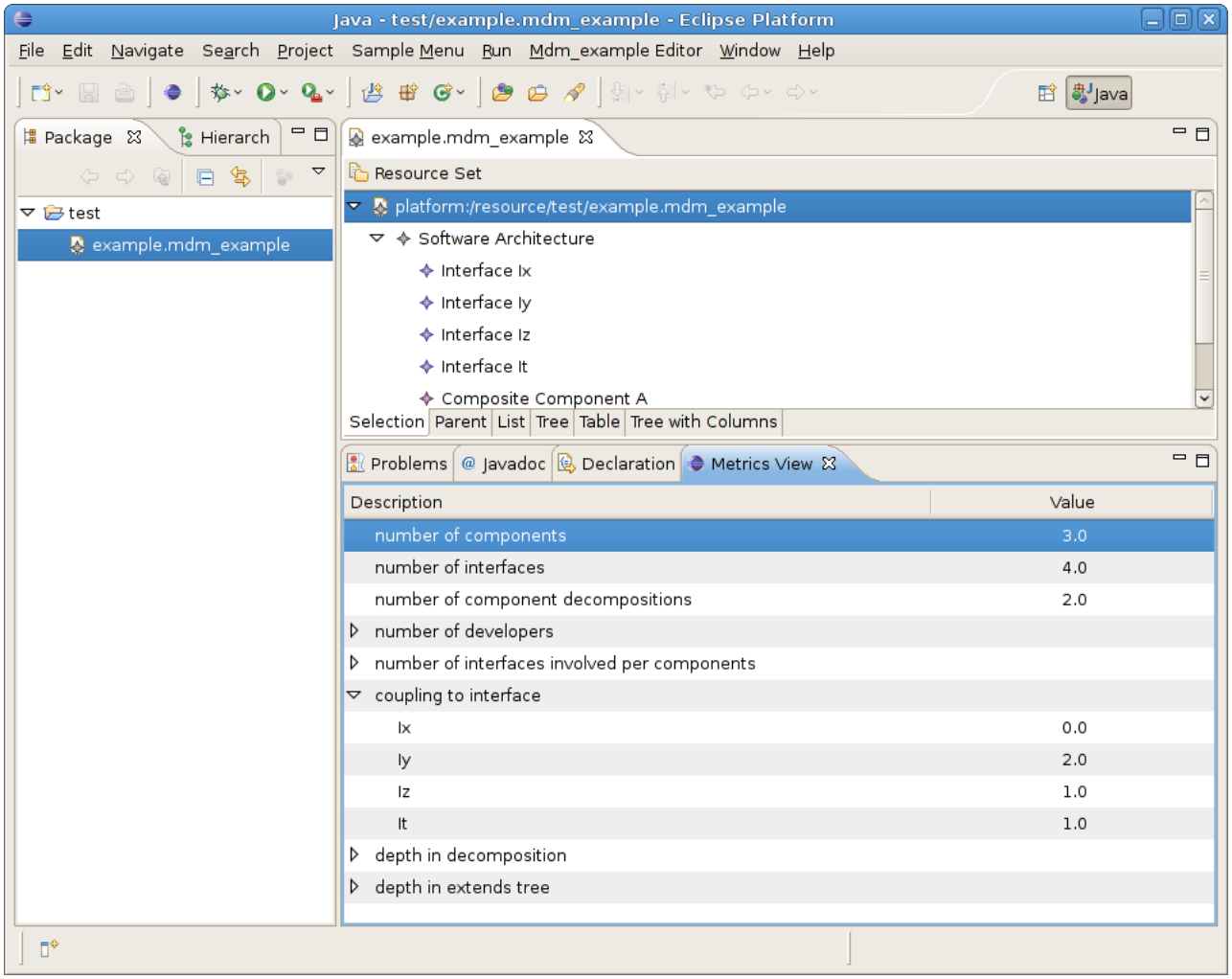

FIG. 3.21 : Outil de mesure généré et intégré dans *Eclipse*

3.4.4 Vue globale de l'architecture logicielle de l'approche MDM

Nous récapitulons l'architecture logicielle de notre approche de la mesure des modèles dans la figure 3.22. Celle-ci montre les composants logiciels mis en oeuvre. Le modèle de spécification de métrique, en haut de la figure, est utilisé pour générer le composant de mesure. Ce composant de mesure comprend le métamodèle, son exécutabilité, les marcheurs, et le code généré. Il s'intègre dans une plate forme [6] avec d'autres composants, dont ceux de création, de modification et d'analyse des modèles de domaine.

3.5 Analyse de la contribution

Dans cette section, nous effectuons une analyse de notre contribution par rapport à un ensemble de critères. Ces critères sont majoritairement ceux de la section 2.5 de l'état de l'art. Les critères supplémentaires viennent de deux sources, soit des arguments secondaires de certains articles de l'état de l'art, soit d'autres publications qui mentionnent un point pouvant éclairer notre contribution. Pour des raisons de lisibilité, nous n'avons mentionné pour chaque critère d'analyse, qu'une seule référence bibliographique. Ces critères sont divisés en cinq catégories: l'environnement de mesure, la validation des métriques, les liens avec le processus de développement, l'orientation IDM de la solution à la mesure des modèles et enfin l'application à différents domaines d'application.

3.5.1 Environnement de mesure

Intégration dans un environnement de développement [Kim et Boldyreff, 2002]

Dans le cas de l'ingénierie dirigée par les modèles, l'environnement de développement est l'environnement de modélisation. Notre approche de la mesure est totalement intégrée à l'environnement de modélisation.

Intégration dans un outil d'analyse des modèles [Mendelzon et Sametinger, 1995]

Dans le cas de l'ingénierie dirigée par les modèles, et contrairement aux approches centrées sur le code source, l'analyse et la génération de code sont faites à partir des mêmes modèles. Les outils d'analyse des modèles sont majoritairement intégrés dans les environnements de modélisation eux-mêmes. Dans cette perspective, notre approche est intégrée dans un environnement d'analyse, et plus exactement, fournit un outil supplémentaire à l'environnement de modélisation.

Efficacité de la mesure [Beyer et al., 2005]

L'efficacité de la mesure dépend de trois facteurs: des librairies utilisées, du langage cible de la génération et des algorithmes de parcours des modèles (les marcheurs). Ces deux derniers facteurs nous ont amenés à faire les modifications suivantes. D'une part, l'algorithme du parcours total fait un usage extensif de l'introspection, ce qui

[6] dans notre cas *Eclipse*, mais nous verrons en 3.5 que l'approche est ouverte

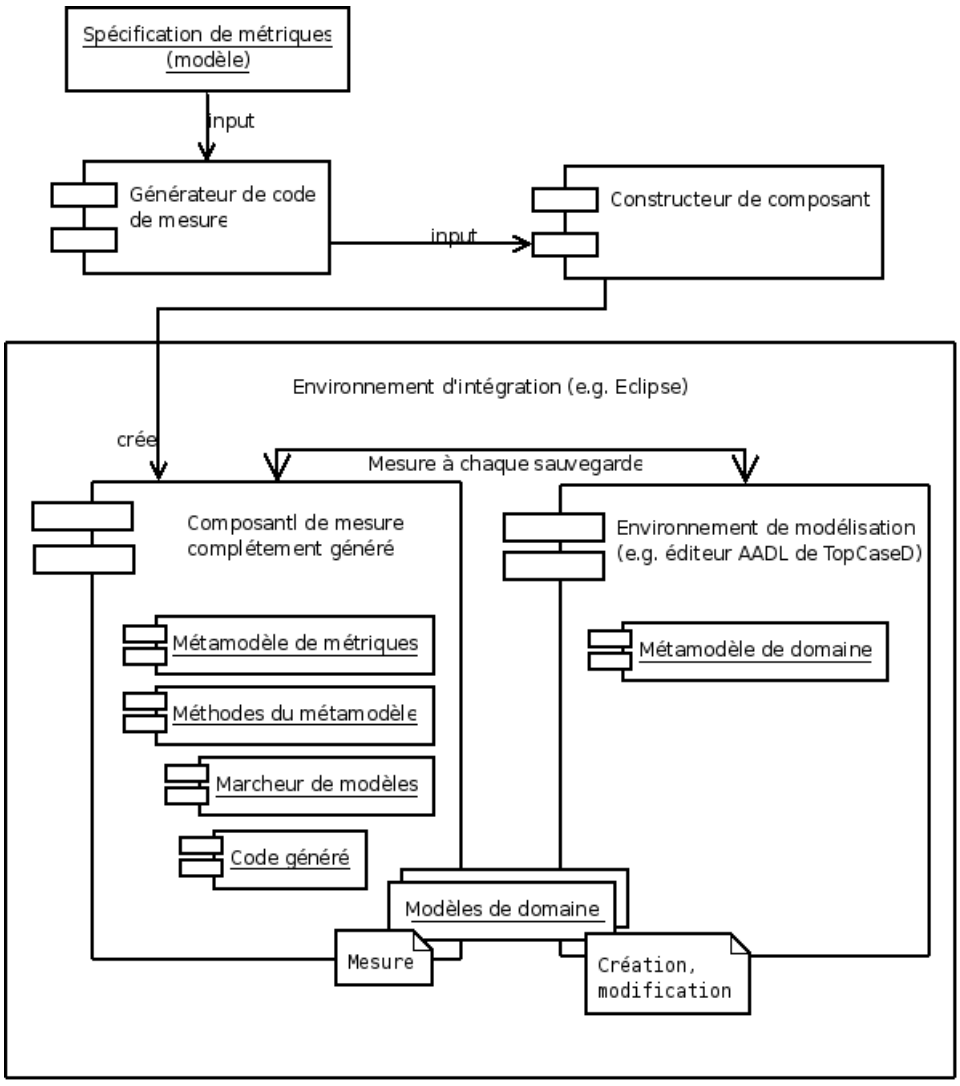

FIG. 3.22 : Vue globale de notre contribution: composants

a un coût significatif à l'exécution. Nous avons donc proposé un parcours des modèles par aspects (cf. 3.6). D'autre part, le prototype de l'approche MDM, écrit en *Kermeta*, langage interprété, s'avère trop lent pour la mesure des gros modèles. Pour maximiser l'efficacité, nous avons implémenté un générateur de code qui génère du Java. Finalement, l'implémentation de l'approche MDM reste tributaire de l'efficacité de la librairie EMF, mais nous avons pu mesurer des modèles de plusieurs centaines de milliers d'éléments (cf. 4.2).

Automatisation de la mesure [Lavazza et Agostini, 2005]

L'automatisation de la mesure fait l'objet de deux points de notre contribution. D'une part, l'outil de mesure généré mesure les modèles de domaine à chaque enregistrement des modèles, de manière non intrusive pour l'utilisateur. D'autre part, le composant généré peut être utilisé en mode *batch* en ligne de commande.

Adaptabilité de l'approche [Tsalidis et al., 1992]

Comme l'outil de mesure est à 100% généré, notre approche permet un maximum de souplesse dans la modification et l'adaptation des modèles de spécifications de métriques. Ceci est valable dans la limite du pouvoir d'expression du métamodèle de spécification de métriques. Dans le cas où la métrique de domaine ne peut pas s'exprimer telle quelle, il est possible de modifier le code généré, ou d'utiliser la partie librairie de notre approche (cf. 3.4.2).

Notons aussi que la spécification des métriques repose sur MOF mais est indépendante de la plateforme finale. L'approche est donc suffisamment ouverte pour générer un composant de mesure dans un autre environnement de modélisation ou d'analyse (e.g.; *Atom3* [de Lara et Vangheluwe, 2002]) à partir de la même spécification de métriques.

Génération de rapports de mesure [Vépa et al., 2006]

L'approche MDM permet d'afficher les résultats de la mesure dans une vue *Eclipse*, totalement intégrée dans l'environnement de modélisation. Cela donne la possibilité à l'utilisateur d'intégrer les résultats de la mesure de façon quasi-transparente dans le processus de création et de modification des modèles de domaine. De même, l'outil généré, appelé en ligne de commande, peut générer un rapport de mesure. La génération de vues graphiques des résultats de mesure, et la prise en compte de format dédié à l'homme (e.g.; un rapport de mesure au format PDF) sont en dehors du périmètre de cette thèse.

Artefacts à mesurer faciles à collecter [Card, 1991]

Card[7] note que, pour qu'une mesure soit effectivement utilisée dans la pratique industrielle, il faut que les artefacts à mesurer soient faciles à collecter. Notre approche a pour but premier de mesurer des modèles de l'IDM. Dans un développement IDM, les modèles sont centraux et accessibles par définition (presque ontologiquement). Le critère de Card est donc satisfait.

[7] cité par [Henderson-Sellers, 1996, p. 37-38].

Facilité de définition des mesures [Jones, 1991]

Ce critère est assez subjectif. De manière relativement objective, notre approche de la mesure est totalement déclarative, ne nécessite pas de connaissance en programmation, et supprime la boucle entre l'expert du domaine et le développeur car étant totalement générative.

Syntaxe graphique [Mendelzon et Sametinger, 1995]

Nous n'avons pas défini de syntaxe graphique pour la spécification des métriques. La définition d'une syntaxe textuelle ou graphique proche du processus cognitif de modélisation et des représentations de l'expert du domaine est en dehors du périmètre de cette thèse. Nous reviendrons sur ce point dans la section 5 de cette thèse.

3.5.2 Validation

Mesure validée empiriquement [Card, 1991]

Card souligne la nécessité de valider empiriquement les métriques par rapport à des attributs du logiciel. Nous partageons son point de vue. Notre approche fournit l'outil pour mener à bien les études empiriques de validation dans un domaine précis.

Mesure validée théoriquement [Schneidewind, 1992]

Schneidewind a largement souligné l'importance de la validation théorique des métriques [Schneidewind, 1992].

Échelle. Toutes les métriques de l'approche MDM sont de type *nombre de* (dû au caractère fondamentalement discret de la modélisation IDM). En conséquence, elles ont toutes une échelle absolue. D'après Fenton, toutes les analyses arithmétiques sont permises et fondées [Fenton, 1991] et d'après Henderson-Sellers , l'échelle absolue permet d'appliquer la gamme complète des statistiques descriptives [Henderson-Sellers, 1996].

Erreur de mesure. L'approche MDM est valide et fiable au sens de Kan [Kan, 1995, p.70-77]. La mesure donne toujours le même résultat, sans marge d'erreur (fiabilité) du fait du caractère déterministe de l'outil informatique de mesure. La mesure donne un résultat exact (validité) sous l'hypothèse que le générateur de code et la librairie sont exempts de défauts d'implémentation (*bug*).

Type. L'approche MDM ne peut être directement analysée dans un cadre de validation tel celui de Briand et al. [Briand et al., 1996]. En effet, les types de métriques du métamodèle MDM sont génériques, et n'ont donc pas de contexte direct d'application et d'interprétation. En fonction du domaine d'application et de la sémantique des éléments de modèles mesurés, nous faisons l'hypothèse que l'approche MDM permet de spécifier des métriques de taille, de longueur, de complexité, de cohésion ou de couplage, selon le sens de Briand.

Partie	Taille	Cocomo
Outil total	7 KLOC	18 PM
Code généré par EMF total	4K LOC	10 PM
Code de la librairie	2.5 KLOC	6 PM
Code généré par MDM	0.5KLOC	1.2 PM
Modèle de métriques	62 LOC 111 EM	

TAB. 3.2: Taille d'un outil de mesure

3.5.3 Processus

Mesures obtenues à temps [Card, 1991]

Les métriques ont une durée de validité finie et doivent être obtenues à temps pour que l'on puisse tirer parti de leur pouvoir informatif [Card, 1991]. Notre approche satisfait ce critère de deux manières. D'une part, l'outil de mesure est obtenu rapidement par génération de code, d'autre part, les résultats de mesures sont fournis en temps réel dans l'environnement de modélisation.

Mesure ressentie comme importante [Grady, 1992]

Un argument pragmatique formulé par Grady et al. [Grady, 1992] est que les métriques ont un impact d'autant plus grand sur le processus de développement qu'elles sont ressenties comme importantes et porteuses de valeur par les acteurs du processus. Notre approche n'apporte aucune contribution sur ce point.

Liaison de chaque mesure à but d'entreprise [Grady, 1992]

De même, notre approche n'est pas une approche de type processus de mesure comme par exemple l'approche GQM [Basili et al., 1994]. La liaison de la mesure à un but d'entreprise est donc en dehors du périmètre de notre contribution.

Coût du logiciel de mesure [NASA Software Program, 1995]

L'aspect génératif de notre approche a pour but de réduire les coûts de développement d'un outil de mesure des modèles. En effet, l'approche MDM permet de générer un outil de mesure complet et intégrable à l'environnement de modélisation.

Dans quel mesure l'approche MDM est-elle plus productive qu'un développement manuel ? Notons pour commencer qu'il est en dehors du périmètre de cette thèse de mener une expérience contrôlée de validation de ce critère.

La librairie de mesure et le mécanisme de génération de composants encapsulent à la fois le savoir-faire de la mesure des modèles et la complexité accidentelle liée à son implémentation. La définition des métriques avec cette librairie permet de se concentrer uniquement sur l'essentiel. De ce point de vue, l'approche MDM comme librairie permet un gain de productivité au même titre que toute librairie spécialisée.

Ensuite, prenons l'exemple de la mesure des modèles *Ecore* (cf. section 4.6). Le tableau 3.2 montre la taille[8] des différents constituants de l'outil de mesure et le coût[9] correspondant par le modèle Cocomo. D'une part, on remarque que la partie directement liée à la définition des métriques (code généré par MDM) représente une faible proportion de l'outil total. Nous retrouvons là l'argument de la section 3.1.3: la complexité d'un outil de mesure est majoritairement accidentelle. D'autre part, si l'on se concentre sur la spécification des métriques, le fichier XMI de spécifications des métriques fait 111 éléments de modèles EM, soit 62 LOC physiques, alors que le code généré fait 500 LOC. Tout en sachant que l'on compare des choses différentes, il semble que la structuration donnée par le métamodèle de spécification de métriques permette de spécifier celles-ci de manière plus concise qu'avec du code Java.

Par ces arguments qualitatifs et ces considérations quantitatives, nous pensons que le gain de productivité de l'approche MDM est réel. Dans ce cas, notre approche permettrait de garder en rapport constant les coûts de développement de l'IDM (cf. 3.1) et de satisfaire le critère de la NASA sur la place des coûts liés à la mesure dans la somme des coûts d'un projet logiciel [NASA Software Program, 1995, p. 32].

Génération de l'outil de mesure [Guerra et al., 2007]

Notre approche génère totalement un outil de mesure à partir d'un modèle de métriques spécifié par un métamodèle. C'est là le coeur de notre contribution.

Mesure tôt dans le cycle de vie [Jones, 1991]

Jones montre l'importance de définir et d'obtenir des mesures tôt dans le cycle de vie. Ce point est constitutif de notre problématique. Nous avons posé dans la définition du problème en section 3.1 la nécessité d'avoir une approche indépendante du domaine et du cycle de vie. Par exemple, et c'est un cas d'application du chapitre suivant, notre approche permet de mesurer des modèles d'exigences. L'approche MDM satisfait le critère de Jones.

3.5.4 Orientation modèle de la solution

DSL pour la mesure [SDMetrics, 2006]

L'approche MDM définit un langage dédié à la mesure des modèles. Nous continuons naturellement directement sur le point suivant.

DSL pour la mesure basé sur un métamodèle [Guerra et al., 2007]

Ce langage est basé sur un métamodèle de spécification de métriques. Notons que nous avons suivi une approche de modélisation pure. De manière temporelle, nous avons commencé par définir le métamodèle (au contraire des approches qui commencent par définir une syntaxe, avec par exemple, une grammaire BNF). De manière logique, le métamodèle de spécification de métriques reste prépondérant,

[8] LOC: ligne de code, EM: élément de modèle
[9] PM: homme/mois

comme l'a montré l'exposition de notre contribution, la syntaxe restant en second plan, au stade de proto-syntaxe.

Artefacts mesurés spécifiés par un métamodèle [Lanza et Ducasse, 2002]

La nécessité de spécifier les artefacts à mesurer par un métamodèle est constitutive de notre approche. D'un point de vue pratique, sont pris en compte exclusivement des modèles spécifiés par un métamodèle. Du point de vue des principes fondateurs de l'approche, les modèles de spécifications de métriques réfèrent explicitement aux concepts du métamodèle de domaine (cf. figure 3.3).

3.5.5 Domaine d'application

Mesure d'un langage de programmation [Alikacem et Sahraoui, 2006]

La mesure d'un langage de programmation constituera un cas d'application dans le chapitre suivant.

Mesure d'une famille de langages de programmation [Eichberg et al., 2006]

Notre approche permet la mesure d'une famille de langages de programmation. Il faut alors spécifier le métamodèle commun, et mettre en place les mécanismes de transformation des codes source écrits dans des langages différents.

Mesure des modèles UML [Lavazza et Agostini, 2005]

Les modèles UML pourraient être mesurés avec l'approche MDM. Dans ce cas, le métamodèle de domaine est le métamodèle UML.

Mesure de n'importe quel DSL [Guerra et al., 2007]

Enfin, et nous reprenons l'argument de la section *Mesure tôt dans le cycle de vie*, nous avons posé dès la définition de notre problème le besoin de mesurer n'importe quel modèle de n'importe quel DSL. En cela, notre contribution est avec celle de Guerra et al., la première sur le sujet.

Conclusion

Nous avons présenté en détail l'approche MDM pour la mesure des modèles. Nous avons montré en quoi cette approche est déclarative et comment est obtenue sa complète exécutabilité. En fait, l'approche MDM est une réalisation de la vision IDM appliquée au problème de la mesure des modèles. Nous avons ensuite analysé les propriétés de l'approche grâce à différents éclairages issus de la littérature. Il en ressort que l'approche MDM est une synthèse des travaux précédents.

4 Validation de l'approche

Ce chapitre a pour but de valider l'approche MDM. Pour ce faire, nous allons exprimer avec l'approche MDM plusieurs métriques dans plusieurs contextes différents. Chaque cas d'étude apportera un éclairage différent sur les propriétés de l'approche MDM.

4.1 Méthode de validation

Plusieurs points ont guidé la validation de l'approche MDM. Nous présentons ici la méthode suivie.

4.1.1 Mesurer des modèles

Bien entendu nous allons mesurer des modèles, et ce, par rapport à des métamodèles clairement identifiés. Pour illustrer la généricité de notre approche quels que soient les domaines ou les moments du cycle de vie, nous allons mesurer des modèles de différents domaines, et obtenus à différents moments du cycle de vie.

4.1.2 Montrer l'indépendance par rapport au domaine

Les domaines considérés seront: le logiciel exécutable, l'architecture logicielle et matérielle des systèmes embarqués, les exigences, et la patrouille maritime. Ces domaines sont représentés avec autant de métamodèles totalement indépendants les uns des autres et sans recoupements. Ces études de cas illustreront l'indépendance de notre approche par rapport au domaine, i.e. sa généricité.

4.1.3 Montrer l'indépendance par rapport au cycle de vie

La liste des domaines (cf. 4.1.2) peut être aussi être interprétée comme une preuve de l'indépendance par rapport au cycle de vie. Cela est illustré sur la figure 4.1. Notre cas d'études sur la mesure des exigences à base de modèles est une application de notre approche au tout début du cycle de vie d'un produit. Le cas d'étude sur

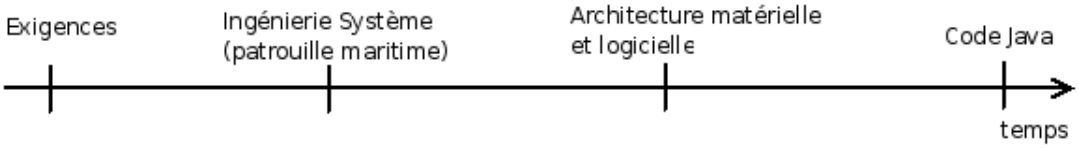

FIG. 4.1 : Cas d'étude montrant l'indépendance de l'approche MDM par rapport au cycle de vie

les systèmes de patrouille maritime se situe à l'étape d'après, à un niveau habituellement appelé ingénierie système, le cas d'étude sur la mesure des modèles AADL (*Architecture Analysis & Design Language*) se situe en phase d'architecture logicielle pour les systèmes embarqués. Enfin, le cas d'étude sur la mesure des programmes Java considérés en tant que modèles se situe à la toute fin du développement d'un produit logiciel.

4.1.4 Identifier et éviter les biais

La validation de notre approche consiste en l'expression de métriques. Nous avons identifié deux biais à la validation et avons essayé de les minimiser.

Le premier biais vient du choix des métriques à exprimer avec notre approche. En définissant nous-mêmes nos métriques, ce biais provient de la possibilité de définir des métriques qui s'expriment dans notre métamodèle presque par construction. Nous avons donc suivi la règle suivante: toutes les métriques considérées proviennent de la littérature ou d'outils existants.

Le second biais vient des métamodèles considérés. Si nous validons notre approche avec des métamodèles de notre cru, une critique recevable pourrait être que les métamodèles ont été construits, consciemment ou non, de manière à supporter la mesure avec notre approche. Pour réduire ce biais autant que possible, les cas d'étude AADL et Java sont faits à partir de métamodèles dont nous ne sommes pas les auteurs.

Nous n'avons pas pu considérer des métamodèles existants pour les cas d'études sur les exigences et sur les systèmes de patrouille maritime. Dans le premier cas (exigences), il n'y a pas de métamodèle dominant pour exprimer des exigences[1]. Dans le second cas, et c'est un point important, nous montrons que la préoccupation de mesure peut être une motivation de premier ordre pour la modélisation, et que celle-ci permet d'unifier des approches existantes. Nous proposons donc un métamodèle pour les exigences.

Dans le second cas (système de surveillance maritime), c'est un modèle de domaine au sens le plus restrictif du terme, le domaine étant la surveillance maritime. Il n'y pas de métamodèle standard de système de patrouille maritime. Notre partenaire, *Thales Airborne Systems* (TAS), est en phase de recherche et de développement vis-à-vis de l'IDM pour l'ingénierie système, et n'a pas non plus de métamodèles disponibles. Nous proposons donc un métamodèle pour les systèmes de surveillance maritime. Notons que pour ce cas d'étude, les modèles en présence servent à la fois à la mesure et à la simulation.

Dans le cas des exigences comme des systèmes de surveillance maritime, l'introduction de l'IDM dans ces ingénieries est une contribution en tant que telle, même si ce n'est pas le but premier de notre thèse. L'introduction de l'IDM était pour nous un passage obligé, une étape nécessaire permettant ensuite d'appliquer l'approche MDM.

[1] L'approche *SysML* [OMG, 2005] n'étant pas encore largement acceptée et utilisée.

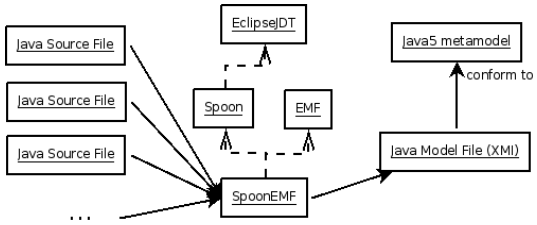

FIG. 4.2 : La transformation Java vers modèle

4.2 Étude de cas: la mesure des programmes Java

Dans cette section, nous considérons la mesure des programmes Java. Cette étude de cas montre que :

- notre approche englobe l'état de l'art autour des métriques sur les programmes;
- notre approche est applicable sur un artefact de la fin du cycle de vie;
- notre approche est applicable à la mesure des programmes Java considérés comme modèle;
- notre approche passe à l'échelle et permet de mesurer des modèles de plusieurs centaines de milliers d'éléments de modèles.

Nous avons choisi de mesurer des programmes Java, car d'une part, Java est un langage très largement utilisé, et d'autre part, il existe de nombreux logiciels écrits en Java et disponibles en open-source.

4.2.1 Métamodèle considéré

Nous avons utilisé le métamodèle de Java de *SpoonEMF*. Il est dérivé du métamodèle implicite de *Spoon* [Pawlak et al., 2006], lui même inspiré de la librairie de JDT[2].

Le métamodèle Java de *SpoonEMF* supporte Java5, dont les types génériques. D'un point de vue quantitatif il comprend 73 classes, 92 références, et 39 attributs. Sa taille le rend impossible à représenter dans un seul diagramme de classe. Le métamodèle *SpoonEMF* est un métamodèle à grain fin, c'est à dire que chaque élément de programme est accessible (i.e.; classes, méthodes, expressions, etc.).

L'outil *SpoonEMF* transforme un programme Java entier (plusieurs fichiers, plusieurs classes) en un modèle unique. Le modèle unique se présente sous la forme d'un fichier XMI. Le processus entier est illustré sur la figure 4.2.

4.2.2 Métriques

Le nombre de lignes de code

Le nombre de lignes de code est la métrique logicielle la plus utilisée [Fenton, 1991, p. 246]. Toutefois sa définition intuitive est ambiguë [Fenton, 1991]. Parmi les ambiguïtés, on peut citer les suivantes : doit on compter les lignes physiques ou logiques ? Doit-on compter les commentaires ? Dans le cas des lignes physiques, doit

[2] JDT est le projet *Eclipse* concernant la manipulation (dont la compilation) des programmes Java, voir www.eclipse.org/jdt.

on compter les lignes vides ? Ce qui amène Kan à affirmer [Kan, 1995, p. 88] que *the lines of code (LOC) metric is anything but simple*[3].

Pourtant la métrique de nombre de lignes de code est essentielle pour analyser la relation entre la taille du logiciel et son coût [Boehm et al., 1995], ou entre sa taille et sa densité de défauts [Fenton et Neil, 1999].

Avec l'approche MDM, une définition de la métrique du nombre de lignes de code est non ambiguë. D'une part, l'ensemble des éléments du programme est clairement et explicitement identifié dans le métamodèle. D'autre part, la métrique est complètement spécifiée comme instance du métamodèle de métriques.

Nous proposons donc la définition suivante de la métrique de nombre de lignes de code pour les programmes Java. Cette définition est de type nombre de lignes de code logiques. *CtStatement* est une classe du métamodèle *SpoonEMF*, dont hérite un grand nombre de classes (e.g.; *CtIf, CtCase, CtLoop*).

```
1 metric SigmaMetric LOC is
2   description "Number of lines of code"
3   x satisfy "this.isInstance(CtStatement)"
4 endmetric
```

Cette métrique illustre bien que notre approche de la mesure dirigée par les modèles permet à la fois de clarifier les concepts manipulés (c'est le métamodèle), et de spécifier de manière non ambiguë les métriques.

Nombre de *try/catch* défectueux

Un grand nombre de métriques pour les logiciels consiste en un nombre de violations d'une certaine bonne pratique [Cheng et al., 2005]. Ces bonnes pratiques peuvent être exprimées comme un prédicat. Par exemple, nous considérons la bonne pratique [Müller et Simmons, 2002] qui consiste à ne pas avoir de bloc *catch* vide dans un logiciel supportant la programmation par exception. Pour un programme Java, et toujours par rapport au métamodèle *SpoonEMF*, cela s'exprime comme suit :

```
1 metric SigmaMetric NTCD is
2   description "Number of dangerous catch"
3   x satisfy "this.isInstance(CtCatch)
4              and this.Block.Statements.size()=0"
5 endmetric
```

Estimation de nombre de défauts

Lo [Lo, 1992] (cité par [Kan, 1995]) a trouvé une relation linéaire avec une haute confiance statistique entre le nombre de défauts et certains éléments du langage. Ces derniers sont le nombre de constructions conditionnelles si/alors et le nombre d'appels de méthodes. La formule est présentée ci dessous, en 4.1 (# symbolise le *nombre de*).

$$\text{Field defects} = 0.11\#ifthen + 0.03\#calls \tag{4.1}$$

Les éléments constitutifs de cette métrique dérivée peuvent s'exprimer avec notre approche par rapport au métamodèle *Spoon*. Nous donnons les définitions correspondantes dans le listing 4.2.2.

[3] La métrique des lignes de code est tout sauf simple.

```
metric SigmaMetric NbIF is
  description "Number of If conditionnals"
  x satisfy "this.isInstance(CtIf)"
endmetric

metric SigmaMetric NbMC is
  description "Number of method calls"
  x satisfy "this.isInstance(CtExecutableReference)"
endmetric

metric DerivedMetric FD is
  description "Number of field defects"
  formula "0.11*NbIF + 0.03*NbMC"
endmetric
```

Métriques de Chidamber et Kemerer

Un des travaux les plus cités du génie logiciel [Wohlin, 2007] est un article de Chidamber et Kemerer [Chidamber et Kemerer, 1991], qui définit un ensemble de métriques sur les classes d'un logiciel orienté-objet. Nous présentons ci-dessous la ré-expression de ces métriques avec notre approche, i.e.; comme instance du métamodèle de métriques et référant au métamodèle Java de *SpoonEMF*. Nous choisissons de garder les noms et abréviations originales pour chacune d'entre elles.

WCM (*Weighted Methods per Class*) Nous considérons la version de Basili et al. [Basili et al., 1996], qui est définie comme le nombre de méthodes dans chaque classe.

```
metric MultiplicityPerX WMC is
    description "Weighted Methods per Class"
    elements satisfy "this.isInstance(CtClass)"
    reference is "Methods"
endmetric
```

DIT (*Depth in Inheritance Tree*) DIT est définie comme la profondeur d'une classe dans le graphe d'héritage.

```
metric PathLengthMetricSpecification DIT is
  description "Depth in Inheritance Tree per Class"
  x satisfy "this.isInstance(CtClass)"
  references followed "Superclass"
endmetric
```

NOC (*Number of Children*) NOC est le nombre de descendants directs pour chaque classe. C'est un exemple typique de l'utilisation du type de métrique *AboutMeMetricSpecification*. Il n'y a pas de référence d'une classe vers ses enfants mais *AboutMeMetricSpecification* permet la spécification et le calcul de la métrique NOC.

```
metric AboutMeMetricSpecification NOC is
  description "Number of Children of a Class"
  x satisfy "this.isInstance(CtClass)"
```

```
4   elements satisfy "this.Superclass == __me__"
5 endmetric
```

CBO (*Coupling Between Object Classes*) CBO est une mesure de couplage définie comme le nombre de classes couplées à une classe donnée.

```
1 metric AboutMeMetricSpecification CBO is
2   description "Coupling Between Object Classes"
3   x satisfy "this.isInstance(CtClass)"
4   input metric SetOfEdgesPerX CBO is
5   target satisfy "this.isInstance(CtTypeReference)
6         and this.QualifiedName == __me__.SimpleName"
7   endmetric
8 endmetric
```

RFC (*Response For a Class*) RFC est le nombre de méthodes qui peuvent être potentiellement exécutées en réponse à un message.

```
1 metric SetOfElementsPerX RFC is
2   description "Response For a Class"
3   x satisfy "this.isInstance(CtExecutableReference)"
4   elements satisfy "this.isInstance(CtClass)"
5   // ce sont des noms de références du métamodèle Spoon :
6   // on retrouve les relations entres éléments et blocs
7   // d'un programme
8   references followed "Methods,Body,Statements,
9    Expression,AssertExpression,CaseExpression,
10   ThenStatement,Condition,ElseStatement,Selector,
11   Cases,Block,Finalizer,Catchers,AssertExpression,
12   CaseExpression,Finalizer,Executable,
13   DeclaringExecutable"
14 endmetric
```

Lack of Cohesion on Methods La métrique LCOM est définie comme le nombre de couples de méthodes ne partageant aucune variable d'instance (nous nous référons à [Briand et al., 1998] pour une discussion approfondie sur le sujet). Il n'est pas possible de spécifier la métrique LCOM de Chidamber et Kemerer comme instance du métamodèle de spécification de métrique. La raison est que cette métrique est fondée sur le concept de couple de méthodes. Ce concept est artificiel par rapport au langage Java et n'est donc pas présent dans le métamodèle. D'un point de vue pratique, c'est une limite de notre approche: il n'est pas possible de définir LCOM.

D'un point de vue conceptuel, nous posons l'hypothèse qu'il est important que les métriques de domaine manipulent des concepts du domaine dans le but de respecter les critères de [Grady, 1992] cités plus haut. En particulier, cette condition nous semble nécessaire pour garder des métriques compréhensibles et ressenties comme importantes. Dans cette perspective, il est souhaitable qu'un métamodèle de métriques ne permette pas de créer des concepts ad hoc. Alors, l'apparition d'une métrique non exprimable comme instance du métamodèle de métriques peut être interprétée comme un indice d'un besoin d'adaptation du métamodèle de domaine.

Métrique	umlgraph	log4j-1.2.14	eclipse.osgi-3.2	regexp-1.4	bcel-5.2
Nb éléments de modèle	19800	160653	516413	30446	286605
Statements	1100	12286	38608	2735	22499
Empty catch blocks	1	14	157	6	16
Ifs	109	898	4392	263	1502
Calls	785	7624	22913	1345	15409

TAB. 4.1: Valeurs de mesures pour certains logiciels Java

4.2.3 Résultats de mesure

L'approche MDM génère ensuite l'outil de mesure des programmes Java. Pour ce faire, un modèle de spécification de métriques (fichier XMI) est donné en entrée au générateur décrit en 3.4. Celui-ci construit un outil de mesure, qui est utilisé pour mesurer les modèles de programme Java (fichier XMI) issus de *SpoonEMF*. Nous avons mesuré cinq logiciels open-source écrits en Java. Des résultats de mesure sont donné dans la table 4.1. La table donne aussi la taille des modèles: certains sont constitués de plusieurs centaines de milliers d'éléments. Les logiciels mesurés sont les suivants:

UmlGraph *UmlGraph* permet une spécification déclarative des diagrammes de classes et de séquences UML. (1250 LOC)

regexp *regexp* est une librairie Java fournissant le support des expressions régulières. (4950 LOC)

log4j log4j est une librairie de log. Elle fournit un service avancé de log, qui met l'accent sur la performance, i.e.; la vitesse nécessaire pour déterminer si une trace doit être enregistrée ou non. (30195 LOC)

BCEL BCEL (*Byte Code Engineering Library*) donne la possibilité d'analyser, de créer et de manipuler des fichiers compilés binaire Java (les fichiers *.class*) (46430 LOC)

org.eclipse.osgi org.eclipse.osgi est le coeur d'*Eclipse*. C'est une implémentation du standard *Open Services Gateway Initiative* (OSGI). (65837 LOC)

4.2.4 Travaux similaires

MoDisco [4] est un composant du projet *Eclipse* GMT (*Generative Modeling Technologies*) dédié à la rétro-ingénierie dirigée par les modèles. Une fonctionnalité annoncée de *MoDisco* est l'exact équivalent de *SpoonEMF*, à savoir la transformation d'un programme Java en instance d'un métamodèle. Toutefois, le composant n'était pas disponible au moment de l'écriture de cette thèse (avril 2008).

Nous avons listé dans le tableau 2.3 un ensemble d'outils (commerciaux et libres) qui mesurent des logiciels Java. Notons que, d'une part, aucun de ces logiciels n'est généré et que, d'autre part, ils ne sont pas basés sur un métamodèle explicite. Toutefois, il est clair que tous reconstruisent implicitement, par leurs structures de données, un métamodèle de programme Java (plus exactement un métamodèle de *bytecode* pour ceux qui travaillent à partir du *bytecode* et un métamodèle de source pour les autres).

[4] http://www.eclipse.org/gmt/modisco

Nous avons montré dans cette étude de cas l'applicabilité de notre approche de la mesure dirigée par les modèles à la mesure des programmes Java considérés comme modèle. Cela a d'abord consisté à identifier un métamodèle de programmes Java (*Spoon*) et un outil permettant de transformer un programme en un modèle conforme au métamodèle. Ensuite, nous avons exprimé plusieurs métriques de la littérature en fonction des concepts du métamodèle de spécification de métriques. L'outil généré permet de mesurer des programmes Java d'envergure (plusieurs centaines de milliers d'éléments de modèles).

4.3 Étude de cas: la mesure des modèles d'architecture temps-réel

Dans cette section, nous considérons la mesure des modèles d'architecture AADL (*Architecture Analysis & Design Language*) [Monperrus et al., 2008c, SAE, 2006]. Cette étude de cas montre que :

- notre approche est applicable sur un artefact utilisé en phase de conception (milieu du cycle de vie);
- notre approche est applicable à la mesure des artefacts de conception;
- notre approche à base de modèles apporte une amélioration des pratiques d'ingénierie par rapport aux pratiques de mesures à base de documents.

Nous avons choisi de mesurer des modèles AADL, car AADL est un métamodèle standardisé par la SAE (*Society of Automotive Engineers*). AADL est un langage de modélisation très actif comme en témoigne le nombre de publications et d'ateliers sur le sujet. De plus, il existe des modèles AADL open-source à mesurer.

4.3.1 Métamodèle considéré

Le langage AADL est un langage de modélisation à base de composants qui a pour vocation de permettre l'analyse de l'architecture des systèmes embarqués. Par exemple, AADL permet la vérification de propriétés de performance grâce à une sémantique bien définie. Les principaux concepts d'AADL, i.e.; les principales classes du métamodèle, sont les concepts de système, de périphérique (*device*) de bus, de processus, et de processus légers (*threads*). Nous nous référons à [Feiler et al., 2006] et à la spécification elle même [SAE, 2006] pour une présentation complète. Un extrait du métamodèle AADL est montré sur la figure 4.3.

Le modèle AADL que nous allons mesurer est un modèle open-source d'un système d'affichage aéronautique. Ce modèle a été crée par *Rockwell Collins* et fait partie des ressources disponibles sur le site d'AADL [5]. Il consiste en 30400 éléments de modèles. Un extrait de ce modèle avec la syntaxe textuelle d'AADL est présenté dans le listing 4.1.

Listing 4.1 : Extrait du modèle AADL mesuré (syntaxe textuelle)

```
1
2 bus Node_Switch
3   properties
```

[5] http://www.aadl.info

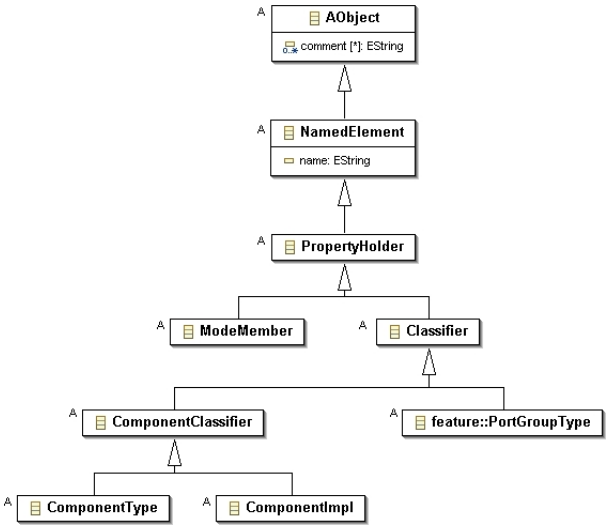

FIG. 4.3 : Extrait du métamodèle AADL

```
    Network_Speed => 120 Kbps;
    Max_Transmission_Rate => 160 Kbps;
    Max_Packet_Size => 1 KB;
end Node_Switch;

thread Periodic_1_Hz
  properties
    Dispatch_Protocol => Periodic;
    Period => 1000 Ms;
end Periodic_1_Hz;
```

4.3.2 Métriques

Nous avons pris un parti très pragmatique pour choisir les métriques sur les modèles AADL. Nous avons interviewé un architecte de systèmes embarqués d'une compagnie d'automobile, qui a plus de vingt ans d'expérience. Il nous a listé les métriques qu'il utilise de manière informelle sur les systèmes qu'il développe.

Notons que les métriques utilisées sont actuellement calculées ou mentalement par l'expert, ou à partir d'un ensemble de documents en langage naturel (e.g.; documents *Word*) et de feuilles de calcul (e.g.; documents *Excel*).

Ensuite nous avons étudié comment exprimer ces métriques par rapport au métamodèle AADL d'une part, et par rapport au métamodèle de spécification de métriques d'autre part.

Le nombre de périphériques (*device*)

Le nombre de périphériques est une métrique essentielle, directement liée au coût matériel du système à construire. D'après l'expert interviewé, dans l'industrie automobile, la problématique du coût matériel est cruciale, au regard du nombre d'unités produites.

```
metric SigmaMetric NOD is
  description "Number of devices"
  x satisfy "this.isInstance(Component::DeviceImpl)"
endmetric
```

Le nombre de processus

Le nombre de processus est une métrique qui servira de composant dans des métriques dérivées présentées plus loin.

```
metric SigmaMetric NOP is
  description "Number of processes"
  x satisfy "this.isInstance(Component::ProcessImpl)"
endmetric
```

Le nombre de *threads*

Le nombre de *threads* est une métrique qui servira de composant dans des métriques dérivées présentées plus loin.

```
metric SigmaMetric NOT is
  description "Number of threads"
  x satisfy "this.isInstance(Component::ThreadImpl)"
endmetric
```

Le nombre de *threads* non périodiques

Le nombre de *threads* non périodiques est une métrique qui servira de composant dans des métriques dérivées présentées plus loin.

```
metric SigmaMetric NONPT is
  description "Number of non periodic threads"
  x satisfy "this.isInstance(Component::ThreadImpl)
  and this.propertyAssociation.exists{ x |
  x.propertyValue.select{z|~property::EnumValue.isInstance(z)}
   .exists{z |
    z.asType(~property::EnumValue).enumLiteral.name=="Periodic"}}
     or this.compType.extend.properties
     .propertyAssociation.exists{ x |
      x.propertyValue.select{z|
       ~property::EnumValue.isInstance(z)}.exists{z |
        z.asType(~property::EnumValue)
        .enumLiteral.name=="Periodic"}}"
endmetric
```

Le nombre de composants orphelins

Le nombre de composants orphelins est une métrique de type nombre de violations d'une bonne pratique. Un composant orphelin est un composant qui n'est relié à aucun autre. La présence de composants orphelins est une faute grave à corriger, et un signe du besoin de revue générale du modèle. Dans un modèle AADL, cela se traduit par le nombre de composants qui ne sont pas connectés à au moins un port.

```
metric SigmaMetric NOOC is
  description "Number of orphan components"
  x satisfy "this.isInstance(core::ComponentImpl)
and (this.connections.eventConnection.size ==0
and this.connections.dataConnection.size ==0
and this.connections.eventDataConnection.size ==0
and this.connections.portGroupConnection.size ==0
and this.connections.dataAccessConnection.size ==0
and this.connections.parameterConnection.size ==0
and this.connections.busAccessConnection.size ==0"
endmetric
```

L'indicateur de difficulté d'ordonnancement

L'indicateur de difficulté d'ordonnancement est une estimation du nombres maximal de tâches que l'ordonnanceur du système temps réel devra gérer. Les architectes temps réels recommandent que des alertes soient levées si cette métrique dépasse 30.

Cette métrique est calculé à partir des métriques NOD, NOP, NOT, NONPT présentées dans les paragraphes précédents (l'identifiant est dans le code MDM de la métrique, e.g.; `metric SigmaMetric NOT is`).

```
metric D1 is
  formula "NOP/NOD"
end

metric D2 is
  formula "NOT/NOP"
end

metric SchedPB is
  description "schedulability difficulty indicator"
  formula "D1*D2"
endmetric
```

La prédictibilité de charge

La prédictibilité de charge est $NONPT/NOT$. Comme le nom l'indique, cette métrique montre s'il est possible de prédire la charge du système en terme de consommation CPU.

```
metric LP is
  description "Load predictability"
  formula "NONPT/NOT"
endmetric
```

Cette métrique a les propriétés suivantes : si tous les *threads* sont périodiques i.e.; $NONPT \rightarrow 0$, la charge des périphériques est facilement prédictible. Au contraire, si tous les *threads* sont sporadiques ou non périodiques i.e.; si $NONPT/NOT \rightarrow 1$, il est extrêmement difficile de prédire la charge. Dans ce cas, les architectes augmentent leurs marges de sécurité.

ID	Value	Interprétation
NOD	5	valeurs enregistrées dans une base afin de calibrer à terme des modèles de coûts et de risques.
NOP	11	idem.
NOT	69	idem.
NOPTT	0	cf. D4
NOOC	0	le modèle est bien formé.
D1	2.2	OK d'après le savoir-faire empirique.
D2	6.27	idem.
SchedPB	13.8	inférieur à 30, acceptable.
LP	0	ceci indique un risque bas par rapport à la charge.

TAB. 4.2: Les résultats obtenus grâce à l'outil généré à partir de la spécification des métriques sur AADL

4.3.3 Résultats de mesure

Suivant notre approche, ces spécifications abstraites sont automatiquement transformées en un outil de mesure exécutable. Nous avons mesuré le modèle AADL présenté ci-dessus. Les résultats sont affichés dans la table 4.2. Tous les résultats indiquent que le modèle est correct au regard de l'expertise de l'architecte interviewé. Par exemple, nous notons que l'indicateur de difficulté d'ordonnancement *SchedPB* est inférieur à la limite empirique de 30. Du point de vue des gestionnaires, les métriques NOD, NOP, NOT sont importantes. Elles ont vocation à être sauvegardées dans une base de données, afin d'être utilisées dans des analyses futures des processus de développement et la calibration de modèle de coût ou de risque.

4.3.4 Travaux similaires

À notre connaissance, il n'y a pas d'autres travaux visant à définir des métriques sur des modèles de systèmes temps-réels, et en particulier des modèles AADL.

4.4 Étude de cas: la mesure d'un modèle de système de surveillance maritime

Dans cette section, nous présentons l'application de notre approche dans un contexte industriel [Monperrus et al., 2008a]. Notre partenaire industriel est *Thales Airborne Systems* (TAS). Cette étude de cas montre:

- une application industrielle;
- des métriques de domaine au sens le plus strict du terme, le domaine étant les systèmes de surveillance maritime;
- une instance de l'IDM au niveau de l'ingénierie système. Notons que l'ingénierie système est au début du cycle de vie et découplée de l'architecture et du développement strictement logiciel.

La ligne de produits de *Thales Airborne Systems* que nous considérons est constituée des systèmes de surveillance maritime [Ince et al., 2000]. La figure 4.4 montre les

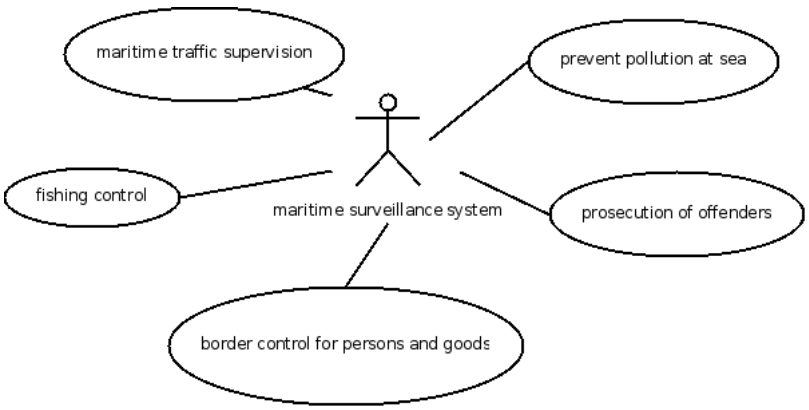

FIG. 4.4 : Cas d'utilisation d'un système de surveillance maritime.

principales fonctionnalités d'un système de surveillance maritime sous la forme d'un diagramme de cas d'utilisation UML. Par la suite, nous utiliserons l'abréviation MSS pour désigner ce système. MSS est l'acronyme du nom anglais *maritime surveillance system*. Un MSS a pour but de superviser le trafic maritime, d'éviter les pollutions maritimes (e.g.; dégazage de pétrolier), de contrôler les activités de pêche, de contrôler l'activité aux frontières. Il est habituellement composé d'un aéronef (avion ou hélicoptère), d'un ensemble de capteurs, d'un équipage et d'un grand nombre d'artefacts logiciels. Le nombre des fonctionnalités, les relations entre composants logiciels et matériels, et la communication entre le système et les autres entités de surveillance (bases terrestres, autres MSS) indiquent le niveau de complexité de ces systèmes.

Le développement d'un système de surveillance maritime commence dans le département ingénierie système. Cela commence avant même que le contrat ne soit finalisé, afin de mener des études préliminaires pour maintenir en cohérence le prix du produit vendu et son coût de développement estimé. À ce moment du cycle de vie, l'ingénierie consiste en des études de faisabilité et d'estimation des coûts, d'élicitation des exigences, de vérification de cohérence des exigences. Ces activités passent par des définitions d'architecture à gros grain, et la validation de l'architecture par la simulation.

La simulation est une étape essentielle dans les développements des systèmes de patrouilles maritimes [Ince et al., 2000, chapitre 9]. Au tout début du cycle de vie, c'est-à-dire au niveau de l'ingénierie système, la simulation est un moyen de communiquer avec le client afin d'éliciter les besoins. En cela la simulation permet d'améliorer la cohérence entre le produit livré et le produit attendu. D'un point de vue technique, la simulation permet de valider des parties de l'architecture et le comportement du produit à gros grain. Enfin, la simulation permet une vision à la fois plus globale et plus complète du système et permet de mieux estimer et planifier les ressources techniques et humaines à mettre en oeuvre.

Chez notre partenaire, la plupart des activités d'ingénierie système sont aujourd'hui centrées sur les documents. Or le développement des simulateurs est principalement centré sur le code. Il y a donc une grande distance conceptuelle et pratique entre les modèles mis en oeuvres (parfois totalement implicites) dans les documents, dans les modèles mentaux des ingénieurs système et dans les modèles de simulation.

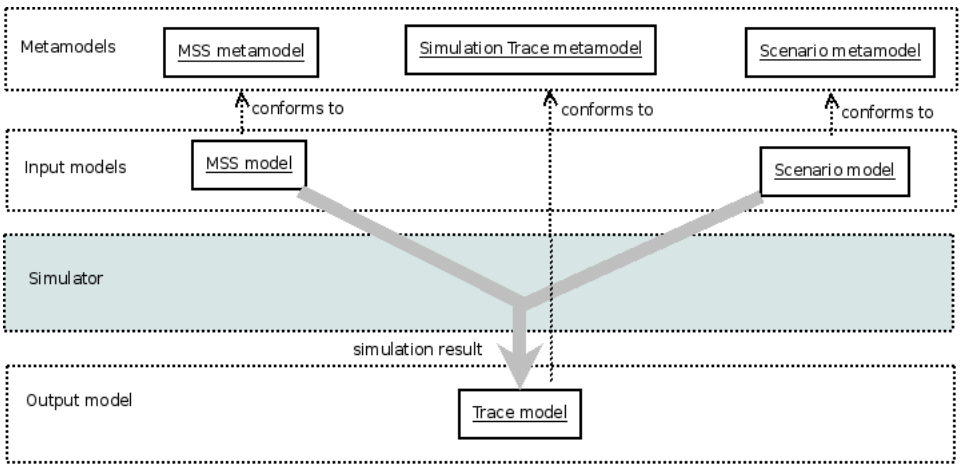

FIG. 4.5 : L'architecture du processus de simulation

Notre idée est donc d'introduire l'IDM et ses modèles en ingénierie système, puis d'appliquer notre approche pour mesurer des caractéristiques du modèle, donc d'estimer des caractéristiques du système. Notons que ce projet est un projet parmi de nombreux autres à TAS, comme dans d'autres filiales de *Thales*, visant à étudier l'utilisation des modèles dans les phases d'ingénierie système.

Nous avons donc développé un prototype de simulateur de système de patrouille maritime à base de modèle afin d'appliquer notre approche sur des modèles et d'obtenir des mesures de domaine.

4.4.1 Métamodèles considérés

Contrairement aux études de cas précédentes, nous n'avons pas ici un seul mais 3 métamodèles mis en oeuvre. Ceci est dû aux besoins de la simulation. Les responsabilités de ces métamodèles et les liens entre les modèles attenants sont montrés figure 4.5. Ces métamodèles sont présentés par la suite.

Le coeur du simulateur est une instance du patron en Y du MDA [Soley, 2000] (la forme de la lettre Y est similaire à la forme des graphiques utilisés pour représenter les transformations PIM vers PSM). Le simulateur prend en entrée deux modèles: un modèle de MSS et un modèle de situation tactique appelé modèle de scénario. Il produit en sortie un modèle de trace de simulation. Ces trois modèles sont spécifiés par trois métamodèles qui sont respectivement le métamodèle d'architecture, le métamodèle de scénario et le métamodèle de trace de simulation. Ces métamodèles sont présentés ci-dessous.

Cette architecture est fondée sur deux principes de l'IDM: 1) toutes les données échangées sont explicitement spécifiées par un métamodèle; 2) les concepts du domaine sont indépendants des concepts d'implémentation. Dans notre cas, ce deuxième principe consiste à dissocier les modèles du système (architecture et scénario) et la trace de simulation. Cette architecture possède plusieurs propriétés remarquables.

Toute la variabilité du simulateur est encodée dans des modèles. Par exemple, ajouter un radar donné, ou changer une caractéristique du radar embarqué dans le système consiste seulement à modifier le modèle d'architecture d'entrée de la simu-

lation.

Le modèle de MSS et le modèle de scénario ne sont pas couplés du tout. On peut donc créer plusieurs modèles d'architecture et plusieurs modèles de scénarios, et étudier toutes les combinaisons possibles, c'est-à-dire étudier le comportement de chaque architecture dans chaque scénario.

Le métamodèle de système de patrouille maritime

Le métamodèle de MSS est divisé en plusieurs paquetages. Le critère utilisé pour la séparation en paquetage est un critère de décomposition fonctionnelle. Le paquetage du système de navigation contient les classes dont le rôle est de représenter les composants de positionnement et de routage (matériel, logiciel et composite). Par exemple, il y a une classe GPS (*Global Positioning System*) Les attributs de cette classe sont les caractéristiques principales d'un GPS. Le paquetage de détection comprend les classes qui représentent les composants de détection d'un MSS. Par exemple, un modèle de MSS peut être simulé avec un radar classique, un IFF (*Identification Friend and Foe*) ou encore un FLIR (*Forward Looking Infra Red*). Les attributs d'un radar (de la classe Radar), comprennent l'angle d'ouverture de l'antenne, la vitesse de rotation, ou la période d'impulsion. Le paquetage de communication explicite les types de communication et les composants mis en oeuvre pour la communication. Cela va de la communication interne entre les membres de l'équipage à la communication avec des acteurs externes. Les classes couvrent donc les supports de communication (e.g.; VHF, IHF, satellite), et les propriétés de canaux de communication (e.g; chiffrage, protocole). Enfin le paquetage dédié à la description de l'équipage contient les classes permettant de spécifier le nombre des membres de l'équipage, et leurs fonctions respectives. Le modèle principalement utilisé lors du développement du prototype représente un système de surveillance maritime composé d'un avion de type *Falcon*, d'un radar, d'un système inertiel, et d'un calculateur tactique.

Le métamodèle de scénario

Le métamodèle de scénario contient toutes les classes nécessaires pour représenter une situation tactique. Un modèle, instance de ce métamodèle, spécifie la zone de surveillance, le nombre et le type des objets qui se trouvent dans la zone. Chaque objet est spécifié par sa trajectoire, et sa vitesse, et sa surface équivalent-radar (SER). Enfin la zone a un attribut décrivant la météo. Durant la simulation, cet attribut est utilisé pour calculer la qualité de l'information renvoyée par les capteurs.

Le métamodèle de trace de simulation

Le simulateur prend en entrée deux modèles: un modèle de MSS et un modèle de scénario. À partir de la sémantique de simulation, il produit des traces de simulation. D'après les principes de l'IDM, ces traces de simulation forment aussi un modèle, spécifié par un métamodèle de trace de simulation. Les traces de simulation contiennent tous les évènements qui ont une sémantique par rapport à la simulation, et par rapport aux préoccupations de l'ingénierie système. Ces traces sont:

- le position du MSS à chaque instant;
- la position de chaque objet du système à chaque instant;

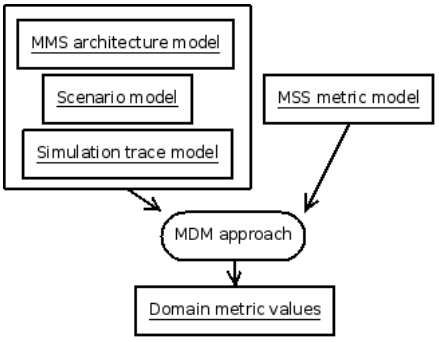

FIG. 4.6 : L'approche MDM appliquée aux modèles de systèmes de surveillance maritime

TAB. 4.3: Métriques pour les modèles de système de surveillance maritime

Système	Estimation du coût
	Quantification de la taille de la base de données embarquée
	Nombre de pistes par niveau de danger
Scénario	Difficulté de la situation tactique
	Régularité des trajectoires
	Homogénéité de la situation tactique
Trace	Nombre de cible détectées
	Régularité des détections
	Temps de simulation
	Consommation de carburant

- les caractéristiques internes du MSS: carburant, champ de détection du radar;
- les événements sémantiques, par exemple la détection et/ou l'identification des objets par le calculateur tactique.

À tous ces types de traces corresponde une classe dans le métamodèle de trace de simulation.

4.4.2 Métriques

Les métriques de domaine dans le contexte du développement des MSS au niveau de l'ingénierie système sont de différents types.

Les métriques sur l'architecture du système Ces métriques concernent l'architecture proprement dite e.g.; le nombre de capteurs embarqués. Elles sont habituellement utilisées pour des estimations de coût et de planning.

Les métriques de scénario Ces métriques concernent la situation tactique dans laquelle le système va évoluer. Par exemple, le nombre de navires dans la zone de surveillance. Le dimensionnement du système dépend de ce type de métrique de domaine.

Les métriques de performance technico-opérationnelle Ces métriques sont des estimations des propriétés du système, par exemple la proportion d'objets identifiés dans la zone de surveillance.

Comme le montre la figure 4.6, chacun de ces trois types de métriques est lié à un seul type de modèle, i.e.; à un seul métamodèle mis en oeuvre. Ce dernier point est notable, car l'architecture des métamodèles de cette étude de cas n'a pas été dictée par les métriques de domaine, mais par les besoins de la simulation (cf. figure 4.5). On retrouve dans les métriques de domaine la même décomposition que dans la simulation.

À partir des métamodèle d'une part, et des métriques de domaine d'autre part, nous sommes donc à même de spécifier des métriques de modèles suivant notre approche. Celles-ci sont présentées dans le listing 4.2 et en annexe.

Listing 4.2 : Métriques sur des modèles de systèmes de surveillance maritime

```
1
2  metric ArithmeticMetric R is
3    description "Ratio of detected ships over the whole
4                 tactical situation"
5    formula is "N1/N2"
6  endmetric
7
8  metric SigmaMetric N1 is
9    description "Number of detected ships"
10   elements satisfy "this.isInstance(SimulationEvent)
11                  and this.detected.iSInstance(Ship)"
12   option unique
13 endmetric
14
15 metric SigmaMetric N2 is
16   description "# of ships in the surveillance zone"
17   elements satisfy "this.isInstance(Ship)
18     and this.x > this.scenario.surveillance_xmin
19     and this.x < this.scenario.surveillance_xmax
20     and this.y > this.scenario.surveillance_ymin
21     and this.y < this.scenario.surveillance_ymax"
22   option unique
23 endmetric
```

4.4.3 Travaux similaires et conclusion

Nous n'avons pas trouvé de travaux relatifs à la mesure des modèles au niveau de l'ingénierie système. De même, les publications relatives à l'application de l'IDM en ingénierie système sont très peu nombreuses [Ogren, 2000, Axelsson, 2002, OMG, 2005, Estefan, 2007].

Dans le contexte du développement des systèmes de patrouille maritime et au niveau de l'ingénierie système, nous avons appliqué notre approche de la mesure des modèles. La figure 4.6 illustre l'obtention de valeurs de mesure de métriques de domaine à partir des modèles manipulés.

Par rapport aux pratiques existantes, notre approche permet l'obtention de métriques opérationnelles qui n'étaient pas disponibles avec les pratiques d'ingénierie système centrées sur les documents.

4.5 Étude de cas: la mesure des exigences

Dans cette section, nous étudions le problème de la mesure des exigences en appliquant l'approche MDM. Pour ce faire, nous proposons un métamodèle d'exigence, et exprimons des métriques d'exigences de la littérature par rapport à ce métamodèle et comme instance du métamodèle de spécification de métriques. Cette étude de cas montre:

- une application de notre approche au tout début du cycle de vie d'un produit;
- l'aspect unificateur de notre approche[6] – nous ré-exprimons plusieurs métriques de la littérature comme instance du métamodèle de métriques. Il devient donc possible de comparer à la fois les définitions des métriques et les valeurs de mesure.

4.5.1 Pourquoi mesurer des exigences ?

L'idée de mesurer de exigences est assez ancienne. Ainsi, Hammer a fait une présentation à la conférence RE'97 intitulée *Requirements Metrics - Value Added*[7] [Hammer et al., 1997]. L'intuition est qu'il doit être possible d'identifier des risques et des défaillances très tôt dans le cycle de vie des systèmes en mesurant les exigences. Plusieurs études ont été conduites pour trouver des relations entre le processus d'ingénierie des exigences et le succès des projets (e.g.; [Kamata et Tamai, 2007], et ses références).

D'un point de vue normatif, dans le processus CMM, la gestion des exigences est une des 7 pratiques du niveau 2 du CMM. Un processus de développement qui suit les standards du CMM doit être supporté par des mesures. Dans *Key Practices of the Capability Maturity Model* [Paulk et al., 1993] publié par le *Software Engineering Institute* (SEI), il est spécifié que *measurements are made and used to determine the status of the activities for managing the allocated requirements*[8].

Des approches de l'ingénierie des exigences par les modèles ont été explorées. Par exemple, Nakatani et al. décrivent [Nakatani et al., 2001] un métamodèle de cas d'utilisation. Solheim et al. présentent [Solheim et al., 2005] un processus et un outillage basé sur les modèles autour de AKM (*Active Knowledge Modeling*).

De même, plusieurs auteurs ont défini des métriques sur les exigences [Davis et al., 1993, Costello et Liu, 1995, Marchesi, 1998, Loconsole, 2001, Henderson-Sellers et al., 2002, Singh et al., 2004, Berenbach et Borotto, 2006]. Tous s'attachent à couvrir un aspect particulier du processus ou un contexte d'ingénierie des exigences. Ils font tous des hypothèses différentes sur les exigences elles-mêmes. Le problème est qu'il n'y a pas de cadre commun pour exprimer les exigences, les métriques d'exigences, et pour obtenir des valeurs de mesure.

Ce problème est polymorphe. Premièrement, il n'est pas possible de comparer entre elles les définitions de métriques de la littérature. Deuxièmement, il n'est généralement pas possible d'obtenir automatiquement des valeurs de mesure à partir d'un modèle d'exigences. C'est-à-dire que pour obtenir l'ensemble des valeurs de mesure,

[6] Ce point est comparable à l'aspect unificateur de UML dans les années 1990.
[7] Mesures des exigences = Valeur ajoutée
[8] Des mesures sont effectuées et utilisées pour déterminer le statut des activités de gestion des exigences

on doit créer autant de modèles que de papiers de la littérature. Enfin, il n'est pas possible de comparer les valeurs de mesure elles-mêmes, étant donné que la définition des métriques et les hypothèses implicites sont différentes.

Nous proposons d'appliquer notre approche au problème de la mesure des exigences. Pour ce faire, nous proposons un cadre commun pour spécifier les exigences, ce cadre étant complètement spécifié par un métamodèle. Celui-ci contient tous les concepts nécessaires à la mesure des exigences. Ensuite, nous pouvons exprimer les métriques d'exigences de la littérature par rapport à ce métamodèle d'exigence et comme instance de notre métamodèle de spécification de métriques. Nous rappelons que l'approche étant entièrement basée sur des modèles, il est donc possible d'obtenir un outil de gestion des exigences (comprenant des fonctionnalités de mesure) entièrement par génération de code.

4.5.2 La mesure des exigences dans la littérature

Dans cette section, nous présentons notre analyse de la littérature sur les métriques d'exigences. Nous commencerons par présenter nos critères d'analyse, puis nous décrirons les métriques définies dans 11 articles et enfin nous analyserons les relations entre chacune de ces métriques.

Critères pour sélectionner des métriques

Nous utilisons trois critères pour sélectionner des métriques d'exigences parmi les travaux existants.

Métriques sur les produits Notre approche considère les produits issus des processus d'ingénierie des exigences. Une métrique de produit s'applique à une exigence atomique ou à un ensemble d'exigences, appelé habituellement une spécification (*requirements specification*). D'un point de vue plus formel, une métrique de produit est un morphisme d'un document vers une valeur. En conséquence, nous excluons les métriques concernant les processus ou la gestion des exigences.

Métriques sur la sémantique Nos travaux sont clairement IDM, donc essaient de se concentrer sur la sémantique des concepts manipulés. Dans le cas des exigences, nous excluons donc les métriques basées sur le langage naturel (comme par exemple le nombre des occurrences de *shall* dans la spécification).

Métriques non syntaxiques Pour la même raison, nous excluons les métriques purement syntaxiques, comme par exemple le nombre de pages de [Costello et Liu, 1995].

Papiers sélectionnés

Baumert et al. ont proposé [Baumert et McWhinney, 1992] un ensemble de métriques compatibles avec les méthodes de mesure du CMM (*Capability Maturity Model for Software*). Ces métriques sont classées par catégorie. L'une d'elle est dédiée aux exigences (table 4.4).

Tab. 4.4: Métriques de Baumert et al. (1992)

total number of requirements
number of requirements changes proposed
number of requirements changes open
number of requirements changes approved
number of req. changes incorporated into baseline
number of TBDs in requirements specifications
number of req.s scheduled for each software release
number of waivers requested from requirements
number of waivers approved from requirements
time for analysis and action per change request
amount of effort required per change request

Le but du papier de Davis et al. [Davis et al., 1993] est d'explorer rigoureusement le concept de qualité d'une SRS (*Software Requirements Specification*) et de définir des attributs de qualité qui peuvent être réellement mesurés. Ils définissent 24 attributs de qualité pour une SRS[9] et montrent des exemples d'exigences qui satisfont ou pas ces exigences de qualité.

Costello et Liu [Costello et Liu, 1995] estiment que la rigueur des métriques logicielles peut être appliquée aux exigences par des métriques sur les exigences. Leur but final est de valider totalement et objectivement les processus et les produits de l'ingénierie des exigences. À notre connaissance, ils sont les premiers à introduire le terme de *measurable requirements specification*[10]. Cette expression met l'accent sur le rôle clé de la mesure dans l'ingénierie des exigences. Dans une certaine mesure, cela veut dire que vouloir mesurer des exigences est une raison suffisante pour modifier les produits ou les processus de l'ingénierie des exigences afin de rendre ceux-ci mesurables. Costello et Liu définissent un certain nombre de métriques liées à trois attributs de qualité (volatilité, traçabilité, complétude).

Les exigences à base de cas d'utilisation peuvent être métamodélisées [Nakatani et al., 2001]. Le papier de Marchesi [Marchesi, 1998] contient des métriques qui s'appliquent aux cas d'utilisation UML. Comme UML est spécifié par un métamodèle, ces métriques sont donc les premières métriques de type IDM pour la mesure des exigences.

Contrairement à ce que le titre de [Loconsole, 2001] (*Measuring the Requirements Management Key Process Area*) laisse supposer, Loconsole définit un ensemble de métriques sur les produits de l'ingénierie des exigences (e.g; une SRS). Dans son papier, Loconsole applique l'approche GQM (*Goal/Question/Metric* [Basili et al., 1994]) aux processus CMM. Les questions identifiées par GQM contiennent des indices sur les éléments qui doivent apparaître dans un métamodèle d'exigences.

Le papier de Henderson-Sellers et al. [Henderson-Sellers et al., 2002] fait une synthèse entre les objections de Costello [Costello et Liu, 1995] et l'idée de Marchesi

[9] Non-ambiguë, complète, correcte, compréhensible, vérifiable, cohérente, réalisable, concise, indépendante du design, traçable, modifiable, au format électronique, exécutable, annotée selon l'importance, annotée selon la stabilité, annotée par version, sans redondance, au bon niveau de détail, précise, réutilisable, tracée, organisée, avec des références croisées.

[10] une spécification mesurable

[Marchesi, 1998]. Henderson-Sellers et al. définissent un patron de cas d'utilisation qui permet de mesurer ceux-ci (*so that use cases can be metricated*[11]). À partir d'un ensemble d'exigences exprimées suivant ce patron, il est alors possible de définir des métriques d'exigences et d'obtenir des valeurs de métriques. En conséquence, ils définissent 12 métriques sur les exigences, comme par exemple le nombre d'actions atomiques dans le flot principal.

En 2004, deux *white papers*[12] ont été publiés sur le sujet par deux compagnies différentes. Kolde [Kolde, 2004] [Kolde, 2004, Douglass, 2004] définit un ensemble de métriques à propos du développement des logiciels. Il remarque que beaucoup de projets n'ont pas de mesures au niveau des exigences et que les documents liés aux exigences n'ont pas de forme standard. Nous faisons l'hypothèse que le deuxième point est une des causes du premier. Une approche IDM pour les exigences pourrait remplacer la variété des documents par un ensemble de modèles conformes à un seul métamodèle. Le sujet de *Computing Model Complexity* [Douglass, 2004] est plus vaste. Pourtant comme la compagnie vend un outil de type modeleur UML, le papier contient plusieurs définitions de métriques sur les cas d'utilisation.

Singh et al. définissent [Singh et al., 2004] une métrique de complexité pour une exigence atomique et pour un groupe d'exigences. Afin de mesurer effectivement des exigences, il proposent un métamodèle d'exigences. À notre connaissance, c'est le premier essai de définition d'un métamodèle d'exigences dans le seul but de rendre une SRS mesurable.

Le projet de recherche européen Modelware a livré un document intitulé *MDD Engineering Metrics Catalogue* [Modelware Project, 2006]. Nous incluons dans notre analyse les métriques liées aux cas d'utilisation. Enfin, Berenbach et al. [Berenbach et Borotto, 2006] décrivent une approche IDM , conforme au CMM, pour la mesure des exigences. Ce travail projette le processus CMM sur des modèles, principalement des modèles de cas d'utilisation, afin d'automatiser la mesure.

Analyse des métriques sélectionnées

À partir des métriques sélectionnées, nous faisons les observations suivantes :

Évolution de la recherche Si l'on suit l'ordre chronologique, la tendance des recherches sur la mesure des exigences va vers de plus plus d'approches à base de modèles.

Répétition Certaines métriques d'exigences apparaissent dans plusieurs papiers. Dans cette perspective, c'est une preuve de l'existence d'une sorte de consensus autour d'un noyau de métriques d'exigences, par exemple le nombre total d'exigences ou la volatilité des exigences en terme de nombre de changements dans les exigences par unité de temps.

Chevauchement Certaines métriques sont différentes mais répondent à des questions similaires. Dans une certaine mesure, elles se chevauchent. Par exemple, le

[11] afin de pouvoir mesurer les cas d'utilisation.
[12] Nous n'avons pas trouvé d'équivalent français à ce terme.

nombre de exigences fonctionnelles allouées à une livraison [Kolde, 2004], et le nombre d'exigences qui se reflètent dans un ou plusieurs CSCI[13] [Costello et Liu, 1995].

Terminologie Certaines métriques sont décrites avec des termes différents. Par exemple, les notions de *time frame*, *unit of time*, *project release* ont probablement le même sens. Ce qui nous amène directement au dernier point.

Interprétation Certaines métriques sont sujettes à interprétation. Comme elles sont définies avec le langage naturel, leur sens n'est pas toujours clair. Cette remarque peut s'appliquer à la courte description de la métrique, comme à son explication complète. Par exemple, le coût de changement d'une exigence de [Loconsole, 2001] peut s'interpréter comme le coût de vérification de la cohérence des exigences impactées ou comme le coût de modification des artefacts logiciels correspondants.

4.5.3 L'application de notre approche à la mesure des exigences

Nous voulons pouvoir obtenir toutes les valeurs de mesures pour toutes les métriques de la littérature, et ce, à partir d'un seul modèle d'exigences. En l'état, le problème principal est que toutes les métriques s'appliquent à des exigences sous des formes différentes, avec des hypothèses implicites.

Notre intuition est que la littérature sur les métriques d'exigences contient implicitement des concepts communs pour la modélisation des exigences, c'est à dire cache un métamodèle commun.

Nous allons donc commencer par identifier un langage d'expression des exigences via un métamodèle, puis nous allons appliquer notre approche de la mesure et respécifier les métriques de la littérature comme instances du métamodèle de spécification de métriques. Alors, grâce à cette double formalisation et suivant l'approche proposée dans cette thèse, nous pourrons exprimer des exigences comme instance du métamodèle d'exigences, puis obtenir toutes les valeurs de mesure à partir d'un seul modèle.

Création du métamodèle d'exigences

Nous avons suivi un processus pragmatique pour créer le métamodèle d'exigences. Celui-ci est décrit figure 4.7. Notre but est d'avoir un métamodèle qui supporte de manière complète et non ambiguë les métriques de la littérature. Nous avons analysé les métriques de la littérature une à une pour déterminer si les concepts ou relations mis en oeuvre existent dans le métamodèle courant. Sinon, nous ajoutions le concept correspondant dans le métamodèle. Dans certains cas, une étape d'unification terminologique était nécessaire. À la fin de ce processus, nous avons obtenu un métamodèle commun pour les exigences qui contient tous les concepts nécessaires à la mesure des exigences. La table ci-après montre les liens entre les métriques et les concepts du métamodèle. Chaque élément du métamodèle a été créé en réponse au besoin du concept dans une des métriques considérées.

[13] Computer Software Configuration Item

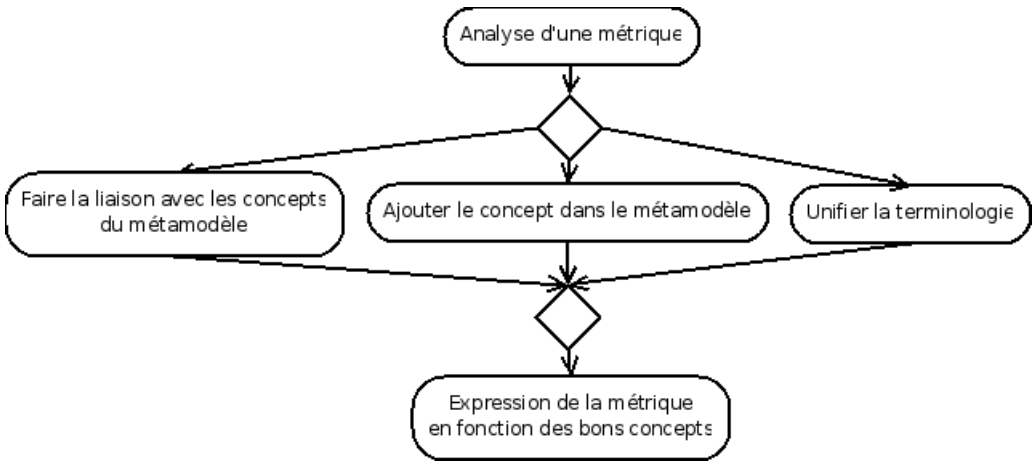

FIG. 4.7 : Le processus de métamodélisation dirigé par les métriques

Choix des métriques

L'analyse une par une des métriques décrites dans la section précédente nous a amené à identifier les répétitions, faire des choix sémantiques, et procéder à des unifications terminologiques. Notre approche permet donc d'unifier les contributions précédentes sur la mesure des exigences. La table ci-dessous montre une vue complète des 79 métriques résultantes sur les exigences et leurs origines. Certaines métriques apparaissent dans différents papiers, avec des définitions similaires, en conséquence elle ont plusieurs items dans la colonne *Origins*.

Métriques sur les exigences (consolidées)

#	Metric	Related elements of the metamodel	Origins
1	number of requirements (NR)	Requirement(c)	Costello, Kolde, Davis, Loconsole, Baumert
2	number of initial requirements	Requirement(c), TimeFrame(c)	Loconsole
3	number of requirements added per time frame	TimeFrame(c), RequirementAddition(c)	Costello, Kolde
4	number of requirements modified per time frame	TimeFrame(c), RequirementModification(c)	Costello,
5	number of requirements deleted per time frame	TimeFrame(c),, RequirementDeletion(c)	Costello, Baumert
6	number of changes per time frame	TimeFrame(c), RequirementChange(c),	Costello, Kolde, Loconsole
7	number of changes per requirement	RequirementChange(c), Requirement(c),	Loconsole
8	number of requirements that trace to the next level up	ConceptionLevel(c), originalRequirement(r)	Costello
9	number of requirements that trace to the next level down	ConceptionLevel(c), decomposedIn(r)	Costello
10	number of requirements that trace to the next level in both directions	ConceptionLevel(c), decomposedIn(r), originalRequirement(r)	Costello
11	number of requirements that trace from highest to lowest	ConceptionLevel(c), decomposedIn(r), originalRequirement(r)	Costello, Kolde
12	number of requirements that trace from lowest to highest	ConceptionLevel(c), decomposedIn(r), originalRequirement(r)	Costello
13	number of CSCI linked to a requirement	CSCI(c), allocatedTo(r), decomposedIn(r)	Loconsole
14	number of requirements per level that have inconsistent traceability links upward	decomposedIn(r), originalRequirement(r), ConceptionLevel(c)	Costello
15	number of req. per level that have inconsistent traceability links downward	decomposedIn(r), originalRequirement(r), ConceptionLevel(c)	Costello, Kolde

16	number of requirements per level that have no traceability links upward	decomposedIn(r), originalRequirement(r), ConceptionLevel(c)	Costello
17	number of requirements per level that have no traceability links downward	decomposedIn(r), originalRequirement(r), ConceptionLevel(c)	Costello
18	degree of decomposition per requirement per time frame	decomposedIn(r), TimeFrame(c)	Costello
19	number of requirements per status	Requirement(c), Status(c)	Costello, Kolde, Davis, Berenbach, Loconsole
20	number of req. that trace to one or more incomplete req.	decomposedIn(r)	Costello
21	num. of req. that trace to TBx	Requirement(c), Status(c), decomposedIn(r)	Costello
22	number of incomplete requirements	Requirement(c) and its attributes	Costello
23	number of requirements reflected in one or more CSCI	Requirement(c), decomposedIn(r), allocatedTo(c), CSCI(c)	Costello
24	number of use cases per status	UseCase(c), Status(c)	Modelware
25	number of use cases per status per time frame	UseCase(c), Status(c), TimeFrame(c)	Modelware, Loconsole
26	number of use cases	UseCase(c)	Douglass, Marchesi
27	number of functions specified (NF)	CapabilityRequirement(c)	Davis
28	number of unique functions specified (NUF)	CapabilityRequirement(c), dependsOn(r)	Davis
29	number of requirements traced to incomplete CSCI	Requirement(c), decomposedIn(r), allocatedTo(c), CSCI(c)	Costello
30	number of accepted use case diagrams	UseCaseDiagram(c), Status(c)	Berenbach
31	number of non submitted use case diagrams	UseCaseDiagram(c), Status(c)	Berenbach
32	number of sequence diagrams per use case	SequenceDiagram(c), UseCase(c)	Douglass
33	number of submitted use case diagrams	UseCaseDiagram(c), Status(c)	Berenbach
34	number of boundary that does not communicate with an actor	Boundary(c), Actor(c)	Berenbach

35	number of boundary that does not communicate with a concrete use case	Boundary(c), Actor(c), UseCase, concrete(a)	Berenbach
36	number of use cases per actor	UseCase(c), Actor(c)	Douglass, Marchesi
37	number of actors	Actor(c)	Berenbach, Douglass, Henderson-Sellers
38	number of use cases non described by one or more behavioral diagram	UseCase(c), Diagram(c) and its subclasses	Berenbach
39	number of use cases that do not appear on a diagram	UseCase(c), UseCaseDiagram(c)	Berenbach
40	number of circular dependency between use cases	UseCase(c), extends(r), includes(r)	Berenbach
41	number of uses cases that do not appear on a parent behavioral diagram	UseCase(c), extends(r), includes(r), describedBy(r), Diagram(c)	Berenbach
42	number of mixed use cases (including one abstract and one concrete)	UseCase(c), includes(r)	Berenbach
43	number of impacted requirements per change	supplierFor(r), decomposedIn(r), RequirementChange(c)	Modelware, Loconsole, Baumert
44	number of input states per function (A)	CapabilityRequirement(c), State(c)	Davis
45	number of states per use cases	UseCase(c), State(c)	Douglass
46	number of activities per use cases	UseCase(c), Activity(c)	Douglass, Henderson-Sellers
47	number of activities in the main flow per use case	UseCase(c), Activity(c), Flow(c)	Henderson-Sellers
48	number of activities per alternative flow per use case	UseCase(c), Activity(c), Flow(c)	Henderson-Sellers
49	number of activities in the alternative flows per use case	Activity(c), Flow(c), UseCase(c)	Henderson-Sellers
50	number of activities per actor	Activity(c), Actor(c)	Henderson-Sellers, Marchesi
51	number of activities per goal	Activity(c), Goal(c)	Henderson-Sellers
52	number of goals per stakeholder	Goal(c), StakeHolder(c)	Henderson-Sellers
53	number of dependencies per use case (includes, extends)	UseCase(c), includes(r), extends(r)	Douglass
54	number of base-lined requirements	Requirement(c), Baseline(c)	Loconsole, Baumert

55	number of requirements changes to a requirements baseline	RequirementChange(c), Baseline(c)	Kolde
56	number of requirements by responsible	Requirement(c), Responsible(c)	Kolde
57	number of responsible by requirements	Requirement(c), Responsible(c)	Loconsole
58	number of functional requirements allocated to a project release	CapabilityRequirement(c), Realease(c)	Kolde, Loconsole, Baumert
59	strength of an individual requirement	Requirement(c), dependsOn(r), supplierFor(r)	Singh
60	strength of a category	Requirement(c), RequirementCategory(c)	Singh
61	number of req. for which all reviewers presented identical interpretations (NU)	Requirement(c), reviewed(a), Individual(c)	Davis, Loconsole
62	unambiguity	NU/NR	Davis
63	number of input stimulus per function (B)	CapabilityRequirement(c), Stimulus(c)	Davis
64	number of flows per function (C)	CapabilityRequirement(c), Flow(c)	Davis
65	completeness per function	$C/(ABb)$	Davis
66	number of correct requirements (NC)	Requirement(c), correct(a), EndUser(c)	Davis
67	correctness	NC/NR	Davis
68	verifiability $= NR/(NR + \sum_i cost_i + \sum_i time_i)$	Requirement(c), TestCase(c), costOfTest(a)	Davis
69	number of test cases per requirement	Requirement(c), TestCase(c)	Loconsole, Baumert
70	number of fun. that are not deterministic (NUFND)	Requirement(c)	Davis
71	number of req. that describe pure external behavior	CapabilityRequirement(c)	Davis
72	number of req. that describe architecture and algorithm (NAC)	ArchitecturalConstraint(c)	Davis
73	design dependency	simplified to NAC/NR	Davis
74	redundancy	NF/NUF	Davis
75	size of the longest path between the first activity and the final activity	UseCase(c), Activity(c), next(r)	Henderson-Sellers

76	number of alternative flows	Flow(c), alternative(a)	Henderson-Sellers
77	number of stakeholders	StakeHolder(c)	Henderson-Sellers
78	number of goals	Goal(c)	Henderson-Sellers
79	number of changes to req. incorporated into baseline per time frame	Requirements(c), Baseline(c), TimeFrame(c)	Loconsole

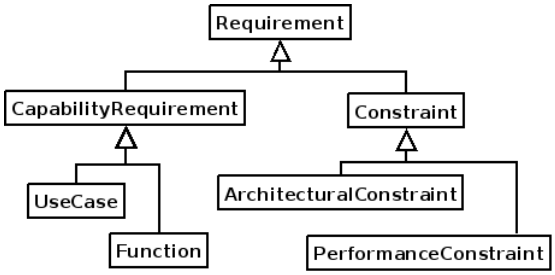

FIG. 4.8 : L'arbre d'héritage des types d'exigences

4.5.4 Métamodèle d'exigences

Dans cette section, nous présentons les principaux aspects du métamodèle obtenu. Le métamodèle complet est présenté en annexe.

Le métamodèle est centré sur la notion d'exigence, comme le montre la figure 4.8. Cependant, une exigence peut être plus finement typée. Une exigence de capacité (*CapabilityRequirement*) spécifie une capacité atomique du système. Cela peut être fait par une description textuelle (la classe *Function*) ou en employant un cas d'utilisation (la classe *UseCase*). Le métamodèle contient une partie dédiée aux cas d'utilisation. Certaines exigences sont exprimées comme des contraintes sur le système (classe *Constraint*), qui peuvent être raffinées en contrainte d'architecture (classe *ArchitecturalConstraint*) ou en contrainte de performance (classe *PerformanceConstraint*). Notons à nouveau que toutes ces classes proviennent de métriques qui s'intéressent à ces types d'exigence. Par exemple, la métrique *nombre d'exigences qui décrivent une contrainte d'architecture ou d'algorithme* [Davis et al., 1993] compte le nombre d'instances de la classe *ArchitecturalConstraint*.

Les principaux attributs et références de la classe *Requirement* sont montrés figure 4.9. Une exigence est identifiée par son ID, dont le codage est dépendant du processus d'ingénierie des exigences. Enfin vient l'exigence elle-même, en une ou quelques phrases maximum, comme le préconise les bonnes pratiques. Notons que ce métamodèle n'a pas pour but de modéliser à un grain plus fin, donc ne modélise pas le contenu des exigences. Une date de création est un marquage pour la traçabilité. Plusieurs métriques concernent l'histoire d'une exigence, en conséquence une exigence peut être marquée comme version courante, tout en gardant un lien de traçabilité vers les versions plus anciennes avec la référence *pastVersions*. À chaque exigence est associé un ou plusieurs statuts, instance de la classe *Status*, qui n'apparaît pas sur la figure. Les différents statuts ne sont pas codés dans le métamodèle mais dans une librairie dépendante du processus. Les exigences peuvent être structurées en catégories, c'est pourquoi il y a un attribut *catégorie*. Une catégorie est un groupe logique d'exigences atomiques, qui facilite la compréhension des exigences. Par exemple, une catégorie peut correspondre à une macro-fonction du système e.g.; *Système de divertissement* dans un avion de ligne. Une exigence est associée à des cas de test (la classe *TestCase*). La modélisation des cas de test est en dehors du périmètre de ce métamodèle. Enfin, une exigence est associée à zéro ou plusieurs items logiciels. Les associations restantes expriment la décomposition en sous-exigences, et les liens de dépendance entre exigences. Ce dernier point est équivalent aux matrices de dépendance de la

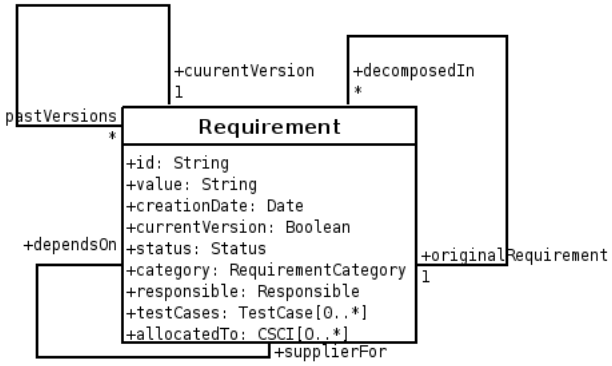

FIG. 4.9 : La classe représentant une exigence

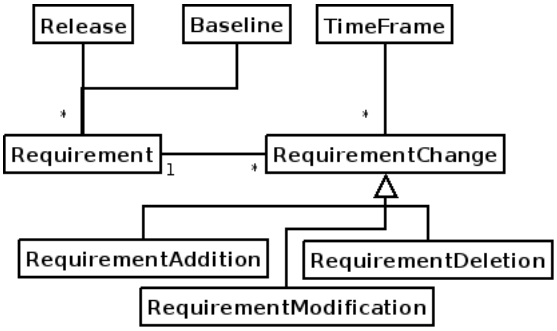

FIG. 4.10 : Gestion du cycle de vie des exigences

littérature [Moisiadis, 2002].

La figure 4.10 montre les concepts liés au cycle de vie des exigences. Cette famille de concepts intervient dans une part importante de métriques de la littérature. Une *Baseline* est composée d'un ensemble d'exigences bien formées. Une *Release* satisfait un ensemble d'exigences. À l'intérieur d'une période, dont le sens dépend du processus d'ingénierie des exigences, il y a plusieurs *RequirementChange*. *RequirementChange* est une classe abstraite spécialisée en *RequirementAddition*, *RequirementModification*, et *RequirementDeletion* (respectivement changement, ajout et suppression d'exigence). Toutes ces classes permettent une description complète du cycle de vie des exigences.

Tous ces concepts, classes, références ou attributs du métamodèle ont été identifiés dans une ou plusieurs métriques de la littérature. Ce métamodèle constitue un cadre:

- dans lequel nous pouvons exprimer les exigences;
- qui est référencé par un modèle de spécification de métriques sur les exigences.

Listing 4.3 : Extrait de la spécification des métriques sur les exigences suivant l'approche MDM

```
1 metric SigmaMetric 01_NOR is
2   description "Total number of requirements
3             (excluding those who are past versions)"
4   elements satisfy "(Requirement.isInstance(self)
```

```
5                      and !self.currentVersion!=void)"
6  endmetric
7
8
9  // (Loconsolle 2001)
10 //     number of requirements affected by a change
11 // (Modelware 2006)
12 //     number of impacted model elements per change
13 metric SetOfElementsPerX NoRpC is
14   elements satisfy "this.isInstanceOf(RequirementChange)"
15   phi2 is  "this.isInstanceOf(Requirement)"
16   references are
17     "requirementChanged, decomposedIn, supplierFor"
18 endmetric
19
20 metric SetOfElementsPerX 04_NRMPTF is
21   description "number of requirements modified per time frame"
22   elements satisfy "TimeFrame.isInstance(self)"
23   count "RequirementModification.isInstance(self)"
24   references followed "changes"
25 endmetric
26
27
28 metric SigmaMetric 08_NORTNLU is
29   description "number of requirements that
30               trace to the next level up"
31   elements satisfy "(Requirement.isInstance(self)
32                     and this.originalRequirement!=void)"
33 endmetric
34
35
36 metric SetOfElementsPerX 13_N is
37   description "number of CSCI linked to a requirement"
38   elements satisfy "Requirement.isInstance(self)"
39   count "CSCI.isInstance(self)"
40   references followed "decomposedIn,allocatedTo"
41 endmetric
42
43 metric SigmaMetric 31_ACUCD is
44   description "number of non sumbitted case diagrams"
45   elements satisfy "(UseCaseDiagram.isInstance(self)
46                     and !exists{x|x.status=='non-submitted'"
47 endmetric
48
49 metric SetOfElementsPerX 46_N is
50   description "number of activities per use cases"
51   elements satisfy "UseCase.isInstance(self)"
52   count "Activity.isInstance(self)"
53   references followed "describedBy,states"
54 endmetric
55
56 end of metric specification
```

Le listing 4.3 montre certaines de ces métriques. Le reste des métriques est donné en annexe. Commentons la première métrique: les quatre premières lignes sont des commentaires qui expliquent que cette métrique est le résultat de la fusion de métriques de [Loconsole, 2001, Modelware Project, 2006]. La métrique est de type *SetOfElementsPerX* et son nom est *NoRpC*. Cette métrique spécifie que nous voulons connaître le nombre d'instances de *Requirement* liés à un *RequirementChange* donné par un chemin ne contenant que des pointeurs dont la référence correspondante est incluse dans `requirementChanged, decomposedIn, supplierFor`.

4.5.5 Discussion

Toutefois, cette étude de cas contient plusieurs axes d'interprétation.

Applicabilité de l'approche MDM en amont du cycle de vie Le but premier de cette étude de cas a été de montrer l'applicabilité de l'approche MDM en amont du cycle de vie. Pour ce faire, nous avons créé un métamodèle d'exigences, et exprimé 127 métriques de la littérature en fonction de ce métamodèle. Cette approche montante (*bottom-up*) pose la question de la validité du métamodèle. Nous pensons que la validité provient du processus de création du métamodèle (cf. figure 4.7): tous les concepts, références et attributs du métamodèle sont issus d'un besoin précis.

Unification des travaux existants Notre étude est constituée de plusieurs contributions sur le sujet des métriques sur les exigences. Chacune de ces contributions adresse un aspect du processus d'ingénierie des exigences. À cause d'hypothèses différentes, de problèmes terminologiques, et de l'ambiguïté du langage naturel, il n'est pas possible en l'état de calculer les valeurs de toutes ces métriques à partir de la même SRS. La métamodélisation des exigences dirigée par les métriques met en place un espace d'expression des exigences cohérent et complet. Il est alors possible d'exprimer les exigences dans un seul langage, c'est à dire comme instance de ce métamodèle, et d'obtenir toutes les valeurs de mesure. Cette étude de cas montre donc le pouvoir d'unification de l'IDM.

Calculabilité La calculabilité des métriques sur les exigences est la possibilité d'obtenir automatiquement des valeurs de mesure à partir d'un document d'exigence. Le problème de la calculabilité pour les exigences réside dans la forme des exigences, par exemple le langage naturel, dont la sémantique est non explicite[14]. L'approche MDM, car dirigée par les modèles, permet une calculabilité totale, à partir d'exigences spécifiées comme instance d'un métamodèle.

Génération automatique des outils liés aux exigences Dans une approche IDM globale, le même métamodèle qui sert à la mesure, sert aussi d'artefact d'entrée à des outils génératifs d'environnement de gestion d'exigence. De même, les modèles

[14] D'ailleurs, nous connaissons seulement deux outils de mesures pour les modèles d'exigences. Le premier, ARM est développé à la NASA et fait une mesure sur des exigences écrites en langage naturel (`http://satc.gsfc.nasa.gov/tools/arm/`). Le second, l'outil *Telelogic Doors*[15] intègre aussi des fonctionnalités de mesure, mais la documentation officielle n'est pas accessible.

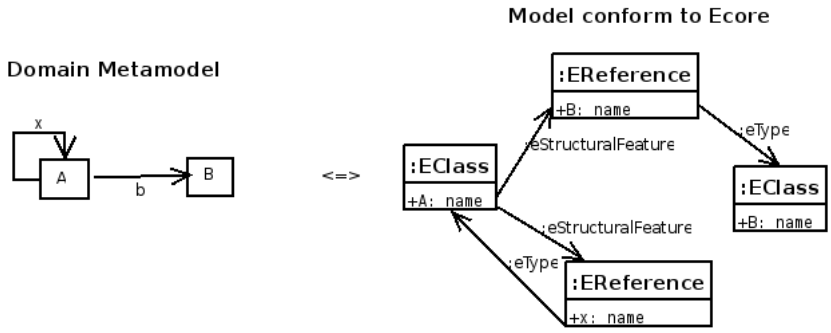

FIG. 4.11 : Un métamodèle de domaine vu en tant que modèle

peuvent servir d'artefact d'entrée à des outils de génération automatique de test ou d'architecture.

Ouverture: extensibilité et adaptabilité Comme la majeure partie du code est généré, l'approche est ouverte. Le métamodèle d'exigences, comme les spécifications de métriques peuvent être adaptées ou étendues pour une compagnie donnée. Par exemple, l'ensemble des statuts pour une exigence peut être réduit ou augmenté, en fonction du processus de validation des exigences. Une compagnie peut aussi ajouter une classe au métamodèle, comme par exemple une classe *BudgetConstraint* héritant de la classe *ConstraintRequirement* (cf. figure 4.8). De façon similaire, les spécifications de métriques sur les exigences peuvent être adaptées. Il est aussi possible d'écrire de nouvelles métriques sur les exigences, non identifiées dans la littérature, afin de mesurer un aspect particulier en ingénierie des exigences. Ces points montrent les possibilités d'ouverture de l'approche MDM pour les exigences.

4.6 Un cas particulier: la mesure des métamodèles *Ecore*

Pouvons-nous appliquer l'approche MDM à la mesure des métamodèles ? Pratiquement, il est souhaitable d'avoir un outil de mesure des métamodèles exprimés en *Ecore* et intégré dans *Eclipse*. Pour l'approche MDM, le domaine considéré est celui de la métamodélisation. Le métamodèle de domaine est le métamodèle des métamodèles *Ecore*. Les modèles de domaine sont donc des métamodèles *Ecore* considérés en tant que modèle. Ce dernier point est illustré sur la figure 4.11. Ce point de vue réflexif est aussi celui de l'implémentation EMF. En conséquence, il est possible d'appliquer l'approche MDM à la mesure des métamodèles *Ecore* eux-mêmes. Certaines métriques sont présentées dans le listing 4.4. La capture d'écran de la figure 4.12 montre l'outil de mesure des métamodèles *Ecore*.

Listing 4.4 : Extrait des métriques pour *Ecore*

```
1
2
3 metric SigmaMetric NOAC is
4   description "number of abstract classes"
5   elements satisfy "(EClass.isInstance(self)
6                     and this.abstract == true))"
```

```
7  endmetric
8
9
10 metric DerivedMetric Abstractness is
11   description "Abstractness of the metamodel see jdepend"
12   formula is "(NOAC/NOC)"
13 endmetric
14
15
16 metric SigmaMetric NOA is
17   description "number of attributes (primitive types/slots)"
18   elements satisfy "EAttribute.isInstance(self)"
19 endmetric
20
21
22 metric SigmaMetric NOR is
23   description "number of references"
24   elements satisfy "EReference.isInstance(self)"
25 endmetric
26
27 metric DerivedMetric DataQuantity is
28   description "Data Quantity Metric
29            (see [Measuring Models], Monperrus et al."
30   formula is "(NOA/(NOA+NOR))"
31 endmetric
32
33 // the concept of assocation is not reified in EMOF
34 // we have to use this trick
35 metric DerivedMetric NoAc is
36   description "number of associations"
37   formula is "(norwo/2)"
38
39   metric SigmaMetric norwo is
40     description "part of number of associations"
41     elements satisfy "(EReference.isInstance(self)
42                      and this.eOpposite!=void)"
43   endmetric
44
45 endmetric
46
47
48 // strange thing in ecore,
49 // an exception can be a datatype
50 // we want to identify these things
51 metric PredicateBasedTau ExceptionIsDataType is
52   source satisfies "EOperation.isInstance(self)"
53   target satisfies "EDataType.isInstance(self)"
54 endmetric
55
56
57
58 // En Ecore, pour connaître le nombre de classes couplées
59 // à une autre,  il faut commencer par déclarer une
```

```
60 // spécification  de métrique AboutMe, qui sélectionne
61 // les classes  une par une, et pour chacune,
62 // recalcule sur le modèle entier la métrique interne,
63 // ici une métrique Sigma.
64
65 metric AboutMeMetricSpecification CBO is
66   description "coupling to this class"
67   elements satisfy "EClass.isInstance(self)"
68   internal metric spec is metric SigmaMetric CBO_ is
69   description "used for CBO"
70   elements satisfy "this.eType==__me__"
71 endmetric
72
73
74 metric SetOfEdgesPerX NoPpC is
75   description "number of parents per class"
76   references followed "eSuperTypes"
77 endmetric
78
79 metric PathLengthMetricSpecification DIT is
80   description "depth in inheritance tree"
81   elements satisfy "EClass.isInstance(self)"
82   references followed "eSuperTypes"
83 endmetric
```

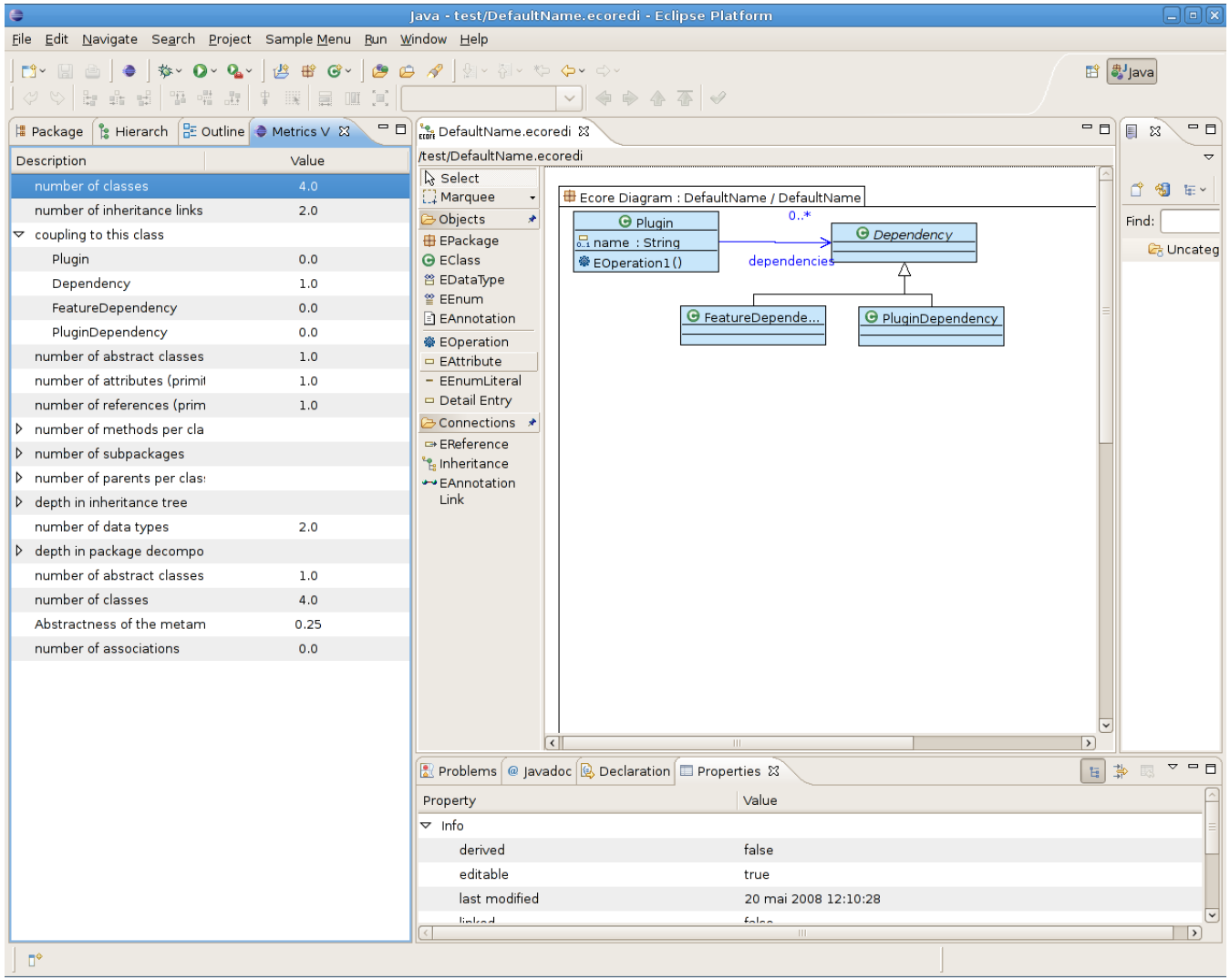

FIG. 4.12 : Outil de mesure des métamodèles *Ecore*

5 Conclusion & Perspectives

Conclusion

Nous avons présenté une approche de la mesure des modèles entièrement dirigée par les modèles. Suivant cette approche, il est possible de spécifier des métriques sur les modèles comme des modèles structurés par un métamodèle de métriques. A partir de cette spécification déclarative, la production de l'outil de mesure des modèles est complètement automatisée. L'automatisation permet de s'affranchir des complexités accidentelles telle l'intégration dans un environnement de modélisation.

Nous avons discuté en 3.5 des caractéristiques de l'approche proposée afin de prouver qu'elle répond aux problèmes réels de la mesure des artefacts logiciels. Par exemple, l'approche MDM permet d'intégrer la mesure dans le processus de création des modèles, et donc de lui donner le rôle de boucle de rétroaction. A chaque modification est fait un ensemble de mesure; pour chaque valeur de mesures, l 'ingénieur peut analyser et valider les propriétés de son modèle.

La validation de l'approche MDM a été effectuée grâce à quatre études de cas qui ont montré son applicabilité et sa généricité, c'est-à-dire son indépendance vis-à-vis du domaine et du cycle de vie.

Perspectives

Il nous reste maintenant à présenter les perspectives de recherche ouvertes par ces travaux. Ces perspectives sont organisées en quatre parties. Dans une première partie, nous listerons les instances de notre approche qui nous semblent être importantes. Dans une seconde partie, nous proposerons quelques pistes liées à la productivité de l'IDM. Dans une troisième partie, nous verrons que les modèles de métriques peuvent être couplées à d'autres approches basées sur des modèles. Enfin, nous verrons qu'il est possible d'utiliser les modèles de métriques à d'autres fins que celle présentée dans cette thèse.

Application de l'approche dans d'autres domaines

Une perspective logique de cette thèse est d'appliquer notre approche sur d'autres modèles que les modèles Java, AADL, de patrouille maritime ou d'exigences.

Métriques sur les processus de développement logiciel spécifiés par un métamodèle de processus

La modélisation des processus de développement logiciel est un domaine de recherche et d'application actif, comme en témoigne la standardisation de SPEM [OMG, 2008] par l'OMG. Cette modélisation a pour but d'automatiser et d'outiller le

développement pour réduire les coûts en enlevant des tâches répétitives et augmenter la qualité en vérifiant des règles de qualité automatiquement.

La mesure des modèles de processus a été explorée une première fois dans [Garcia et al., 2004]. Nous proposons donc de mesurer des modèles de processus exprimés dans un métamodèle par exemple SPEM. La contribution serait alors du même ordre que celles sur les exigences, la modélisation permettrait une mesure effective des processus par rapport à la mesure de processus écrit en langage naturel.

Métriques sur les transformations de modèles

La transformation de modèles est un pilier de l'IDM [Schmidt, 2006]. Comme c'est un domaine récent, et endogène à l'IDM, la plupart des outils de transformation de modèles donnent directement accès aux transformations en tant que modèle par rapport à un métamodèle de transformation. Par exemple, un programme *Kermeta* peut être analysé comme modèle.

La mesure des transformations de modèles est un domaine émergent [Saeki et Kaiya, 2006]. Elle pourrait permettre d'évaluer leur complexité ou encore de calibrer des modèles de coût. Comme les transformations sont disponibles comme modèle, il est possible d'étudier ce point en appliquant l'approche MDM.

Métriques à l'exécution (*@runtime*)

Un des courants de recherche de l'IDM consiste à utiliser les modèles à l'exécution, pour par exemple, avoir des logiciels auto-adaptables et auto-reconfigurables. Il nous semble que notre approche pourrait être fertile dans ce contexte. Prenons un exemple.

Soit un logiciel qui maintient un modèle de son environnement à jour à l'exécution et qui adapte son comportement en fonction de mesures faites sur ce modèle (par exemple, une station *Wifi* qui adapte sa puissance au nombre de répétiteurs environnants). L'approche MDM permettrait une grande agilité dans la définition et l'obtention des mesures sur les modèles utilisés à l'exécution.

Comparaison des métamodèles

Une des questions récurrentes en IDM réside dans l'évaluation du pouvoir d'expression des métamodèles, et de la comparaison des métamodèles. Une classe d'équivalence de métamodèle peut être définie comme l'ensemble des métamodèles sur lesquels on peut avoir des transformations bi-directionnelles.

Nous pensons qu'une classe d'équivalence de N métamodèles pourrait être définie par l'obtention des mêmes mesures sur N ensembles de modèles. Ce point est illustré sur la figure 5.1.

À propos du gain de productivité

Dans cette thèse, nous avons émis l'hypothèse qu'une approche générative pour les logiciels de mesure des modèles permettrait de réduire leur coût. Il est nécessaire de continuer dans cette direction pour à la fois améliorer et valider empiriquement un éventuel gain de productivité.

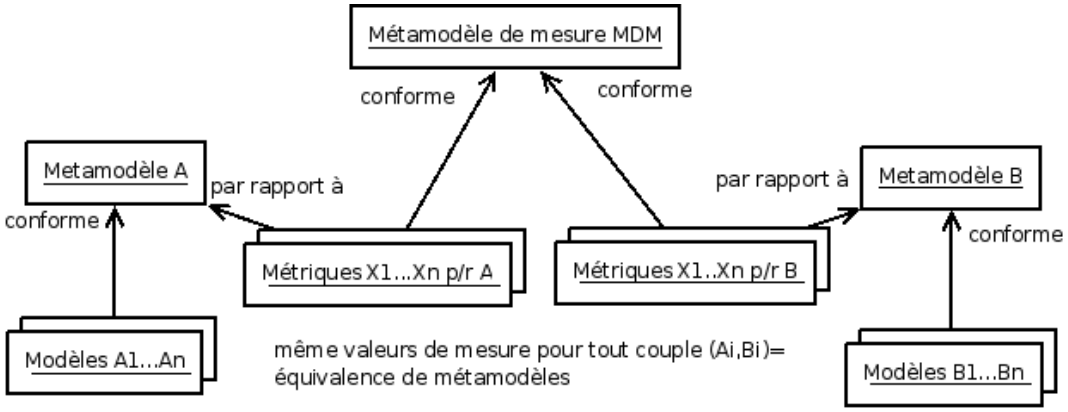

FIG. 5.1 : Classe d'équivalence des métamodèles définie par la mesure

Mise en place d'une expérience contrôlée

Afin de mesurer ce gain de productivité, il faudrait mener une expérience empirique. Les hypothèses seraient les suivantes.

H0: L'approche MDM ne réduit pas le coût du développement des logiciels de mesure des modèles; H1: L'approche MDM réduit le coût du développement des logiciels de mesure des modèles.

Les sujets de l'expérience seraient deux groupes de N développeurs qui reçoivent la même spécification, par exemple pour mesurer des modèles AADL. Le groupe 1 aurait le générateur de composant de mesure des modèles, et la documentation de l'approche MDM (par exemple cette thèse). Le groupe B n'aurait que les outils de développement classique (*Eclipse/Java/EMF*). On récolterait les N durées de développement nécessaires pour obtenir le logiciel final. Le test statistique associé est un test de *Student* à deux échantillons [Morgenthaler, 2007, p.156].

Syntaxe pour les modèles de métriques

Nous avons défini dans cette thèse une proto-syntaxe pour les métriques spécifiées avec l'approche MDM. Les modèles de métriques sont voués à être écrit par des experts d'un domaine précis. Ces experts peuvent n'avoir absolument aucune culture du logiciel.

Nous faisons l'hypothèse que plus la syntaxe est intuitive: 1) plus l'approche MDM a de chance d'être acceptée et 2) moins long sera le temps passé à définir des métriques.

Pour rendre la spécification des mesures efficace, il faut que la syntaxe associée soit la plus intuitive possible. La définition d'une syntaxe optimale nécessite d'être étudiée non seulement d'un point de vue informatique mais aussi d'un point de vue psychologique et cognitif. Une syntaxe pourrait être textuelle, tel un langage de programmation ou un langage naturel contraint. Elle pourrait être aussi graphique, dans l'esprit de l'approche *GraphLog* [Mendelzon et Sametinger, 1995]. De même, et du même point de vue cognitif, on pourrait envisager un guidage automatique de la spécification des métriques par une interface graphique ou par un jeu de questions/-réponses posées à l'expert du domaine (une sorte de *wizard*).

Insertion dans une métamodélisation à un grain différent

Notre approche de la mesure est totalement basée sur un métamodèle de spécification de métriques. Ce métamodèle pourrait être couplé à d'autres métamodèles, qui abordent un point ou un grain d'analyse différent.

Liens avec d'autres métamodélisations de la mesure

La métamodélisation de la mesure repose sur trois points:
- la métamodélisation du processus de mesure [Kitchenham et al., 2001] et de l'interprétation des résultats;
- la métamodélisation des résultats de la mesure[Kitchenham et al., 2001];
- la métamodélisation de la définition des métriques.

Notre contribution se situe sur ce troisième point. D'un point unificateur, et dans le but de définir une approche globale de la mesure des logiciels et des systèmes, une perspective de recherche consiste à composer le métamodèle de l'approche MDM avec d'autres contributions de la littérature qui s'attaquent aux deux autres points.

La mesure dans les modèles de processus de développement

La mesure est un point important dans les modèles de processus de développement comme en témoigne la présence de la classe *Metric* dans le métamodèle SPEM. Il serait donc possible de donner plus de sémantique à cette classe, en la spécialisant comme métrique de l'approche MDM.

Ce faisant, les approches génératives à partir des modèles de processus pourraient être enrichies par le logiciel de mesure des modèles lui-même généré. L'exécutabilité des processus de développement IDM passe par des enchaînements automatiques de transformations de modèles. On pourrait coupler ces transformations à des mesures automatiques sur les modèles d'entrée et de sortie.

De la macro à la micro-métamodélisation des exigences

Le but de notre étude de cas sur l'ingénierie des exigences était de montrer l'application de l'approche MDM en tout début du cycle de vie, et sur des modèles totalement indépendants du logiciel.

Par ricochet, nous l'avons souligné en conclusion de la section correspondante, notre étude de cas constitue un argument en faveur de l'IDM car elle montre une possibilité de métamodélisation des exigences et son pouvoir d'unification.

Il se trouve que les métriques sur les exigences de la littérature s'arrêtent au niveau de l'exigence, considérée comme une ou quelques phrases. Or des travaux explorent la métamodélisation d'une exigence elle-même [Nebut et al., 2003, Brottier et al., 2007]. Une perspective de nos travaux sur les exigences pourrait être de coupler notre métamodèle, de type macroscopique, à un métamodèle des exigences elles-mêmes, de type microscopique.

La possibilité d'étendre le métamodèle cible

La puissance d'expression du métamodèle de spécification de métriques dépend des éléments du métamodèle de domaine considéré. Nous avons montré en 4.2 que

certaines métriques ne peuvent être exprimées de manière déclarative du fait de l'absence du concept correspondant dans le métamodèle de domaine (e.g; le concept de paire de fonctions dans le métamodèle de *SpoonEMF*). Une perspective de cette thèse consiste à étudier la possibilité d'ajouter des concepts dans le métamodèle de domaine de manière non intrusive, afin de permettre la spécification de nouvelles métriques. Cet ajout agirait donc comme un trait d'union entre l'approche MDM et le métamodèle de domaine.

Utiliser les modèles de métriques pour mesurer les implémentations

Notre approche a pour but principal de mesurer des modèles. Nous avons ainsi présenté la mesure des modèles AADL par des métriques spécifiées comme modèle, instance du métamodèle de spécification de métriques.

La mesure effective est obtenue en générant entièrement l'outil de mesure. Dans notre prototype, cet outil de mesure est un composant *Eclipse* de type *plugin*, entièrement écrit en Java.

Reprenons un instant l'étude de cas sur les modèles AADL. Les modèles AADL définissent l'architecture des systèmes embarqués. L'approche IDM permet de générer automatiquement une implémentation à partir de cette architecture abstraite. Dans le cas des systèmes embarqués critiques, cette implémentation est très largement testée, statiquement et dynamiquement. Les tests dynamiques sont effectués par des sondes ou des espions, développés ad hoc, et qui sont branchés directement sur la carte électronique embarquée.

De la même manière que nous utilisons le modèle de mesure pour générer l'outil de mesure des modèles, nous pourrions utiliser le modèle de mesure pour générer l'outil de mesure de l'implémentation. Dans le cas d'AADL, pour générer effectivement la sonde[1] qui, sur telle carte électronique et tel OS, va mesurer le nombre de *threads* en cours d'exécution.

Suivant cette idée, il devient possible d'obtenir automatiquement deux outils de mesure, sur le modèle et sur l'implémentation, et de comparer les valeurs de mesure prédites par le modèle et effectives sur l'implémentation, ce qui pourrait être considéré comme une validation croisée à la fois du modèle, de l'implémentation, et du processus de génération de cette dernière.

Sur cette dernière perspective se termine l'exposé de notre thèse. Nous espérons que nos travaux soient – et resteront – une contribution heureuse issue de la rencontre des deux mondes de la mesure des logiciels et de l'ingénierie dirigée par les modèles.

[1] Une sonde est un composant à la fois matériel et logiciel, qui se branche sur une carte électronique afin de permettre une analyse de l'exécution d'un programme embarquée en conditions réelles. Une sonde est utilisée pour faire des analyses de fautes (*debug*) et de performances (*profiling*).

Bibliographie

[Abounader et Lamb, 1997] Abounader, J. et Lamb, D. (1997). A data model for object-oriented design metrics. Technical report, Queen's University, Kingston, ON.

[Alikacem et Sahraoui, 2006] Alikacem, E. et Sahraoui, H. (2006). Generic metric extraction framework. In *Proceedings of IWSM/MetriKon'2006*.

[Andler et al., 2006] Andler, D., Lascar, D., et Sabbagh, G. (2006). Théorie des modèles. In *Encyclopedia Universalis*. Les éditions de l'Encyclopedia Universalis.

[Aspray et al., 1996] Aspray, W., Keil-Slawik, R., et Parnas, D. (1996). The history of software engineering. Technical report, Schloss Dagstuhl.

[Atkinson et Kühne, 2003] Atkinson, C. et Kühne, T. (2003). Model-driven development: A metamodeling foundation. *IEEE Softw.*, 20(5):36–41.

[Axelsson, 2002] Axelsson, J. (2002). Model based systems engineering using a continuous-time extension of the unified modeling language (UML). *Systems Engineering*, 5:165–179.

[Baader et al., 2003] Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., et Patel-Schneider, P. F. (2003). The description logic handbook: Theory, implementation, and applications. In *Description Logic Handbook*. Cambridge University Press.

[Baker, 2004] Baker, A. (2004). Simplicity. In *Stanford Encyclopedia of Philosophy*. The Metaphysics Research Lab, Stanford University.

[Baroni et Abreu, 2002] Baroni, A. et Abreu, F. (2002). Formalizing object-oriented design metrics upon the uml meta-model. In *Proc. of the 16th Brazilian Symposium on Software Engineering*.

[Baroni et al., 2002] Baroni, A., Braz, S., et Abreu, F. (2002). Using OCL to formalize object-oriented design metrics definitions. In *ECOOP'02 Workshop on Quantitative Approaches in OO Software Engineering*.

[Baroni, 2005] Baroni, A. L. (2005). Quantitative assessment of uml dynamic models. In *Proceedings of the doctoral symposium at the 10th European Software Engineering Conference held jointly with 13th International Symposium on Foundations of Software Engineering (ESEC-FSE'05)*, pages 366–369. ACM Press.

[Basili et al., 1996] Basili, V. R., Briand, L. C., et Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761.

[Basili et al., 1994] Basili, V. R., Caldiera, G., et Rombach, H. D. (1994). The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley.

[Batory, 2006] Batory, D. (2006). Multilevel models in model-driven engineering, product lines, and metaprogramming. *IBM Syst. J.*, 45(3):527–539.

[Baumert et McWhinney, 1992] Baumert, J. et McWhinney, M. (1992). Software measures and the capability maturity model. Technical report, Software Engineering Institute, Carnegie Mellon University.

[Berenbach et Borotto, 2006] Berenbach, B. et Borotto, G. (2006). Metrics for model driven requirements development. In *Proceeding of the 28th International Conference on Software Engineering (ICSE '06)*, pages 445–451. ACM Press.

[Beyer et al., 2005] Beyer, D., Noack, A., et Lewerentz, C. (2005). Efficient relational calculation for software analysis. *IEEE Trans. Softw. Eng.*, 31(2):137–149.

[Boag et al., 2007] Boag, S., Chamberlin, D., Fernández, M. F., Florescu, D., Robie, J., et Siméon, J. (2007). XQuery 1.0: An XML query language. Technical report, W3C.

[Boehm, 2006] Boehm, B. (2006). A view of 20th and 21st century software engineering. In *ICSE '06: Proceeding of the 28th International Conference on Software Engineering*, pages 12–29. ACM.

[Boehm et al., 1995] Boehm, B., Clark, B., Horowitz, E., Shelby, R., et Westland, C. (1995). An Overview of the COCOMO 2.0 Software Cost Model. In *Software Technology Conference*.

[Boehm et Sullivan, 2000] Boehm, B. W. et Sullivan, K. J. (2000). Software economics: a roadmap. In *ICSE - Future of SE Track*, pages 319–343.

[Briand et al., 1998] Briand, L. C., Daly, J. W., et Wüst, J. (1998). A unified framework for cohesion measurement in object-oriented systems. *Empirical Softw. Eng.*, 3(1):65–117.

[Briand et al., 1996] Briand, L. C., Morasca, S., et Basili, V. R. (1996). Property-based software engineering measurement. *Software Engineering*, 22(1):68–86.

[Brinkkemper et al., 2001] Brinkkemper, S., Saeki, M., et Harmsen, F. (2001). A method engineering language for the description of systems development methods. In *CAiSE*, pages 473–476.

[Brottier et al., 2007] Brottier, E., Baudry, B., Traon, Y. L., Touzet, D., et Nicolas, B. (2007). Producing a global requirement model from multiple requirement specifications. In *EDOC*, pages 390–404.

[Budinsky et al., 2004] Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., et Grose, T. J. (2004). *Eclipse Modeling Framework*. Addison-Wesley.

[Bézivin, 2005] Bézivin, J. (2005). On the unification power of models. *Software and System Modeling*, 4(2):171–188.

[Cabot et Teniente, 2006] Cabot, J. et Teniente, E. (2006). A metric for measuring the complexity of ocl expressions. In *Model Size Metrics Workshop co-located with MODELS'06*.

[Card, 1991] Card, D. (1991). What makes a software measure successful. *American Programmer*, 4:2–8.

[CEI et ISO, 1993] CEI et ISO (1993). *International vocabulary of basic and general terms in metrology = vocabulaire international des termes fondamentaux et généraux de métrologie*. International Organisation for Standardization.

[Chen, 1976] Chen, P. P. (1976). The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36.

[Cheng et al., 2005] Cheng, B. H. C., Stephenson, R., et Berenbach, B. (2005). Lessons learned from automated analysis of industrial uml class models (an experience report). In *Proceedings of MODELS' 2005*, pages 324–338.

[Chidamber et Kemerer, 1991] Chidamber, S. R. et Kemerer, C. F. (1991). Towards a metrics suite for object-oriented design. In *Proceedings of OOPSLA'91*, pages 197–211.

[Chidamber et Kemerer, 1994] Chidamber, S. R. et Kemerer, C. F. (1994). A metric suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6):293–318.

[Cho et al., 2001] Cho, E. S., Kim, M. S., et Kim, S. D. (2001). Component metrics to measure component quality. In *Proceedings of the 8th Asia-Pacific on Software Engineering Conference (APSEC'01)*, page 419. IEEE Computer Society.

[Chretienne et al., 2004] Chretienne, P., Jean-Marie, A., Le Lann, G., Stefani, J., Atos Origin, et Dassault Aviation (2004). Programme d'Étude Amont Mesure de la compléxité (marché n°00-34-007). Technical report, DGA.

[Clayberg et Rubel, 2006] Clayberg, E. et Rubel, D. (2006). *Eclipse: Building Commercial-Quality Plug-ins*. Addison-Wesley.

[Consens et Mendelzon, 1990] Consens, M. P. et Mendelzon, A. O. (1990). GraphLog: a visual formalism for real life recursion. In *Proc. of the ACM Symp. on Principles of Database Systems*, pages 404–416.

[Costa, 2001] Costa, F. (2001). *Combining meta-information management and reflection in an architecture for configurable and reconfigurable middleware*. PhD thesis, University of Lancaster.

[Costello et Liu, 1995] Costello, R. J. et Liu, D.-B. (1995). Metrics for requirements engineering. *J. Syst. Softw.*, 29(1):39–63.

[Davis et al., 1993] Davis, A., Overmyer, S., Jordan, K., Caruso, J., Dandashi, F., Dinh, A., Kincaid, G., Ledeboer, G., Reynolds, P., Sitaram, P., Ta, A., et Theofanos, M. (1993). Identifying and measuring quality in a software requirements specification. In *Proceedings of the First International Software Metrics Symposium*.

[de Lara et Vangheluwe, 2002] de Lara, J. et Vangheluwe, H. (2002). Atom3: A tool for multi-formalism and meta-modelling. In *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering (FASE'02)*, pages 174–188. Springer-Verlag.

[Demeyer et Ducasse, 1999] Demeyer, S. et Ducasse, S. (1999). Metrics, do they really help? In Publications, H. S., editor, *Proceedings of LMO'99*, pages 69–82.

[Demeyer et al., 2001] Demeyer, S., Tichelaar, S., et Ducasse, S. (2001). Famix 2.1 - the famoos information exchange model. Technical report, University of Bern.

[Dijkstra, 1972] Dijkstra, E. W. (1972). The humble programmer. *Commun. ACM*, 15:859–866.

[Douglass, 2004] Douglass, B. P. (2004). Computing model complexity. White paper, I-Logix.

[Ducasse et Gîrba, 2006] Ducasse, S. et Gîrba, T. (2006). Using Smalltalk as a reflective executable meta-language. In *International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006)*, volume 4199 of *LNCS*, pages 604–618. Springer-Verlag.

[Ducasse et al., 2008] Ducasse, S., Gîrba, T., Kuhn, A., et Renggli, L. (2008). Meta-environment and executable meta-language using smalltalk: an experience report. *Journal of Software and Systems Modeling (SOSYM)*.

[Dufour et al., 2003] Dufour, B., Driesen, K., Hendren, L., et Verbrugge, C. (2003). Dynamic metrics for java. In *Proceedings of the ACM SIGPLAN 2003 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, pages 149–168. ACM Press.

[Eichberg et al., 2006] Eichberg, M., Germanus, D., Mezini, M., Mrokon, L., et Schäfer, T. (2006). Qscope: an open, extensible framework for measuring software projects. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR'06)*.

[El Wakil et al., 2005] El Wakil, M., El Bastawissi, A., Boshra, M., et Fahmy, A. (2005). A novel approach to formalize and collect object-oriented design-metrics. In *Proceedings of the 9th International Conference on Empirical Assessment in Software Engineering*.

[Estefan, 2007] Estefan, J. A. (2007). Survey of model-based systems engineering (MBSE) methodologies. Technical report, INCOSE MBSE Focus Group.

[Favre, 2006] Favre, J.-M. (2006). Megamodelling and etymology. In Cordy, J. R., Lämmel, R., et Winter, A., editors, *Transformation Techniques in Software Engineering*, number 05161 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany.

[Feiler et al., 2006] Feiler, P. H., Gluch, D. P., et Hudak, J. J. (2006). The architecture analysis and design language (aadl): An introduction. Technical report, CMU/SEI.

[Fenton, 1991] Fenton, N. E. (1991). *Software Metrics: A Rigorous Approach*. Chapman and Hall.

[Fenton et Neil, 1999] Fenton, N. E. et Neil, M. (1999). A critique of software defect prediction models. *IEEE Trans. Softw. Eng.*, 25(5):675–689.

[Ferneley, 1997] Ferneley, E. (1997). An empirical study of coupling and control flow metrics. *Information and Software Technology*, 39(13):879–887.

[France et Rumpe, 2007] France, R. et Rumpe, B. (2007). Model-driven development of complex software: A research roadmap. In *Future of Software Engineering (FOSE'07)*, pages 37–54. IEEE Computer Society.

[Frigg et Hartmann, 2006] Frigg, R. et Hartmann, S. (2006). Models in science. In *Stanford Encyclopedia of Philosophy*. The Metaphysics Research Lab.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., et Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA, USA.

[Garcia et al., 2004] Garcia, F., Ruiz, F., et Piattini, M. (2004). Definition and empirical validation of metrics for software process models. In *Proceedings of the International Conference on Product Focused Software Process Improvement (PROFES'2004)*.

[Garcìa et al., 2006] Garcìa, F., Bertoa, M. F., Calero, C., Vallecillo, A., Ruiz, F., Piattini, M., et Genero, M. (2006). Towards a consistent terminology for software measurement. *Information and Software Technology*, 48(8):631–644.

[Genero et al., 2002a] Genero, M., Miranda, D., et Piattini, M. (2002a). Defining and validating metrics for uml statechart diagrams. In *Proceedings of QAOOSE'2002*.

[Genero et al., 2000] Genero, M., Piattini, M., et Calero, C. (2000). Early measures for UML class diagrams. *L'Objet*, 6(4):489–505.

[Genero et al., 2002b] Genero, M., Piattini, M., et Calero, C. (2002b). Empirical validation of class diagram metrics. In *Proceedings of the International Symposium on Empirical Software Engineering (ISESE'02)*, page 195. IEEE Computer Society.

[Genero et al., 2005] Genero, M., Piattini, M., et Caleron, C. (2005). A survey of metrics for UML class diagrams. *Journal of Object Technology*, 4:59–92.

[Gill et Grover, 2003] Gill, N. S. et Grover, P. S. (2003). Component-based measurement: few useful guidelines. *SIGSOFT Softw. Eng. Notes*, 28(6):4–4.

[Grady, 1992] Grady, R. B. (1992). *Practical software metrics for project management and process improvement*. Prentice-Hall, Inc.

[Granger, 1993] Granger, G.-G. (1993). *La science ou les sciences*. PUF.

[Guerra et al., 2007] Guerra, E., de Lara, J., et Díaz, P. (2007). Visual specification of measurements and redesigns for domain specific visual languages. *Journal of Visual Languages and Computing*, in Press:1–27.

[Guerra et al., 2006] Guerra, E., Diaz, P., et de Lara, J. (2006). Visual specification of metrics for domain specific visual languages. In *Graph Transformation and Visual Modeling Techniques (GT-VMT 2006)*.

[Hammer et al., 1997] Hammer, T., Rosenberg, L., Huffman, L., et Hyatt, L. (1997). Requirements metrics - value added. In *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE'97)*, page 141.1. IEEE Computer Society.

[Harel et Rumpe, 2004] Harel, D. et Rumpe, B. (2004). Meaningful modeling: what's the semantics of semantics? *Computer*, 37(10):64–72.

[Harmer et Wilkie, 2002] Harmer, T. J. et Wilkie, F. G. (2002). An extensible metrics extraction environment for object-oriented programming languages. In *Proceedings of the International Conference on Software Maintenance*.

[Henderson-Sellers, 1996] Henderson-Sellers, B. (1996). *Object-Oriented Metrics, measures of complexity*. Prentice Hall.

[Henderson-Sellers et al., 2002] Henderson-Sellers, B., Zowghi, D., Klemola, T., et Parasuram, S. (2002). Sizing use cases: How to create a standard metrical approach. In *Proceedings of the 8th International Conference on Object-Oriented Information Systems (OOIS '02)*, pages 409–421. Springer-Verlag.

[Henry et Kafura, 1981] Henry, S. et Kafura, D. (1981). Software structure metrics based on information flow. *IEEE Trans. Softw. Eng.*, 7(5):510–518.

[Hitz et Montazeri, 1996] Hitz, M. et Montazeri, B. (1996). Chidamber and kemerer's metrics suite: A measurement theory perspective. *IEEE Trans. Softw. Eng.*, 22(4):267–271.

[Ince et al., 2000] Ince, A., Topuz, E., Panayirci, E., et Isik, C. (2000). *Principles of Integrated Maritime Surveillance Systems*, volume 527. Kluwer Academic Publishers.

[ISO/IEC, 2001] ISO/IEC (2001). *ISO/IEC 9126. Software engineering – Product quality.* ISO/IEC.

[Jones, 1991] Jones, C. (1991). *Applied software measurement: assuring productivity and quality.* McGraw-Hill.

[Jouault et Bézivin, 2006] Jouault, F. et Bézivin, J. (2006). Km3: a dsl for metamodel specification. In *Proceedings of the 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems.*

[Jouault et Kurtev, 2005] Jouault, F. et Kurtev, I. (2005). Transforming models with ATL. In *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005.* Workshop Model Transformations in Practice, part of the MoDELS 2005 Conference.

[Jézéquel, 2008] Jézéquel, J.-M. (2008). Model driven design and aspect weaving. *Journal of Software and Systems Modeling (SoSyM)*, 7(2):209–218.

[Kamata et Tamai, 2007] Kamata, M. I. et Tamai, T. (2007). How does requirements quality relate to project success or failure? In *Proceedings of the15th IEEE International Requirements Engineering Conference (RE'07)*, pages 69–78.

[Kan, 1995] Kan, S. H. (1995). *Metrics and Models in Software Quality Engineering.* Addison Wesley, Reading, MA.

[Kastenberg, 2004] Kastenberg, H. (2004). Software metrics as class graph properties. Master's thesis, University of Twente.

[Kent, 2002] Kent, S. (2002). Model driven engineering. In *Proceedings of the Third International Conference Integrated Formal Methods (IFM'2002).*

[Khoshgoftaar et Munson, 1990] Khoshgoftaar, T. et Munson, J. (1990). The lines of code metric as a predictor of program faults: A critical analysis. In *Proceedings of the Computer Software and Applications Conference*, pages 408–413.

[Kiczales et al., 1991] Kiczales, G., des Rivières, J., et Bobrow, D. G. (1991). *The Art of the Metaobject Protocol.* The MIT Press.

[Kieburtz et al., 1996] Kieburtz, R. B., McKinney, L., Bell, J. M., Hook, J., Kotov, A., Lewis, J., Olivia, D. P., Sheard, T., Smith, I., et Walton, L. (1996). A software engineering experiment in software component generation. In *Proceeding of the 18th International Conference on Software Engineering (ICSE'96)*, pages 542–552.

[Kim et Boldyreff, 2002] Kim, H. et Boldyreff, C. (2002). Developing software metrics applicable to UML models. In *Proceedings of QAOOSE'2002.* QAOOSE 2002.

[Kitchenham et al., 2001] Kitchenham, B. A., Hughes, R. T., et Linkman, S. G. (2001). Modeling software measurement data. *IEEE Trans. Softw. Eng.*, 27(9):788–804.

[Kitchenham et al., 1990] Kitchenham, B. A., Pickard, L. M., et Linkman, S. J. (1990). An evaluation of some design metrics. *Softw. Eng. J.*, 5(1):50–58.

[Kolde, 2004] Kolde, C. (2004). Basic metrics for requirements management. White paper, Borland.

[Krantz et al., 1971] Krantz, D., Luce, R., Suppes, P., et Tversky, A. (1971). *Foundations of measurement*. Academic Press, New York.

[Kuehne, 2006] Kuehne, T. (2006). Matters of (meta-) modeling. *Software and System Modeling*, 5(4):369–385.

[LaLonde et Pugh, 1994] LaLonde, W. R. et Pugh, J. R. (1994). Gathering metric information using metalevel facilities. *JOOP*, 7(1):33–37.

[Lanza et Ducasse, 2002] Lanza, M. et Ducasse, S. (2002). Beyond language independent object-oriented metrics: Model independent metrics. In Abreu, F., Piattini, M., Poels, G., et Sahraoui, H. A., editors, *Proceedings of the 6th International Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, pages 77–84.

[Lavazza et Agostini, 2005] Lavazza, L. et Agostini, A. (2005). Automated measurement of uml models: an open toolset approach. *Journal of Object Technology*, 4(4):115–134.

[Ledeczi et al., 2001] Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., et Volgyesi, P. (2001). The generic modeling environment. In *Proceedings of the Workshop on Intelligent Signal Processing (WISP'2001)*.

[Lehman, 1994] Lehman, M. (1994). Models and modelling in software engineering. In J.Marciniak, editor, *Encyclopaedia of Software Engineering*, volume 1, pages 698 – 702. Wiley and Co.

[Lethbridge et al., 2004] Lethbridge, T., Tichelaar, S., et Ploedereder, E. (2004). The dagstuhl middle metamodel: A schema for reverse engineering. *Electr. Notes Theor. Comput. Sci.*, 94:7–18.

[Lewerentz et Simon, 1998] Lewerentz, C. et Simon, F. (1998). A product metrics tool integrated into a software development environment. In *Proceedings of the Workshop on Object-Oriented Technology at ECOOP'98*.

[Lo, 1992] Lo, B. (1992). Syntactical construct based apar projection. Technical report, IBM Santa Teresa Laboratory Technical Report, California.

[Loconsole, 2001] Loconsole, A. (2001). Measuring the requirements management key process area. In *Proceedings of the 12th European Software Control and Metrics Conference (ESCOM'2001)*.

[Mahmood et Lai, 2005] Mahmood, S. et Lai, R. (2005). Measuring the complexity of a uml component specification. In *QSIC '05: Proceedings of the Fifth International Conference on Quality Software*, pages 150–160, Washington, DC, USA. IEEE Computer Society.

[Marchesi, 1998] Marchesi, M. (1998). OOA metrics for the Unified Modeling Language. In *Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR'98)*, page 67. IEEE Computer Society.

[Marinescu et al., 2005] Marinescu, C., Marinescu, R., et Gîrba, T. (2005). Towards a simplified implementation of object-oriented design metrics. In *IEEE METRICS*, page 11.

[Martin, 2000] Martin, R. C. (2000). Design principles and design patterns.

[McCabe, 1976] McCabe (1976). A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320.

[McQuillan et Power, 2006a] McQuillan, J. et Power, J. (2006a). Towards the reusability of software metrics definitions at the meta level. In *Proceedings of the ECOOP'2006 Doctoral Symposium*.

[McQuillan et Power, 2006b] McQuillan, J. A. et Power, J. F. (2006b). Experiences of using the dagstuhl middle metamodel for defining software metrics. In *Proceedings of the 4th International Conference on Principles and Practices of Programming in Java*.

[Mellor et al., 2003] Mellor, S. J., Clark, A. N., et Futagami, T. (2003). Guest editors' introduction: Model-driven development. *IEEE Softw.*, 20(5):14–18.

[Mendelzon et Sametinger, 1995] Mendelzon, A. et Sametinger, J. (1995). Reverse engineering by visualizing and querying. *Software - Concepts and Tools*, 16(4):170–182.

[Mens et Lanza, 2002] Mens, T. et Lanza, M. (2002). A graph-based metamodel for object-oriented software metrics. *Electronic Notes in Theoretical Computer Science*, 72:57–68.

[Metrotrade Project et Regmet Project, 2003] Metrotrade Project et Regmet Project (2003). *Metrology - in short*. European Association of National Metrology Institutes.

[Mettala et Graham, 1992] Mettala, E. et Graham, M. H. (1992). The domain specific software architecture program. Technical Report MU/SEI-92-SR-009, Software Engineering Institute.

[Miller et Mukerji, 2003] Miller, J. et Mukerji, J. (2003). Mda guide version 1.0.1. Technical report, Object Management Group (OMG).

[Misic et Moser, 1997] Misic, V. B. et Moser, S. (1997). From formal metamodels to metrics: An object-oriented approach. In *Proceedings of the Technology of Object-Oriented Languages and Systems Conference (TOOLS'97)*, page 330.

[Modelware Project, 2006] Modelware Project (2006). D2.2 MDD Engineering Metrics Definition. Technical report, Framework Programme Information Society Technologies.

[Moisiadis, 2002] Moisiadis, F. (2002). The fundamentals of prioritising requirements. In *Proceedings of the Systems Engineering, Test and Evaluation Conference (SETE'2002)*.

[Monperrus et al., 2007] Monperrus, M., Champeau, J., et Hoeltzener, B. (2007). Counts count. In *Proceedings of the 2nd Workshop on Model Size Metrics (MSM'07) co-located with MODELS'2007*.

[Monperrus et al., 2008a] Monperrus, M., Jaozafy, F., Marchalot, G., Champeau, J., Hoeltzener, B., et Jézéquel, J.-M. (2008a). Model-driven simulation of a maritime

surveillance system. In *Proceedings of the 4th European Conference on Model Driven Architecture Foundations and Applications (ECMDA'2008)*.

[Monperrus et al., 2008b] Monperrus, M., Jézéquel, J.-M., Champeau, J., et Hoeltzener, B. (2008b). Measuring models. In Rech, J. et Bunse, C., editors, *Model-Driven Software Development: Integrating Quality Assurance*. IDEA Group.

[Monperrus et al., 2008c] Monperrus, M., Jézéquel, J.-M., Champeau, J., et Hoeltzener, B. (2008c). Model-driven engineering metrics for real time systems. In *Proceedings of the 4th European Congress on Embedded Real Time Software (ERTS'2008)*.

[Monperrus et al., 2008d] Monperrus, M., Jézéquel, J.-M., Champeau, J., et Hoeltzener, B. (2008d). A model-driven measurement approach. In *Proceedings of the ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MODELS'2008)*.

[Mora et al., 2008] Mora, B., García, F., Ruiz, F., Piattini, M., Boronat, A., Gómez, A., Carsí, J. Á., et Ramos, I. (2008). Software measurement by using qvt transformations in an mda context. In *Proceedings of the International Conference on Enterprise Information Systems (ICEIS'2008)*.

[Morgenthaler, 2007] Morgenthaler, S. (2007). *Introduction à la statistique*. Presses polytechniques et universitaires romandes, 3 edition.

[Mouloud, 2006] Mouloud, N. (2006). Modèle. In *Encyclopedia Universalis*. Les éditions de l'Encyclopedia Universalis.

[Muller et al., 2005] Muller, P. A., Fleurey, F., et Jézéquel, J. M. (2005). Weaving executability into object-oriented meta-languages. In *Proceedings of MODELS/UML 2005*.

[Müller et Simmons, 2002] Müller, A. et Simmons, G. (2002). Exception handling: Common problems and best practice with java 1.4. In *Proceedings of Net.ObjectDays'2002*.

[Nakatani et al., 2001] Nakatani, T., Urai, T., Ohmura, S., et Tamai, T. (2001). A requirements description metamodel for use cases. In *Proceedings of the 8th Asia-Pacific on Software Engineering Conference (APSEC'01)*, page 251. IEEE Computer Society.

[Narasimhan et Hendradjaya, 2004] Narasimhan, V. L. et Hendradjaya, B. (2004). Component integration metrics. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP'04)*.

[NASA Software Program, 1995] NASA Software Program (1995). Software measurement guidebook. Technical report, National Aeronautics and Space Administration.

[Nebut et al., 2003] Nebut, C., Fleurey, F., Traon, Y. L., et Jézéquel, J.-M. (2003). Requirements by contracts allow automated system testing. In *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE'03)*, page 85.

[Ogren, 2000] Ogren, I. (2000). On principles for model-based systems engineering. *Systems Engineering*, 3:38–49.

[OMG, 2004] OMG (2004). MOF 2.0 specification. Technical report, Object Management Group.

[OMG, 2005] OMG (2005). System modeling language (SysML) specification. Technical report, Object Management Group.

[OMG, 2008] OMG (2008). Software Process Engineering Metamodel (SPEM). Technical report, Object Management Group.

[Park et Kim, 1993] Park, S. J. et Kim, H. D. (1993). Constraint-based meta-view approach for modeling environmentinitl generation. *Decision Support System*, 9:325–348.

[Paulk et al., 1993] Paulk, M. C., Weber, C. V., Garcia, S. M., Chrissis, M. B., et Bush, M. (1993). Key practices of the capability maturity model. Technical report, Software Engineering Institute.

[Pawlak et al., 2006] Pawlak, R., Noguera, C., et Petitprez, N. (2006). Spoon: Program analysis and transformation in java. Technical Report 5901, INRIA.

[Poels et Dedene, 2000] Poels, G. et Dedene, G. (2000). Distance-based software measurement: necessary and sufficient properties for software measures. *Information and Software Technology*, 42(1):35–46.

[Reissing, 2001] Reissing, R. (2001). Towards a model for object-oriented design measurement. In *ECOOP'01 Workshop QAOOSE*.

[Revault, 1996] Revault, N. (1996). *Principes de méta-modélisation pour l'utilisation de canevas d'applications à objets (MétaGen et les frameworks)*. PhD thesis, Université P. et M. Curie (Paris 6), Paris, France.

[Reynoso et al., 2003] Reynoso, L., Genero, M., et Piattini, M. (2003). Measuring ocl expressions: a "tracing"-based approach. In *Proceedings of QAOOSE'2003*.

[Rothenberg, 1989] Rothenberg, J. (1989). *The nature of modeling*. John Wiley and Sons, Inc.

[SAE, 2006] SAE (2006). AADL Standard. Technical report, Society of Automotive Engineers.

[Saeki, 2003] Saeki, M. (2003). Embedding metrics into information systems development methods: An application of method engineering technique. In *CAiSE*, pages 374–389.

[Saeki et Kaiya, 2006] Saeki, M. et Kaiya, H. (2006). Model metrics and metrics of model transformation. In *Proc. of 1st Workshop on Quality in Modeling*, pages 31–45.

[Schmidt, 2006] Schmidt, D. C. (2006). Model-driven engineering. *IEEE Computer*, 39(2):25–31.

[Schmietendorf et al., 2000] Schmietendorf, A., Dumke, R., et Foltin, E. (2000). Metrics based asset assessment. *SIGSOFT Softw. Eng. Notes*, 25(4):51–61.

[Schneidewind, 1992] Schneidewind, N. F. (1992). Methodology for validating software metrics. *IEEE Trans. Software Eng.*, 18(5):410–422.

[SDMetrics, 2006] SDMetrics (2006). The software design metrics tool for the UML (http://www.sdmetrics.com/).

[Seidewitz, 2003] Seidewitz, E. (2003). What models mean. *IEEE Softw.*, 20(5):26–32.

[Selic, 2003] Selic, B. (2003). The pragmatics of model-driven development. *IEEE Softw.*, 20(5):19–25.

[Singh et al., 2004] Singh, Y., Sabharwal, S., et Sood, M. (2004). A systematic approach to measure the problem complexity of software requirement specifications of an information system. *Information and Management Sciences*, 15:69–90.

[Smacchia S.A., 2006] Smacchia S.A. (2006). Code query language 1.5 specification. Technical report, Smacchia S.A.

[Soley, 2000] Soley, R. (2000). Model driven architecture. Technical report, Object Management Group.

[Solheim et al., 2005] Solheim, H., Lillehagen, F., Petersen, S. A., Jorgensen, H., et Anastasiou, M. (2005). Model-driven visual requirements engineering. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE'05)*, pages 421–428.

[Spector et Gifford, 1986] Spector, A. et Gifford, D. (1986). A computer science perspective of bridge design. *Commun. ACM*, 29(4):267–283.

[Sztipanovits et Karsai, 1997] Sztipanovits, J. et Karsai, G. (1997). Model-integrated computing. *Computer*, 30(4):110–111.

[Szyperski, 1998] Szyperski, C. (1998). *Component Software*. Addison-Wesley.

[Tang et Chen, 2002] Tang, M.-H. et Chen, M.-H. (2002). Measuring OO design metrics from UML. In *Proceedings of MODELS/UML'2002*. UML 2002.

[The Standish Group, 1995] The Standish Group (1995). Chaos. Technical report, The Standish Group.

[Tichelaar et Demeyer, 1998] Tichelaar, S. et Demeyer, S. (1998). An exchange model for reengineering tools. In *Proceedings of the Workshop on Object-Oriented Technology at ECOOP'98*, pages 82–84.

[Tolvanen, 1998] Tolvanen, J.-P. (1998). *Incremental Method Engineering with Modeling Tools: Theoretical Principles and Empirical Evidence*. PhD thesis, University of Jyväskylä.

[Tsalidis et al., 1992] Tsalidis, C., Christodoulakis, D., et Maritsas, D. (1992). Athena: a software measurement and metrics environment. *Journal of Software Maintenance*, 4(2):61–81.

[van Deursen et al., 2000] van Deursen, A., Klint, P., et Visser, J. (2000). Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36.

[Vépa et al., 2006] Vépa, E., Bézivin, J., Brunelière, H., et Jouault, F. (2006). Measuring model repositories. In *Proceedings of the 1st Workshop on Model Size Metrics (MSM'06) co-located with MoDELS'2006*.

[Waismann, 1983] Waismann, F. (1983). *Ludwig Wittgenstein and the Vienna Circle*. Blackwell Publishing.

[Warmer et Kleppe, 2003] Warmer, J. et Kleppe, A. (2003). *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley.

[Washizaki et al., 2003] Washizaki, H., Yamamoto, H., et Fukazawa, Y. (2003). A metrics suite for measuring reusability of software components. In *METRICS '03: Proceedings of the 9th International Symposium on Software Metrics*, page 211. IEEE Computer Society.

[Weil et al., 2006] Weil, F., Neczwid, A., et Farbelow, K. (2006). Model size metrics research in motorola. In *Proceedings of the Model Size Metrics workshop at Models'2006*.

[White, 2000] White, K. P. (2000). Model theory. In *McGraw-Hill Encyclopedia of Science and Technology*. The McGraw-Hill Companies.

[Withrow, 1990] Withrow, C. (1990). Error density and size in ada software. *IEEE Softw.*, 7(1):26–30.

[Wittgenstein, 1975] Wittgenstein, L. (1975). *Philosophical Remarks*. Blackwell Publishing Ltd.

[Wohlin, 2007] Wohlin, C. (2007). An analysis of the most cited articles in software engineering journals - 2000. *Inf. Softw. Technol.*, 49(1):2–11.

[Yu-ying et al., 2007] Yu-ying, W., Qing-shan, L., Ping, C., et Chun-de, R. (2007). Dynamic fan-in and fan-out metrics for program comprehension. *Journal of Shanghai University*, 11(5):474–479.

[Zuse, 1991] Zuse, H. (1991). *Software Complexity*. Walter de Gruyter, Berlin.

A Glossaire

AADL Architecture Analysis & Design Language
API Application Programming Interface
BNF Backus-Naur Form
CASE Computer aided software engineering
CMM Capability Maturity Model
DSL Domain specific language
DSML Domain specific modeling language
DSVL Domain specific visual language
DTD Document Type Definition
EJB Enterprise Java Beans
EMF Eclipse Modeling Framework
EMOF Essential Meta Object Facility
IDE Integrated Development Environment
IDM Ingénierie dirigée par les modèles
ISO International Standardization Organization
KLOC Kilo LOC
LOC Lines of Code
MDA Model-driven Architecture
MD{S}D Model-driven {Software} Development
MDE Model-driven Engineering
MDM Model-driven Measurement (l'objet de cette thèse)
MOF Meta Object Facility
OCL Object Constraint Language
OO orienté objet / object-oriented
PM Platform model
PIM Platform independent model
PSM Platform specific model
OMG Object Management Group
UML Unified Modeling Language
XMI XML Metadata Interchange
XML eXtended Markup Language

B Annexe

La page http://www.monperrus.net/martin/phd-thesis/ regroupe toutes les ressources informatiques liées à cette thèse.

- Métamodèle de spécification de métriques;
- Code source du générateur de composants de mesure;
- Métamodèles de domaine utilisés;
- Spécifications de métriques (modèles XMI);
- Modèles mesurés.