SOMMAIRE

TABLE DES FIGURES

LISTE DES TABLEAUX

Première partie

Introduction

Chapitre 1

Introduction

1.1 Introduction

Le logiciel est devenu un élément, non seulement indispensable, mais aussi omniprésent dans notre société [1]. Par ailleurs, la taille et la complexité de celui-ci n'ont cessé de croître. Ainsi, le développement du logiciel est devenu une tâche tellement complexe qu'il devient rare d'avoir un logiciel sans faille.

Le génie logiciel a été introduit pour résoudre les problèmes des projets de développement du logiciel qui mènent à des résultats de faible qualité. Les problèmes surviennent lorsque le développement dépasse les délais et/ou les budgets ou que la qualité n'est pas au niveau des exigences [2]. Ainsi, le génie logiciel a pour objectif la construction de logiciels de manière cohérente, correctement, dans les délais, sans dépassement du budget et répondant aux exigences. Cependant, face à la croissance continue du besoin logiciel, ainsi que la complexité des logiciels qui ne cesse d'augmenter [3, 4], une approche a vu le jours qu'est l'Ingénierie Dirigée par les Modèles (IDM). Apparue dans les années 1992, le message central de l'IDM est de déplacer le cœur central du développement logiciel du programme (code) aux modèles, jusqu'à même arriver à concevoir des modèles qui peuvent être directement compilés et exécutés [5, 6]. Étant donné que les modèles sont plus proches de la compréhension humaine que le code, manipuler les modèles serait moins sujet à des erreurs que le code source [7]. Ainsi, l'ingénierie dirigée par des modèles est une approche de développement logiciel qui propose d'élever le niveau d'abstraction des langages afin de déplacer l'effort de conception et de compréhension du point de vue du programmeur vers celui du concepteur. Pour mettre cette vision en pratique, des Langages de Modélisation Spécifiques aux Domaines (DSML) ont émergé en industrie [8]. L'utilisation de ces derniers dans différents cas d'études industriels a augmenté la productivité de 500% à 1000% selon [9].

Pour concevoir des DSML de qualité, la construction de métamodèles précis demeure une tâche primordiale. Un métamodèle représente la syntaxe abstraite d'un langage de modélisation spécifique au domaine [10]. Étant l'artefact principal autour duquel est construit le DSML, le métamodèle est composé d'une partie structurelle, qui capture tous les concepts du domaine, ainsi que les relations entre les différents concepts [9, 11]. Les métamodèles sont souvent définis avec des langages, comme MOF (*Meta-Object Facility*) [12], un standard du consortium OMG (*Object Management Group*). Considérant la difficulté ou l'impossibilité d'exprimer certaines informations par le biais de diagrammes, des contraintes textuelles, autrement appelées des règles de bonne formation (ou Well-Formedness Rules (WFR)) et qui sont souvent spécifiées avec le langage OCL (*Object Constraint Language*) [13], sont souvent ajoutées afin de préciser certains concepts. Ceci assure que la sémantique structurelle des modèles générés à partir du métamodèle soit en adéquation avec le domaine visée. Par conséquent, on ne peut bénéficier de toute la puissance de l'ingénierie dirigée par les modèles que si le métamodèle est suffisamment précis pour décrire adéquatement les parties syntaxiques et sémantiques du domaine visé.

1.2 Motivation

Actuellement, la plupart des métamodèles présents dans les référentiels, comme celui de l'OMG [14], ne comprennent qu'une description de la partie structurelle [15]. Les règles de bonne formation sont rarement incluses, et parfois l'ensemble des règles n'est pas complet et n'empêche donc pas toutes les imprécisions sémantiques. Cela est dû principalement au fait que l'élicitation de ces règles est effectuée manuellement. Cela prend beaucoup de temps et est sujet à des erreurs. Simplifier cette tâche par des procédures était l'objectif de nombreux travaux. Plusieurs approches ont été explorées, notamment l'utilisation d'algorithmes génétiques pour générer des contraintes OCL à partir d'un ensemble de modèles corrects et incorrects [15, 16]. Bien que ces approches soient très prometteuses, elles restent difficiles à utiliser car il n'est pas aisé de disposer d'exemples de modèles valides et de modèles invalides, surtout pour des utilisateurs débutants. D'autres approches ont directement ciblé le langage OCL pour identifier des patrons de contraintes permettant de spécifier la plupart des contraintes OCL [17-22]. Bien que ces approches soient d'une grande utilité pour un concepteur ayant connaissance de ce qu'il doit décrire. Cependant, sans grande maîtrise du langage OCL, ils deviennent difficilement applicables. En effet,

ces approches fournissent rarement une assistance pour montrer où et comment utiliser ces patrons de contraintes OCL.

Dans la littérature, deux types de contraintes OCL peuvent être distingués : le premier concerne les contraintes qui sont relatives à des structures de métamodèle, car elles ne sont pas toujours suffisamment précises pour décrire avec exactitude le domaine applicatif [17]. On retrouve fréquemment ces contraintes dans les métamodèles, quel que soit le domaine applicatif auquel ils appartiennent, et qui sont souvent signe de la faiblesse du langage de modélisation MOF. Le deuxième type de contraintes concerne celles qui sont très spécifiques au domaine applicatif. Leur but est de préciser la sémantique du domaine applicatif visé.

Dans cette thèse, nous nous intéressons particulièrement au premier type de contraintes OCL. L'objectif étant de proposer une assistance à l'identification des parties d'un métamodèle nécessitant une précision, à l'aide de contraintes OCL, ainsi qu'à l'écriture de celle-ci. Pour la deuxième catégorie de contraintes OCL, il sera très difficile de proposer une approche générique vu que celles-ci sont directement liées au domaine applicatif visé. Il est important de souligner que selon le domaine visé, la présence de ces structures n'implique pas automatiquement des imprécisions. Cependant, elle indique un problème potentiel qui devrait être inspecté par le concepteur du métamodèle pour vérifier s'il s'agit d'un problème réel. D'où le nom suggéré de "Structure imprécises dans les métamodèles" (MIS).

1.3 Contributions de la thèse

Pour répondre à la problématique de l'absence de contraintes OCL dans les métamodèles, nous avons proposé les deux contributions décrites ci-dessous.

1.3.1 Identification et catégorisation des structures imprécises dans les métamodèles

Partant de métamodèles bien définis, comme le métamodèle d'UML, nous avons identifié les raisons qui ont menées leurs concepteurs d'y adjoindre des contraintes OCL. Les seules raisons que nous avons retenues sont celles qui résultent de la faiblesse de l'expres-

sivité du langage utilisé, ici MOF. Ainsi, nous avons proposé :

1. Le concept de MIS : qui correspondent à des catégories de structures d'un métamodèle qui, sans ajout de contraintes OCL, pourraient être imprécises.
2. des patrons de contraintes OCL correcteur de MIS : à chaque MIS nous avons proposé un ou plusieurs patrons de contraintes OCL qui aident, le concepteur du métamodèle, à préciser la sémantique visée par le MIS.
3. Enfin, un outil pour identifier, automatiquement, la présence d'un MIS dans un métamodèle. Cet outil est destiné à être intégré à un outil de (méta)modélisation. Ces apports ont été validé autant en qualitatif et en plus en quantitatif en ce qui concerne les MIS.

1.3.2 Identification de l'apparition de MIS lors de la co-évolution métamodèle/contrainte

La deuxième contribution porte sur une approche d'identification des occurrences de MIS qui peuvent apparaître suite à l'évolution d'un métamodèle. L'approche vise principalement à compléter les approches existantes, dans la littérature, pour la co-évolution métamodèle/contrainte, en proposant de nouvelles contraintes suite à l'apparition de MIS.

1.4 Structure du rapport

Cette thèse est composée de 4 parties majeures :

— **Partie 1 : Introduction**
Cette partie comprend le chapitre présent qui introduit le contexte de la thèse ainsi que la problématique de recherche.

— **Partie 2 : Contexte de travail et état de l'art**
Cette partie comprend 2 chapitres qui posent les bases nécessaires à la compréhension de cette thèse et passent en revue les travaux existants relatifs à la métamodélisation :

— Le chapitre 2 définit les différents concepts utilisés dans ce rapport de thèse, à savoir l'ingénierie dirigée par les modèles et l'assistance à la métamodélisation

précise.

— Le chapitre 3 présente l'état de l'art relatif aux travaux portant sur l'assistance à la métamodélisation précise. Notre attention sera particulièrement portée sur l'assistance à l'identification de structures imprécises dans les métamodèles. La fin de ce chapitre est consacrée au positionnement de nos contributions par rapport aux travaux existants dans l'état de l'art.

— **Partie 3 : Contributions**
La partie "Contributions" contient trois chapitres :

— Chapitre 4 (*Étude d'investigation sur les structures imprécises dans les métamodèles*) : ce chapitre détaille notre première contribution, qui consiste à étudier empiriquement des métamodèles avec leur contraintes OCL pour identifier les MIS. Ce chapitre détaille chacune de ces structures, ainsi que les différentes méthodes qui permettent de les rendre plus précises. La fin de ce chapitre est consacrée à l'interprétation des différents résultats obtenus dans l'étude empirique.

— Chapitre 5 (*Approche d'identification de MIS lors de la coévolution métamodèle/contrainte*) : ce chapitre détaille notre approche d'identification de MIS pendant la phase d'évolution des métamodèles. Nous commençons par un exemple qui illustre l'impact de l'évolution d'un métamodèle sur ses contraintes OCL, consistant, principalement, à l'introduction de MIS dans les métamodèles. Ensuite, notre approche complétant les approches de coévolution existantes est détaillée. Dans ce même chapitre, nous présentons notre étude sur les liens entre les différents opérateurs d'évolution et les MIS.

— Chapitre 6 (*Évaluation*) : L'ensemble des évaluations, des apports de la présente thèse, sont regroupées dans ce chapitre. Nous commençons par une évaluation quantitative puis qualitative des MIS. Ensuite, une évaluation de l'utilité d'un outil d'assistance à la recherche automatique des occurrences de MIS. Enfin, nous terminons par l'évaluation de notre approche d'identification de MIS lors de la coévolution métamodèle/contraintes.

— **Partie 4 : Conclusion**
Enfin, le dernier chapitre 7 conclut cette thèse en fournissant un résumé des contributions ainsi que des perspectives sur les travaux futurs.

Deuxième partie

Contexte de travail et état de l'art

Chapitre 2

Contexte de travail

Dans ce chapitre, nous commençons par présenter une vue d'ensemble du large domaine de l'ingénierie dirigée par les modèles et l'assistance à la métamodélisation précise. Plus précisément, nous introduisons les concepts relatifs à la métamodélisation, le langage de métamodélisation MOF et le langage de contraintes d'objets OCL. Nous concluons ce chapitre en présentant les méthodes utilisées dans le contexte de métamodélisation précise.

Sommaire

2.1 Ingénierie dirigée par les modèles (IDM)

Souvent, les développeurs font une démarcation entre la modélisation et le codage. Les modèles sont utilisés pour concevoir des systèmes, mieux les comprendre, spécifier des fonctionnalités requises, et créer de la documentation. Par conséquent, le code est écrit pour implémenter les modèles, ce qui fait que le débogage, les tests, ainsi que la maintenance sont faits au niveau du code. Cette séparation entre le code et le modèle n'est pas nécessaire car il existe plusieurs façons d'aligner le code et les modèles. La figure 2.1 proposée d'abord par [23] puis complétée par [9] décrit les différents cas possibles :

— D'abord, dans le cas le plus extrême, aucun modèle n'est créé. La spécification des fonctionnalités se fait directement sur le code source. Cette approche fonctionne bien dans le cas où le projet est de petite taille.

— Ensuite, certains développeurs créent des modèles, mais qui servent juste à abstraire les détails d'implémentation pour se focaliser sur le domaine. Cependant, ces derniers sont indépendants du code. Cette méthode n'est pas rentable puisque le coût nécessaire pour garder à jour à la fois le code source, mais aussi le modèle, est supérieur au bénéfice qu'apporte l'utilisation des modèles.

— Le troisième cas est celui de la visualisation de code. Cette méthode est surtout utilisée en rétro-ingénierie pour permettre de visualiser le fonctionnement d'un code déjà fonctionnel.

— Le quatrième cas est celui du modèle "Aller-retour" ou "Round-Trip" en anglais. Ce dernier automatise le processus de mise à jour entre le code source et le modèle. Ce modèle fonctionne seulement quand les formats du code et du modèle sont similaires, et quand il n'y a pas de perte d'information entre les différentes transformations. Aussi, ce mode de fonctionnement est contraire au concept d'utiliser les modèles pour abstraire des détails d'implémentation et donc de se focaliser sur le domaine. Si le modèle inclut tous les détails d'implémentation, ce dernier ne sera pas d'une grande utilité.

— Enfin, dans la dernière approche, le modèle est d'abord créé. Ce dernier étant

abstrait, il représente les différents concepts du domaine et les relations entre les concepts, sans aucun détail d'implémentation, ce qui le rend beaucoup moins complexe à manipuler en comparaison au code. Ainsi, le modèle devient l'artefact principal dans ce processus de développement. À partir du modèle, un langage d'implémentation est ciblé puis du code source est généré pour ensuite être compilé ou interprété pour l'exécution. Cette approche est le fondement de l'ingénierie dirigée par des modèles.

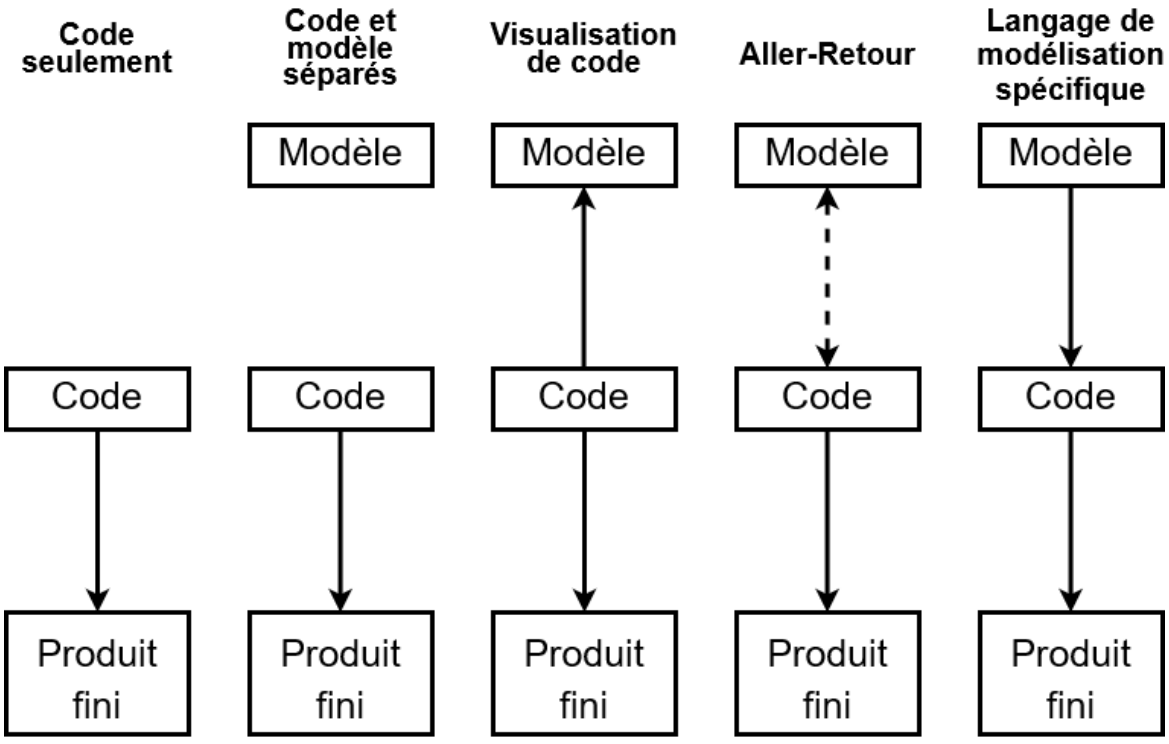

FIGURE 2.1 – Aligner le code et les modèles [9]

Compte-tenu de la croissance continue de la complexité logicielle [4, 24, 25], l'ingénierie dirigée par les modèles permet donc de élever le niveau d'abstraction, du code source vers les modèles [26]. De ce fait, les modèles sont placés comme artefacts centraux dans le processus de développement. Ainsi, les développeurs se concentrent sur le domaine du problème, plutôt que sur les technologies sous-jacentes pour créer, maintenir, tester et manipuler des modèles.

2.2 Modèle

Un modèle est une abstraction d'un système souvent utilisé pour étudier un système [27-29]. La pratique de modélisation est adoptée par les domaines de l'ingénierie ainsi que d'autres domaines tels que la physique, les mathématiques, la biologie, l'économie, la politique et la philosophie [30]. Dans la littérature, il existe une multitude de définitions de ce qu'est un *modèle*. Nous en citerons les suivantes :

1. *"le modèle est un ensemble de déclarations concernant le système étudié"* [31] ;
2. *"Le modèle est une abstraction d'un système (réel ou basé sur le langage) permettant de faire des prédictions ou des déductions"* [29] ;
3. *"Le modèle est une représentation réduite d'un système qui met en évidence les propriétés intéressantes d'un point de vue donné"* [32] ;
4. *"un modèle est une simplification d'un système construit dans un but précis ; un modèle doit donc être capable de répondre à des questions à la place du système original"* [33].

Pour soutenir l'utilisation de modèles comme artefacts principaux, l'OMG a défini quatre niveaux d'abstraction comme illustré dans la figure 2.2, à savoir les systèmes modélisés, les modèles, les métamodèles, et les méta-métamodèles. Il existe trois relations en IDM : la conformité (x), la représentation (μ) et l'appartenance ($\in$).

Par conséquent, l'OMG définit 4 niveaux de modélisation, chaque niveau (sauf le niveau M3) étant constitué d'éléments qui sont des instances d'éléments du niveau supérieur :

— M0 : système réel, système modélisé.
— M1 : modèle du système réel défini dans un certain langage. Ce dernier représente un système réel. Il est conforme à un métamodèle et appartient donc à un langage de modélisation.
— M2 : métamodèle définissant le langage. Il représente un langage de modélisation. Un métamodèle est conforme à un méta-métamodèle, et donc appartient à un langage de métamodélisation.
— M3 : méta-métamodèle définissant le métamodèle. Suivant le concept de méta-circularité, il est conforme à lui-même puisqu'il doit permettre d'être créé à partir de lui-même. Un méta-métamodèle appartient et représente un langage de méta-modélisation.

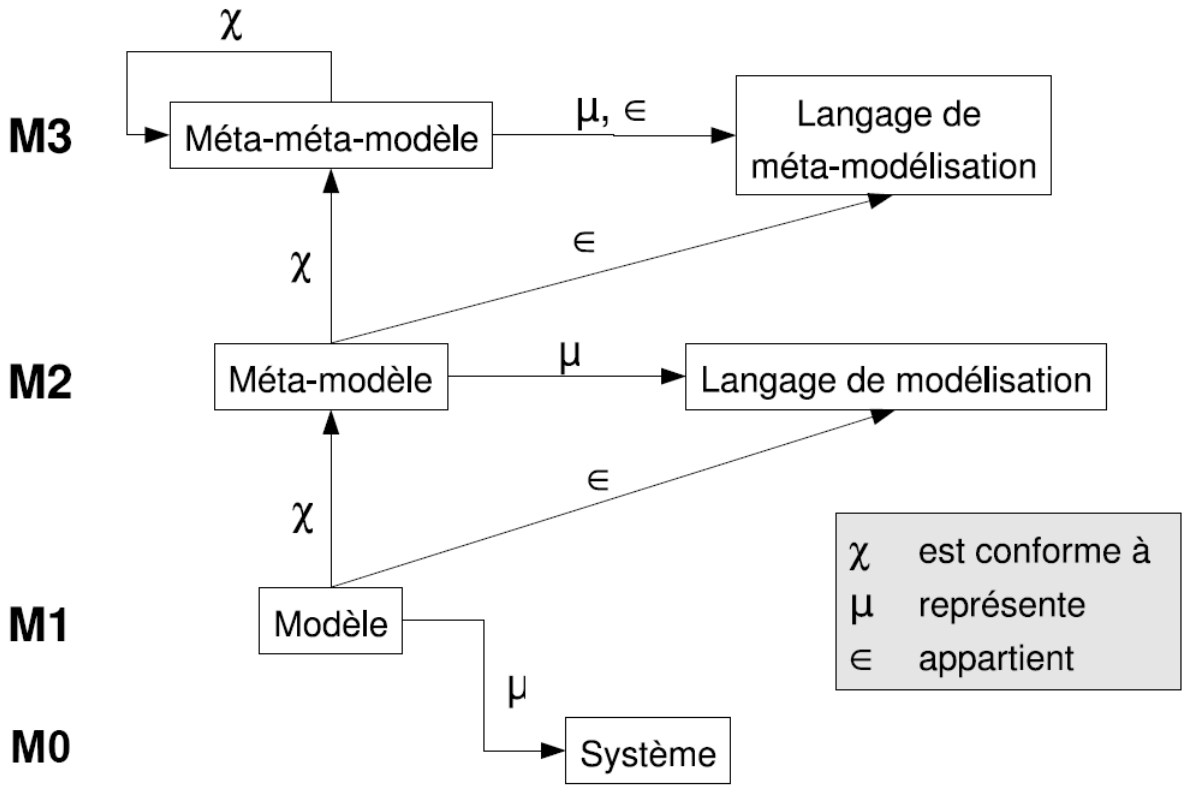

FIGURE 2.2 – L'infrastructure de modélisation de l'OMG, définie par Fleurey [34]

2.3 Métamodèles

Bien que les modèles non structurés qui servent comme représentation graphique soient utiles pour soutenir la communication entre différentes parties prenantes [35], les modèles ne sont plus considérés comme une documentation basique avec l'avancée de l'IDM. Ces derniers deviennent une force dans le processus de développement logiciel [36].

Afin d'apporter une valeur tangible aux processus automatiques impliquant des représentations abstraites, la traduction des modèles en code exécutable efficace doit être automatisée. En effet, lorsque les modèles sont bien définis (c'est-à-dire qu'une méta définition est fournie), les processus automatisés peuvent interpréter leur structure. En IDM, la conversion des représentations abstraites en implémentions concrètes se fait à l'aide d'une séquence de transformations automatisées qui génèrent, à partir d'un modèle de haut niveau, le code exécutable de bas niveau [37, 38]. Ce processus entraîne un gain de productivité considérable [39, 40]. Dans la pratique, l'utilisation de l'IDM accélère sans

doute les réponses aux changements d'exigences, facilite la communication avec les parties prenantes et augmente la portabilité, la maintenabilité et bien sûr la productivité [26, 41].

Pour soutenir l'automatisation proposée par l'IDM, les métamodèles jouent un rôle majeur. Les métamodèles sont des composants essentiels de l'écosystème de langage de modélisation [26]. Ils définissent les aspects structurels d'un domaine métier qui composeront les modèles, c'est-à-dire les principaux concepts, leurs propriétés, les relations entre eux, ainsi que leurs occurrences [42].

Parmi les différentes définitions de métamodèle de la littérature, Nous pouvons citer les suivantes :

1. *"Un métamodèle est un modèle qui définit le langage pour exprimer un modèle"* [14]
2. *"Un métamodèle est une spécification formelle d'une abstraction, généralement consensuelle et normative. A partir d'un système donné, nous pouvons extraire un modèle particulier à l'aide d'un métamodèle spécifique. Un métamodèle agit comme un filtre défini avec précision exprimé dans un formalisme donné"* [43].
3. *"Un métamodèle fait des déclarations sur ce qui peut être exprimé dans les modèles valides d'un certain langage de modélisation"* [31]
4. *"Un métamodèle est un modèle spécifiant un langage de modélisation"* [44]

Le métamodèle est au coeur de l'écosystème des DSL. Celui-ci constitue une base à laquelle les différents artefacts de l'IDM sont conformes. La figure 2.3 illustre les artefacts qui sont conformes au métamodèle. Il représente la syntaxe abstraite d'un DSL.

Dans cette thèse, nous utilisons la définition de métamodèles proposée par [45]. Cette définition stipule que le métamodèle comprend deux aspects : la structure du domaine, ainsi que les règles de bonne formation.

(1) Structure du domaine

Cette partie du métamodèle détaille tous les concepts utilisés à travers des métaclasses. Elle décrit aussi les différentes relations existantes entre ces concepts. Celle-ci est formalisée dans la présente thèse avec le langage Meta-Object Facility (MOF).

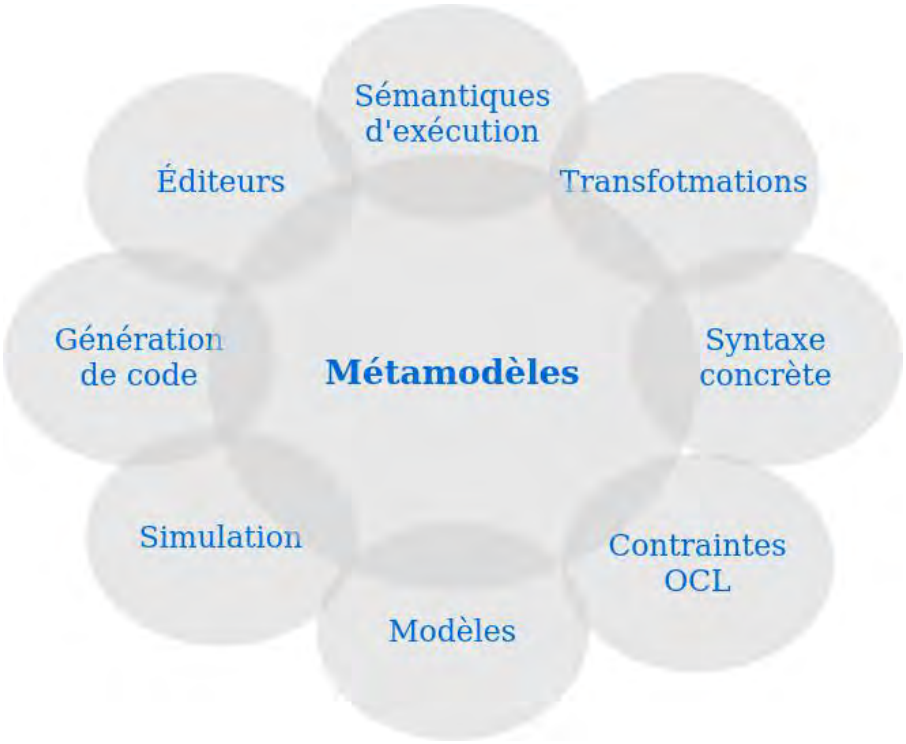

FIGURE 2.3 – Le métamodèle au coeur de l'écosystème de l'IDM

(2) Règles de bonne formation

Ces règles sont appliquées à la structure du domaine afin de préciser certaines informations qui sont difficiles, voire impossibles, à exprimer structurellement. Elles représentent la sémantique statique, autrement appelée sémantique structurelle du métamodèle.Dans la présente thèse, ces règles sont exprimées sous forme d'invariants avec le Langage de Contraintes d'Objects OCL.

Dans ce rapport nous utilisons les termes "structure du métamodèle" pour désigner la structure du domaine, et "contraintes OCL" pour désigner les règles de bonne formation. Nous introduisons dans ce qui suit les deux langages standards pour la métamodélisation selon l'OMG, à savoir le langage MOF et le langage OCL.

2.4 Meta-Object Facility (MOF)

Le Meta-Object Facility (MOF) est un langage de modélisation pour les métamodèles. En particulier, il est utilisé pour les métamodèles de la famille de langages maintenue par l'OMG, dont le langage UML. MOF fait également partie de UML, car il réutilise des parties de la syntaxe et de la sémantique de modélisation de "classes" UML (avec un ensemble de contraintes OCL appliquées pour exclure des concepts que les concepteurs de l'OMG ne souhaitaient pas inclure dans MOF). Il contient deux langages, le premier étant un ensemble minimal d'éléments nommés EMOF. Le EMOF est conçu pour la simplicité et la réutilisation. le principal objectif du langage EMOF est de permettre à des métamodèles simples d'être définis à l'aide de concepts simples tout en prenant en charge les extensions [46]. Ensuite, le langage MOF Complet (CMOF), qui étend EMOF par des capacités de langage plus sophistiquées [46].

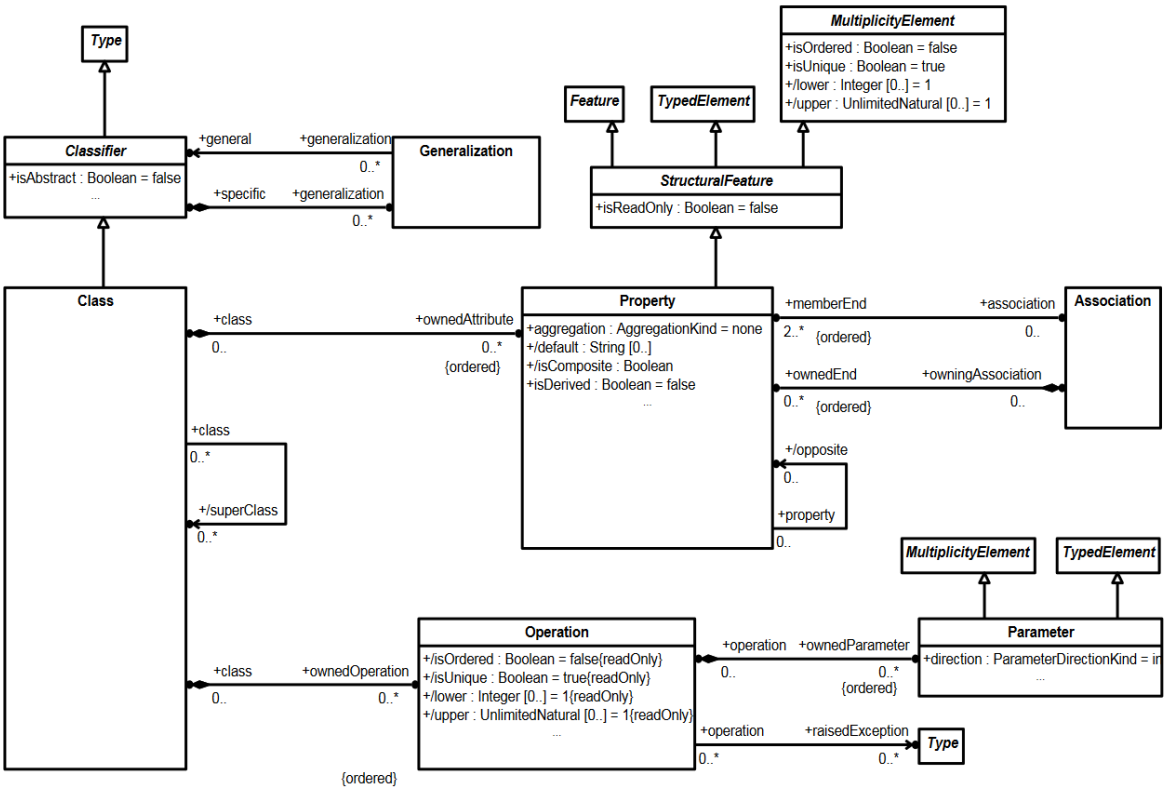

FIGURE 2.4 – Syntaxe abstraite du langage EMOF [46]

Le MOF a été conçu comme un cadre d'intégration global pour tous les méta-modèles de l'industrie du développement de logiciels [33].

EMOF et le CMOF correspondent à deux niveaux distincts de la spécification du MOF. L'EMOF fusionne le package Classes : :Kernel de la superstructure UML avec ses propres capacités linguistiques. Classes : :Kernel fusionne à son tour le package Core : :Basic de l'infrastructure UML [46]. Le CMOF fusionne l'EMOF et l'étend avec des fonctionnalités d'extension et de réflexion plus sophistiquées [46].

Les deux niveaux MOF sont conçus pour être autonomes, c'est-à-dire que le métamodèle EMOF est méta-circulaire (réflexif). Ainsi, le métamodèle de l'EMOF est le langage EMOF. Le CMOF sert en outre de métamodèle pour l'infrastructure UML, la superstructure et les niveaux de conformité, tels qu'implémentés par les modèles MOF XMI des "fichiers consommables de la machine normative", dans le cadre de la norme UML [47].

Dans le métamodèle EMOF, chaque (méta)classe désigne un concept du domaine. Les attributs et les opérations (pouvant avoir des paramètres) détaillent le concept décrit par la classe. Les associations (compositions incluses) ainsi que les liens d'héritage décrivent les différents liens existants entre les concepts. Étant donné que son infrastructure repose sur le métamodèle des classes UML, les types primitifs sont les mêmes. De plus, il est possible de définir de nouveaux types de données ainsi que des énumérations.

Dans le reste du présent rapport, nous utilisons la version EMOF pour présenter les différents métamodèles. Par exemple, la figure 2.5 illustre un extrait du métamodèle Activités d'UML créé avec EMOF. Cette structure représente les métaclasses qui correspondent aux Nœuds (nœud initial, final, et de jointure) et les connexions (flux d'objets et flux d'activité) dans ce métamodèle. Elle décrit aussi les relations existantes entre les nœuds et les connexions. Ces relations sont sous la forme d'associations (incoming et outgoings), chacune ayant une multiplicité pour désigner le nombre d'objets pouvant être liés via l'association. Cet exemple est repris dans les sections suivantes.

2.5 Le Langage de Contraintes d'Objets (OCL)

Le Langage de Contraintes d'Objets (OCL) est la norme OMG pour la spécification des contraintes d'un (méta)modèle. OCL est un langage formel pour exprimer des contraintes

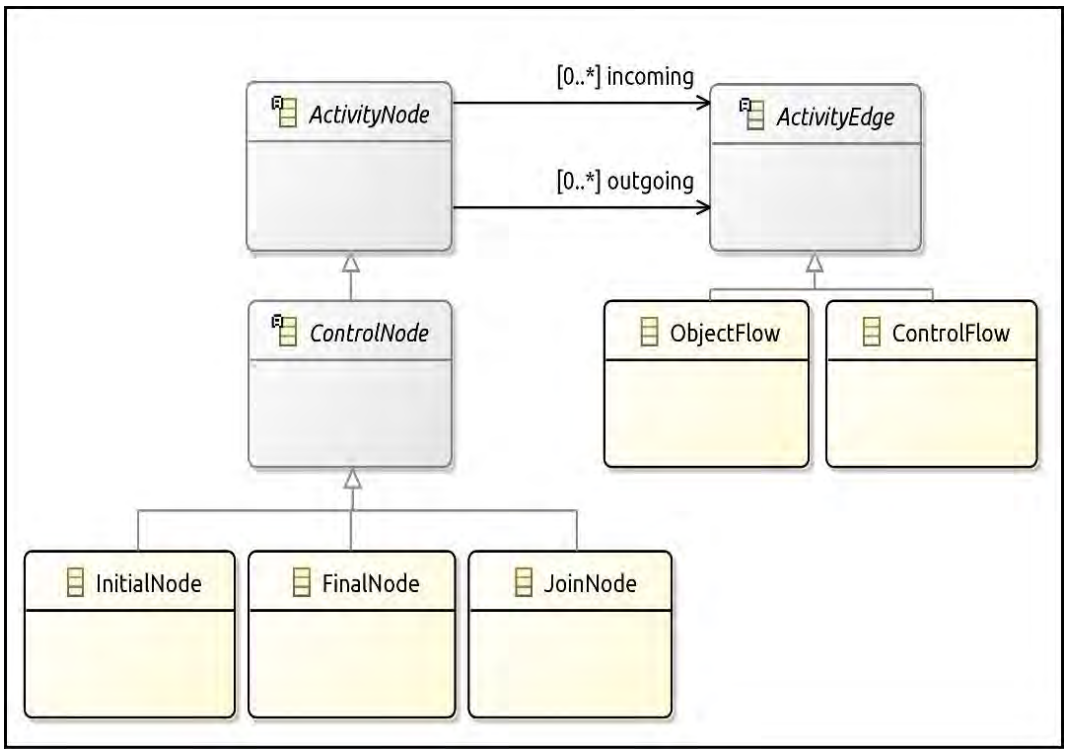

FIGURE 2.5 – Extrait de métamodèle "Activités" d'UML

sans effets secondaires, c'est-à-dire que leur évaluation sur les modèles ne peut modifier ni leur contenu ni leur état [13]. Cabot et al. [42] définissent OCL comme "*un langage de spécification typé, déclaratif et sans effets secondaires. Typé signifie que chaque expression OCL est évaluée dans un type (soit l'un des types OCL prédéfinis, soit un type du modèle dans lequel l'expression OCL est utilisée) et doit se conformer aux règles et opérations de ce type. L'absence d'effets secondaires implique que les expressions OCL peuvent interroger ou contraindre l'état du système mais pas le modifier. Déclaratif signifie qu'OCL n'inclut pas de constructions impératives comme les affectations. Et enfin, la spécification fait référence au fait que la définition du langage n'inclut aucun détail d'implémentation ni aucune directive d'implémentation*".

Comme déjà signalé précédemment, la partie structurelle du métamodèle ne permet pas d'exprimer toutes les déclarations d'une spécification approfondie [48]. Ainsi, la création de modèles précis, représentant le système réel, est très difficile et peut rendre l'utilisation de modèles encore plus difficile. Ceci devient encore plus contraignant dans le cas des technologies de génération automatique de code qui peut, ainsi, résulter à des

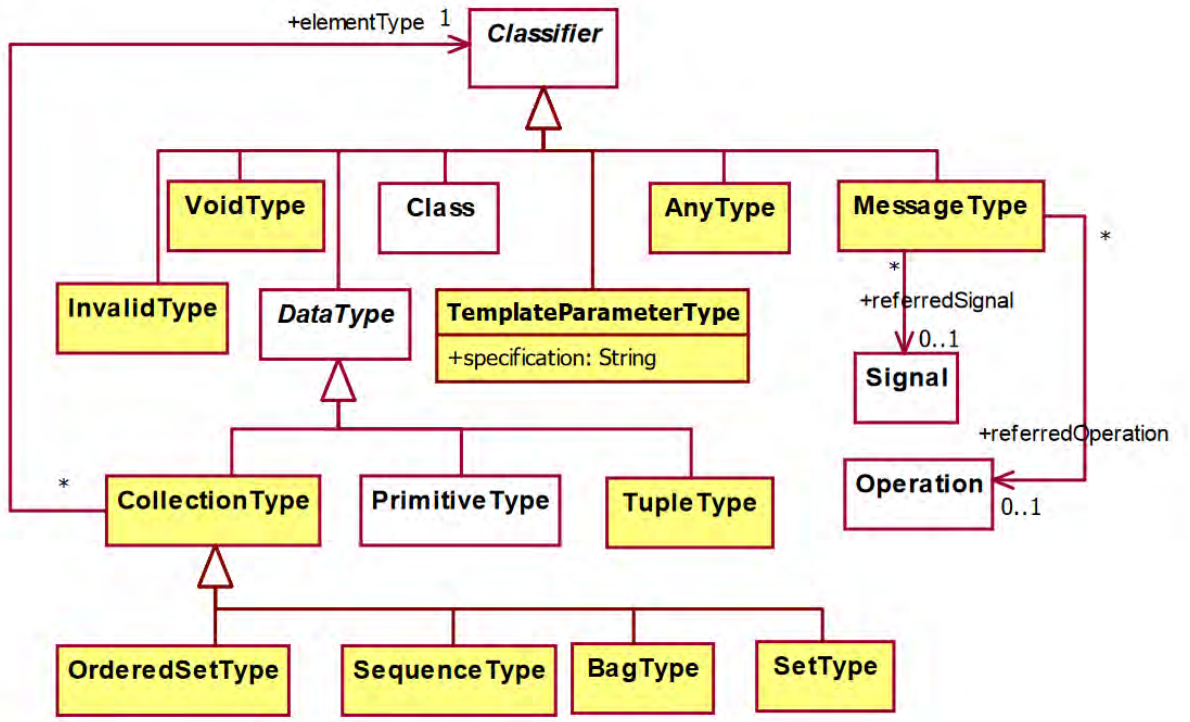

FIGURE 2.6 – La syntaxe abstraite du langage OCL [13]

incorrections [49]. OCL peut être utilisé pour définir différents types de contraintes sur le métamodèle pour préciser certaines spécificités d'un domaine métier. L'utilisation de contraintes OCL sur un métamodèle peut être considérée comme un moyen de restreindre davantage l'espace de couverture d'un métamodèle [48]. Cela signifie que parmi l'ensemble de tous les modèles possibles, pouvant être créés à partir du métamodèle, seuls les modèles qui vérifient les contraintes OCL sont considérés comme corrects ou valides. Ces derniers sont également appelés états de modèles valides dans [50-52]. La figure 2.6 représente la syntaxe abstraite du langage OCL.

Le langage OCL peut être utilisé dans plusieurs cadres, nous en citons quelques utilisations :

— utilisation comme langage de requêtage ;
— spécification d'invariants (contraintes) sur des classes et des types, qui doivent être respectées pendant toute la durée de vie des modèles. Un invariant peut être défini sur tous les éléments du métamodèle ;
— Des préconditions ou des postconditions sur les opérations ou les méthodes ;

— Des règles de dérivation et d'initialisation de propriétés.

Pour montrer l'utilité des contraintes OCL, nous utilisons l'extrait du métamodèle Activités d'UML présenté précédemment dans la figure 2.5. Bien que la structure de ce métamodèle regroupe tous les concepts nécessaires pour représenter des activités, l'absence de contraintes OCL dans certaines parties du métamodèle est problématique. En effet, la figure 2.7 illustre le modèle Activités pouvant être créé à partir du métamodèle d'Activité en l'absence de contraintes OCL. Bien que le langage MOF soit expressif pour décrire les différents Nœuds, Connexions (ActivityEdges) ainsi que les relations existantes entre les deux concepts (les associations incomings et outgoings), ce langage n'est pas suffisamment précis pour capturer tous les détails sémantiques. Ainsi, à partir de cette structure, les trois types de Nœuds (Initial, Final et Join) héritent les associations *incoming* et *outgoing* avec des multiplicités 0..*. Or, selon la spécification du métamodèle UML de l'OMG [53], nous avons les contraintes suivantes :

1. un noeud *Initial* ne doit pas avoir de connexions (*ActivityEdges*) en entrée ;
2. un noeud *Final* ne doit pas avoir de connexions en sortie ;
3. un noeud *Join*, qui permet de fusionner plusieurs connexions entrantes pour en créer une seule sortant, ne doit pas avoir plus d'une connexion sortante.

Pour exclure la possibilité de créer des modèles invalides qui ne représentent pas le domaine applicatif, les trois contraintes OCL suivantes sont spécifiées pour accompagner le métamodèle.

```
1 context InitialNode inv aucuneEntrée : self.incoming -> isEmpty()
```

```
1 context FinalNode inv aucuneSortie : self.outgoing -> isEmpty()
```

```
1 context JoinNode inv uneSeuleSortie : self.outgoing -> size() = 1
```

2.6 Métamodélisation précise

Pour montrer les différentes conditions nécessaires à la création d'un métamodèle précis, nous définissons d'abord l'espace de modélisation.

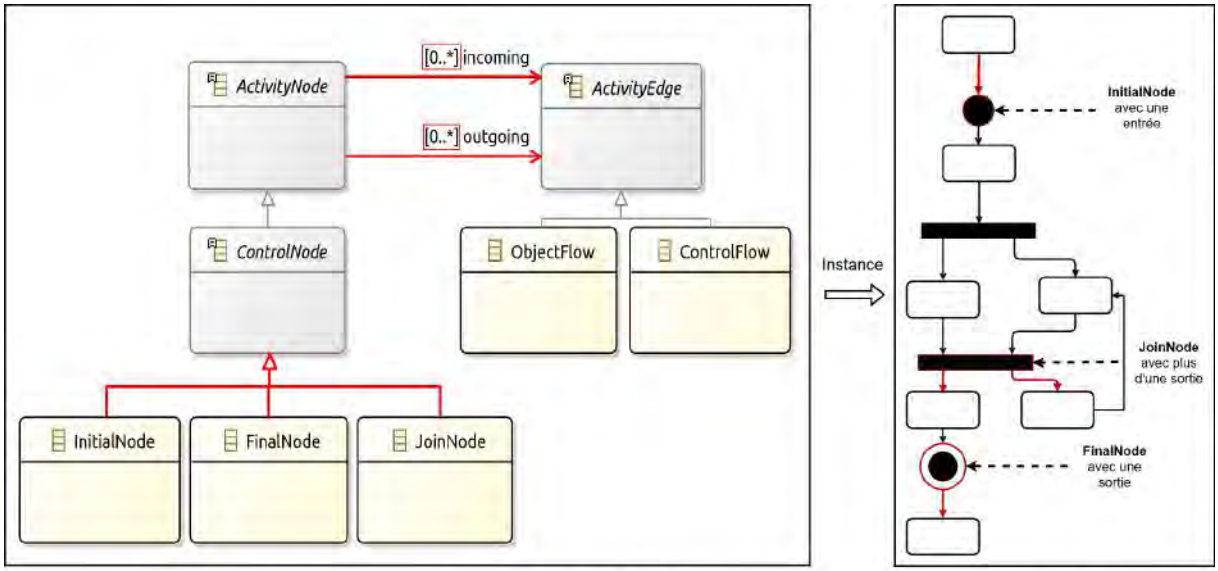

FIGURE 2.7 – Exemple de l'impact de l'absence de contraintes OCL sur le métamodèle Activités d'UML

Espace de modélisation

L'espace de modélisation (EM) désigne l'ensemble de tous les modèles qui peuvent être créés à partir d'un métamodèle. Par conséquent, cela correspond à tous les modèles appartenant au domaine de modélisation d'un métamodèle et qui lui sont conformes.

En règle générale, il existe deux espaces de modélisation :

— Espace de modélisation souhaité (théorique) qui doit inclure tous les modèles valides du domaine et exclure tous les autres modèles qui ne correspondent pas au domaine applicatif ;

— Espace de modélisation réel (en pratique) qui est influencé par les deux éléments constituant le métamodèle : i) la complétude de la structure du métamodèle ; ii) la complétude de l'ensemble des contraintes OCL.

L'objectif de tout concepteur est de créer un métamodèle pour lequel l'espace de modélisation réel soit égal à l'espace de modélisation souhaité. Cependant, le concepteur peut être confronté à diférentes situations. La figure 2.8 illustre ces différentes situations et que nous abordons ci-dessous :

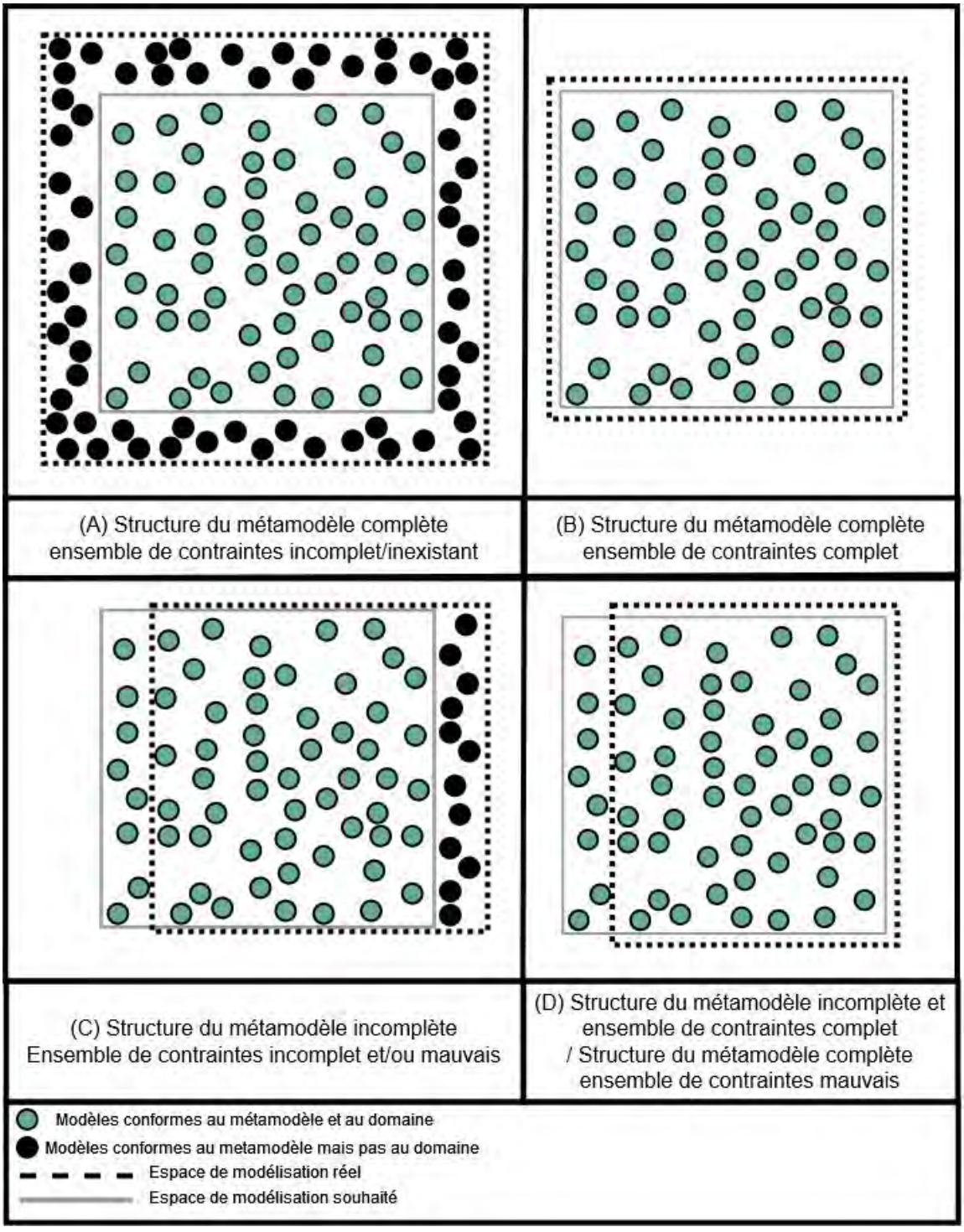

FIGURE 2.8 – Espace de modélisation

— La situation A survient lorsque la structure du métamodèle est complète, c.à.d. elle comporte tous les concepts nécessaires du domaine applicatif, mais que l'ensemble des contraintes OCL est inexistant ou n'est pas complet. Dans cette situation, l'espace de modélisation regroupe tous les modèles corrects qui représentent le domaine, mais aussi des modèles incorrects qui ne sont pas compatibles avec le domaine applicatif.

— La situation B survient lorsque le métamodèle comporte à la fois une structure complète, mais aussi un ensemble de contraintes OCL complet. Dans ce cas, l'espace de modélisation souhaité est égal à l'espace de modélisation réel.

— La situation C se manifeste quand l'espace de modélisation réel ne comprend qu'un sous-ensemble des modèles corrects et comprend un ensemble de modèles incorrects. Cette situation survient dans deux cas : i) lorsque la structure du métamodèle est incomplète et, donc, ne représente qu'un sous-ensemble des modèles corrects et que l'ensemble des contraintes OCL est correct mais incomplet, c.à.d ne comprend pas toutes les contraintes permettant d'exclure les modèles incorrects ; ii) lorsque la structure du métamodèle est complète, mais l'ensemble de contraintes OCL est mal défini, ce qui fait qu'il exclue des modèles qui sont corrects (qui représentent le domaine), et qu'il n'exclue pas tous les modèles incorrects (qui ne représentent pas le domaine).

— La situation D se manifeste quand l'espace de modélisation réel ne comprend qu'un sous-ensemble des modèles corrects, mais ne comprend pas de modèles incorrects. Cette situation survient dans deux cas : i) quand la structure du métamodèle est incomplète, et que l'ensemble des contraintes OCL est complet. Dans ce cas, des modèles qui représentent le domaine ne sont pas couvert par l'espace de modélisation réel ; ii) quand la structure du métamodèle est complète, mais que l'ensemble des contraintes OCL est trop sévère. Ce phénomène se nomme la surcontrainte [54]. Dans ces conditions, l'ensemble des contraintes OCL arrive à exclure tous les modèles qui ne représentent pas le domaine applicatif, mais exclue aussi un ensemble de modèles qui représentent le domaine.

Pour assister les concepteurs à spécifier des métamodèles de qualité, qui permettent de représenter tous, et seulement, les modèles qui représentent le domaine applicatif, une

multitude de travaux ont été proposés dans la littérature. Ces travaux seront détaillés dans le chapitre "Etat de l'art". Dans ce rapport, nous nous intéressons, particulièrement, à l'assistance pour l'identification de structures pouvant nécessiter des contraintes OCL.

2.6.1 Transformation de modèles

Une transformation de modèle est un ensemble de règles déclaratives représentant un mapping entre deux ensembles de modèles à exécuter par un moteur de transformation. Elle se compose d'un ensemble de règles qui décrivent comment des éléments de la langue cible peuvent être créés automatiquement à partir d'éléments de la langue source. Par exemple, il est concevable de supposer que les programmes soient des modèles, car ils représentent un système abstrait se conformant à leur langage de programmation et existent dans un but spécifique [43]. Ce type de transformation est appelé "modèle-vers-texte". En IDM, la transformation de modèles signifie plus une transformations de type "modèle-vers-modèle".

Les transformations de modèles peuvent être vues comme des modèles décrivant la fonction de transformation [29]. La transformation est conforme à son langage de transformation (métamodèle), elle représente la fonction de transformation entre un ou plusieurs ensembles de modèles et est créée à des fins d'exécution par un moteur de transformation.

La figure 2.9 illustre le pattern décrivant la transformation de modèles selon [55]. Il illustre les rôles que jouent les éléments d'une transformation de modèle-vers-modèle en termes de relations ReprésentationDe (μ), ÉlémentDe ($\in$) et ConformeÀ (x). La transformation elle-même est donnée comme une fonction de transformation, définie entre un langage de modélisation source et un langage de modélisation cible. Lorsqu'elle est exécutée pour une paire spécifique de modèles, l'instance de transformation est un élément de la fonction de transformation, car les modèles source et cible sont des éléments de l'ensemble de tous les modèles valides, c'est-à-dire les langages de modélisation source et cible. Les langages sont représentés par leurs métamodèles, auxquels se conforment tous les modèles valides. La fonction de transformation est représentée par un modèle de transformation (une description exécutable de la transformation) qui est un élément de toutes les transformations valides. Le langage de transformation est représenté par un métamodèle auquel se conforme le modèle de transformation. Lorsque les fonctions de transformation sont représentées par des modèles, ces modèles peuvent eux-mêmes servir d'entrée ou de sortie de transformations. De cette manière, les transformations peuvent

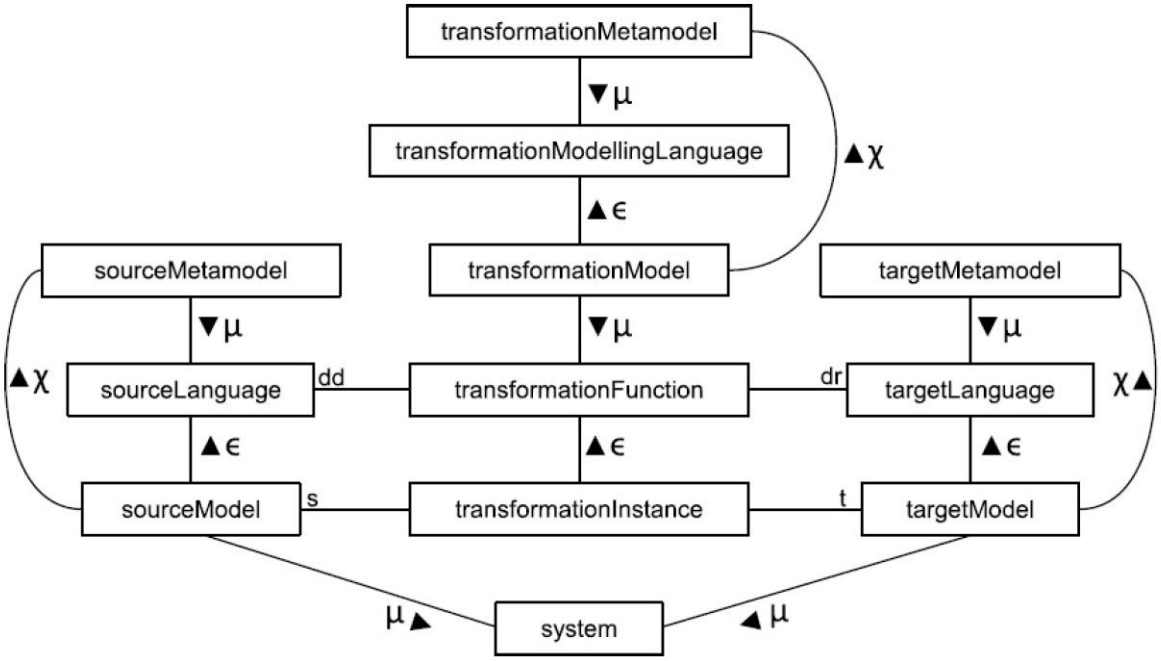

FIGURE 2.9 – Le schéma de transformation de Favre [55]

être gérées par les mécanismes et outils communs à l'IDM et être traitées d'une façon automatique [45, 56].

2.6.2 Les différents types de transformation

Dans la littérature, il existe quatre types de transformation : l'évolution, la maintenance, le refactoring, et la coévolution. Chaque type est appliqué dans un contexte particulier.

2.6.2.1 L'évolution

L'évolution en ingénierie logicielle permet d'adapter un logiciel pour y inclure de nouvelles exigences de l'utilisateur ou environnementales. L'évolution, ainsi, répond à la fois à l'apprentissage des développeurs et des utilisateurs, où des exigences plus précises sont basées sur l'expérience passée avec l'application [57]. L'évolution est parfois aussi appelée adaptation [58-60].

2.6.2.2 La maintenance

La maintenance est définie comme le processus de modification d'un produit logiciel dans le but de le corriger, améliorer les performances, ou pour adapter le produit à un environnement ayant changé [61]. La maintenance est souvent utilisée à tort comme équivalent au terme évolution [62, 63]. Cependant, la maintenance fait référence à une activité réalisée après la remise du produit au client, et qui suggère la réparation du produit, tandis que l'évolution comprend à la fois la maintenance, mais aussi la mise en place de nouveaux concepts qui évoluent par rapport aux précédents [64].

2.6.2.3 Le refactoring

Le refactoring est le terme orienté-objet qui désigne la restructuration, et qui a été introduit dans [65]. Le refactoring et la restructuration sont des modifications apportées au logiciel pour pour le rendre plus facile à comprendre et à modifier et/ou le rendre moins susceptible aux erreurs lorsque de futurs changements sont introduits [65, 66]. Dans les modèles orientés objets, le refactoring est une sorte de redistribution des classes, des variables et des méthodes, à travers la hiérarchie des classes, pour améliorer la qualité et faciliter les futures adaptations et extensions, mais sans introduire de nouveau comportement au niveau conceptuel [67, 68].

Contrairement à ce qui est communément admis, le refactoring ne fait pas exclusivement partie de la phase de maintenance. Il peut également être appliqué dans les phases de conception et de développement antérieures [65]. L'activité de refactoring peut être considérée comme faisant partie d'une évolution perfective.

2.6.2.4 La coévolution

La coévolution fait référence au processus d'adaptation et de correction d'un ensemble d'artefacts en réponse à l'évolution du métamodèle dont dépendent fortement ces artefacts. Par exemple, les instances de modèle, les scripts de transformation et les contraintes OCL, dépendent tous du métamodèle. Les modèles doivent être conformes au métamodèle, alors que les scripts de transformation et les contraintes OCL dépendent des éléments du métamodèle pour être spécifiés et utilisables.

De plus, l'activité de coévolution dénote l'idée de propager l'évolution du métamodèle dans ses artefacts afin qu'ils restent cohérents avec la nouvelle version du métamodèle.

Par conséquent, la coévolution peut être considérée comme une évolution adaptative en réponse à une autre évolution de haut niveau. La coévolution est aussi parfois appelée coadaptation [58] ou migration [69, 70].

2.7 En résumé

Dans ce chapitre, nous avons introduit les différents concepts relatifs à l'Ingénierie Dirigée par les Modèles. Nous avons d'abord commencé par expliquer comment l'IDM est apparue, pour ensuite, présenter les divers artefacts utilisés tels que les métamodèles, les modèles, ainsi que les contraintes OCL. Nous avons aussi présenté les langages proposés par l'OMG pour représenter ces différents artefacts, ainsi que l'importance de la métamodélisation précise. Dans le chapitre suivant, nous présentons les différents travaux, existant dans la littérature, qui proposent des approches d'assistance à la métamodélisation.

Chapitre 3

Etat de l'art

Dans ce chapitre, nous présentons les études qui portent sur l'ingénierie dirigée par les modèles en mettant l'accent sur les travaux portant sur l'assistance à la métamodèlisation précise. Nous commençons par présenter les travaux présentant l'assistance à la métamodélisation de qualité, que ce soit en évaluant la qualité des métamodèles et/ou leurs contraintes OCL, ou bien en assistant la spécification des contraintes OCL. Ensuite, nous présentons les travaux de maintenance et de coévolution des métamodèles.

Sommaire

3.1 Assistance à la métamodélisation de qualité

3.1.1 Évaluation de la qualité des métamodèles

Dans la littérature, on retrouve une multitude de travaux portant sur l'évaluation de la qualité des métamodèles. Nous présentons quelques travaux relatifs à l'évaluation des métamodèles, ainsi que d'autres qui proposent des métriques qualitatives pour les métamodèles.

Dans le contexte de la qualité des métamodèles, une étude empirique menée par [71] porte sur la perception de la qualité des métamodèles. Les résultats montrent que la majorité estiment que la qualité dépend de la complétude, correction, et de la modularité, tandis que d'autres critères de qualité comme la consistance n'ont pas été souvent validés par les participants à l'étude.

Afin d'évaluer les métamodèles utilisant les langages de métamodélisation MOF et OCL, Cadavid et al. [19] ont réalisé une analyse empirique sur des métamodèles avec des règles de bonne formation visant à comprendre la manière dont les concepteurs articulent les deux langages, et explorant les pratiques de métamodélisation au cours des dix dernières années. L'analyse des métamodèles a été réalisée en utilisant un ensemble de métriques écrites dans le langage OCL. Cette étude a été menée sur 33 métamodèles de différentes sources (universitaires, industriels et standards) avec leurs contraintes OCL respectives. L'étude a répondu à de nombreuses questions intéressantes en relation avec les langages OCL et MOF, en particulier la cohérence des contraintes OCL dans les métamodèles étudiés, ainsi que la couverture des contraintes OCL de la partie structurelle du métamodèle. En outre, les auteurs ont conclu l'étude par un ensemble de patterns OCL, présentés en détail dans [54]. Ces patrons ont été identifiés comme des contraintes fréquentes apparaissant de manière récurrente dans les métamodèles étudiés.

Le métamodèle UML est l'un des métamodèles les plus connus dans le domaine de l'ingénierie dirigée par les modèles. De par sa taille, le nombre de ses contraintes, et les évolutions qu'il a connues, UML a été étudié de nombreuses fois à des fins différentes. [72] a été l'un des premiers travaux à étudier le métamodèle UML. Ce travail consiste à exploiter un ensemble de métriques orientées objet (OO) inspirées de [73] et de [74] pour mesurer à la fois la stabilité et la qualité du métamodèle. L'étude a été réalisée sur cinq versions d'UML (de 1.1 à 2.0) en utilisant des métriques orientées objet adaptées. L'objectif principal de cette étude était l'évolution de la qualité (principalement la maintenabilité

et la réutilisabilité) de la structure du métamodèle UML.

Dans [75], une étude qualitative du métamodèle PCM [76] a été faite. Ce métamodèle décrit les architectures logicielles à base de composants avec une attention particulière aux propriétés de performance. Ce dernier est un vieux métamodèle qui a évolué dans le temps avec de nombreuses fonctionnalités. Les évolutions que le métamodèle a connues ont rendu le métamodèle plus susceptible de contenir des bad smells. Le travail se concentre à la fois sur les smells qui peuvent être détectés automatiquement, mais aussi sur les smells qui peuvent être détectés manuellement. Les auteurs ont identifié 10 types de bad smells, dont les effets ont été expliqués dans l'article. Dans l'ensemble, l'étude a montré que l'évolution du métamodèle a fait apparaître différentes bad smells qui ont un impact négatif sur la maintenabilité du métamodèle.

Compte tenu de la tâche fastidieuse que représente l'analyse manuelle des modèles et des artefacts dans l'ingénierie dirigée par les modèles, et afin de simplifier l'obtention de données pour les études empiriques, les auteurs ont proposé dans [77] EMMA (EMF Meta-Model Analysis), un outil qui effectue automatiquement l'extraction, l'analyse et la visualisation des données et des métriques sur les artefacts dans le contexte de l'ingénierie dirigée par les modèles. Les auteurs ont présenté divers exemples d'application du cadre sur de nombreuses études de cas industrielles. L'approche est très prometteuse et pourrait faciliter la création de jeux de données IDM et les études empiriques.

D'autres travaux ont proposé des métriques pour les métamodèles. D'abord, le travail présenté dans [78] propose des métriques qui peuvent être utilisées pour acquérir des mesures objectives, transparentes et reproductibles sur les métamodèles. L'objectif principal est de mieux comprendre les principales caractéristiques des métamodèles, comment ils sont couplés et comment ils évoluent. Une analyse de corrélation a été effectuée pour identifier les métriques les plus réticulées, qui ont, à leur tour, été calculées sur 450 métamodèles. Ensuite, Kudo et al. [79] proposent le framework MQuare, qui inclut une description détaillée de 19 exigences de qualité, ainsi que 23 mesures de qualité. Enfin, Lopez et al. [80] proposant le langage "mmSpecand" et son outil de support (metaBest), dédié à la spécification des propriétés souhaitées du métamodèle avec une approche pour évaluer la qualité du méta-modèle en utilisant des métriques.

3.1.2 Assistance à la production de contraintes OCL

Lorsqu'il s'agit de l'assistance à la production de contraintes OCL, quelques travaux ont été proposés dans la littérature. Le premier type de travaux concerne ceux dans lesquels des patterns OCL ont été proposées pour faciliter l'écriture des contraintes.

Les patterns de contraintes ont d'abord été introduits pour la programmation orientée objet dans [81], puis adaptés aux modèles conceptuels et aux métamodèles. Par exemple, les auteurs de [82, 83] ont révélé des types de contraintes pertinentes pour la conception de modèles conceptuels bien formés sous forme de taxonomie. Cela a été fait en analysant des méthodes de modélisation conceptuelle les plus importantes pour identifier les situations où ces contraintes devraient être utilisées. Ensuite, dans [84], ils ont défini un profil qui étend l'ensemble des contraintes prédéfinies UML avec certains types de contraintes qui sont très fréquemment utilisés dans les schémas conceptuels. De plus, les travaux de [18, 20] visent à adapter certains patterns de contraintes existants pour augmenter l'efficacité des processus de test et de débogage. De plus, la collection de modèles de contraintes publiés a été étendue dans [21]. De l'autre côté, suite à l'étude empirique de Cadavid et al. [19], un ensemble de patterns de contraintes OCL a été identifié dans [54]. Enfin, Wahler et al. [85] ont proposé un ensemble de patrons OCL pour simplifier l'écriture des contraintes OCL. Les auteurs ont également proposé un ensemble d'anti-patterns, qui sont des structures qui peuvent ne pas être assez précises pour capturer pleinement le domaine du métamodèle, ce qui peut conduire à un modèle immature. À cette fin, l'auteur a indiqué les structures dans lesquelles les patrons OCL pourraient être utilisés et a fourni un outil qui développe des spécifications de contraintes concises et cohérentes. L'outil vérifie également les chevauchements potentiels entre différentes contraintes. La principale lacune de cette approche est le manque de preuves empiriques. En effet, l'ensemble des anti-patterns n'a pas été validé par des études empiriques et est donc difficile à généraliser. Malgré la similitude entre les anti-patterns de Wahler et les MIS, nous pouvons noter quelques différences. Tout d'abord, nous nous sommes appuyés sur une approche empirique pour identifier les MIS. En revanche, Wahler et al. ont proposé les anti-patterns en se basant sur leurs connaissances en matière de modélisation. De plus, les MIS ont été identifiés sur des métamodèles basés sur le MOF, et non sur des diagrammes de classes UML, ce qui les rend propres aux métamodèles uniquement.

Le deuxième type de travaux concerne les approches ayant comme objectif de générer

d'une façon automatique ou semi-automatique des contraintes OCL. Tout d'abord, [15] a proposé une approche heuristique pour rechercher automatiquement les règles de conformité des métamodèles à partir de modèles valides et invalides. L'approche fait évoluer et entraîne une population de contraintes OCL jusqu'à ce qu'elles deviennent capables de distinguer les modèles valides et invalides. L'évolution se fait à l'aide d'opérateurs de croisement et de mutation. Contrairement à notre approche, l'approche génétique n'a pas besoin d'analyser les structures des méta-modèles pour proposer des contraintes OCL, mais se base sur les modèles valides et invalides pour récupérer l'information. Les résultats sont très prometteurs pour les métamodèles étudiés. De plus, cette approche a été améliorée dans [86] en injectant de la diversité sémantique sociale pour améliorer le processus de recherche automatique des contraintes OCL. Ce processus prend en considération non seulement l'aptitude d'un individu (un ensemble de contraintes OCL) à distinguer entre les modèles valides et invalides, mais aussi ce que l'individu ramène de plus à la population. Ainsi, l'approche devient multi objective. Les auteurs ont validé l'approche à travers trois métamodèles.

Dans le même contexte, Dang et al. [16, 87] proposent un framework pour inférer des règles métier écrites avec OCL à partir d'ensembles valides et invalides de scénarios utilisateurs. Leur travail consiste à transformer les patterns OCL en un Problème de Satisfaction de Contraintes (CSP), puis d'utiliser un moteur d'inférence prolog pour résoudre le problème CSP. Étant donné que le processus est itératif, l'utilisateur est amené à vérifier si les contraintes générées sont correctes ou pas, ou à préciser les parties qui posent des problèmes.

L'approche de Bajwa et al. [88] propose un framework pour la génération dynamique de contraintes OCL à partir de langage naturel. Les utilisateurs ainsi expriment les contraintes en anglais qui sont ensuite transformées automatiquement. La vision des auteurs est de permettre de mettre en avant l'utilité du langage OCL sans avoir à maîtriser le langage. De la même manière, Bajwa et al. [89] ont proposé une autre alternative à la transformation du langage naturel vers OCL, en transformant les contraintes spécifiées en langage "Semantics Of Business Vocabulary and Rules" SBVR [90], un standard de l'OMG, vers le langage OCL.

3.2 Gestion de la maintenance et évolution des métamodèles

3.2.1 Maintenance et évolution des métamodèles

Dans cette partie, nous présentons quelques travaux relatifs à la maintenance et l'évolution des métamodèles ainsi qu'à la coévolution des modèles.

L'évolution des modèles est un sujet qui a été étudié par de nombreux travaux de recherche. Ces travaux peuvent être classés en quatre catégories selon [91]. La première concerne les approches où les concepteurs encodent manuellement la stratégie de migration en utilisant des langages de programmation comme Java, ou des langages de transformation de modèles comme QVT. Dans [60], Garces et al. ont comparé deux versions du métamodèle pour capturer les différences entre elles. Les règles de transformation sont ensuite utilisées pour adapter automatiquement les modèles. [92] propose une approche pour faire coévoluer les modèles. Elle commence par détecter les changements soit en comparant les métamodèles, soit en analysant la séquence de changements appliquée à l'ancienne version du métamodèle. Ensuite, les changements identifiés sont classés en fonction de leur impact sur les instances du modèle. Enfin, un algorithme de migration approprié pour la migration du modèle est déterminé. Les approches mentionnées ci-dessus reposent sur les règles de copie, qui peuvent être très complexes et longues à réaliser manuellement. Par conséquent, la deuxième catégorie d'approches d'évolution des métamodèles est apparue.

Cette catégorie concerne les approches qui utilisent des techniques de correspondance pour déduire des stratégies de migration en comparant les anciennes et les nouvelles versions du métamodèle. Dans ces approches [93-97], les règles de coévolution sont spécifiées comme des règles de transformation utilisant un métamodèle unifié qui représente les deux versions du métamodèle. Ensuite, une analyse est effectuée pour supprimer les éléments de modélisation qui ne sont pas présents dans la nouvelle version du métamodèle. Ainsi, les modèles peuvent être mis à jour pour être conformes au nouveau métamodèle.

La troisième catégorie englobe les approches qui capturent les changements de métamodèles en tant qu'opérateurs d'évolution [98]. La performance de ces approches dépend de la complétude de l'ensemble des opérateurs d'évolution. Wachsmuth [58] a enregistré un ensemble d'opérateurs pour faire évoluer les métamodèles et coévoluer les modèles.

Marković et Barr [99] ont proposé des règles pour coévoluer les diagrammes de classes et les contraintes OCL. Hassam et al. [100] ont proposé des opérations pour coévoluer des métamodèles et des contraintes OCL. Kruse et al. [101] ont rassemblé tous les opérateurs présents dans la littérature et les ont résumés. De plus, inspirés par Fowler [102] qui a proposé des opérateurs de refactoring de code, Kruse et al. ont proposé quelques opérateurs complexes d'évolution de métamodèles tels que *Introduce Composite Pattern*. Chacun des articles mentionnés ci-dessus a contribué à compléter la bibliothèque d'évolution des modèles en fournissant des opérateurs supplémentaires qui ont considérablement amélioré l'approche d'évolution des métamodèles basée sur les opérateurs.

Enfin, Kessentini et al. [103, 104] proposent une approche interactive multiobjective qui vise à adapter dynamiquement des modèles suite à l'évolution du métamodèle et interactivement suggérer des opérations d'édition aux concepteurs pour prendre leur avis en considération. Le processus de recherche est guidé par des fonctions objectives. L'avis du concepteur est utilisé pour réduire l'espace de recherche et converger vers de meilleures solutions. L'approche est évaluée sur divers scénarios d'évolution.

Certains travaux se sont intéressés aux homologues des bad smells de code introduits par Fowler et al.[102] puis complétés par [105], mais sur les modèles UML et des métamodèles. Ainsi, [106-109] ont proposé des bad smells des modèles UML ainsi que des métamodèles et des méthodes de refactoring pour retirer chaque bad smell.

3.2.2 Maintenance et coévolution des contraintes OCL

Cette partie regroupe les différents travaux qui ont traité la maintenance et la coévolution des contraintes OCL. Nous présentons d'abord les différents travaux de maintenance et de refactoring des contraintes OCL dans la partie suivante, ensuite, nous présentons les différents travaux portant sur la coévolution des contraintes OCL.

3.2.2.1 Maintenance et refactoring de contraintes OCL

Dans cette partie, nous présentons les différents travaux existant dans la littérature, et qui portent sur le refactoring et la maintenance des contraintes OCL.

D'abord, Cabot et al. [110] ont proposé pour la première fois une technique de trans-

formation de contraintes OCL qui permet d'obtenir des contraintes OCL sémantiquement équivalentes à celles prises en entrée. L'approche repose sur des techniques permettant de réaliser des changements non seulement sur le corps de la contraintes OCL, mais aussi de modifier le contexte de la contrainte OCL. L'approche se base sur quelques métriques pour évaluer les transformations.

Dans [111, 112], les auteurs proposent un catalogue d'optimisations pour les expressions OCL. En raison de l'absence d'outils automatiques de refactorisation des contraintes, les auteurs ont spécifiquement ciblé les optimisations pour les contraintes OCL générées automatiquement, qui ont tendance à être très complexes. L'étude a commencé par regrouper un ensemble d'optimisations OCL sous la forme d'un catalogue. Ce catalogue a été implémenté comme un outil générique qui prend chaque optimisation et la vérifie. Il a été validé sur un ensemble de contraintes OCL, et a prouvé son efficacité en réduisant la complexité des contraintes de 35%. Ce travail vise exclusivement les contraintes OCL, indépendamment du métamodèle.

Un autre travail portant sur les contraintes OCL a été rapporté par Correa et al. [113, 114], qui ont réalisé une étude expérimentale contrôlée pour analyser si la compréhension des contraintes OCL peut être affectée par la structure des expressions OCL. En particulier, l'objectif était de vérifier si la présence de smells OCL rendait les expressions OCL moins compréhensibles. L'étude révèle que la présence de smells OCL dans les expressions a eu un impact négatif à la fois sur la correction et sur le temps nécessaire à la compréhension des contraintes.

Dans le travail réalisé par Hong et al. [115], les auteurs ont proposé une approche de refactorings de contraintes OCL. D'abord, le refactoring de contraintes a été formalisé sous forme de problème d'optimisation multi-objectifs, où le but est de proposer de nouvelles contraintes OCL qui soient moins complexes, moins couplées, et avec plus de cohésion. Pour résoudre ce problème multi-objectifs, l'approche a été testée avec six algorithmes et sur quatre cas d'études. [116] a proposé une approche permettant de réduire le nombre d'éléments contraintes OCL, et ainsi réduire l'effort de maintenance et d'évolution des contraintes OCL. L'approche a été réalisée en proposant une formalisation des contraintes OCL sous forme d'aspects, et a résulté par l'outil AspectOCL. L'approche ainsi que l'outil ont été testé sur sept cas d'études ont permis de réduire de 55% le nombre

d'éléments des contraintes sans altérer la syntaxe OCL ni leur sémantique.

Reynoso et al, [117] puis [118] ont proposé un ensemble de métriques qualitatives pour les contraintes OCL. Enfin, Gogolla et al. [119] ont proposé un ensemble de métriques pour les contraintes OCL, et ont implémenté toutes les 27 métriques OCL connues dans la littérature dans leur outil USE [120]. Les résultats ont été validés par un expert en OCL qui a prouvé que les nouvelles métriques proposées sont en concordance avec l'opinion de l'expert.

Nous mentionnons ces travaux ci-dessous bien qu'ils ne portent pas sur la maintenance et le refactoring de contraintes OCL, car ils ont un apport très important dans le domaine de l'IDM, et plus particulièrement sur l'étude des contraintes OCL. D'abord, Mengerinks et al. [121] ont effectué des mesures pour tester les différences entre la complexité des contraintes OCL open-source et les contraintes industrielles. Les auteurs ont expliqué que la complexité est un bon indicateur de la similarité entre les artefacts. Cette étude empirique réalisée sur un ensemble de données regroupant plus de 9000 contraintes OCL a conclu que les contraintes open-source et industrielles ont la même complexité. Pour s'assurer que les ensembles de données disponibles dans la littérature sont représentatifs des données industrielles, une analyse plus approfondie est obligatoire pour tirer de meilleures conclusions. Enfin, pour faciliter les études empiriques du langage OCL, les auteurs dans [122] ont proposé un ensemble de 9188 contraintes OCL extraites de GitHub. L'ensemble de contraintes a été normalisé et analysé avec succès afin de supprimer les contraintes qui n'ont pas pu être analysées. Les auteurs ont étendu le jeu de données dans leur travail suivant [123] pour proposer un autre jeu de données de 103 262 contraintes OCL, ce qui est une grande contribution au domaine de la métamodélisation. Ce jeu de données englobe à la fois les invariants du métamodèle et les contraintes de transformation du modèle.

Les différents travaux présentés dans cette partie proposent de nouvelles techniques permettant de maintenir et de refactoriser les contraintes OCL. Pendant que certains travaux proposent des métriques pour mesurer divers aspects qualitatifs des contraintes OCL, d'autres approches exploitent ces métriques pour guider et évaluer leurs approches.

3.2.2.2 Coévolution de contraintes OCL

Dans cette partie, nous introduisons les travaux existants dans la littérature portant sur la coévolution de contraintes OCL. La coévolution des contraintes OCL est de deux types : la coévolution en ligne, qui consiste à coévoluer itérativement les contraintes OCL pendant l'évolution du métamodèle, c.à.d. à chaque évolution du métamodèle, les contraintes impactées par le changement sont coévoluées. Le deuxième type de coévolution est hors ligne, ce qui veut dire que la coévolution est faite après l'évolution complète du métamodèle. En plus de cette catégorisation, certaines approches sont automatiques, tandis que d'autres sont semi-automatiques, donc requièrent l'intervention du concepteur. Pour les approches en ligne, Markovic et al. [99, 124] ont d'abord proposé une approche dans laquelle ils formalisent les règles de refactoring les plus importantes pour les diagrammes de classes et les classent en fonction de leur impact sur les contraintes OCL annotées. Ces opérateurs de refactorings sont formalisés avec le langage Query View Transformation (QVT) [125]. Cette approche se concentre principalement sur la classification des règles de refactoring en fonction de leur impact sur les contraintes OCL. Ensuite, Hassam et al. [100, 126] ont proposé l'outil de coévolution METAEVOL qui se base aussi sur le langage QVT. L'analyse de l'impact de l'évolution du métamodèle est réalisée pendant l'évolution du métamodèle, mais la coévolution des contraintes ne se fait qu'après l'évolution du métamodèle. Un tableau intermédiaire permettant de connaître si les contraintes sont impactées par l'évolution ou pas est utilisé dans cette approche. Enfin, Demuth et al. [127, 128] proposent une approche de coévolution de contraintes OCL basée sur les modèles. Ils ont défini 11 structures génériques pour les contraintes OCL qui peuvent ensuite être instanciées pour mettre à jour les contraintes après l'évolution du métamodèle.

De l'autre côté, il existe des travaux proposant des approches hors ligne. Par exemple, Cabot et al. [129] se concentrent principalement sur les opérations de suppression. Ainsi, cette approche automatique a pour objectif de supprimer des contraintes ou des parties de celles-ci. Elle cible les éléments qui ne sont plus présents dans le métamodèle. Cette approche se concentre sur la suppression des contraintes qui provoquent des incohérences avec les schémas conceptuels après les opérations de soustraction. Cette approche est très puissante pour éviter les incohérences, mais elle n'est applicable que sur les contraintes OCL définies en forme normale conjonctive (FNC). De l'autre côté, Cabot et al. [110] ont aussi étudié la possibilité de changer le contexte de la contraintes OCL tout en la

maintenant.

Kusel et al. [130, 131] proposent des actions de résolution semi-automatiques pour la coévolution des expressions OCL dans les transformations de modèles en réponse à l'évolution du métamodèle. Khelladi et al. [132, 133] proposent une approche semi-automatique pour enregistrer les changements de métamodèles par ordre chronologique, et ainsi détecter les deux types de changements atomiques et complexes. Ensuite, ils appliquent des stratégies de résolution pour adapter les contraintes impactées par le changement, en proposant à l'utilisateur des contraintes OCL alternatives.

Batot et al. [134] proposent un processus en deux étapes pour faire co-évoluer automatiquement les métamodèles et les contraintes OCL à l'aide d'un algorithme génétique. Le problème de coévolution est ainsi traduit en un problème multi-objectifs, où pour chaque contrainte OCL, l'algorithme génétique explore les changements possibles dans l'espace de modélisation. Les auteurs ont défini trois fonctions objectives qui aident à guider l'exploration de l'espace de modélisation de telle sorte que la contrainte ne viole pas la nouvelle version du métamodèle, qu'il y ait un minimum de changement, et un minimum de perte d'information. La principale force de cette approche est l'utilisation de l'aléatoire à l'aide d'opérateurs génétiques, qui peut être intéressant pour un problème de coévolution.

Les approches proposées par Markovic et al. [99, 124], cabot et al. [129], Demuth et al. [127, 128], et Batot et al. [134] sont des approches complètement automatiques, tandis que Hassam et al. [100, 126], Khelladi et al. [132, 133], Kusel et al. [130, 131] sont des approches semi-automatiques.

3.3 Positionnement de la problématique de thèse

La littérature comprend une multitude de travaux portant sur l'assistance à la métamodélisation précise. Ces travaux, tels que présentés précédemment, se focalisent sur deux aspects : l'assistance à la création/maintenance de la structure du domaine ; et l'assistance à la création/co-évolution/refactoring de contraintes.

Si on se focalise sur les travaux portant sur l'assistance à l'écriture des contraintes OCL, ces derniers ont apporté une aide précieuse quand il s'agit d'écrire des contraintes OCL à travers la proposition de patterns de contraintes OCL qui servent à créer des

contraintes relatives à un métamodèle. Le principal manquement dans ces approches réside dans le fait que ces dernières se focalisent sur "Comment" simplifier la grammaire OCL pour permettre à un concepteur qui ne maîtrise pas le langage OCL de spécifier ces contraintes aisément, mais ne répondent pas aux questions "quand" et "pourquoi" des contraintes OCL doivent être spécifiées. En effet, ces approches sont avantageuses lorsqu'il s'agit de concepteurs ayant une bonne maîtrise du langage de modélisation MOF, ce qui fait qu'il/elle arrive à identifier les structures nécessitant des contraintes OCL. Cependant, lorsqu'il s'agit d'un concepteur n'ayant pas de connaissances approfondies en métamodélisation, ces approches lui seront bénéfiques qu'à moitié. En effet, si le concepteur souhaitant décrire des informations purement liées à la sémantique du domaine, l'utilisation de patterns OCL lui sera d'une grande utilité. En revanche, même si ce type de contraintes OCL est spécifié, il y a toujours moyen que des structures MOF qui ne sont pas toujours précises aient besoin de contraintes OCL.

Concernant les différentes approches qui génèrent d'une façon automatique ou semi-automatique les contraintes OCL en se servant de modèles valides et invalides pour valider l'ensemble de contraintes généré [15, 16, 87], le problème de savoir quelles structures cibler par OCL ne se pose pas. En effet, ces approches s'occupent de la génération automatique de contraintes OCL en se basant sur un ensemble valide et invalide de modèles. La faiblesse de ces approches réside dans le fait que leurs performances reposent sur l'ensemble des modèles valides et invalides. En effet, cet ensemble doit être suffisamment complet et divers pour permettre de distinguer entre les contraintes à garder ou à retirer. De plus, l'acquisition de modèles valides et invalides doit se faire en utilisant des technologies de génération automatique de modèles. Ensuite, le tri des différents modèles générés en "valides" et "invalides" doit se faire par quelqu'un ayant au préalable une bonne connaissance du domaine applicatif. Ce type d'approches n'est pas adapté aux concepteurs qui débutent en modélisation.

Pour répondre à la problématique d'assistance à l'identification de structures pouvant nécessiter des contraintes OCL, nous proposons une approche qui permet d'identifier ces structures. Cette approche repose sur une étude de métamodèles ayant des contraintes OCL dans l'optique de déterminer pourquoi les contraintes OCL sont spécifiées, et donc déterminer les différentes structures qui sont souvent contraintes. Notre travail vient en complément de l'étude empirique de Cadavid et al. [19] portant sur l'étude de métamodèles

et de contraintes OCL. Il complète aussi les travaux de Wahler et al., [85] qui pointent des anti-patterns dans les métamodèles. Cette contribution est détaillée dans le chapitre 4.

Quand il s'agit des travaux portant sur la coévolution des contraintes OCL suite à l'évolution de leur métamodèle, il existe un axe auquel les différents travaux n'ont pas répondu. Les différentes approches existantes dans la littérature ont comme principal objectif de faire coévoluer des contraintes OCL pour qu'elles soient conformes à la nouvelle version du métamodèle. Ces différentes approches sont très performantes pour répondre à cette problématique. Cependant, les approches ne tiennent pas en compte la complétude de l'ensemble de contraintes OCL. Effectivement, pour faire évoluer le métamodèle, un ensemble d'opérateurs d'évolution est appliqué. L'application de ces opérateurs crée, modifie, ou supprime des concepts du métamodèles. Par conséquent, les approches existantes ne vérifient pas si l'évolution du métamodèle a engendré de nouveaux concepts qui pourrait nécessiter des contraintes OCL. Ainsi, même en ayant co-évolué toutes les contraintes OCL, il est fort possible que le nouvel ensemble de contraintes OCL qui est compatible avec la nouvelle version du métamodèle soit incomplet, et permettrait de créer des modèles qui ne représentent pas le domaine.

Pour répondre à cette problématique, nous proposons une approche qui complète les approches de coévolution existantes dans la littérature, et ceci en notifiant les concepteurs qui évoluent le métamodèle de l'apparition de structures pouvant nécessiter des contraintes OCL. Cette contribution est détaillée dans le chapitre 5.

3.4 En résumé

Dans ce chapitre, nous avons présenté les travaux existant dans la littérature et qui portent sur l'assistance à la métamodélisation précise. Ensuite, nous avons identifié des limitations dans l'état de l'art quant à l'absence d'une assistance à l'identification de structures pouvant nécessiter des contraintes OCL. Nous avons identifié aussi une deuxième limitation dans les approches de coévolution de contraintes OCL qui existent dans la littérature. Dans les chapitres suivants, nous proposons des approches pour combler les limitations que nous avons identifié dans ce chapitre.

Troisième partie

Contributions

Chapitre 4

Étude d'investigation sur les structures imprécises dans les métamodèles

Sommaire

4.1 Introduction

Dans ce chapitre, nous détaillons notre contribution portant sur l'étude des métamodèles et leurs contraintes OCL. La section 4.2 introduit la problématique traitée dans la contribution ainsi que les motivations qui nous ont conduit à étudier des métamodèles avec leurs contraintes OCL. Ceci est expliqué avec un exemple de métamodèle utilisé par des ingénieurs de notre partenaire industriel, Segula Engineering. Ensuite, la section 4.3 détaille le processus expérimental que l'on a employé pour l'étude empirique, en commençant par la projection des contraintes sur le métamodèle pour obtenir les fragments des structures contraintes, en allant jusqu'à l'étude de chaque structure avec sa contrainte pour déterminer l'intérêt de la contraintes OCL. Avant de conclure le travail dans la section 4.5, nous présentons les résultats de notre étude empirique dans la section 4.4.

4.2 Problématique et Motivations

Afin de garantir la précision d'un métamodèle, l'utilisation conjonctive des deux langages MOF et OCL est indispensable. Les concepteurs de métamodèles définissent la syntaxe abstraite du langage de modélisation à travers le métamodèle, puis l'enrichissent d'informations sémantiques avec des contraintes textuelles définies avec OCL. Cependant, écrire des contraintes OCL reste une tâche qui consomme énormément de temps, et qui peut très bien induire en erreur [15], surtout si le concepteur qui écrit les contraintes ne maîtrise pas le langage OCL. Ce fait décourage beaucoup de concepteurs de métamodèles et limite l'utilisation du langage OCL en pratique. L'utilisation des contraintes OCL reste cependant indispensable. Dans le cas où elles sont absentes, des modèles et artefacts incorrects peuvent être dérivés du métamodèle, ce qui le rend immature [85]. En conséquence, la précision et la complétude du langage de modélisation dépend énormément de l'ensemble de ses contraintes. Ainsi, un métamodèle est mature si son espace de modélisation, qui représente l'ensemble de tous les modèles qui peuvent être créés à partir de lui regroupent seulement les modèles qui représentent le domaine.

Les approches existantes dans la littérature se focalisent souvent sur la façon d'écrire les contraintes OCL, ce qui a donné lieu à beaucoup d'approches. Certaines approches génèrent automatiquement les contraintes OCL utilisant des approches métaheuristiques telles que les algorithmes génétiques [15], d'autres approches proposent des patrons OCL

qui sont des contraintes OCL génériques paramétrables, qui, suite au remplacement des paramètres par des valeurs réelles du métamodèle donne lieu à des contraintes OCL propres au métamodèle [16]. D'autres travaux proposent des transformations du langage naturel vers le langage OCL afin de faciliter aux concepteurs l'écriture des contraintes [88]. Enfin, des approches sont proposées pour faciliter la coévolution des contraintes OCL suite à l'évolution du métamodèle [100], [133]. Le point fort de ces approches est de permettre au concepteur de pouvoir exprimer ces contraintes OCL plus facilement. Le principal manquement dans ces approches réside dans le fait qu'elles ne proposent pas au concepteur une assistance sur "où" appliquer des contraintes OCL. En pratique, le manque de certaines contraintes OCL est facilement identifiable dans le métamodèle. Cependant, le manque d'autres contraintes est moins évident à détecter. Par conséquent, un concepteur débutant ne saura pas toujours quand faudrait-il écrire des contraintes OCL, ce qui donne lieu à des métamodèles immatures.

4.2.1 Types de contraintes OCL

Si nous analysons les contraintes OCL de la littérature (UML par exemple), nous pouvons distinguer deux types de contraintes OCL. Le premier type comprend les contraintes qui sont écrites pour détailler la sémantique des concepts du domaine, et qui ne sont pas complètement liées à la structure du métamodèle. Le second type comprend les contraintes qui sont également utilisées pour contraindre les concepts du domaine, mais qui sont plus étroitement liées à la structure du métamodèle.

Contrairement aux contraintes du premier type, la nécessité d'exprimer certaines des contraintes du second type peut être perçue simplement en analysant la structure du métamodèle. Nous appelons les structures à l'origine du second type de contraintes : *Structures imprécises du métamodèle* (MIS).

4.2.2 Exemple Illustratif

Pour illustrer la différence entre les deux types de contraintes, nous nous appuyons sur le métamodèle P&ID (*Piping and Instrumentation Diagram*). Le P&ID est le langage normalisé pour la modélisation des architectures physiques des processus industriels, tels que les systèmes hydrauliques, chimiques et nucléaires [135]. Dans [136], le P&ID est défini comme : "*la description détaillée de l'architecture des flux de processus illustrant la*

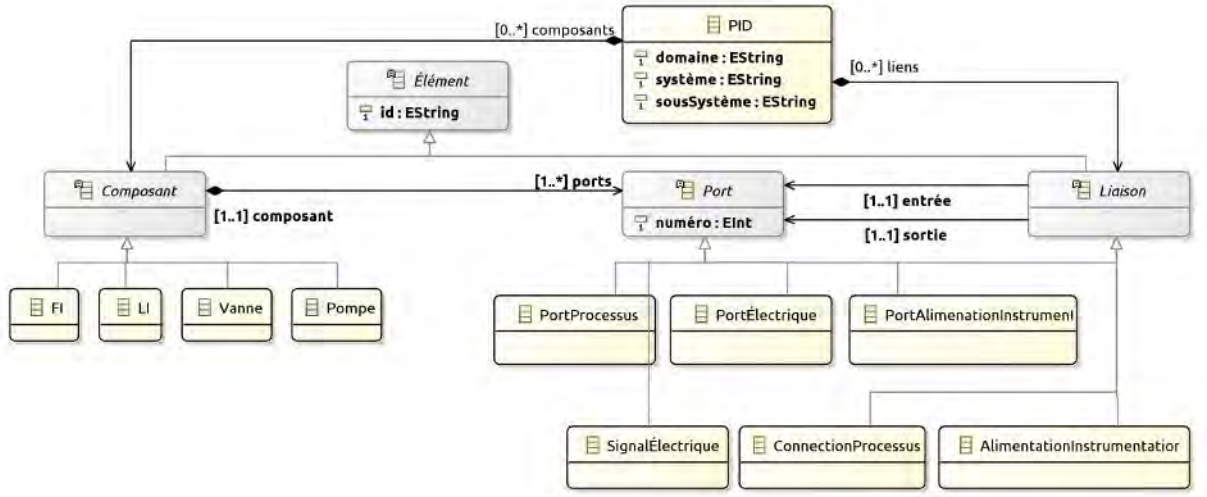

FIGURE 4.1 – Le métamodèle P&ID

tuyauterie, les composants et l'instrumentation associés au système". Un P&ID est essentiellement constitué de composants et de liaisons (Fig. 4.1). Il existe différents types de composants : *Pompe, Vanne, FI, LI*. Les pompes et les vannes, par exemple, sont des composants de processus qui entraînent et régulent le transport des fluides [135]. *FI* et *LI* sont des indicateurs ou des transmetteurs qui sont utilisés pour mesurer et/ou contrôler une variable de processus (comme le niveau, la pression, etc.) [135]. Le composant comprend un ensemble de ports de différents types (processus, électrique, etc.) permettant son interaction avec l'environnement. Les liaisons (par exemple *ConnectionProcessus, AlimentationInstrumentation, SignalÉlectrique*, etc.) assurent l'interaction entre les composants. Chaque liaison a deux extrémités (*entrée, sortie*), où chaque extrémité est connectée à un port.

Le métamodèle P&ID est accompagné d'un ensemble de contraintes OCL pour préciser certaines informations qui sont difficiles ou impossibles à préciser dans le diagramme. Par exemple, l'association avec *entrée* comme rôle, et qui cible la super-classe *Port* dans la Fig. 4.1. Cette structure nécessite une contrainte OCL pour spécifier le type exact (la sous-classe spécifique) qui doit être pris en compte pour les instances obtenues lors de la navigation dans cette association. Cette contrainte stipule que le *SignalÉlectrique* doit être connecté, par l'intermédiaire de l'association *entrée*, uniquement au *PortÉlectrique* (le type de l'association *entrée* est spécifié par la sous-classe *PortÉlectrique*). Sans cette contrainte, le métamodèle P&ID permettrait de créer un modèle imprécis avec des asso-

ciations *entrée* ayant pour type les différentes sous-classes de la classe *Port*. Le même MIS est également identifié pour l'association *sortie*. La contrainte ajoutée au métamodèle est spécifiée comme suit :

context SignalÉlectrique **inv** compatibilitéPort :
self.entrée.oclIsTypeOf(PortÉlectrique) **and**
self.sortie.oclIsTypeOf(PortÉlectrique)

La raison principale de la spécification de cette contrainte est la présence de l'association "entrée" ou "sortie" qui relient deux super-classes chacune ayant plusieurs sous-classes qui représentent un type. Souvent, cette structure qui représente un MIS n'est pas assez précise pour bien représenter le domaine applicatif.

La présence de MIS dans un métamodèle émet juste un doute sur la nécessité d'ajouter une contrainte OCL. C'est au concepteur du métamodèle, qui connaît mieux le domaine, de décider si les instances de MIS nécessitent des contraintes ou non. Une analogie peut être faite avec les smells de code [102] qui suggèrent la présence potentielle d'une mauvaise structure dans le code. Lorsqu'un code smell est identifié dans un extrait de code, il appartient au programmeur de le valider en effectuant un refactoring ou de l'ignorer tout simplement si ce dernier ne pose aucun problème.

Dans ce travail, nous avons étudié des métamodèles existants pour identifier empiriquement une liste de MIS. Puis nous avons validé quantitativement et qualitativement leur utilité. La conception de cette étude expérimentale est présentée dans la section suivante.

4.3 Processus expérimental

Nous estimons que la meilleure façon de mettre en évidence les structures imprécises des métamodèles est d'analyser des métamodèles existants comportant des contraintes OCL. En étudiant les parties du métamodèle où les contraintes ont été ajoutées, et les raisons pour lesquelles ces contraintes ont été ajoutées, nous serons en mesure de conclure si nous sommes confrontés à un MIS ou non.

4.3.1 Objectifs et question de recherche

L'étude vise à répondre à la question de recherche suivante :

QR : Quelles sont les structures de métamodèles (MIS) qui sont souvent complétées par des contraintes OCL ?

Cette question vise à caractériser les structures de métamodèles qui sont fréquemment raffinées avec des contraintes OCL ainsi que la raison pour laquelle ces structures sont plus contraintes que d'autres.

L'hypothèse associée et son alternative sont les suivantes :

— H_0 : l'étude nous permettra de trouver au moins une structure récurrente imprécise et donc complétée avec OCL pour être plus précise.

— H_1 : l'étude ne nous permettra pas d'identifier de structure récurrente imprécises.

4.3.2 Approche et données

Pour répondre à la question de recherche, nous avons établi le processus expérimental suivant, qui est illustré dans la figure 4.2 :

1. Pour chaque contrainte, nous la projetons sur le métamodèle pour identifier la structure ciblée dans le métamodèle. Ainsi, nous obtenons un ensemble de couples (contrainte, structure).

2. Chaque structure de métamodèle et sa contrainte sont étudiées par cinq personnes. Chacune ayant un profil différent : i) Un étudiant qui mène un doctorat sur OCL et MOF ; ii) Un docteur en informatique ayant travaillé sur MOF et OCL pendant quatre années ; et iii) Trois chercheurs seniors qui travaillent sur/enseignent OCL et MOF durant plus de 15 ans. L'objectif de cette étude est de savoir pourquoi chaque contrainte a été définie et si : i) la contrainte est étroitement liée aux faiblesses du langage MOF qui ne permet pas d'exprimer cette contrainte, ii) elle est liée à un choix du concepteur, ou bien : iii) si elle est purement spécifique au domaine applicatif du métamodèle.

3. Après avoir obtenu un ensemble de structures récurrentes qui sont souvent contraintes, nous généralisons la structure et expliquons la faiblesse de chacune ainsi que les

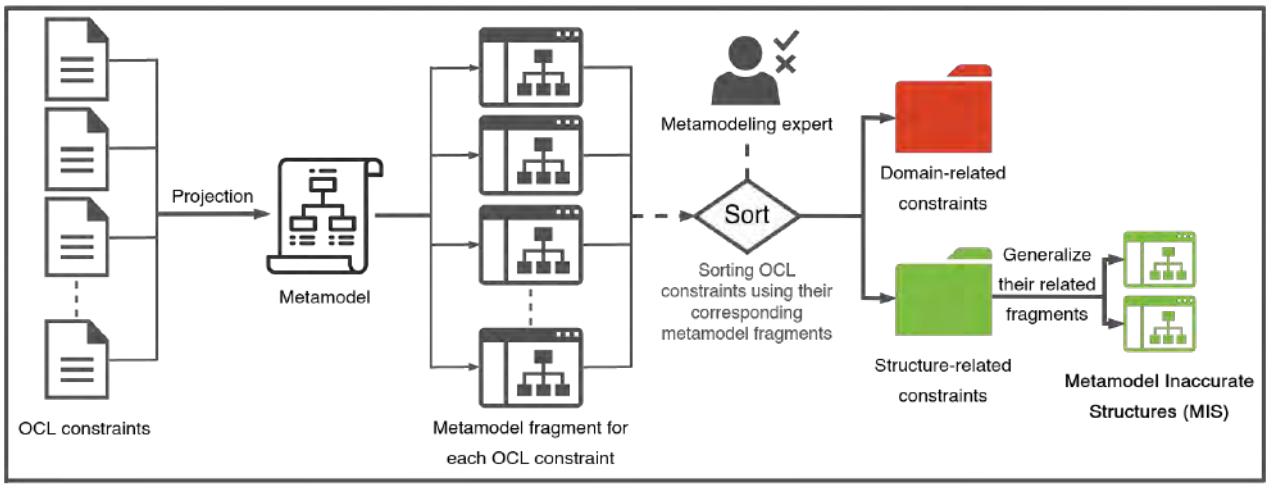

Figure: MIS Identification Process

FIGURE 4.2 – Processus d'identification des MIS

différentes contraintes OCL qui peuvent y être appliquées afin de la rendre plus précise.

Nous avons porté une attention particulière à la qualité des métamodèles et des contraintes étudiées. Parmi tous les métamodèles disponibles dans l'état de l'art, nous avons sélectionné le métamodèle UML version 2.5 [53] qui répond le mieux à nos besoins. En effet, ce métamodèle comprend 242 classes distinctes issues de 14 packages et englobe 416 contraintes OCL. Il s'agit d'une base cohérente pour ce type d'études empiriques, d'autant plus que le métamodèle est créé et maintenu par l'OMG, les créateurs des deux langages MOF et OCL.

Un prétraitement a été effectué sur certaines de ces contraintes. Celles-ci ont été divisées en plusieurs contraintes plus petites, car elles ciblaient différentes structures. L'ensemble final qui a été considéré dans cette étude est composé de 450 contraintes.

La version 2.5 du métamodèle a bénéficié de plusieurs itérations de changement corrigeant plusieurs erreurs qui étaient présentes dans les contraintes OCL de la version 2.4 [19], ce qui en fait une source fiable depuis lors. Nous avons également plus de chances de trouver des structures diverses en raison de la taille du métamodèle (indiquée ci-dessus).

Nous avons volontairement choisi de ne pas étudier les métamodèles de la littérature

contenant des contraintes OCL qui ne sont pas destinées à préciser la sémantique du métamodèle. Par exemple, l'ensemble de données présenté dans [123] contient plus de 100 000 contraintes OCL. Cependant, beaucoup de ces contraintes servent à la transformation de modèles. De plus, ce jeu de données a été créé en rassemblant des métamodèles et leurs contraintes OCL sur GitHub [137], indépendamment de leurs concepteurs et de leur niveau d'expertise en métamodélisation. Comme notre étude vise à récupérer les bonnes pratiques à partir des métamodèles et de leurs contraintes, nous devons nous assurer qu'ils sont bien spécifiés. Cependant, ceci n'est pas garanti pour les métamodèles et les contraintes dans les dépôts publics, où aucune vérification n'est faite avant leur publication.

4.4 Résultats

Les résultats sont composés d'un ensemble de structures imprécises de métamodèles (MIS) identifiées. Chaque MIS sera présenté individuellement avec sa définition et un exemple. De plus, pour chaque MIS, nous décrivons sous forme de question le problème éventuel soulevé. Nous proposons également des patterns de contraintes OCL pour chaque MIS, ce qui représente une contrainte générique qui doit être personnalisée avec les informations du métamodèle afin d'obtenir une contrainte OCL concrète qui correspond à une solution possible au problème. Pour les MIS qui peuvent être évités par le refactoring, nous proposons comment éviter le problème sans utiliser les contraintes OCL.

4.4.1 Restriction de la valeur d'un attribut

Le langage MOF permet de détailler les concepts par le biais d'attributs. Chaque attribut est nommé et typé en fonction du domaine. Cependant, il n'y a aucun moyen de spécifier le domaine de définition précis auquel l'attribut appartient. Par exemple, un attribut entier peut prendre n'importe quelle valeur de l'ensemble des entiers naturels N. Comme l'illustre la Fig. 4.3, si l'attribut désigne l'âge d'un employé qui doit être compris entre 18 et 65, ce détail ne peut pas être spécifié via un diagramme, ce qui peut conduire à la création de modèles avec des informations incorrectes par rapport au domaine. Pour y remédier, le seul moyen de spécifier le domaine de définition précis des attributs est d'écrire des contraintes OCL comme la contrainte C1. Ce MIS est le plus basique et est composé d'un seul élément de modélisation (attribut). Ce MIS est lié à la question suivante *Est-ce que le domaine de définition de l'attribut par défaut correspond au domaine*

de définition dicté par la sémantique du domaine ?

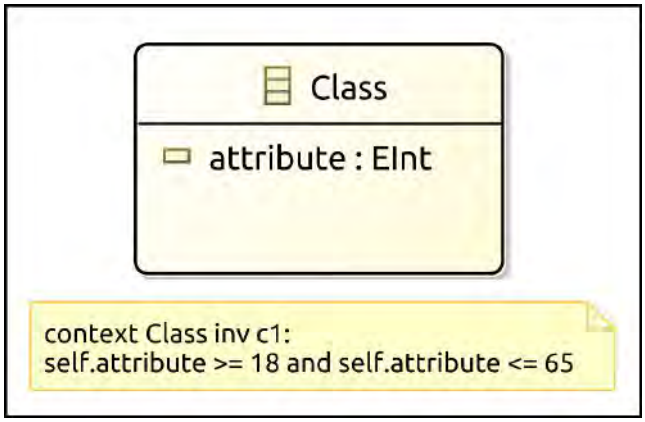

FIGURE 4.3 – Restriction de la valeur d'un attribut

Concernant le MIS *Restriction de la valeur d'un attribut*, le pattern suivant décrit la structure de contrainte OCL qui peut être exprimée pour le préciser, où {Classe, navigation, attribut, OpérateurArithmétique, Valeur} sont des paramètres qui doivent être remplacés par des valeurs réelles du métamodèle pour obtenir une contrainte OCL.

```
1  context Classe inv p1:
2  self.attribut OpérateurArithmétique Valeur
```

Dans certains cas, la contrainte ne sert pas à restreindre la valeur de l'attribut, mais à restreindre les éléments qui peuvent être associés à l'attribut en fonction de sa valeur. Dans ce cas, le pattern OCL se manifeste comme suit :

```
1  context Classe inv p2:
2  self.navigation.attribut OpérateurArithmétique Valeur
```

Bien que les patterns OCL mentionnés ci-dessus soient très similaires, la sémantique des contraintes est très différente. Le tableau 4.1 regroupe l'ensemble des opérateurs arithmétiques qui peuvent être utilisés avec le pattern OCL susmentionné en fonction du type d'attribut.

Type de l'attribut	Opérateurs
Attribut Numérique	> , < , >= , <= , = , <>
Booléen	= , <>
Énumération	= , <>

TABLE 4.1 – Les opérateurs arithmétiques qui peuvent être utilisés dans les patterns P1 et P2

4.4.2 Restriction des Littéraux d'une énumération

Les énumérations sont des types de données dont les valeurs sont énumérées dans le métamodèle comme des littéraux [53]. Par conséquent, un attribut de type énumération est un attribut qui peut prendre comme valeur l'un des littéraux définis dans l'énumération. Lorsque le métamodèle comporte de nombreux attributs du même type Enumeration, un problème peut survenir. Comme l'illustre la structure A.1 de la Fig. 4.4, dans une classe "Class1", si un attribut "attribute1" de type "Enum" peut prendre comme valeurs l'ensemble des littéraux {litA,litB}, et qu'un autre attribut "attribut2" de la "Classe2", également de type "Enum", peut prendre comme valeurs {litA,litC}, l'ensemble des littéraux de "Enum" sera l'union des deux ensembles {litA,litB} et {litA,litC}, ce qui donne {litA,litB,litC}. En conséquence, l'attribut "attribut1" peut prendre comme valeur le littéral 'litC', et "attribut2" peut prendre comme valeur le littéral 'litB', ce qui est faux par rapport au domaine d'application. La première solution consiste à garder "Enum" et de spécifier le sous-ensemble de littéraux que chaque attribut peut prendre comme valeur à travers des contraintes OCL. Par exemple, les contraintes c2 et c3 de la figure 4.4 A.2 sont appliquées aux classes Class1 et Class2 respectivement de la structure A.1, chacune visant à définir les littéraux qui peuvent être pris comme valeur pour chaque attribut. La deuxième solution consiste à fractionner l'énumération "Enum" en deux énumérations "Enum1" et "Enum2", avec des ensembles de littéraux {litA,litB} et {litA,litC} respectivement, comme dans la figure B de 4.4. Cette solution n'est pas recommandée car elle divise le même concept en plusieurs selon la situation, ce qui rend le métamodèle moins modulaire et plus difficile à maintenir et à réutiliser.

Lorsque l'on trouve plusieurs attributs ayant le même type d'énumération, nous pouvons nous poser la question suivante : *Lorsqu'il existe dans un métamodèle plusieurs attributs ayant la même énumération comme type, est-ce que tous les attributs dont le type est cette énumération acceptent tous ses littéraux comme valeur ?*

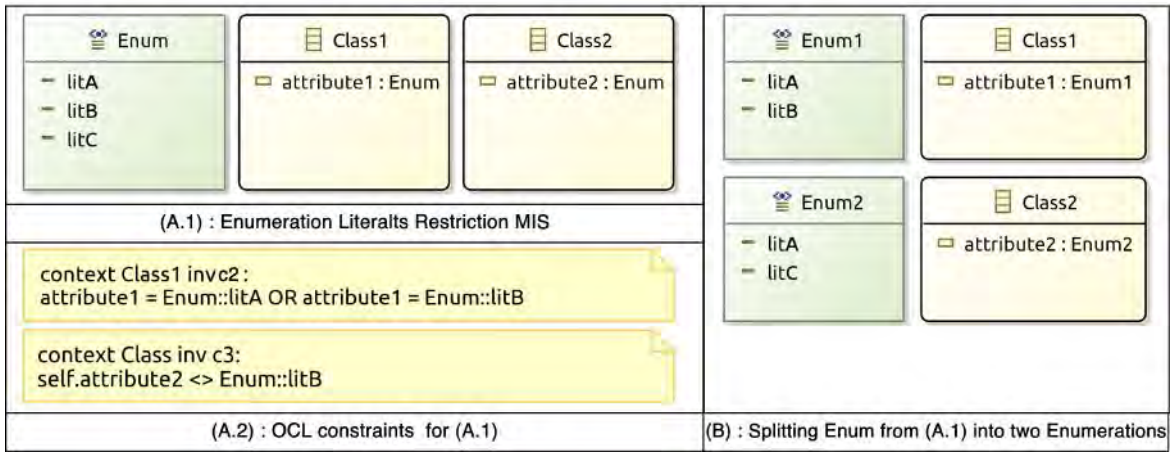

FIGURE 4.4 – Exemple du MIS Restriction des Littéraux d'une Énumération

En considérant le MIS *Restriction des Littéraux d'une Énumération*, le pattern suivant décrit la structure de la contrainte OCL qui peut être exprimée pour la préciser, où {Classe, navigation, attribut, OpérateurArithmétique, NomEnumération, littéral} sont des paramètres qui doivent être remplacés par des valeurs réelles du métamodèle afin d'obtenir une contrainte OCL.

```
1  Context Classe inv p3:
2  self.navigation.attribut OpérateurArithmétique NomEnumération::litté
       ral
```

Ce pattern permet d'écrire deux différents types de contraintes OCL, le premier pour spécifier les littéraux que cet attribut peut prendre comme valeur (comme la contrainte c2 de la figure A.2 de la Fig.4.4). Le second type permet d'exclure les littéraux qui ne doivent pas être pris comme valeur (comme la contrainte c3 de la même figure).

4.4.3 Attribut optionnel hérité

Étant donné que l'héritage est utilisé pour rendre les métamodèles plus réutilisables et plus faciles à faire évoluer, ce concept est fréquemment utilisé, en particulier dans les grands métamodèles comme UML. Nous avons remarqué que de nombreux attributs dans les super-classes ont été définis comme optionnels (avec des bornes [0..1]) comme pour la structure A.1 dans la figure 4.5. Cela est dû au fait que cet attribut est facultatif ou n'est

pas obligatoire à spécifier dans au moins une sous-classe. Par conséquent, même si dans certaines autres sous-classes ce même attribut est obligatoire, il est possible de ne pas le spécifier du fait de sa multiplicité 0..1, ce qui peut conduire à un métamodèle incomplet. Pour éviter cela, trois solutions sont possibles.

La première solution, et la plus recommandée, consiste à définir des contraintes OCL dans les sous-classes qui nécessitent des multiplicités différentes. L'avantage de cette solution est que le métamodèle reste inchangé. Pour la structure A.1 de 4.5, la contrainte c4 dans A.2 est écrite pour restreindre "attribut" dans la classe 2 et le rendre obligatoire (avec des bornes de [1..1]). La deuxième solution consiste à spécialiser l'attribut dans les sous-classes où cet attribut doit avoir des multiplicités différentes et à spécifier les multiplicités correctes. Dans notre exemple 4.5 structure B, redéfinir l'attribut "attribute" dans la sous-classe2 avec une multiplicité de [1..1]. La dernière solution consiste à réduire l'utilisation de l'héritage lorsque cela peut poser problème. Cela peut être fait en déplaçant l'attribut de la super-classe vers les sous-classes lorsque cela est possible. Par exemple, l'"attribut" de la "Classe1" de la figure 4.5 structure A.1 (C) est déplacé vers "Subclass1" avec une multiplicité de [1..1], et vers "Subclass2" avec une multiplicité de [0..1]. Cette solution est coûteuse et peut donner lieu à des métamodèles complexes comportant de nombreuses redondances, mais ne nécessite pas l'ajout de contraintes OCL.

Conernant ce MIS, nous pouvons nous poser la question suivante : *Ayant un attribut optionnel (0..1) défini dans une super-classe, existe-t-il une sous-classe dans laquelle cet attribut est obligatoire et doit toujours être spécifié ? ou ambigu, et doit être exclu ?*

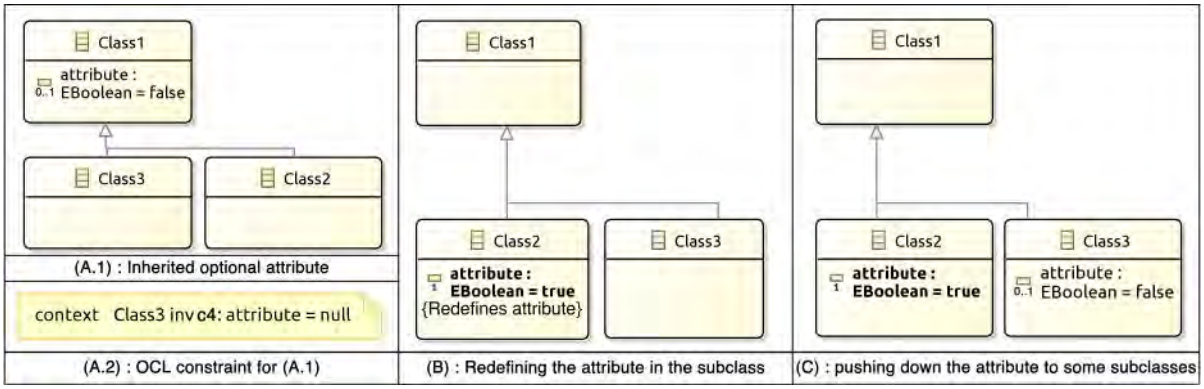

FIGURE 4.5 – Attribut Optionnel hérité

En ce qui concerne le MIS *Attribut Optionnel Hérité*, le pattern suivant décrit la structure de la contrainte OCL qui peut être exprimée pour le préciser, où {Classe, navigation, attribut, Opérateur, Valeur} sont des paramètres qui doivent être remplacés par des valeurs réelles du métamodèle pour obtenir une contrainte OCL concrète.

```
1        Context Classe inv p4:
2        self.navigation.attribut ->size() Opérateur Valeur
3        self.navigation.attribut $->$ notEmpty()
```

4.4.4 Restriction de la Multiplicité d'une Association Héritée

Tout comme les attributs, lorsqu'une super-classe contient une association, ses sous-classes en héritent avec la même valeur de multiplicité. Souvent, la multiplicité d'association spécifiée dans les super-classes est suffisamment large pour englober toutes les multiplicités dont chaque sous-classe a besoin pour l'association. Par exemple, si une association "asso" doit être généralisée des sous-classes "SubC1" et "SubC2" à la super-classe "Class1", où la multiplicité de "asso" dans "SubC1" est [1..*] et [0..*] dans "SubC2", cette association deviendra [0..*] une fois généralisée dans "Class1" ce qui donne la structure A.1 de la figure 4.6). Nécessairement, la multiplicité doit être détaillée dans les sous-classes selon la sémantique du domaine.

Encore une fois, cela peut être fait en spécialisant l'association dans certaines sous-classes avec les multiplicités correctes. Par exemple, dans la structure B de la figure 4.6), l'association "assoRed" redéfinit "asso" dans "SubC1" en changeant sa multiplicité de [0..*] à [1..*]. La deuxième option est de déplacer l'association de la super-classe vers les sous-classes "SubC1" et "SubC2" et de définir la multiplicité exacte dans chaque sous-classe comme dans la structure C. Cette méthode n'est pas toujours applicable et dépend de si l'association est nécessaire dans la super-classe ou non. Par exemple, si la super-classe est abstraite, il est possible syntaxiquement de déplacer l'association vers les sous-classes. La dernière option consiste à ajouter des contraintes OCL pour spécifier la multiplicité correcte de l'association dans certaines sous-classes. Par exemple, les contraintes C5 et C6 de la structure A.2 sont ajoutées à la structure A.1. Parfois, la multiplicité est liée à une valeur d'attribut, ce qui nécessite la contrainte C6 au lieu de C5.

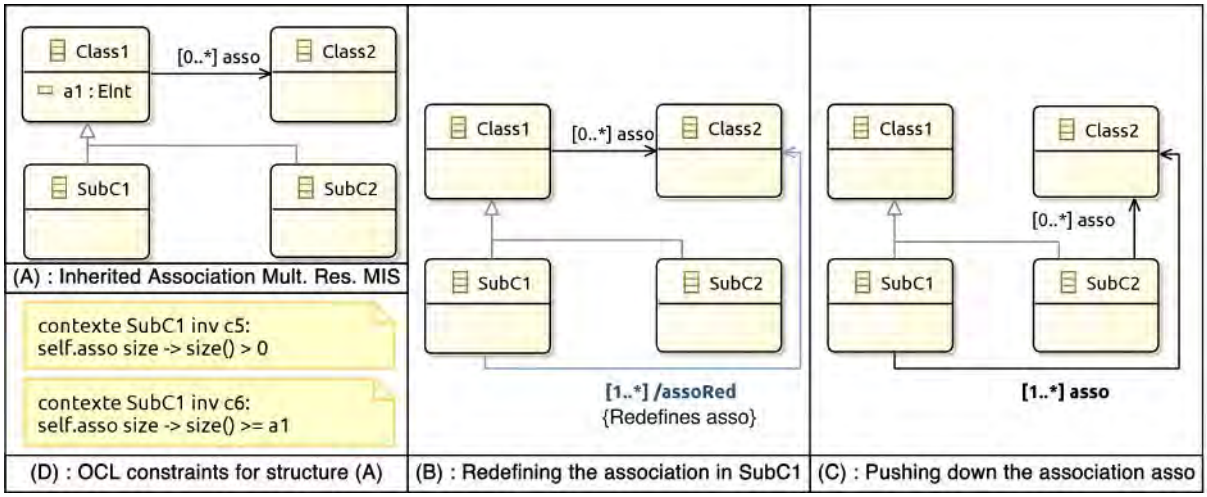

FIGURE 4.6 – Restriction de la Multiplicité d'une Association Héritée

En ce qui concerne le MIS, la question est la suivante : *Ayant une association définie dans une super-classe, et ayant des multiplicités inférieure et superieure différentes, est-ce que les sous-classes héritent l'association avec les mêmes valeurs de multiplicité qui sont spécifiées dans la super-classe ?*

Pour le *Restriction de multiplicité d'association héritée*, le pattern suivant décrit la structure de contrainte OCL qui peut être exprimée pour le préciser, où {Classe, navigation, association, Opérateur, Valeur} sont des paramètres qui doivent être remplacés par des valeurs réelles du métamodèle pour obtenir une contrainte OCL.

```
1 Context Classe inv p5:
2 self.navigation.association $->$ size() Opérateur Valeur
```

```
1 Context Classe inv p6:
2 self.navigation.association $->$ isEmpty()/notEmpty()
```

Pour le pattern P5, les opérateurs arithmétiques sont les suivants : >, <, >=, <=, =, <>. Ce pattern peut être remplacé par p6 dans certaines situations où l'utilisation des opérations OCL isEmpty() et notEmpty() est possible.

4.4.5 Restriction de la Valeur d'un Attribut Hérité

Dans certains cas, même si le domaine de définition de l'attribut dans la super-classe où il est défini est précis, il doit être redéfini dans les sous-classes avec les valeurs adéquates. Par conséquent, les contraintes OCL sont nécessaires comme seul moyen de spécifier le domaine de définition attributs. Par exemple, la structure A.1 de la figure 4.7, illustre un attribut hérité "attrib". Considérons que la contrainte C7 est appliquée à Class1 pour que "attrib" soit supérieur ou égal à 0. Si cet attribut dans SubC1 doit être différent de zéro, la contrainte C8 est nécessaire. Il est possible de redéfinir l'attribut dans la sous-classe "SubC1" comme dans la figure B.1, mais la contrainte C10 sera toujours nécessaire dans la sous-classe. De plus, si nous déplaçons l'attribut vers les sous-classes, nous devrons modifier les contraintes qui lui ont été appliquées dans la super-classe. Dans notre exemple, l'attribut "attribut" est déplacé vers les sous-classes 1 et 2 de la figure C.1. Dans ce cas, le MIS de restriction de valeur d'attribut hérité deviendra un MIS de restriction de valeur d'attribut, ce qui signifie que pour chaque sous-classe nous pouvons écrire une contrainte pour préciser le domaine de définition de l'attribut. Dans notre exemple de la figure C.1, les contraintes C11 et C12 sont écrites, chacune restreint l'attribut selon le domaine.

La question que l'on peut se poser est la suivante : *Ayant un attribut hérité, y aurait t-il des valeurs qu'il ne doit pas prendre dans une des sous-classes qu'elles l'héritent ?*

Pour le MIS *Restriction de la Valeur d'un Attribut Hérité*, le pattern suivant décrit la structure de contrainte OCL qui peut être exprimée pour le préciser, où Classe, navigation, attribut, Opérateur, Valeur, TypeEnumération, Littéral sont des paramètres qui doivent être remplacés par des valeurs réelles du métamodèle pour obtenir une contrainte OCL.

```
1 context Classe inv p7:
2 self.navigation.attribut Opérateur Valeur
```

```
1 context Classe inv p8:
2 self.navigation.attribut Operateur TypeEnumération::Littéral
```

Alors que le pattern p7 est utilisé pour tous les types d'attributs, le motif p8 n'est utilisé que pour les attributs d'énumération. Le tableau 4.2 précise les opérateurs qui peuvent être utilisés selon le type d'attribut.

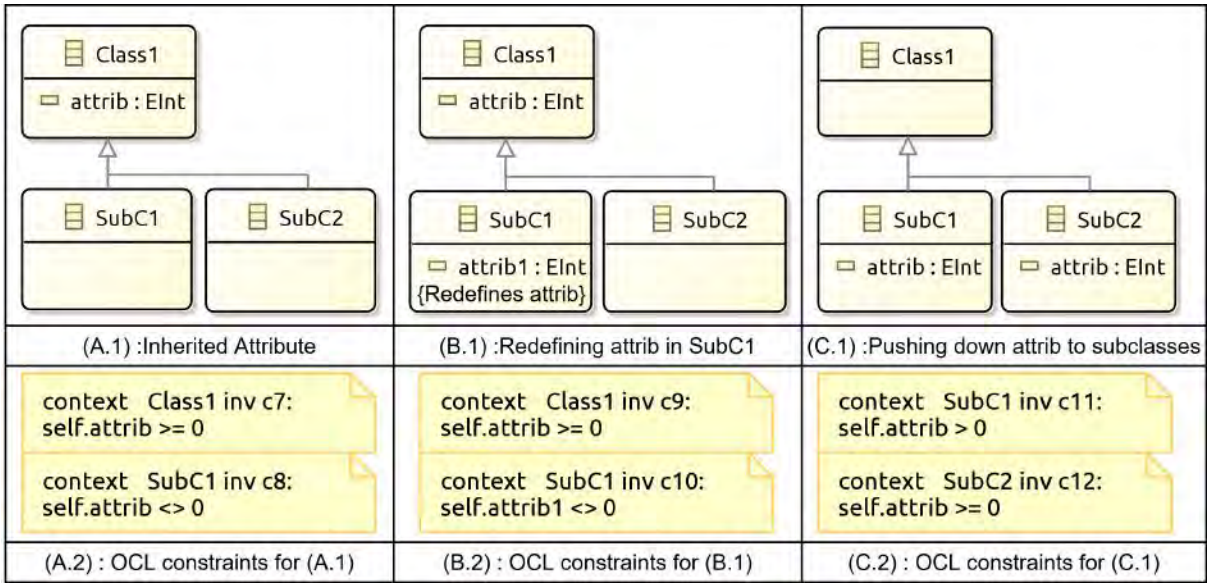

FIGURE 4.7 – Restriction de la Valeur d'un Attribut Hérité

Type d'Attribut	Opérateurs
Attribut Numérique	> , < , >= , <= , = , <>
Booléen	= , <>
Énumeration	= , <>

TABLE 4.2 – Les opérateurs pouvant être utilisés pour le pattern OCL P7

4.4.6 Restriction de la Valeur d'une Opération Héritée

Les opérations définies dans une super-classe décrivent un comportement commun aux sous-classes [53]. Nous avons remarqué l'existence de deux types d'opérations sur UML. La première concerne les opérations qui retournent des collections résultant de navigations. Par conséquent, elles ont des classes comme type de retour. Le second type concerne les opérations qui évaluent l'état du modèle et renvoient des résultats qui sont utilisées pour contraindre certaines classes spécifiques.

Parfois, les résultats qu'une opération renvoie doivent être spécifiques à certaines sous-classes selon la sémantique du domaine. Par exemple, si l'opération "oper()" dans la structure A.1 de la figure 4.8 définie dans la classe Class1 doit retourner false pour toutes les instances de la sous-classe SubC1, le concepteur est contraint d'exprimer cette information.

Cela peut être fait en utilisant des contraintes OCL telles que la contrainte C13 de la sous-figure A.2 (Fig. 4.8. Cette solution ne nécessite aucune modification de la structure du métamodèle. La deuxième solution consiste à redéfinir l'opération dans les sous-classes où elle est nécessaire, puis à apporter les modifications nécessaires pour la préciser dans la sous-classe, comme dans la structure B. Cette solution repose sur la modification du corps de l'opération redéfinie pour qu'elle renvoie toujours la valeur prévue. Dans certains scénarios, une contrainte OCL est obligatoire même après la redéfinition de l'opération, c'est le cas lorsque le concepteur ne veut pas modifier le corps de l'opération redéfinie. La troisième solution consiste à refactoriser la structure, par exemple, en déplaçant l'opération vers les sous-classes et en précisant le comportement des opérations dans leurs classes respectives comme dans la structure C. Cette solution peut être coûteuse surtout s'il y a beaucoup de sous-classes, et elle ne peut pas être appliquée tout le temps. Par exemple, cette solution peut être appliquée si la super-classe est abstraite et qu'elle n'a pas besoin de l'opération.

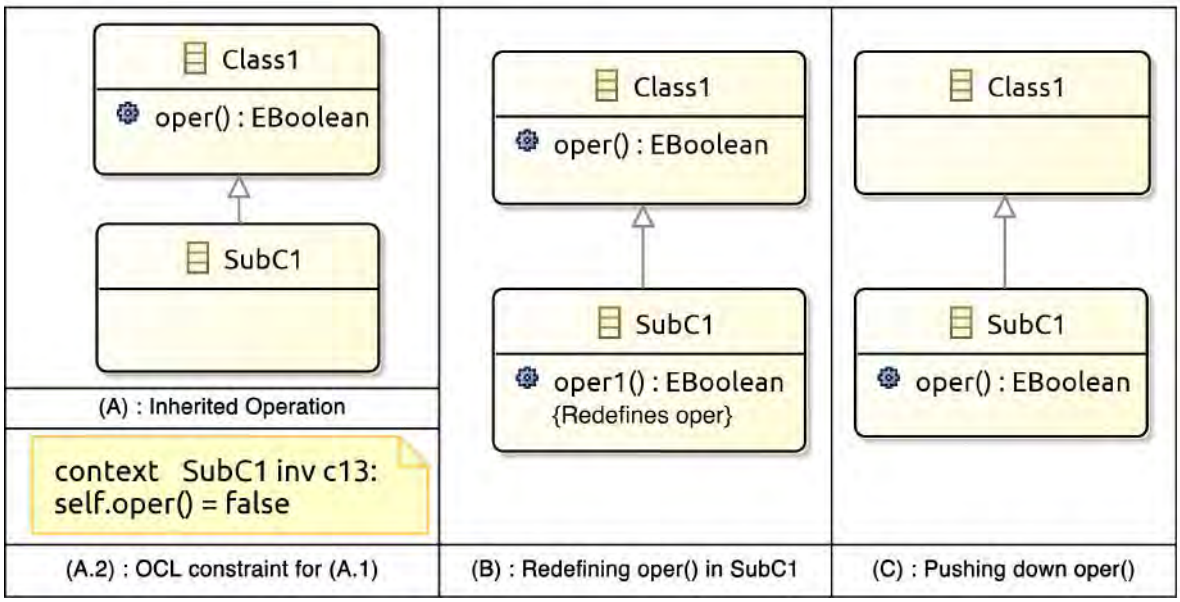

FIGURE 4.8 – Restriction de la Valeur d'une Opération Héritée

Pour le MIS *Restriction de la Valeur d'une Opération Héritée*, le pattern suivant décrit la structure de contrainte OCL qui peut être exprimée pour le préciser, où {Classe, navigation, Opération, Paramètre, Opérateur, Valeur} sont des paramètres qui doivent être remplacés par des valeurs réelles du métamodèle pour obtenir une contrainte OCL.

```
1 context Classe inv p9:
2 self.navigation.Opération(Paramètre) Opérateur Valeur
```

4.4.7 Relation entre Types

Comme l'illustre la structure A.1 de la figure 4.9, lorsqu'une association "asso" relie deux super-classes "Class1" et "Class2" au niveau du métamodèle, toutes les instances de "Class1", y compris ses sous-types, peuvent être associées à n'importe quelle instance de "Class2", y compris ses sous-types. Dans certains cas, certaines relations sont incorrectes par rapport au domaine. Pour éviter la création de métamodèles incorrects, trois solutions sont possibles. Les deux premières solutions nécessitent le refactoring de la structure du métamodèle, où il est possible de redéfinir l'association "asso" dans une ou plusieurs sous-classes comme dans la structure B, ou de supprimer l'association de la super-classe pour la faire descendre dans les sous-classes comme dans la figure C. Dans cette dernière solution, le métamodèle sera redondant ce qui le rend très complexe. La dernière solution consiste à utiliser des contraintes OCL pour spécifier le type d'instances de "Class1" qui peuvent être liées aux instances de "Class2". Dans notre exemple de figure 4.9, la contrainte C14 de la figure A.2 sera ajoutée à la structure A.2. Cette contrainte précise que les instances de SubC1 doivent être liées aux instances de SubC2 uniquement, et non à celles de Class2.

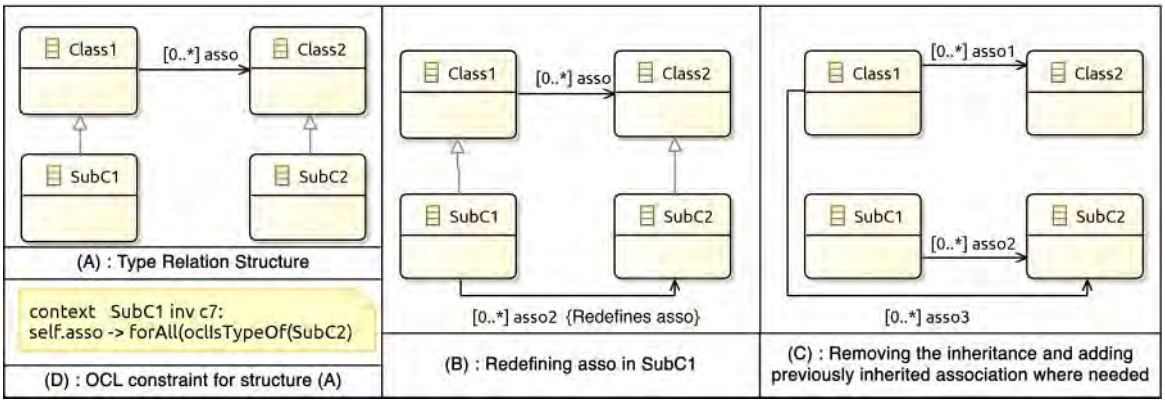

FIGURE 4.9 – Relation de Type

Pour le MIS *Relation de Types*, le pattern suivant décrit la structure de la contrainte OCL qui peut être exprimée pour le préciser, où {Classe1, navigation, Classe2} sont des

paramètres qui doivent être remplacés par des valeurs réelles du métamodèle afin de personnaliser le pattern et obtenir une contrainte OCL concrète propre au métamodèle.

```
1 context Classe1 inv p10:
2 self.navigation.OCLIsTypeOf(Classe2)
```

```
1 context Classe1 inv p11:
2 self.navigation.OCLIsKindOf(Classe2)
```

Dans certains cas, le MIS Relation de Types peut avoir une structure spécifique comme le montre la figure 4.10 : lorsqu'une super-classe "Class1" est liée à une autre classe "Class2" par une association "asso", et que l'une des classes contient un attribut d'énumération nommé "attrib", comme le montre la figure 4.10. Souvent, les liens entre les instances de ces deux classes sont spécifiques à la valeur de l'attribut d'énumération. Par exemple, les instances de SubC1 ne peuvent être associées aux instances de Class2 que si l'attribut "attribut" prend la valeur "SubC2". Ainsi, la contrainte C15 est spécifiée.

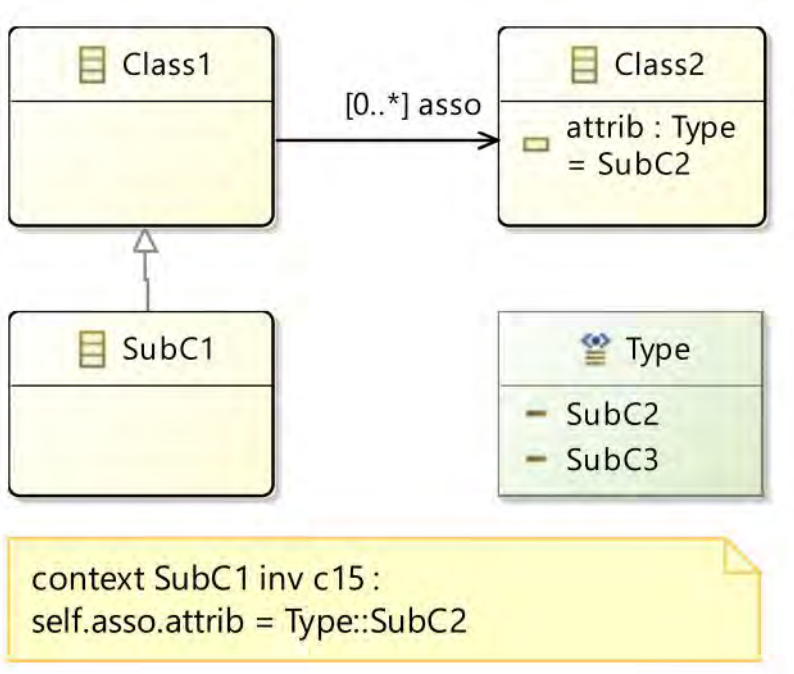

FIGURE 4.10 – Relation de Type avec Énumération

Dans ce cas précis, le modèle de contrainte OCL est le suivant :

```
1 context Class inv p12:
2 self.navigation.association = Énumeration::littéral
```

4.4.8 Restriction des Cycles

Dans les métamodèles, un cycle simple représente une association réflexive “asso” ou une opération “op()” comme le montre la structure A.1 de la figure 4.11. Un cycle complexe peut se manifester comme une succession d’associations et/ou d’opérations dans lesquelles une classe est atteignable à partir d’elle-même, il peut aussi s’agir d’une association bidirectionnelle comme la structure B.1 ou indirecte comme la structure C.1. Les cycles permettent de relier entre elles deux ou plusieurs instances d’une même classe. Souvent, ces structures peuvent entraîner de nombreux problèmes, à commencer par l’auto-association, mais aussi les configurations en diamant [85]. Les concepteurs de métamodèles doivent accorder une attention particulière aux cycles car ces structures sont très fréquentes. La façon la plus simple de résoudre un problème avec un cycle est de le faire par le biais de contraintes OCL afin de détailler toute la sémantique. Les contraintes de cycle sont appliquées pour :

1. restreindre la taille d’une collection d’objets obtenue à partir d’une navigation réflexive, ou spécifier si l’association réflexive doit être acceptée ou non ;
2. comparer la valeur d’attribut d’une instance avec la valeur d’attribut de sa ou ses instances liées par le biais de la navigation réflexive ;
3. spécifier la valeur d’un attribut dans les instances qui sont liées à “self” par la navigation réflexive ;
4. restreindre la valeur que retourne une opération dans les instances liées par la navigation réflexive.

Par exemple, pour la structure A.1 de la figure 4.11, les contraintes C15, C16 et C17 sont appliquées pour restreindre différents éléments. La première est écrite pour éviter l’auto-association. La deuxième contrainte (C16) empêche deux instances de “Class” d’être associées à travers le cycle si elles ont la même valeur “attrib”. La contrainte C17

est spécifiée pour empêcher une instance de "Class" de se retrouver elle-même comme résultat de l'opération op(). Pour le cycle bidirectionnel en B.1, la contrainte C18 est écrite pour spécifier l'auto-inclusion de l'instance dans la collection que nous obtenons si nous naviguons à travers le cycle bidirectionnel (de Class à Class à travers asso, puis l'inverse à travers assoOpposé). Pour les cycles indirects présentés dans la structure C.1, puisque les instances de la sous-classe "SubClass" sont des cas particuliers de "Class", cela permet les auto-associations. Par conséquent, la contrainte C19 de C.2 est spécifiée pour empêcher l'auto-association.

Étant donné un cycle, on peut poser les questions suivantes :

1. une instance peut-elle être associée à elle-même ?
2. les instances qui sont liées à "self" par la navigation réflexive ont-elles des spécificités qui doivent être rendu explicite ?

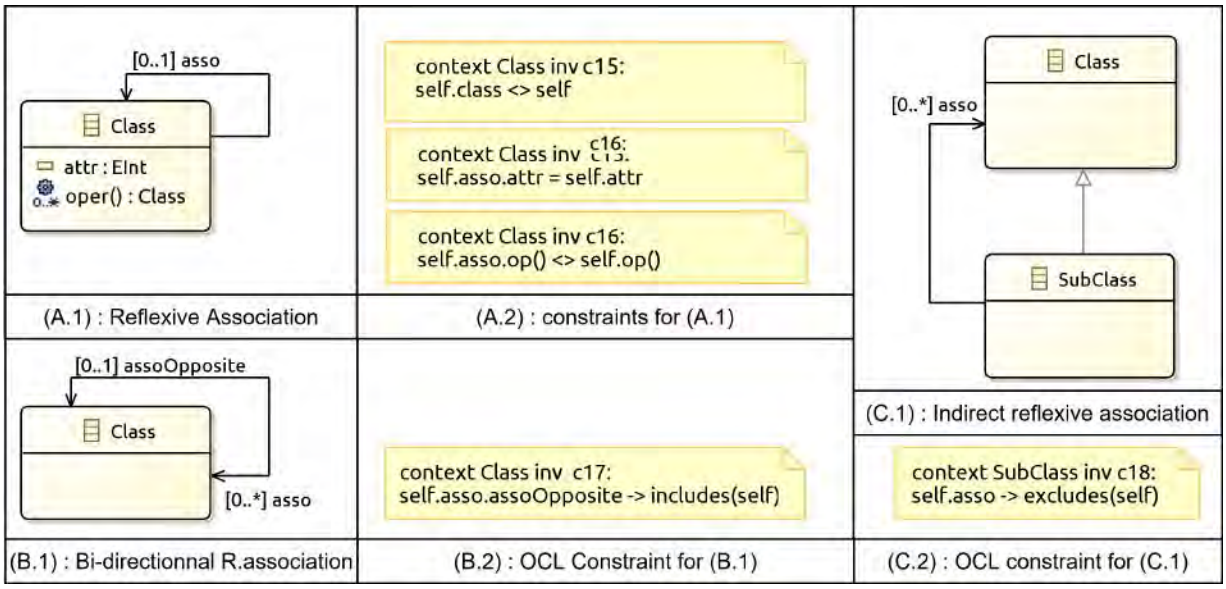

FIGURE 4.11 – Exemples de Cycles

Pour le MIS *Restriction des Cycles*, le pattern suivant décrit la structure de contrainte OCL qui peut être exprimée pour le préciser, où {Classe, assoCycle, Opérateur, attribut} sont des paramètres qui doivent être remplacés par des valeurs réelles pour obtenir une contrainte OCL.

Patterns	Opérateurs			
	Collections	Attribut Simple		
		Numérique	Booléen	Enumération
P13, P17	includes, excludes, includesAll, excludesAll	/		
P14	= , <>	/		
P15, P16	/	>, <, = , <>, <= , >=	= , <>, and , or xor , not	= , <>

TABLE 4.3 – Les opérateurs arithmétiques pour les patterns P13 à P17

```
1  context Classe inv p13:
2  self.navigation -> Opérateur (self)
```

```
1  context Classe inv p14:
2  self.navigation Opérateur self
```

```
1  context Classe inv p15:
2  self.assoCycle -> forAll(x : Classe | x.attribut opérateur Valeur)
```

```
1  context Classe inv p16:
2  self.navigation -> forAll(x : Classe | x.operation() opérator Valeur)
```

```
1  context Classe inv p17:
2  self -> closure(self.assoCycle) opérateur Valeur
```

Alors que les patterns p13, p14 et c17 sont destinés à gérer les auto-associations, les patterns p15 et p16 sont destinés à cibler d'autres éléments tels que les attributs ou les opérations. Le tableau 4.4.8 reprend tous les opérateurs qui peuvent être utilisés pour ces patterns.

4.4.9 Relation entre les Chemins

Dans un contexte de métamodélisation, un chemin représente une navigation entre deux classes. Il peut être composé d'une succession d'associations ou d'opérations avec des classes comme types de valeurs retournées. Si deux chemins peuvent être trouvés pour relier deux classes, il est possible qu'il existe un lien sémantique entre les deux collections résultant de chaque chemin.

Comme l'illustre la figure 4.12, en partant de la classe "Class1", on peut remarquer les deux chemins [Class1 -> asso12 -> Class2 -> asso23 -> Class3] et [Class1 -> asso13 ->

Class3] dans la structure A.1. Dans ce cas, les collections d'instances résultant des navigations sur les deux chemins sont liées. Ce lien peut être exprimé à l'aide de la contrainte C20 qui spécifie qu'une collection doit être incluse dans l'autre. Parfois, la contrainte est appliquée pour comparer la valeur des attributs des instances de la classe 3 des deux collections en utilisant C21 de A.2. Dans la structure B.1, le deuxième chemin est obtenu à partir de l'opération oper() qui retourne les instances de Class3. Pour cet exemple, la contrainte C22 de B.2 est spécifiée.

Souvent, les contraintes OCL sont utilisées pour expliciter le lien entre deux collections obtenues à partir de deux chemins qui relient les mêmes classes. Cela permet d'expliciter le lien caché entre les chemins, et donc d'éviter la création d'instances de modèle incorrectes. Les opérateurs (comme l'inclusion, l'exclusion, l'égalité et la différence) sont utilisés pour lier les deux collections, ou pour lier leurs tailles. La question liée à ce MIS est la suivante : *Existe-t-il un lien sémantique caché entre deux chemins distincts qui ont les mêmes classes sources et cibles ?*

Pour le MIS *Relation entre Chemins*, le pattern suivant décrit la structure de contrainte OCL qui peut être exprimée pour le préciser, où {`Classe, navigation1, Opérateur, navigation2`} sont des paramètres qui doivent être remplacés par des valeurs réelles du métamodèle pour obtenir une contrainte OCL concrète.

```
1  context Classe inv:
2  self.navigation1 Opérateur self.navigation2
```

Il est important de souligner que les refactorings proposés dans cette section ne sont que des exemples d'un ou plusieurs refactorings possibles, cette liste n'est pas exhaustive et d'autres possibilités de refactoring de MIS sont très envisageables. En ce qui concerne les patterns, nous avons extrait les patterns proposés à partir des contraintes étudiées, ce qui signifie également que l'ensemble des patrons que nous proposons n'est pas exhaustif pour couvrir tout le langage OCL. De plus, certains peuvent être exprimés différemment en utilisant des patterns différents que nous n'avons pas mentionnés.

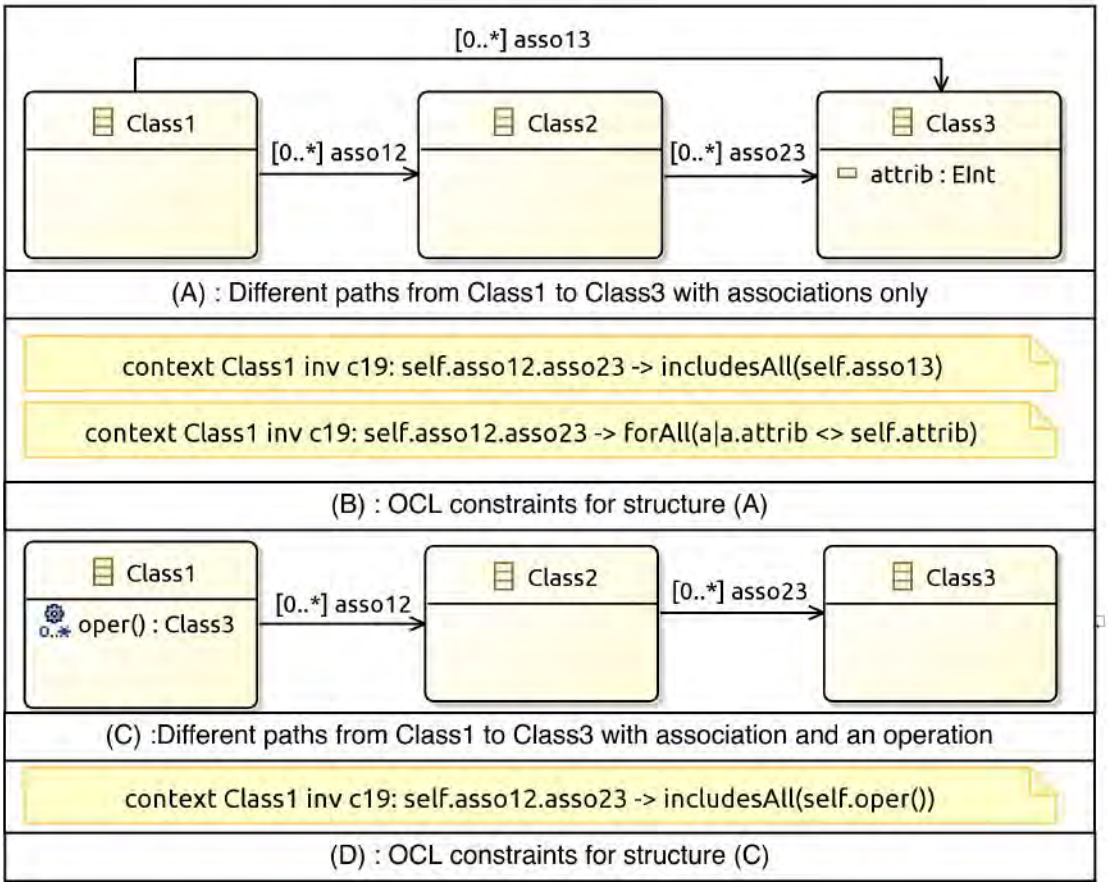

FIGURE 4.12 – Relation entre les Chemins

4.4.10 La distribution des contraintes liées aux MIS dans le métamodèle UML 2.5

Pour quantifier les occurrences de chaque MIS dans le métamodèle UML avec les contraintes OCL, le tableau 4.4 reprend le nombre de contraintes liées à chaque MIS.

Le nombre total de contraintes est de 450. 240 de ces contraintes (plus de la moitié de l'ensemble initial) ont été identifiées comme étant liées au MIS. Les 210 contraintes restantes ont été écartées car notre analyse de chacune d'entre elles nous a amené à conclure qu'elles concernent exclusivement le domaine métier.

Nous pouvons remarquer que le MIS avec le plus grand nombre de contraintes est le *Restriction de Chemins* avec 84 contraintes, où les contraintes sont exprimées pour préciser un lien implicite entre deux collections à partir de deux chemins de navigation. Cela

MIS	# Contraintes	Fréquence
Relation Types	31	13%
Restriction Valeur Attribut	7	3%
Restriction Littéraux Énumération	2	1%
Restriction Attribut Optionnel Hérité	3	1%
Restriction Multiplicité Association Héritée	60	25%
Restriction Valeur Attribut Hérité	12	5%
Restriction Valeur Opération Héritée	23	10%
Cycles	18	7%
Chemins	84	35%
Total	240	100%
proportion contraintes MIS	53%	/

TABLE 4.4 – La distribution des contraintes liées aux MIS dans le métamodèle UML 2.5

représente 35% du nombre total de contraintes liées au MIS. Le MIS présentant le plus petit nombre de contraintes est le *Restriction des Littéraux d'Énumération* avec seulement deux contraintes dans l'ensemble de la spécification du métamodèle UML. Cela est dû au fait que l'énumération est le concept le moins utilisé dans ce métamodèle. Si nous additionnons toutes les contraintes liées à l'héritage des six MIS suivants : 1) *Relation de types* ; 2) *Restriction d'attribut optionnel hérité* ; 3) *Restriction de la multiplicité d'association héritée* ; 4) *Restriction de valeur d'attribut hérité* ; 5) *Restriction de valeur d'opération héritée* ; nous pouvons voir que cela représente 129 contraintes ce qui représente 53% des contraintes liées aux MIS. Ce résultat montre que même si l'utilisation de l'héritage dans UML permet bien de factoriser plusieurs concepts, ainsi de rendre ce dernier réutilisable et plus simple à maintenir, nous constatons que l'utilisation de contraintes OCL pour affiner la sémantique du métamodèle augmente.

Les résultats de l'analyse des contraintes OCL et leurs structures MOF correspondantes confirment l'hypothèse H_0 qui stipule que l'étude empirique sur UML nous permet d'identifier des structures MIS. Il reste ensuite à valider les MIS sur d'autres métamodèles pour confirmer leur présence avec des contraintes OCL. L'hypothèse H_1 est ainsi rejetée.

4.5 En résumé

Dans ce chapitre, nous avons étudié le métamodèle UML avec ses contraintes OCL dans l'objectif d'identifier les structures de métamodèle qui nécessitent souvent l'ajout de

contraintes OCL pour préciser leur sémantique. L'étude a été menée sur UML en raison de sa taille importante et du nombre de contraintes OCL qu'il détient. Suite à cette étude, nous avons identifié neuf MIS qui constituent près de la moitié des structures contraintes d'UML. Chaque MIS a été étudié en profondeur pour comprendre le problème qu'il engendre et pourquoi il nécessite souvent des contraintes OCL. Ainsi, nous avons remarqué que le concept qui nécessite le plus de contraintes OCL est l'héritage, puisqu'il permet de regrouper des concepts redondant existant dans plusieurs classes en une seule super-classe qui sera héritée par la suite par toutes les classes nécessitant ces concepts.

Ensuite, pour chaque MIS, nous avons proposé un ou plusieurs patterns OCL pouvant lui être appliqué afin de préciser sa sémantique. Ces patterns résultent de l'étude empirique menée sur UML.

Dans le chapitre 6, nous évaluons quantitativement et qualitativement l'ensemble des MIS que nous avons identifié dans le métamodèle UML. L'évaluation quantitative consiste à étudier d'autres métamodèles pour déterminer si l'ensemble de MIS figure dans d'autres métamodèles avec des contraintes OCL, et ainsi prouver que ce n'est pas propre au métamodèle UML. Ensuite, l'évaluation qualitative permet d'évaluer l'utilité des MIS. Pour cela, nous avons réalisé une expérimentation contrôlée avec des étudiants afin de déterminer si la connaissance des MIS leur permettrait de trouver plus de contraintes OCL pour un même métamodèle.

Chapitre 5

Approche d'identification de MIS lors de la coévolution métamodèle/contrainte

Sommaire

5.1 Introduction

Pour de nombreuses raisons, les métamodèles sont sujets à des évolutions. Parfois, une évolution est motivée par un changement de syntaxe ou l'introduction de nouveaux éléments de modélisation (par exemple, l'évolution du métamodèle UML de la version 1.5 à 2.0). Elle peut également être motivée par le refactoring du métamodèle afin d'améliorer et de simplifier sa structure (par exemple, de UML 2.4 à UML 2.5). Cependant, pour maintenir la cohérence du métamodèle, il est important de faire évoluer en conséquence les contraintes qui lui sont associées. Ce problème est connu sous le nom de coévolution métamodèle/contrainte. De nombreux travaux dans la littérature ont proposé des méthodes de coévolution des métamodèles et de leurs contraintes [58, 93, 94, 99]. Ces travaux ont fourni des outils efficaces pour automatiser au maximum l'identification des contraintes impactées lors d'un changement dans le métamodèle. Parfois, ils proposent également une solution pour réécrire les contraintes impactées par le changement [100, 133]. Ainsi, ils offrent une aide précieuse aux développeurs lors de la modification de leurs métamodèles, en les déchargeant de la gestion des changements sur les contraintes OCL. Cependant, ces travaux ne concernent que les contraintes déjà définies. Il est vrai qu'une modification sur le métamodèle peut certainement induire une modification sur les contraintes existantes, mais parfois elle peut entraîner le besoin d'ajouter de nouvelles contraintes. En effet, si une partie d'un métamodèle est suffisamment précise sans contraintes, sa modification peut être moins précise et donc nécessitera l'ajout de contraintes. Les travaux existants ne proposent aucun moyen d'identifier de nouvelles contraintes OCL potentielles lorsque le métamodèle évolue.

Par conséquent, nous proposons une approche visant à compléter les approches existantes de coévolution de contraintes OCL. Notre approche permet de vérifier si les nouvelles structures apparues suite à l'évolution du métamodèle sont des MIS, et ainsi pourraient nécessiter des contraintes OCL. Pour ce faire, nous étudions l'impact de chaque opérateur d'évolution sur la nouvelle version du métamodèle, plus particulièrement sur l'apparition des MIS. La suite du chapitre est organisée de la façon suivante : dans la section 5.2, nous illustrons la problématique liée à l'impact de l'application d'un opérateur d'évolution sur les contraintes OCL. Nous montrons aussi pourquoi les approches de coévolution de contraintes OCL ne permettent pas forcément d'avoir systématiquement un métamodèle complet en termes de contraintes OCL.

Dans la section 5.3, nous détaillons l'approche portant sur l'identification de MIS pendant la phase de coévolution de contraintes OCL. Nous présentons d'abord la liste des opérateurs d'évolution qui sont présents dans la littérature. Ensuite, nous étudions chaque opérateur d'évolution de métamodèle dans l'objectif de caractériser ce que celui-ci peut engendrer comme MIS. Cette étude sert à proposer une table de correspondance entre les opérateurs d'évolution et les MIS avec la relation d'implication. Enfin, la section 5.5 conclut le chapitre.

5.2 Problématique et Motivation

Pour illustrer de manière simple le problème abordé dans notre travail, nous prenons un petit exemple du métamodèle UML StateMachine et de son évolution de la version 1.5 à la version 2.0 (Fig. 5.1).

Cette évolution a été réalisée en appliquant les opérations d'évolution suivantes :
— Déplacer la propriété *submachineState* depuis la sous-classe *SubmachineState* vers sa super-classe *CompositeState* avec l'opérateur "pullup property".
— Déplacer la propriété *submachineState* depuis la sous-classe *CompositeState* vers sa super-classe *State* avec l'opérateur "pullup property".

Au cours de cette évolution, l'association *submachineState* initialement typée par la classe *SubmachineState* a été déplacée vers la classe *CompositeState* puis vers la classe *State*. Selon les opérations d'évolution existantes dans la littérature, l'application de cette opération d'évolution nécessiterait une coévolution des contraintes relatives à l'association déplacée *submachineState*. En effet, toutes les contraintes de type :

OCLExpression. submachineState [...], tel que *OCLExpression* est de type *StateMachine*, doivent co-évoluer après l'application du premier opérateur de déplacement de propriété vers la super-classe comme pour la contrainte suivante :

OCLExpression. submachineState ->select(v | v. oclIsTypeOf(SubmachineState)) -> forAll(s | s[...])

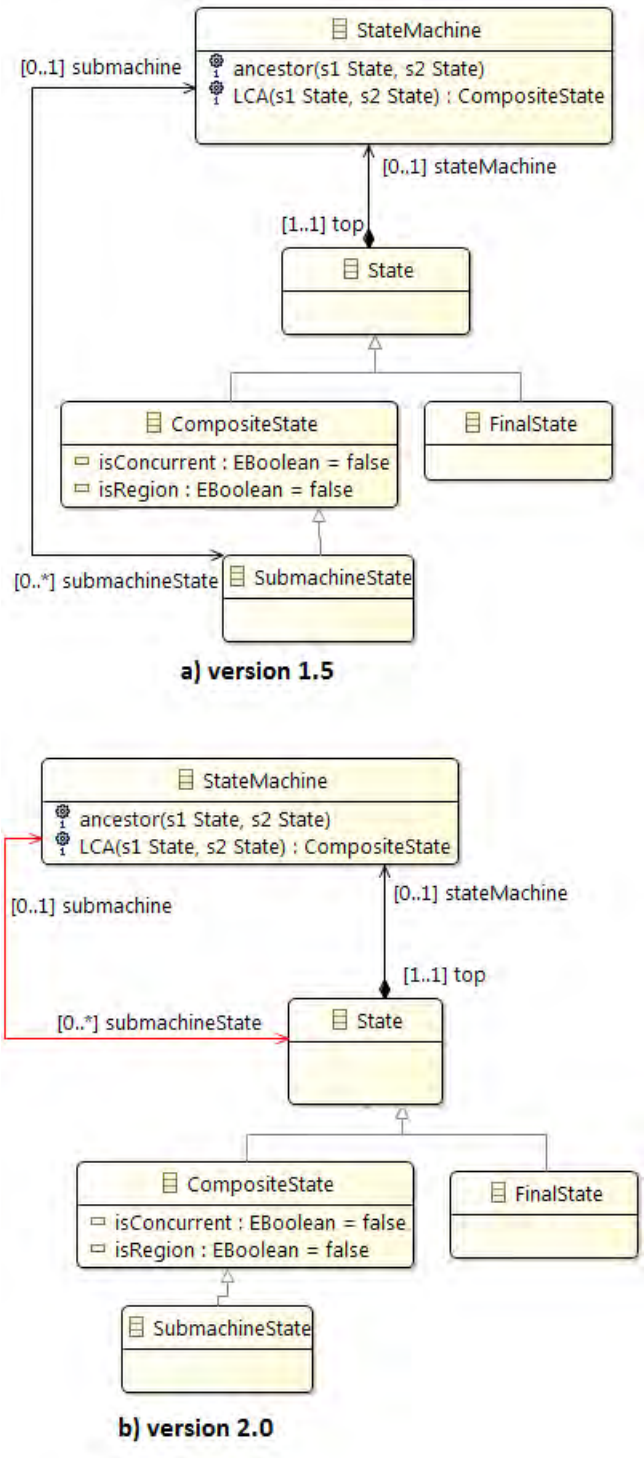

FIGURE 5.1 – Évolution du métamodèle StateMachine (extrait)

Ensuite, après l'application du second opérateur "PullUp", la contrainte co-évolue encore une fois comme suit :

OCLExpression. submachineState -> select(v | v.oclIsTypeOf(CompositeState)) -> select(v | v.oclIsTypeOf(SubmachineState)) -> forAll(s | s[...])

Cependant, suite à l'application de l'opérateur d'évolution "PullUp Property", un nouveau MIS apparaît dans la nouvelle version du métamodèle. En effet, toutes les sous-classes *CompositeState*, *FinalState*, *SubmachineState* de la classe *State*, qui contient maintenant l'association *submachine*, héritent cette référence avec la multiplicité [0..*]. En réalité, un état final *FinalState* ne peut pas avoir d'état sous-machine dans un diagramme d'état-transition. Par conséquent, bien que la nouvelle contrainte OCL soit conforme à la nouvelle version du métamodèle, le MIS engendré par l'évolution du métamodèle rend la version 2.0 incomplète en termes de contraintes OCL. Pour remédier à ça, la contrainte suivante a été ajoutée, celle-ci est appliquée à la sous-classe *FinalState* pour restreindre la multiplicité de l'association héritée *submachine* :

context FinalState
inv inv : self.submachine->isEmpty()

Dans la même optique, nous avons étudié dans ce chapitre les différents MIS qui peuvent nécessiter l'ajout de nouvelles contraintes ou la suppression de celles existantes, après l'application d'une opération d'évolution. Nous présentons notre approche dans la section suivante.

5.3 Approche

5.3.1 Opérateurs d'évolution de métamodèles

Pendant la phase d'évolution, un métamodèle subit plusieurs changements. Des exemples de besoins conduisant à des changements sur le métamodèle sont l'ajout/la suppression de concepts ou le refactoring du métamodèle afin de simplifier sa compréhension et de le rendre facile à maintenir. Suite à l'application de ces changements, les artefacts qui étaient conformes aux métamodèles ne le deviennent plus, d'où le besoin de les coévoluer

pour qu'elles soient à nouveau conformes à la nouvelle version du métamodèle.

Dans la littérature, le problème de coévolution concernant les métamodèles a été abordé par de nombreuses études visant à identifier les changements qui sont habituellement appliqués. Ainsi, un ensemble de 16 opérateurs d'évolution a été introduit par Wachsmuth [58] pour la coévolution des métamodèles et des modèles. Par ailleurs, Marković et Barr [99] ont proposé 15 règles pour faire co-évoluer les diagrammes de classes et les contraintes OCL. Ceux-ci ont été complétés par [100] avec 7 opérations pour co-évoluer les métamodèles et les contraintes OCL. Le tableau 5.1 extrait de [101] résume tous les opérateurs d'évolution de métamodèles EMOF identifiés qui aident à résoudre les problèmes de coévolution.

Les opérateurs d'évolution sont regroupés en quatre catégories :

1. Ceux permettant d'ajouter ou de supprimer de nouveaux éléments de modélisation tels que : classe, association, attribut, opération, généralisation, packages, etc. ;
2. Ceux qui aident à la manipulation des propriétés telles que : Déplacer Propriété, Déplacer vers le bas Propriété, etc ;
3. La troisième catégorie regroupe les opérateurs qui manipulent les hiérarchies (Relations d'héritage) ;
4. Le dernier groupe représente tous les opérateurs complexes qui ont été proposés pour remanier les structures basées sur des patterns.

Nous nous basons sur les différents opérateurs d'évolution du tableau 5.1 ainsi que sur les MIS que nous avons regroupé dans le tableau 5.2 pour la suite de notre approche.

5.3.2 Identification de MIS pendant la coévolution métamodèle/contrainte

Initialement, comme illustré dans la figure 5.2, nous avons identifié 30 opérateurs dans la littérature (A) qui font évoluer le métamodèle de l'ancienne version (B) à la nouvelle version (C). Pour garantir que les contraintes OCL existantes (E) sont conformes à la nouvelle version du métamodèle, un ensemble d'opérateurs de coévolution de contraintes (D) est appliqué à ces contraintes pour les faire co-évoluer afin qu'elles soient conformes à la nouvelle version du métamodèle (F). Comme l'ensemble des contraintes (F) est incomplet et ne couvre donc pas tous les concepts de la nouvelle version du métamodèle, un

TABLE 5.1 – Aperçu des opérateurs d'évolution des métamodèles EMOF

Type	Nom de l'Opérateur
Ajouter / Supprimer Élément	Ajouter Classe
	Supprimer Classe
	Ajouter Package
	Supprimer Package
	Ajouter TypeDonnées, TypePrimitif, Énumeration
	Supprimer TypeDonnées, TypePrimitif, Énumeration
	Ajouter LittéralÉnumération
	Retirer LittéralÉnumération
	Ajouter Attribut
	Supprimer Attribut
	Ajouter Association
	Supprimer Association
	Ajouter Opération (Classe / TypeDonnées)
	Supprimer Opération (Classe / TypeDonnées)
	Introduire Généralisation
	Supprimer Généralisation
Manipulation de propriétés	Déplacer Propriété
	Déplacer un attribut vers les sous-classes
	Déplacer une association vers les sous-classes
	Regrouper un attribut commun dans la super-classe
	Regrouper une association commune dans la super-classe
	Restreindre une association unidirectionnelle
	Généraliser une association unidirectionnelle
Patterns de Refactoring	Extraire Classe
	Fusionner Classe
	Extraire super-classe
	Aplatir Hiérarchie
	Association vers Classe
	Généralisation vers Composition
	Introduire Pattern Composite

ensemble de MIS pouvant découler des opérateurs d'évolution (H) est automatiquement recherché (G). Cet ensemble d'instances MIS est ensuite analysé par le concepteur du métamodèle (I) afin de supprimer toutes les instances qui ne nécessitent pas de contraintes OCL. Après ce tri, l'ensemble d'instances de MIS résultant (J) ne comprend que les instances qui nécessitent des contraintes OCL. À cette fin, nous avons développé un outil qui propose des contraintes OCL (K) pour chaque instance de MIS. Les contraintes proposées

TABLE 5.2 – Aperçu de l'ensemble des MIS

MIS
Restriction de la valeur d'un attribut
Restriction des littéraux d'une énumération
Relation de types
Relation de types avec énumération
Restriction de la valeur d'un attribut hérité
Restriction d'un attribut optionnel
Restriction de la multiplicité d'une association héritée
Restriction de la valeur d'une opération héritée
Cycles
Chemins

sont des instances de modèles OCL qui sont fréquemment utilisés pour préciser les MIS. Ensuite, l'expert du domaine validera l'ensemble des contraintes (L) en se basant sur sa connaissance du domaine. Par conséquent, l'ensemble de contraintes OCL résultant (M) comprend toutes les nouvelles contraintes OCL qui sont appliquées pour préciser les nouveaux éléments du métamodèle résultant de l'évolution du métamodèle. L'union de cet ensemble de contraintes (M) et de l'ensemble de contraintes résultant de la coévolution des contraintes OCL (F) de l'ancien métamodèle à l'aide des opérateurs de coévolution donne lieu à un ensemble complet de contraintes OCL (N) conformes à la nouvelle version du métamodèle. Il est à noter que le nouvel ensemble de contraintes OCL résultats de notre approche ne comprend que des contraintes OCL liées aux MIS, il est possible qu'il y ait d'autres contraintes nécessaires à la précision du métamodèle, qu'il faudra ajouter. Pour le moment, aucune approche n'est capable de proposer de telles contraintes efficacement.

Afin de tirer profit des connaissances acquises avec les MIS et de compléter la coévolution OCL, nous avons effectué une analyse sur chaque opérateur de coévolution du métamodèle. Cette analyse a eu pour but d'identifier tous les MIS qui peuvent apparaître suite à l'application d'un opérateur d'évolution de métamodèle. Pour ce faire, pour chaque opérateur d'évolution, nous avons pris un ensemble de structures et l'avons appliqué, puis nous avons recherché les instances de MIS pour savoir quel MIS peut résulter de quel opérateur d'évolution de métamodèle. Les résultats sont discutés dans la section suivante.

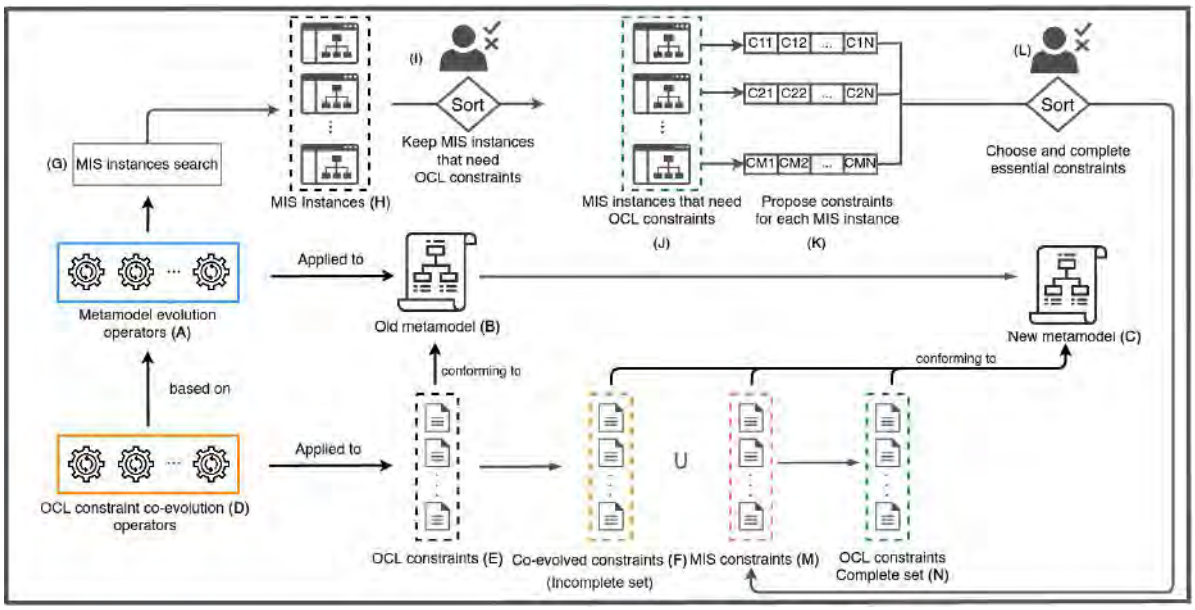

FIGURE 5.2 – L'approche de coévolution de contraintes OCL

5.4 Étude des opérateurs d'évolution de métamodèle pour expliciter le lien de causalité avec les MIS

Dans cette partie, nous détaillons les résultats de l'étude des opérateurs d'évolution présentés dans le tableau 5.4.3.7. Ainsi, nous expliquons pour chaque opérateur d'évolution tous les MIS qui peuvent être générés suite à son application. Compte tenu du nombre conséquent de situations possibles, nous illustrons l'impact de l'application des opérateurs seulement sur l'opérateur "Ajouter association".

5.4.1 Les opérateurs d'ajout/Suppression d'éléments

5.4.1.1 Ajouter Classe

L'opérateur "Ajouter Classe" permet de créer une classe vide dans un package existant. La nouvelle classe étant vide et n'étant pas liée avec aucun autre élément du métamodèle, cet opérateur d'évolution n'engendre aucun MIS. Aussi, l'application de cet opérateur d'évolution sur le métamodèle n'a aucun effet sur les contraintes OCL existantes.

5.4.1.2 Supprimer Classe

L'opération "Supprimer Classe" permet de retirer une classe vide d'un package. Cette opération n'implique pas de MIS du fait qu'elle soit vide. Cependant, les contraintes OCL ayant comme contexte cette classe doivent être supprimées.

5.4.1.3 Ajouter Package

L'opération "Ajouter Package" permet d'ajouter un package vide au métamodèle. Celle-ci n'implique aucun MIS et n'a pas d'impact sur les contraintes OCL existantes.

5.4.1.4 Supprimer Package

L'opération "Supprimer Package" permet de supprimer un package vide du métamodèle. Cette dernière n'implique aucun MIS et n'a pas d'impact sur les contraintes OCL existantes.

5.4.1.5 Ajouter TypeDonnées, TypePrimitif, Énumeration

L'opération "Ajouter TypeDonnées, TypePrimitif, Énumération" permet d'ajouter un type de données, un type primitif, ou une énumération. L'ajout d'un type de données ou bien d'un type primitif n'implique aucun MIS. Cependant, l'ajout d'une énumération peut impliquer le MIS "Restriction des Littéraux d'une énumération" dans le cas où l'énumération n'est pas vide. Si elle est vide, celui-ci n'engendrera donc pas de MIS.

5.4.1.6 Supprimer TypeDonnées, TypePrimitif, Énumeration

L'opération "Supprimer TypeDonnées, TypePrimitif, Énumération" supprime un Type de données, un type primitif, ou une énumération vide. Cet opérateur n'engendre pas de MIS et n'a pas d'impact sur les contraintes OCL existantes.

5.4.1.7 Ajouter LittéralÉnumération

L'opération "Ajouter LittéralÉnumération" permet d'ajouter un littéral dans une énumération. L'application de cet opérateur peut engendrer les MIS suivants :

— Restriction des Littéraux d'une Énumération : dans le cas où l'énumération dans laquelle le littéral a été ajouté était déjà composé d'au moins deux littéraux

— Relation de Types avec Énumération : dans le cas où la classe contenant un attribut de l'énumération dans laquelle le littéral a été ajouté contient une association entrante

Cet opérateur n'a pas d'impact sur les contraintes OCL existantes.

5.4.1.8 Retirer LittéralÉnumération

L'opération "Retirer LittéralÉnumération" permet de retirer un littéral d'une énumération. L'application de cet opérateur d'évolution n'implique aucun MIS, mais impacte les contraintes existantes dans lesquelles le littéral est identifié.

5.4.1.9 Ajouter Attribut

L'opération "Ajouter Attribut" ajoute un attribut d'un type de données existant dans le métamodèle dans une classe existante. Les types de données que l'attribut peut prendre comme type sont les classes, les types de données. L'application de cet opérateur d'évolution pourrait engendrer l'un des MIS suivants :

— Restriction de la valeur d'un attribut : étant donné qu'il n'existe aucun moyen de restreindre la valeur d'un attribut par le biais de diagrammes, les contraintes OCL constituent le seul moyen pour le faire. Ainsi, n'importe quel attribut ajouté pourrait nécessiter une contrainte OCL pour définir avec précision son domaine de définition.

— Attribut optionnel hérité : ce MIS se manifeste si l'attribut est ajouté dans une classe ayant au moins une sous-classe. Aussi, l'attribut pourrait être contraint que si sa borne supérieure est strictement supérieure à sa borne inférieure.

— Restriction de la valeur d'un attribut hérité : ce MIS se manifeste dans le cas où l'attribut est ajouté dans une classe ayant une sous-classe ou plus.

L'application de cet opérateur n'a pas d'impact sur les contraintes OCL déjà existantes dans le métamodèle.

5.4.1.10 Supprimer Attribut

L'opération "Supprimer Attribut" permet de supprimer un attribut d'une classe. L'application de cet opérateur n'engendre aucun MIS mais pourrait avoir un impact sur les contraintes OCL existantes si elles naviguent vers cet attribut. Dans ce cas, ces contraintes

doivent être supprimées ou modifiées pour retirer les parties qui référencent l'attribut supprimé.

5.4.1.11 Ajouter Association

L'opération "Ajouter Association" permet d'ajouter une association entre deux classes existantes dans le même package. Cet opérateur peut impliquer tous les MIS dans lesquels les associations sont présentes, tel que :

— Relation de Types : ce MIS apparaît si au moins une des classes source ou cible de l'association contient au moins une sous-classe 5.3.

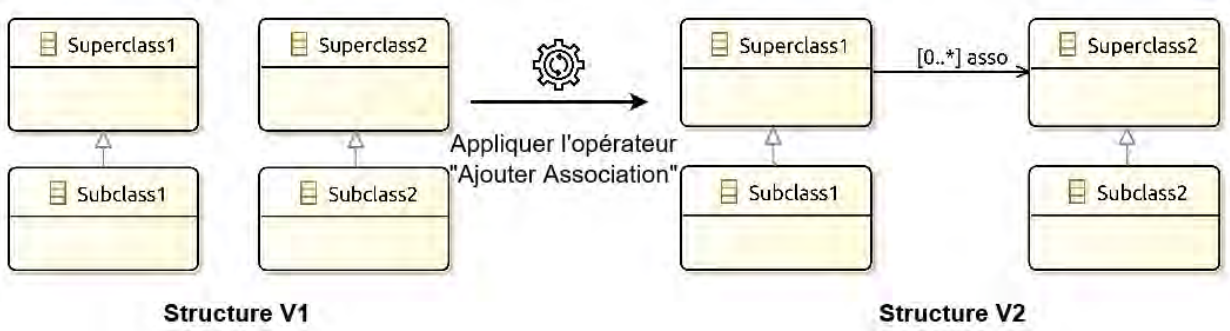

FIGURE 5.3 – Le MIS Relation de Types engendré suite à l'ajout d'une association

La contrainte OCL nécessaire pour préciser ce MIS est la suivante :

context Subclass1 **inv** : **self**.asso.oclIsTypeOf(Subclass2)

— Restriction de la multiplicité de l'association : ce MIS se manifeste dans le cas où la la borne supérieure de la multiplicité de l'association est strictement supérieure à sa borne inférieure. Dans ce cas, une ou plusieurs sous-classes pourraient nécessiter une multiplicité plus fine, ce qui nécessiterait une contrainte OCL pour le faire 5.4.

Pour préciser ce MIS, il est nécessaire d'ajouter la contrainte OCL suivante :

```
1  context Subclass1 inv : self.asso-> isEmpty()
```

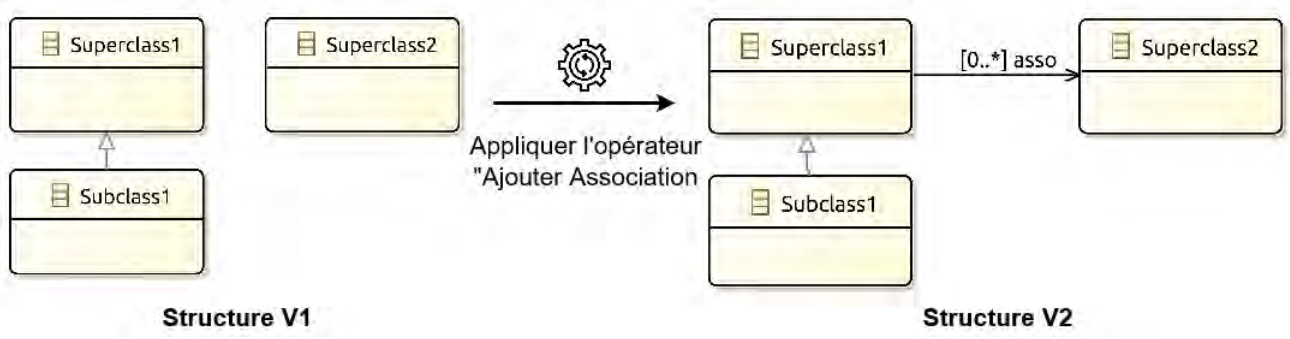

FIGURE 5.4 – Le MIS Restriction de la multiplicité de l'association engendré suite à l'ajout d'une association

— Cycles : ce MIS apparaît dans le cas où l'association est réflexive. Aussi, si l'association complète une navigation qui relierait une classe à elle-même 5.5.

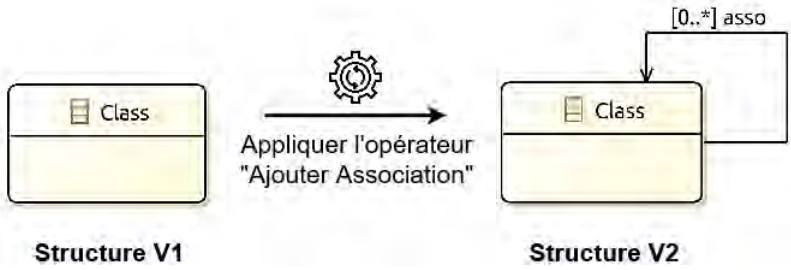

FIGURE 5.5 – Le MIS Cycles engendré suite à l'ajout d'une association

La contrainte OCL nécessaire pour préciser ce MIS est la suivante :

```
1  context Class1 inv : self.asso -> excludes(self)
```

— Chemins : ceci pourrait survenir dans le cas où il existerait une ou plusieurs autre(s) association(s) qui relie(nt) les mêmes classes. Aussi, si l'association ajoutée compléterait une navigation qui relie deux classes, et qu'il existerait une autre navigation qui ferait la même chose 5.6.

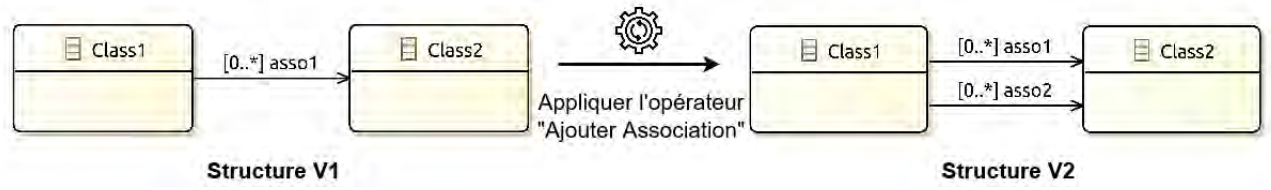

FIGURE 5.6 – Le MIS Chemins engendré suite à l'ajout d'une association

Pour préciser ce MIS, il est nécessaire d'ajouter la contrainte OCL suivante :

```
1  context Class1 inv : self.asso1 -> includesAll(self.asso2)
```

L'utilisation de cet opérateur d'évolution n'impacte pas les contraintes OCL déjà existantes.

5.4.1.12 Supprimer Association

L'opération "Supprimer Association" permet de supprimer une association qui existe entre deux classes du même package. Cette opération n'implique aucun MIS, mais impacte les contraintes OCL dans lesquelles l'association est identifiée. Ainsi, la suppression des parties qui identifient l'association ou de toute la contrainte est nécessaire.

5.4.1.13 Ajouter Opération (Classe / TypeDonnées)

L'opération "Ajouter Opération" permet d'ajouter une opération dans une classe. L'opération peut avoir une classe ou bien un type de données comme retour. L'application de cet opérateur d'évolution sur le métamodèle pourrait impliquer les MIS suivants :

- Restriction de la valeur d'une opération héritée : ce MIS se manifeste dans le cas où l'opération est ajoutée à une classe ayant au moins une sous-classe.
- Cycle : ce MIS peut se manifester seulement si l'opération ajoutée dans la classe A retourne des instances de la même classe A dans laquelle elle se trouve. Dans ce cas, souvent des contraintes OCL sont nécessaires pour gérer certaines situations conflictuelles.
- Chemins : ce MIS se manifeste quand le type de retour de l'opération ajoutée dans la classe A est la classe B. S'il existe une autre navigation entre la classe A et B, une contrainte OCL est parfois nécessaire pour rendre explicite le lien entre les deux collections d'éléments que les deux navigations retournent.

L'utilisation de cet opérateur d'évolution n'impacte pas les contraintes OCL déjà existantes.

5.4.1.14 Supprimer Opération (Classe / TypeDonnées)

L'opération "Supprimer Opération" permet de supprimer une opération existante d'une classe. L'application de cette évolution n'implique pas de MIS mais a un impact sur les contraintes OCL. Ainsi, toutes les contraintes qui font appel à l'opération doivent être modifiées ou supprimées.

5.4.1.15 Introduire Généralisation

L'opération "Introduire Généralisation" permet de créer un lien d'héritage entre deux classes (création d'un lien super-classe sous-classe). Ceci peut engendrer différents MIS comme suit :

- Restriction de la valeur d'un attribut hérité : si la super-classe contient un attribut.
- Attribut Optionnel hérité : si la super-classe contient un attribut optionnel [0..1].
- Restriction de littéraux d'une énumération : Si la super-classe contient un attribut, et que celui-ci soit de type énumération
- Restriction de la valeur d'une opération héritée : si la super-classe contient une opération
- Relation de Types : Si la super-classe est liée via une association à une autre classe
- Restriction de la multiplicité d'une association héritée : si la super-classe contient une association avec la borne supérieure de la multiplicité qui soit strictement supérieure à la borne inférieure
- Cycles : s'il existe une association qui va de la sous-classe vers la super-classe, ceci donnerait lieu à un cycle indirect.
- Chemins : si la généralisation permet de créer une deuxième navigation entre deux classes différentes

L'utilisation de cet opérateur d'évolution n'impacte pas les contraintes OCL déjà existantes.

5.4.1.16 Supprimer Généralisation

L'opération "Supprimer Généralisation" permet de supprimer un lien d'héritage entre deux classes. Cet opérateur d'évolution n'implique aucun MIS, cependant, il a un impact

sur les contraintes OCL qui naviguent entre les deux classes, mais aussi sur toutes les contraintes OCL qui sont appliquées dans la sous-classe pour contraindre des éléments de la super-classe.

5.4.2 Les opérateurs de Manipulation de Propriétés

5.4.2.1 Déplacer Propriété

L'opération "Déplacer Propriété" permet de déplacer un attribut ou bien une association entre deux classes ayant une association un-à-un [1..1] entre elles. La multiplicité 1..1 est indispensable afin que les valeurs de la propriété d'une instance après le déplacement peuvent sans ambiguïté être associés à cette même instance.

L'application de cet opérateur d'évolution peut faire apparaître plusieurs MIS. Dans le cas où la propriété est un attribut, les MIS qui peuvent être engendrés sont les suivants :

- Attribut Optionnel hérité : si l'attribut est déplacé vers une classe possédant au moins une sous-classe, et que la multiplicité de l'attribut est de [0..1].
- Restriction de la valeur d'un attribut hérité : si l'attribut est déplacé vers une classe possédant au moins une sous-classe.
- Restriction de la valeur d'un attribut : ce MIS est valable pour n'importe quel attribut.

Si la propriété est une association, les MIS qui peuvent être impliqués sont les suivants :

- Relation de types : si l'association déplacée lie deux classes, et que la classe ciblée par l'association a au moins une sous-classe.
- Restriction de la multiplicité d'une association héritée : si l'association est déplacée vers une classe ayant au moins une sous-classe.
- Cycles : si l'association déplacée relie une sous-classe à sa super-classe, cela implique un cycle indirect. Aussi, si l'association suite à son déplacement complète une navigation qui relie une classe à elle même
- Chemins : s'il existe déjà une navigation qui lie deux classes A et B, et que le déplacement de l'association compléterait une autre navigation qui lierait les mêmes classes A et B.

En plus de l'impact de cet opérateur d'évolution sur l'apparition des MIS, celui-ci impacte aussi l'ensemble des contraintes OCL existantes. En effet, toutes les contraintes OCL qui servaient à contraindre l'attribut ou l'association doivent être obligatoirement modifiées au supprimées. En plus, si la propriété est une association, et que celle-ci est naviguée dans une contrainte OCL, la contrainte doit être modifiée ou supprimée.

5.4.2.2 Déplacer un attribut vers les sous-classes

L'opération "Déplacer un attribut vers les sous-classes" permet de déplacer un attribut de la super-classe vers toutes ses sous-classes directes. L'application de cet opérateur d'évolution sur le métamodèle implique les MIS suivants :

- Restriction de la valeur d'un attribut : qui reste le MIS le plus général lorsqu'il s'agit des MIS relatifs aux attributs.
- Restriction de la valeur d'un attribut hérité : si au moins une des sous-classes auxquelles l'attribut a été déplacé a des sous-classes.
- Restriction de la multiplicité d'un attribut optionnel hérité : si au moins une des sous-classes auxquelles l'attribut a été déplacé a des sous-classes, et que la multiplicité de l'attribut est de [0..1].
- Restriction des littéraux d'une énumération : dans le cas d'un attribut de type énumération
- Relation de types avec énumération : si une des sous-classes ayant reçu l'attribut a une association rentrante.

Cet opérateur impacte aussi les contraintes existantes. Par exemple, s'il existait une contrainte OCL appliquée sur une sous-classe A pour restreindre la multiplicité de l'attribut [0..1] dans la super-classe B, cette contrainte n'a plus lieu d'exister du fait que le déplacement de cet attribut vers toutes les sous-classes permet de choisir directement la multiplicité adéquate de l'attribut dans la sous-classe A.

5.4.2.3 Déplacer une association vers les sous-classes

L'opération "Déplacer une association vers les sous-classes" permet de déplacer une association d'une super-classe vers toutes ses sous-classes directes. Ainsi, l'association change de classe source (la classe source est à la base la super-classe, et devient suite à l'évolution une des sous-classes) mais garde la même classe cible.

L'application de cet opérateur d'évolution pourrait implique une multitude de MIS comme suit :

- Restriction de la multiplicité d'une association héritée : dans le cas où une des sous-classes ayant eu l'association aurait au moins une sous-classe. Dans ce cas, il se pourrait qu'une contrainte OCL soit nécessaire pour contraindre l'association dans la sous-classe.
- Cycle : si l'association avant l'application de l'opérateur d'évolution ciblait déjà une sous-classe, celle-ci suite à son déplacement vers toutes les sous-classes deviendrait réflexive sur la sous-classe qu'elle ciblait avant. Aussi, si l'association suite à son déplacement vers une des sous-classes compléterait une navigation qui permet de relier une classe à elle-même.
- Chemin : s'il existe déjà une navigation qui relie deux classes A et B, et que le déplacement de l'association vers une des sous-classes compléterait une autre navigation qui relierait les mêmes classes A et B.

L'opération a aussi un impact sur les contraintes OCL déjà présentes. Par exemple, s'il existe une contrainte OCL appliquée sur une des sous-classes pour restreindre la multiplicité de l'association qui est dans la super-classe, celle-ci ne sert plus quand l'association est déplacée vers les sous-classes. Ainsi, les contraintes OCL qui ciblaient l'attribut avant l'évolution du métamodèle sont à modifier ou supprimer.

5.4.2.4 Regrouper un attribut commun dans la super-classe

L'opération "Regrouper un attribut commun dans la super-classe" permet de généraliser un attribut qui existe dans toutes les sous-classes directes d'une super-classe A pour le mettre dans la super-classe. Cette opération peut introduire plusieurs MIS comme suit :

- Restriction de la valeur d'un attribut
- Restriction de la valeur d'un attribut hérité
- Restriction de la multiplicité d'un attribut hérité : si l'attribut n'a pas exactement la même multiplicité dans toutes les sous-classes avant l'évolution, l'attribut suite à l'évolution aura une multiplicité qui englobera toutes celles des autres sous-classes avant l'évolution. Par conséquent, des contraintes OCL seront nécessaires dans une ou plusieurs sous-classes pour préciser les multiplicités souhaitées dans les sous-classes.

— Restriction des littéraux d'une énumération : si l'attribut est de type énumération, il se peut qu'il y ait un littéral qui ne doit pas être pris comme valeur dans une des sous-classes.

L'application de cet opérateur peut avoir un impact sur les contraintes OCL existantes.

5.4.2.5 Regrouper une association commune dans la super-classe

L'opération "Regrouper une association commune dans la super-classe" permet de supprimer une association commune entre toutes les sous-classes et de la déplacer vers la super-classe avec une multiplicité qui regroupe toutes les multiplicités que l'association avait dans chaque sous-classe. Aussi, l'association évoluée garde la même classe d'arrivée mais change de classe de départ pour devenir la super-classe. L'application de cet opérateur d'évolution peut engendrer les MIS suivants :

— Relation de types : ce MIS se manifeste suite à l'application de cet opérateur du fait que l'association lie une classe ayant des sous-classes avec une autre.
— Restriction de la multiplicité d'une association héritée : si l'une des sous-classes a besoin d'une multiplicité différente que celle de la super-classe.
— Cycles : avant l'évolution, si l'association relie les super-classes avec la sous-classe, l'évolution rendrait l'association réflexive. Aussi, si l'association après l'évolution compléterait une navigation qui permettrait à une classe d'être reliée à elle-même.
— Chemins : s'il existe déjà une navigation qui relie deux classes A et B, et que le déplacement de l'association vers la super-classe compléterait une autre navigation qui relierait les mêmes classes A et B.

L'application de cet opérateur d'évolution a un impact sur les contraintes qui contraignent ou naviguent à travers l'association. Ces dernières doivent être contraintes.

5.4.2.6 Restreindre une association unidirectionnelle

Cette opération permet de déplacer une association d'une super-classe vers une sous-classe, ce qui permet de restreindre le type de l'association. Cette évolution peut engendrer un ensemble de MIS comme suit :

— Restriction de la multiplicité d'une association héritée : dans le cas où la sous-classe ayant eu l'association aurait au moins une sous-classe.

— Cycle : le même cas que celui de l'opérateur d'évolution "Déplacer une association vers les sous-classes".
— Chemin : le même cas que celui de l'opérateur d'évolution "Déplacer une association vers les sous-classes".

Tout comme pour l'opérateur "Déplacer une association vers les sous-classes", cet opérateur a un impact sur l'ensemble des contraintes déjà présentes sur le métamodèle. Par exemple, une contrainte OCL appliquée sur la sous-classe pour restreindre la multiplicité de l'association n'aura plus de sens après le déplacement de l'association vers la sous-classe.

5.4.2.7 Généraliser une association unidirectionnelle

Cette opération déplace une association unidirectionnelle d'une sous-classe vers sa super-classe. Son application peut avoir comme conséquence l'apparition de certains MIS selon la situation comme suit :

— Restriction de la multiplicité d'une association : après la généralisation de l'association en la déplaçant vers les super-classes, il se peut que sa multiplicité ne correspond pas exactement à celle recherchée dans l'une des sous-classes. Ainsi, une contrainte OCL est nécessaire pour préciser la multiplicité de l'association héritée dans cette sous-classe.
— Relation de types : pareil que pour l'opérateur "Regrouper une association commune dans la super-classe".
— Cycles : pareil que pour l'opérateur "Regrouper une association commune dans la super-classe".
— Chemins : pareil que pour l'opérateur "Regrouper une association commune dans la super-classe".

D'un autre côté, l'application de cet opérateur d'évolution impacte l'ensemble des contraintes OCL existant. En effet, toutes les contraintes OCL dans lesquelles l'association est identifiée doivent être modifiées ou bien retirées du métamodèle.

5.4.3 Les opérateurs de refactoring

5.4.3.1 Extraire Classe

Cet opérateur d'évolution permet d'extraire une classe B d'une classe A qui existe déjà dans le package. Une association est créée entre les deux classes A et B avec une multiplicité 1..1. La classe A qui est ainsi référée comme classe "source" tandis que la nouvelle classe B est référée comme classe "émergente". L'utilisation de cet opérateur d'évolution peut provoquer quelques MIS comme suit :

— Relation de Types : se manifeste si la classe source a des sous-classes.
— Relation de Types avec énumération : se manifeste si la classe source contient un attribut de type énumération.

5.4.3.2 Fusionner Classe

L'opérateur permet de fusionner une classe A avec une classe B si les deux classes sont connectées à travers une association un-à-un 1..1. Étant donné que la classe est vide, aucun MIS n'est entraîné suite à l'application de cet opérateur. Cependant, s'il existe une contrainte OCL qui parcourt ou cible l'association qui relie les deux classes "cible" et "source" ou qui contraint le nombre d'instances de la classe source, celle-ci devra être modifiée ou supprimée.

5.4.3.3 Extraire Super-classe

L'opérateur "extraire super-classe" permet de créer une nouvelle super-classe à partir d'un ensemble de sous-classes. Toutes les propriétés et opérations communes aux sous-classes sont déplacées vers la super-classe puis sont supprimées des sous-classes. Ainsi, la super-classe regroupera toutes les propriétés et opérations communes. L'application de cet opérateur d'évolution peut entraîner une multitude de MIS comme suit :

— Restriction de la valeur d'un attribut : valable pour tous les attributs regroupés dans la super-classe.
— Restriction des littéraux d'une énumération : si la super-classe contient un attribut de type énumération
— Relation de types : si la super-classe a une association
— Relation de types avec énumération : si la super-classe a un attribut de type énumération et une association entrante, ou bien une association qui cible une autre classe ayant un attribut d'énumération.

— Restriction de la valeur d'un attribut hérité : si la super-classe a un attribut
— Restriction d'un attribut optionnel : s'il existe un attribut dans la super-classe avec la multiplicité de 0..1
— Restriction de la multiplicité d'une association héritée : s'il existe une association dans la super-classe avec une borne supérieure de multiplicité strictement supérieure à la borne inférieure.
— Restriction de la valeur d'une opération : s'il existe une opération dans la super-classe qui retourne au moins un élément.
— Cycles : s'il existe une association réflexive dans la super-classe
— Chemins : s'il existe au moins deux navigations de la super-classe qui ciblent la même classe

5.4.3.4 Aplatir Hiérarchie

Cette opération permet de transférer le contenu d'une super-classe dans une hiérarchie dans toutes ses sous-classes, puis de supprimer la super-classe. L'application de cet opérateur d'évolution peut entraîner les MIS suivants :

— Restriction de la valeur d'un attribut : si la super-classe contenait un attribut.
— Restriction des littéraux d'une énumération : si la super-classe contenait un attribut de type énumération, et que le métamodèle contient un autre attribut typé avec la même énumération
— Relation de types : si la super-classe avant l'évolution avait une association, et si au moins une des sous-classes est héritée.
— Relation de types avec énumération : si une sous-classe après l'évolution a un attribut de type énumération et une association.
— Restriction de la valeur d'un attribut hérité : si un attribut est présent dans l'une des sous-classes, et que cette même sous-classe est héritée.
— Restriction d'un attribut optionnel : si au moins une des sous-classes contient un attribut optionnel (avec la multiplicité 0..1), et que cette même sous-classe est héritée.
— Restriction de la multiplicité d'une association héritée : si une association a été déplacée depuis la super-classe vers les sous-classes, et qu'une des sous-classes est héritée.
— Restriction de la valeur d'une opération héritée : si une opération a été déplacée depuis la super-classe vers les sous-classes, et qu'une des sous-classes est héritée.

— Cycles : si la super-classe avait une association réflexive.
— Chemins : si après le déplacement de toutes les propriétés et opérations de la super-classe vers toutes les sous-classes, une des sous-classes a deux navigations qui ciblent la même classe.

5.4.3.5 Association vers Classe

Cette opération permet de remplacer une association existante entre deux classes A et B par une classe C. Ensuite, deux nouvelles associations reliant la classe A avec C puis C avec B sont créées. L'application de cet opérateur d'évolution peut entraîner les MIS suivants :

— Relation de types : si la classe A ou B ont des sous-classes.
— Relation de types avec énumération : Si la classe A ou B ont des sous-classes et un attribut de type énumération
— Cycles : si les classes A et B sont reliées entre elles avec une association, il est possible qu'une des deux classes A ou B soit atteignable d'elle même avec une suite de navigations (en partant de la classe A par exemple, A-B-C-A)
— Chemins : s'il existe une navigation entre A et B, et que la deuxième navigation serait celle en passant par la classe C.

5.4.3.6 Généralisation vers Composition

L'opération permet de transformer un lien d'héritage en une association, cette association prend comme borne supérieure 1 du côté de l'ancienne classe fille. Du côté de l'ancienne classe mère, la borne supérieure de l'association est de 1 si la classe était abstraite, la borne inférieure est fixée à 0 ou bien 1. L'application de cet opérateur d'évolution peut engendrer les MIS suivants :

— Relation de types : si une des deux classes a toujours des sous-classes.
— Relation de types avec énumération : si l'une des deux classes a un attribut de type énumération
— Cycles
— Chemins : s'il existe deux chemins distincts à partir de l'une des deux classes A ou B vers une autre classe C.

5.4.3.7 Introduire Pattern Composite

Cet opérateur est appliqué à une association père-fils réflexive existante sur une classe A. Par conséquent, la classe A devient abstraite et deux sous-classes jouant le rôle de classe "fille" et "composite" sont dérivés de la classe A. Ensuite, un lien de composition est créé entre la classe "composite" et la classe A. L'application de cet opérateur d'évolution peut entraîner l'apparition de certains MIS comme suit :

— Relation de types : le lien de composition qui relie la classe "composite" avec la classe A forme le MIS "Relation de types".
— Relation de types avec énumération : si la classe A contient un attribut de type énumération.

Pour récapituler, nous avons regroupé la cartographie causale entre les opérateurs d'évolution et les MIS dans le tableau 5.4.3.7. Pour être plus précis, nous avons divisé certains opérateurs tels que "Déplacer Propriété vers la super-classe" en "Déplacer Association vers le haut", et "Déplacer Attribut Vers Super-classe" pour préciser les MIS résultants. Le tableau contient trois types d'opérateurs d'évolution du métamodèle (I). Chaque type englobe un ensemble d'opérateurs (II) que nous avons pris dans la littérature. Pour chaque opérateur, nous indiquons si son application peut entraîner un changement ayant un impact sur les contraintes OCL, qui nécessitent la coévolution d'au moins une contrainte (III). Par exemple, l'ajout d'une classe n'affecte aucune contrainte OCL, mais la suppression d'un attribut nécessite la suppression/réfactorisation de toutes les contraintes qui étaient référencées. Ensuite, pour chaque opérateur, nous indiquons s'il peut engendrer un ou plusieurs MIS ou non, et si c'est le cas, nous listons tous les MIS qui peuvent être déclenchés par un opérateur d'évolution (IV). Enfin, nous donnons quelques conditions, principalement liées à la structure du métamodèle, rendant possible l'apparition d'un MIS (V). Par exemple, l'application de l'opérateur "add association" ne peut déclencher le MIS "Cycle" que si l'association est réflexive.

Type	Nom d'opérateur	OCL coevol	Peut engendrer MIS
Ajout / suppression d'Élément	Ajouter Classe	non	non
	Supprimer Classe	oui	non
	Ajouter Package	non	non
	Supprimer Package	non	non
	Ajouter TypeDonnées, TypePrimitif, Énumération	non	non
	Supprimer TypeDonnées, TypePrimitif, Énumération	oui	non
	Ajouter LittéralÉnumération	non	Restriction des Littéraux d'une Énumération
			Relation de Types avec Énumération
	Retirer LittéralÉnumération	oui	non
	Ajouter Attribut	non	Restriction de la valeur d'un attribut
			Attribut optionnel hérité
			Restriction de la valeur d'un attribut hérité
	Supprimer Attribut	oui	non
	Ajouter Association	non	Relation de Types
			Restriction de la multiplicité de l'association
			Cycles
			Chemins
	Supprimer Association	oui	non
	Ajouter Opération (Classe / TypeDonnées)	non	Restriction de la valeur d'une opération héritée
			Cycle
			Chemins

	Supprimer Opération (Classe / TypeDonnées)	oui	non
	Introduire Généralisation	oui	Restriction de la valeur d'un attribut hérité
			Attribut Optionnel hérité
			Restriction de littéraux d'une énumération
			Restriction de la valeur d'une opération héritée
			Relation de Types
			Restriction de la multiplicité d'une association héritée
			Cycles
			Chemins
	Supprimer Généralisation	oui	non
Manipulation de propriétés	Déplacer Propriété	oui	Attribut Optionnel hérité
			Restriction de la valeur d'un attribut hérité
			Restriction de la valeur d'un attribut
			Relation de types
			Restriction de la multiplicité d'une association héritée
			Cycles
			Chemins
	Déplacer un attribut vers les sous-classes	oui	Restriction de la valeur d'un attribut
			Restriction de la valeur d'un attribut hérité
			Restriction de la multiplicité d'un attribut optionnel hérité

		Restriction des littéraux d'une énumération
		Relation de types avec énumération
Déplacer une association vers les sous-classes	oui	Restriction de la multiplicité d'une association héritée
		Cycles
		Chemin
Regrouper un attribut commun dans la super-classe	oui	Restriction de la valeur d'un attribut
		Restriction de la valeur d'un attribut hérité
		Restriction de la multiplicité d'un attribut hérité
		Restriction des littéraux d'une énumération
Regrouper une association commune dans la super-classe	oui	Relation de types
		Restriction de la multiplicité d'une association héritée
		Cycles
		Chemins
Restreindre une association unidirectionnelle	oui	Restriction de la multiplicité d'une association
		Cycles
		Chemins
Généraliser une association unidirectionnelle	oui	Restriction de la multiplicité d'une association
		Relation de types
		Cycles

			Chemins
Patterns de Refactoring	Extraire classe	non	Relation de types
			Relation de types avec énumération
	Fusionner Classe	oui	non
	Extraire Super-classe	oui	Restriction de la valeur d'un attribut
			Restriction des littéraux d'une énumération
			Relation de types
			Relation de types avec énumération
			Restriction de la valeur d'un attribut hérité
			Restriction de la multiplicité d'une association héritée
			Restriction de la valeur d'une opération
			Cycles
			Chemins
	Aplatir Hiérarchie	oui	Restriction de la valeur d'un attribut
			Restriction des littéraux d'une énumération
			Relation de types
			Relation de types avec énumération
			Restriction de la valeur d'un attribut hérité
			Restriction d'un attribut optionnel
			Restriction de la valeur d'une opération héritée
			Cycles
			Chemins
	Association vers Classe	oui	Relation de types
			Relation de types avec énumération

			Cycles
			Chemins
	Généralisation vers Composition	oui	Relation de types
			Relation de types avec énumération
			Cycles
			Chemins
	Introduire Pattern Composite	oui	Relation de types
			Relation de types avec énumération

5.5 En résumé

Dans ce chapitre, nous avons présenté notre approche visant à identifier les MIS pendant la phase de coévolution de contraintes OCL. L'objectif de cette identification est de signaler les MIS qui sont apparus suite à l'évolution du métamodèle.

Pour le faire, nous avons d'abord regroupé les opérateurs d'évolution de métamodèle qui existent dans la littérature. Suite à cela, nous avons étudié en profondeur chacun des opérateurs d'évolution existants dans la littérature dans le but d'identifier les MIS qui peuvent apparaître suite à son application. Dans le prochain chapitre, nous évaluons les performances de notre approche sur un cas d'étude du métamodèle UML.

Chapitre 6

Evaluation

Sommaire

6.1 Introduction

Dans ce chapitre, nous présentons l'évaluation de nos contributions selon le plan suivant :

1. Évaluation quantitative des MIS identifiés dans la première contribution. Cela consiste à étudier neuf autres métamodèles avec leur contraintes OCL dans l'objectif de conclure si les MIS identifiés dans UML existent dans ces métamodèles avec des contraintes OCL. Ceci vise à prouver empiriquement que les MIS sont des structures génériques qui peuvent être retrouvés quel que soit le métamodèle.

2. Évaluation qualitative des MIS qui consiste à enquêter si la connaissance des MIS permettait à des étudiants qui ne maîtrisent pas beaucoup le langage OCL et MOF à trouver plus de structures nécessitant des contraintes OCL, et ainsi écrire plus de contraintes OCL. Pour cela, nous avons réalisé une expérimentation contrôlée avec deux groupes d'étudiants du même niveau, un groupe avec un cours sur les MIS tandis que l'autre n'a pas été initié aux MIS, puis de comparer leurs performances quand il s'agit de compléter un métamodèle avec des contraintes OCL. Cette évaluation a été réalisée avec les métamodèles State-Machine et Activités d'UML.

3. Évaluation des performances de la recherche automatique de MIS sur le nombre de contraintes OCL. Cette validation vise à tester si une solution basée sur la recherche automatique exhaustive de MIS dans un métamodèle permettrait d'assister un concepteur pendant la phase de spécification de contraintes OCL de trouver plus de contraintes. Aussi, nous souhaitons savoir si la taille du métamodèle pourrait avoir un grand impact sur l'efficacité de cette assistance.

4. Évaluation des performances de notre approche portant sur l'identification des MIS pendant la coévolution de contraintes OCL suite à l'évolution du métamodèle. L'idée est de mesurer la valeur ajoutée de notre approche à côté d'une approche d'évolution existante sur un cas d'étude réel. À cet effet, nous prenons les deux versions 1.5 et 2.0 du métamodèle "State Machine" d'UML pour mesurer les performances de l'approche.

6.2 Évaluation quantitative des MIS

Dans cette section, nous évaluons les MIS identifiés suite à l'étude empirique portée sur le métamodèle UML 2.5. L'objectif est de reproduire les mêmes étapes que nous avons effectuées pour étudier le métamodèle UML, mais cette fois sur neuf autres métamodèles.

6.2.1 Question de recherche et Approche de Validation

QR *Est-ce que les MIS que nous avons identifiés dans le métamodèle UML sont présents dans d'autres métamodèles avec des contraintes OCL, et est-ce que l'ensemble des MIS est complet ?*
Cette question vise à vérifier que tous les MIS identifiés dans la première contribution sont présents dans des métamodèles autres qu'UML. Cela signifie que les MIS identifiés sont indépendants du contexte du métamodèle étudié, et qu'ils peuvent donc être généralisés. De plus, nous devons vérifier si les autres métamodèles utilisés dans cette question de recherche contiennent ou non des structures contraintes récurrentes que nous n'avons pas trouvées précédemment. Ceci a pour but de vérifier si la première liste de MIS est complète.

Pour cette question de recherche, nous avons formulé les deux hypothèses suivantes :

1. Hypothèse 1 : l'ensemble des MIS identifiées à partir de l'étude du métamodèle UML est lié au domaine d'application du métamodèle et n'apparaîtront pas avec des contraintes OCL dans d'autres métamodèles d'autres domaines. Ceci se traduit par les hypothèses nulles et alternatives formelles décrites ci-dessous et qui sont vérifiées en utilisant le nombre de MIS que nous avons trouvé dans le nouvel ensemble de données avec des contraintes OCL ($\#CMIS$). Nous tenons à préciser que cette métrique ne retourne pas le nombre d'instances de MIS trouvées avec des contraintes, mais le nombre de MIS que nous avons précédemment identifiés en UML, et trouvés dans le nouveau jeu de données au moins une fois avec des contraintes OCL. Le résultat de cette métrique est au plus égal au nombre de MIS ($\#KnownMIS$).
   — $H_{1.1}$: tous les MIS identifiés précédemment n'ont pas de contraintes dans ce dataset (CMIS = 0).
   — $H_{1.2}$: certains ou tous les MIS ont été retrouvés dans ce jeu de données avec des contraintes OCL ($0 < \#CMIS <= \#KnownMIS$).

2. Hypothèse 2 : l'ensemble des MIS identifiés précédemment est complet
   — $H_{2.1}$: aucun nouveau MIS n'a été identifié dans les métamodèles du nouveau dataset ($\#NewMIS = 0$)
   — $H_{2.2}$: il existe au moins un nouveau MIS que nous avons identifié dans le nouveau dataset et qui n'existe pas dans UML avec des contraintes OCL ($\#NewMIS > 0$)

6.2.2 Méthode et données

Pour répondre à la première question de recherche, nous avons reproduit le même processus utilisé pour identifier les MIS dans le chapitre 4.3, afin de vérifier leur occurrence dans d'autres métamodèles et de vérifier s'il existe d'autres MIS qui n'ont pas été identifiés précédemment.

Le tableau 6.1 détaille la liste des métamodèles qui constituent le nouveau dataset, avec le nombre de contraintes OCL pour chacun.

Metamodèle	Nb de contraintes
SysML 1.5	21
ODM 1.1	19
CWM 1.1	96
Diagram Definition	16
SAD3	8
CPFSTool	27
ER2RE	59
RBAC	34
SAM	82
Total	362

TABLE 6.1 – Le dataset pour l'évaluation quantitative des MIS

Ces métamodèles ont été pris dans l'ensemble des métamodèles utilisés par [19]. Certains des métamodèles de cet ensemble ont été écartés car ils ont déjà été utilisés dans l'étape d'identification de notre étude, comme les métamodèles constituant UML.

6.2.3 Résultats et Discussion

Après avoir caractérisé un ensemble de MIS du métamodèle UML, nous souhaitons vérifier s'ils sont également présents dans les autres, et surtout accompagnés avec des

contraintes OCL pour compléter leur sémantique. Pour ce faire, nous avons évalué la liste des MIS d'un point de vue quantitatif en comptant les occurrences de chaque contrainte liée aux MIS.

Comme le montre la Fig. 6.1, nous pouvons constater que la proportion de contraintes liées aux MIS change d'un métamodèle à l'autre. Par exemple, dans le métamodèle Diagram Definition, nous pouvons voir que les contraintes liées aux MIS dépassent 90%, alors que dans le métamodèle SAM, elles ne représentent que 17% du nombre total de contraintes OCL spécifiées. En moyenne, la proportion de contraintes liées au MIS avoisine les 48%.

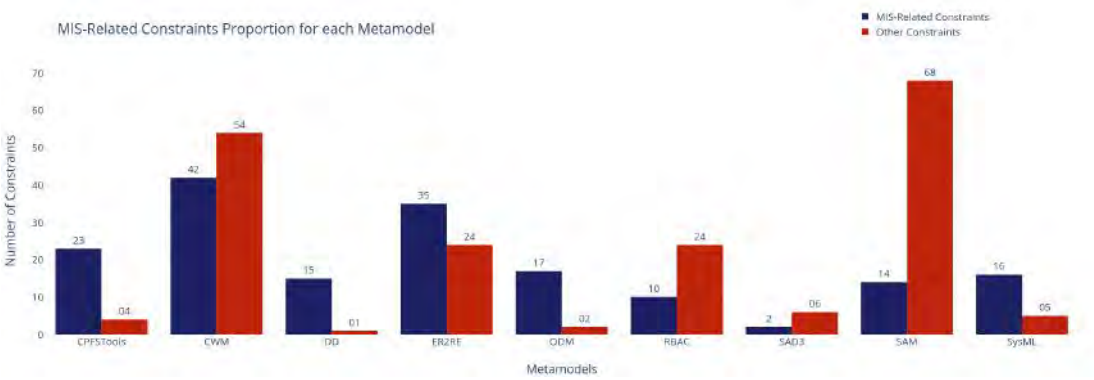

FIGURE 6.1 – La proportion des contraintes liées aux MIS pour chaque métamodèle

La figure 6.2 représente le nombre de contraintes par MIS. Nous pouvons observer que le MIS le plus contraint est *Chemins*, qui représente plus du tiers (34%) des contraintes liées aux MIS (174 au total). En effet, l'absence de contraintes qui explicitent le lien entre certains ensembles d'instances obtenues par la navigation à travers des chemins peut conduire à des incohérences majeures qui ne devraient pas apparaître dans un artefact bien formé. Pour cette raison, l'explicitation de tous les liens entre les concepts liés doit être effectuée par les concepteurs de métamodèles par le biais de contraintes OCL. En revanche, aucune contrainte liée au MIS *Restriction des littéraux d'énumération* n'a été trouvée. La raison de l'absence de ces contraintes est dû au fait que les métamodèles constituant le dataset n'utilisent pas le concept d'énumération (excepté CWM), ce qui justifie l'absence des contraintes d'énumération. Pour les autres MIS, trois sont contraints par plus de 20 et moins de 45 contraintes, et quatre MIS sont contraints par 1 à 7 contraintes.

Si nous additionnons les MIS liés à l'héritage (*Relation de type, Attribut optionnel*

hérité, Restriction de valeur d'opération héritée, Restriction de valeur d'attribut héritée, Restriction de multiplicité d'association héritée), nous trouvons 45 contraintes, ce qui correspond à 26% des contraintes liées aux MIS. Ceci montre que la présence de l'héritage dans un métamodèle permet une grande flexibilité de conception et de réutilisation des descriptions, mais implique également un grand nombre de contraintes à définir pour ce métamodèle afin de préciser sa sémantique.

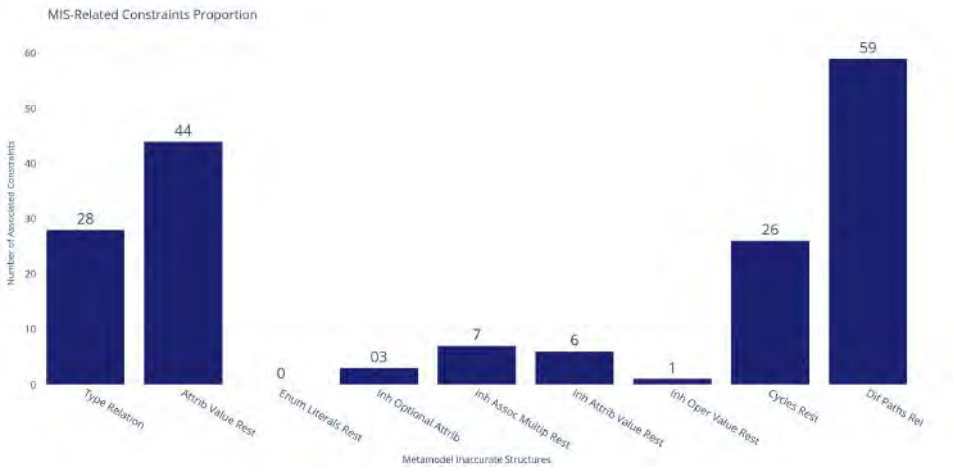

FIGURE 6.2 – Number of MIS-related constraints in all MMs

Le tableau 6.2 présente les occurrences des contraintes liées au MIS pour chaque métamodèle. Nous pouvons remarquer que certains MIS sont présents dans plusieurs métamodèles raffinés avec des contraintes OCL. Par exemple, la restriction de valeur d'attribut MIS et les chemins sont accompagnés de contraintes OCL dans six des neuf métamodèles. Inversement, le MIS sur les énumérations n'a pas été trouvé avec des contraintes OCL (sauf dans UML).

Compte tenu de l'impossibilité de préciser de manière schématique les ensembles de valeurs ou les domaines de définition que l'attribut peut prendre pour respecter la sémantique du domaine, les contraintes OCL qui sont utilisées pour restreindre la valeur des attributs sont très fréquentes.

En ce qui concerne les *Cycles*, ces structures ont déjà été signalées dans de nombreux travaux ([54, 85]) principalement pour les incohérences qu'ils entraînent. Décider si un cycle donné entraîne des incohérences sémantiques reste une tâche obligatoire pour un

concepteur de métamodèle.

Étant donné que sur les 9 MIS identifiés dans le métamodèle UML, 8 ont été retrouvés sur au moins un métamodèle du dataset de validation avec des contraintes OCL ($\#CMIS = 8$), l'hypothèse 1.1 est rejetée. Par conséquent, l'hypothèse 1.2 est validée. De plus, nous n'avons pas identifié d'autres MIS sur le nouveau dataset ($\#NewMIS = 0$). Ainsi, nous rejetons l'hypothèse 2.2 et nous validons l'hypothèse 2.1.

	SysML	ODM	CWM	DD	SAD3	CPFST	ER2RE	RBAC	SAM	**Total**
Relation Type	1	14	13	0	0	0	0	0	0	28
Rest Val Attr	0	0	1	15	2	6	13	0	7	44
Rest Lit Énum	0	0	0	0	0	0	0	0	0	0
Rest Attr Opt Hérit	0	3	0	0	0	0	0	0	0	3
Rest Multip Asso Hérit	0	0	6	0	0	1	0	0	0	7
Rest Val Attr Hérit	1	0	1	0	0	4	0	0	0	6
Rest Val Oper Hérit	0	0	0	0	0	0	0	0	1	1
Cycles	6	0	17	0	0	0	0	3	0	26
Chemins	8	0	4	0	0	12	22	7	6	59
Total	16	17	42	15	2	23	35	10	14	**174**
CMIS proportion	76%	89%	43%	93%	25%	85%	59%	29%	17%	48%

TABLE 6.2 – Les occurrences de MIS associés avec des contraintes OCL

6.2.4 Menaces à la validité

Comme pour toute expérimentation, certaines menaces peuvent affecter la validité de nos résultats. Pour les menaces de construction, nous estimons que l'analyse manuelle des métamodèles et des contraintes OCL pour identifier les MIS est une tâche subjective qui peut être biaisée, et qui dépend de l'expérience des personnes qui procèdent à l'analyse. Par exemple, un concepteur ayant déjà travaillé à compléter la sémantique du métamodèle avec des contraintes OCL sera plus à même d'identifier des MIS en comparaison à un concepteur inexpérimenté ou un étudiant. Pour éviter cela, l'étude a été menée par cinq individus, chacun ayant une connaissance différente des métamodèles et un niveau de compétence différent. Aussi, chacun des quatre individus a mené l'analyse individuellement afin de créer son ensemble de MIS, puis, nous avons procédé à un vote pour décider de chaque MIS pour obtenir à la fin la liste des MIS présentée.

Pour la validité interne, les données pourraient affecter la validité des résultats. En effet, nous nous sommes appuyés sur des métamodèles qui contiennent des contraintes OCL. Pour certains métamodèles, l'ensemble des contraintes OCL pourrait ne pas être complet. Pour éviter cela, nous avons choisi des métamodèles provenant de sources et de domaines différents. Par exemple, les métamodèles créés par l'OMG ont plus de chances d'être complets puisque ce groupe est celui qui a créé à la fois MOF et OCL et que, de plus, ces concepteurs de métamodèles ont une grande expérience dans la création de modèles [138]. Cependant, nous ne sommes pas en mesure d'affirmer que l'ensemble de contraintes OCL était complet dans tous les métamodèles.

La validité externe concerne la généralisation de nos résultats à d'autres métamodèles. Dans nos expériences, nous avons étudié 812 contraintes OCL provenant de 10 métamodèles (le métamodèle UML inclus). Nous ne savons pas jusqu'à quelle mesure cela pourrait être généralisé aux contraintes OCL qui définissent les langages d'autres domaines. Nous avons essayé de nous munir de métamodèles qui proviennent de diverses sources (académiques, groupes de travail et industrie) afin que l'échantillon de métamodèles choisi soit assez représentatif de la réalité.

6.2.5 Conclusion

Dans cette section, nous avons évalué quantitativement la liste de MIS que nous avons identifié en étudiant le métamodèle UML avec ses contraintes OCL dans le chapitre 4.3. L'objectif de cette évaluation est de déterminer si les MIS présents dans UML sont présents dans d'autres métamodèles avec des contraintes OCL, ce qui prouve que leur présence dans d'autres métamodèle pourrait nécessiter l'ajout de contraintes OCL. Dans la section suivante, nous évaluons la liste des MIS d'un point de vue qualitatif.

6.3 Évaluation qualitative des MIS

6.3.1 Question de recherche et Approche de Validation

QR *Est-ce que la connaissance des MIS permet au concepteur de concevoir des métamodèles plus précis ?*

Cette question de recherche vise à étudier la capacité du concepteur à identifier les instances de MIS quand il/elle connaît a priori la définition générale des MIS.

Dans cette expérimentation, nous avons fixé les hypothèses suivantes :

1. Hypothèse 1 : les deux groupes d'étudiants auront les mêmes performances. Cette hypothèse peut être exprimée suivant les deux hypothèses (nulle et alternative) suivantes :
   — $H1.1$: les deux groupes d'étudiant vont retrouver le même nombre de structures à contraindre et vont trouver approximativement le même nombre de contraintes.
   — $H1.2$: Un des deux groupes obtiendra de meilleurs résultats, en étant capable de trouver plus de structures à contraindre, et en spécifiant plus de contraintes.
2. Hypothèse 2 : le groupe avec les connaissances sur les MIS sera capable de trouver plus de contraintes OCL liées aux MIS par rapport au deuxième groupe. Cette hypothèse est valide seulement si $H1.2$ est valide.
   — $H2.1$: le groupe avec les connaissances sur les MIS est capable de trouver plus de contraintes OCL que le groupe sans connaissances de MIS.
   — $H2.2$: le groupe avec les connaissances sur les MIS n'est pas capable de trouver plus de contraintes OCL que le groupe sans connaissances de MIS.

6.3.2 Méthode et données

Pour répondre à cette question de recherche, nous avons mené une expérience contrôlée avec des étudiants d'une école d'ingénieurs [1]. Tous les étudiants sont en quatrième année d'informatique, et ont suivi un cours sur la métamodélisation, incluant les langages OCL et MOF.

1. École Polytechnique de l'Université de Montpellier, France

L'étude consiste à donner des métamodèles à deux groupes d'étudiants afin qu'ils mettent en évidence les structures qui nécessitent des informations supplémentaires et qui doivent être complétées par des contraintes OCL. Nous avons choisi les deux métamodèles "Activités" et "Machines à états" d'UML. Tout d'abord, les étudiants doivent identifier les structures qui nécessitent des contraintes et expliquer pourquoi ces structures ne sont pas complètes. Ensuite, ils doivent exprimer les contraintes qui complètent ces structures.

Pour aider les étudiants dans leur quête de compréhension des métamodèles, nous leur avons fourni une documentation qui définit et explique chaque élément de modélisation des métamodèles. Cette documentation a été tirée de la spécification UML. En effet, cette documentation était importante car les métamodèles incluent certains concepts qui font partie de la syntaxe abstraite du langage spécifié par le métamodèle, mais qui ne font pas partie de sa syntaxe concrète.

Les étudiants ont été répartis aléatoirement en deux groupes, chaque groupe comprenant neuf étudiants en autonomie : le premier (*Groupe 1*) dispose d'un métamodèle, d'un exemple illustratif de contraintes OCL de niveau métamodèle, et de la documentation des métamodèles Activité et State Machine d'UML. Le second groupe (*Group 2*) a reçu en plus par rapport au premier groupe une documentation sur les MIS avec une définition et un exemple illustratif pour chaque MIS. Chaque groupe a été divisé en deux sous-groupes ayant chacun un métamodèle différent. Les deux groupes ont eu 105 heures pour réaliser l'expérience. Nous comparons les résultats des deux groupes (1 et 2) pour évaluer si la documentation du MIS (et ce que les étudiants du groupe en ont compris) les a aidé à identifier les problèmes potentiels du métamodèle.

Pour répondre à cette question, nous avons choisi deux métamodèles, tous deux tirés de la spécification UML [139]. Le premier métamodèle est le métamodèle State Machine comportemental, et le second est le métamodèle activités intermédiaires. Comme l'expérience était destinée à des étudiants qui n'avaient pas de connaissances approfondies sur le métamodèle UML par rapport aux experts, nous avons pris des métamodèles de taille moyenne traitables par l'homme. En outre, les sujets (étudiants) qui ont participé à l'expérimentation ont suivi des cours UML et se sont donc familiarisés avec la syntaxe concrète des diagrammes State Machine et Activities.

Afin de préparer l'expérience contrôlée, nous avons extrait toutes les contraintes OCL liées aux métamodèles Activités intermédiaires et State Machine comportemental. Ensuite, nous avons utilisé notre outil basé sur Prolog pour obtenir toutes les occurrences MIS des métamodèles. Pour ce faire, l'outil transforme les métamodèles Ecore en une base de connaissances qui inclut toutes les informations du métamodèle sous forme de prédicats. La transformation a été très rapide et a pris 0,2 seconde pour les deux métamodèles. Ensuite, l'outil recherche toutes les occurrences des MIS les retourne sous format csv. Le tableau 6.3 représente le nombre d'occurrences de chaque MIS que l'outil basé sur prolog a retourné.

MIS	Nombre d'occurrences	
	Activités	State Machine
Rest Val Attr	2	4
Rest Lit Énum	0	2
Relation de Types	3	8
Rest Multip Asso Hérit	10	11
Rest Val/mult Attr Hérité	0	4
Rest Val Oper Héritée	0	3
Cycles	3	4
Chemins	2	3
Total	20	39

TABLE 6.3 – Occurrences de MIS retournées par l'outil basé sur prolog

Après cela, nous avons étudié chaque occurrence de MIS séparément afin de marquer les occurrences ayant des contraintes OCL s'appuyant sur la spécification UML. Pour certaines occurrences, bien que nous ayons trouvé des textes de la spécification UML affirmant certaines informations qui auraient dû être spécifiées dans le langage OCL, nous n'avons pas trouvé de contrainte OCL correspondante. Pour combler cette lacune, nous avons écrit les contraintes manquantes qui apparaissent dans le texte de la spécification UML mais pas en tant que contraintes OCL. Par exemple, dans la sémantique de la State Machine comportementale, nous pouvons trouver qu'un *ForkNode* doit avoir exactement un *ActivityEdge* entrant, bien qu'il puisse avoir plusieurs *ActivityEdges* sortants [139]. Dans la spécification, nous pouvons constater que la contrainte qui restreint l'association *incoming [O..*]* dans la classe *ForkNode* est présente. Cependant, l'association *outgoing [0..*]* n'est pas contrainte. Par conséquent, il est possible d'avoir un *ForkNode* sans sorties, ce qui n'est pas correct vis-à-vis de la sémantique d'UML. Dans ce cas, nous avons

ajouté la contrainte suivante pour compléter le métamodèle.

context FinalNode **inv** outgoingSize :
self.outgoing $->$ size() > 0

Pour analyser équitablement le travail des étudiants, nous avons considéré comme correctes certaines contraintes écrites différemment de la spécification. Par exemple, un *FinalState* dans un *StateMachine* ne peut pas être composite. Par conséquent, nous pouvons trouver une contrainte dans la spécification UML qui restreint la région d'association à 0. Il est également possible de restreindre l'attribut *isComposite* à la valeur *false* à la place. Même si les contraintes sont différentes et restreignent des éléments de modélisation différents, elles conduisent au même résultat et sont sémantiquement équivalentes. Dans un souci de précision, nous avons supprimé toutes les contraintes qui limitent les éléments de modélisation dans la spécification UML mais qui ne sont pas incluses dans les métamodèles utilisés dans l'expérimentation. Les résultats de notre analyse des données sont détaillés dans le Tableau 6.4.

Metamodel	Nb Of Constraints		Total
	MIS Related	Others	
Activités Intermédiaire	20	6	26
State Machine Comportementale	40	16	56

TABLE 6.4 – Nombre de contraintes dans les métamodèles étudiés

6.3.3 Résultats et Discussion

Les résultats de l'expérience pour les deux groupes d'étudiants sont décrits dans le Tableau 6.5.

Groups	Correct constraints			Incorrect	Total
	MIS related	Other	Total		
G1	16	3	19	16	35
G2	34	5	39	11	50

TABLE 6.5 – Résultats globaux des deux groupes regroupés par catégorie de contraintes

D'après le tableau 6.5, nous pouvons voir que le premier groupe a écrit 35 contraintes OCL pour les deux métamodèles, dont 16 sont incorrectes (sémantiquement) et 19 cor-

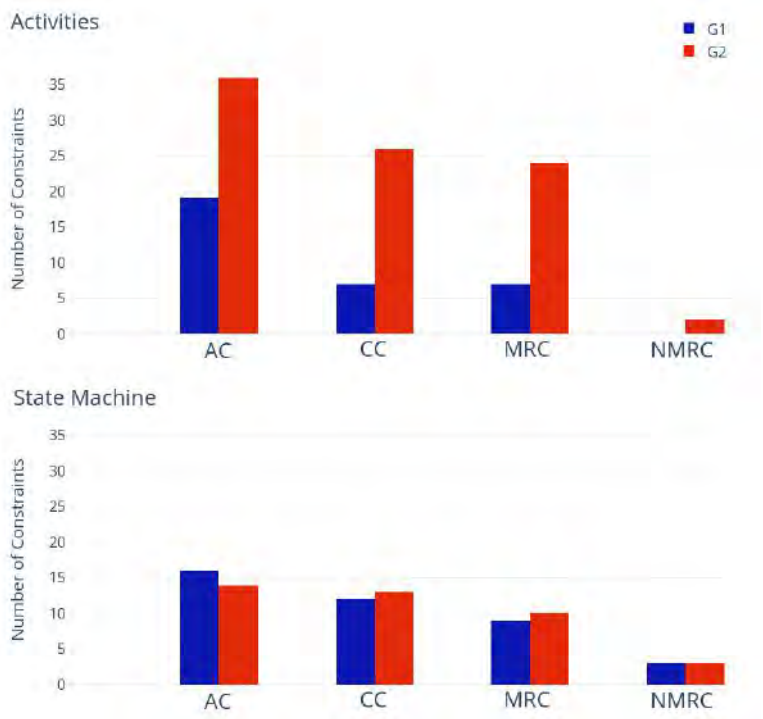

FIGURE 6.3 – Résultats détaillés de l'expérimentation pour les deux groupes

rectes. Parmi les contraintes correctes, 16 sont liées à des MIS, et 3 ne sont pas liées aux MIS. De l'autre côté, le deuxième groupe a écrit au total 50 contraintes OCL pour les deux métamodèles, dont 11 sont incorrectes et 39 correctes. Parmi les contraintes correctes, 34 sont liées aux MIS, et 5 ne le sont pas. Les résultats montrent qu'au total, le deuxième groupe a obtenu de meilleurs résultats que le premier.

Nous avons illustré les résultats détaillés dans la Figure 6.3. Pour le métamodèle Activités, le premier groupe a écrit 19 contraintes OCL au total (AC), dont 7 étaient à la fois correctes (CC) et liées au MIS (MRC). De l'autre côté, le second groupe a spécifié 36 contraintes OCL au total (AC), 26 d'entre elles étaient correctes (CC), dont 24 étaient liées au MIS (MRC) et 2 n'étaient liées à aucun MIS (NMRC). Pour le métamodèle State Machine, le premier groupe a écrit 16 contraintes OCL au total (AC). 12 contraintes sont correctes (CC) dont 9 contraintes sont liées à MIS (MRC) et 3 à d'autres structures (NMRC). De l'autre côté, le deuxième groupe a écrit 14 contraintes OCL au total (AC), soit 2 contraintes de moins que le premier groupe. Cependant, le deuxième groupe a obtenu plus de contraintes correctes avec 12 contraintes (CC), dont 9 sont des contraintes

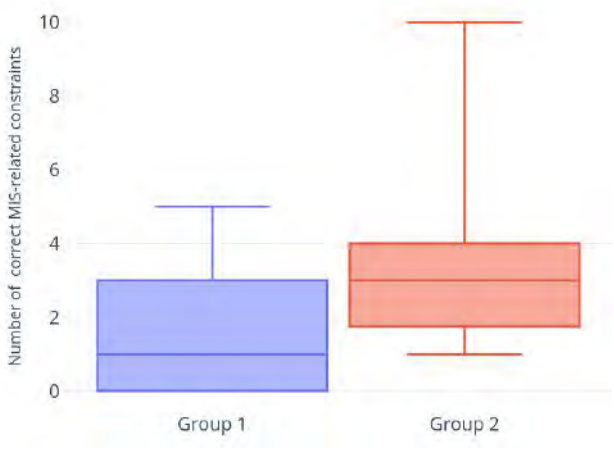

FIGURE 6.4 – Le nombre de contraintes écrites par étudiant par métamodèle

liées aux MIS (MRC) et 3 ne le sont pas (NRMC). Ces résultats montrent que le deuxième groupe qui s'est appuyé à la fois sur la spécification du métamodèle des activités et sur la spécification du MIS a été en mesure de trouver plus de contraintes OCL correctes que le premier groupe qui s'est appuyé uniquement sur la spécification du métamodèle des activités. Cela montre que la performance globale du second groupe était meilleure que celle du premier. Pour vérifier les performances individuelles, nous avons illustré le nombre de contraintes correctes pour chaque étudiant sous la forme d'un diagramme en boîte à moustache (*boxplot*) dans la Figure 6.4. Là encore, les résultats montrent que les étudiants du premier groupe qui n'avaient pas de documentation sur les MIS ont spécifié entre 0 et 5 contraintes correctes. De même, 50% des étudiants ont spécifié entre 0 et 3 contraintes. Les étudiants du groupe 2 disposant d'une documentation sur le MIS ont pu spécifier entre 1 et 10 contraintes, 50% d'entre eux ont spécifié entre 2 et 4 contraintes, ce qui confirme que la performance individuelle globale des étudiants du second groupe était meilleure que celle des étudiants du premier groupe.

La première partie de cette évaluation nous a permis de comparer le nombre de contraintes OCL. Maintenant, nous allons comparer le nombre de MIS que les contraintes spécifiées visent à vérifier. Cela permet de voir si les contraintes écrites par le second groupe ciblent plus de structures MIS ou non. Pour ce faire, pour chaque étudiant, nous avons pris l'ensemble des contraintes OCL spécifiées et l'avons analysé pour obtenir le nombre de MIS distincts pour chaque étudiant. Le diagramme en boîte à moustache de

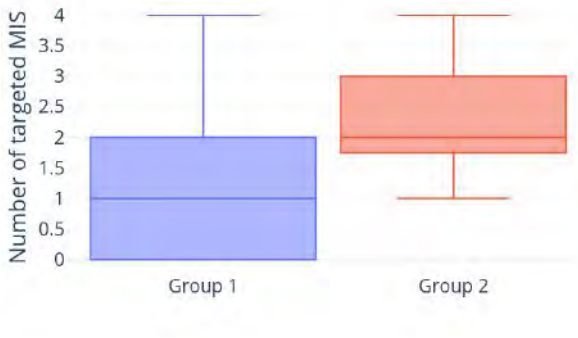

FIGURE 6.5 – Le nombre de MIS que chaque étudiant a identifié

la Figure 6.5 illustre les résultats.

Suivant la Figure 6.5, les étudiants du premier groupe ont ciblé entre 0 et 4 MIS. Au même niveau, les étudiants du second groupe ont ciblé entre 1 et 4 MIS avec leurs contraintes OCL. Même si la borne la plus élevée était la même pour les deux groupes, la médiane pour le premier groupe est de 1 MIS comparé au second groupe qui a 2 MIS. Si nous nous basons sur ces résultats, nous pouvons voir que la documentation des MIS a aidé les étudiants à comprendre plus profondément les problèmes et à trouver certaines contraintes que le premier groupe n'a pas été en mesure de trouver. Suivant ces résultats, nous rejetons les deux hypothèses 1.1 et 2.2 du fait que le groupe ayant la connaissance des MIS a obtenu de meilleures performances. Nous validons ainsi les hypothèses 1.2 et 2.1.

6.3.4 Menaces à la validité

Pour cette évaluation qualitative des MIS, certaines menaces peuvent avoir un impact sur les résultats. D'abord, la qualité des données recueillies lors de l'expérience. Les étudiants sont censés fournir des contraintes OCL dans le cadre du projet de spécification de contraintes OCL. Il y a donc un risque que les données soient incorrectes en raison d'erreurs, ou mal exprimées en raison du manque d'expertise du langage OCL. Pour éviter cette menace, nous avons demandé aux étudiants de décrire textuellement les problèmes que chaque contrainte OCL cible, afin de comprendre l'idée derrière chaque contrainte, même si la contrainte n'est pas correcte.

Un autre problème qui pourrait affecter négativement les résultats est celui des métamodèles utilisés. Même si les étudiants ont l'habitude de créer des diagrammes de classe UML dans leurs projets, ils ont trouvé de nombreuses parties des métamodèles très abstraites par rapport aux diagrammes de classe représentant des systèmes réels. Par conséquent, la mauvaise compréhension du métamodèle pourrait affecter négativement les résultats. Pour y remédier, nous avons choisi des métamodèles de taille moyenne pour l'expérimentation. En effet, les métamodèles Intermediate Activities et Behavioral State Machine de la spécification UML s'adaptent parfaitement à l'expérimentation avec les étudiants : i) premièrement, parce que les étudiants travaillent sur les deux types de diagrammes UML pendant le cours UML et affirment qu'ils sont habitués à travailler avec leur syntaxe concrète ; ii) deuxièmement, parce que nous avons fourni une spécification claire pour les métamodèles, contenant la définition de chaque métaclasse, un exemple de la syntaxe concrète (pour les classes concrètes), et des définitions et exemples pour chaque propriété (attributs et associations) et opérations au sein du métamodèle.

Une autre menace est liée aux contraintes disponibles dans la spécification des métamodèles utilisés pour l'expérimentation. En effet, la possibilité que les contraintes OCL qui sont écrites dans la spécification ne soient pas complètes n'était pas nulle. Pour atténuer ce risque, nous avons utilisé l'approche Prolog pour rechercher toutes les occurrences de MIS dans les deux métamodèles, et nous les avons analysées pour compléter les contraintes OCL du métamodèle UML sur la base du texte de la spécification UML comme moyen d'étudier chaque MIS pour juger s'il nécessite une contrainte OCL. Nous nous sommes également appuyés sur la spécification UML dans certains cas où les étudiants ont donné des réponses douteuses pour lesquelles nous ne connaissions pas la réponse. Le niveau des étudiants peut avoir un impact sur cette expérimentation. En effet, certains bons étudiants ont été capables de trouver plus de contraintes que certains moins bons étudiants disposant de la documentation MIS. Pour éviter cela, nous avons réparti équitablement les deux groupes d'étudiants pour éviter les biais liés au niveau des étudiants. L'expérimentation a été effectuée au deuxième semestre ce qui nous a permis de nous baser sur les résultats du premier semestre pour les classer. Enfin, nous sommes conscients que les performances de l'outil ne peuvent être généralisées à tous les domaines d'activité et à tous les langages de métamodélisation. De nombreux paramètres peuvent avoir un impact sur ce point, tels que la spécificité de certains domaines d'activité, la taille du métamodèle, le langage de métamodélisation et l'expérience du concepteur du métamodèle. Ces

paramètres doivent être étudiés en profondeur afin de pouvoir généraliser les résultats de cette expérimentation. Dans notre expérimentation, nous avons essayé d'atténuer ce risque en considérant deux parties du métamodèle UML (activités et machines à états) qui correspondent à deux types différents de domaines de modélisation.

6.3.5 Conclusion

Dans cette section, nous avons évalué notre liste de MIS d'un point de vue qualitatif. L'objectif était de comprendre si la connaissance apportée par les MIS permet à un concepteur de mieux identifier des structures ayant besoin de contraintes OCL, et ainsi de spécifier plus de contraintes pour préciser la sémantique du métamodèle. L'étude menée avec des étudiants de l'école polytechnique de Montpellier a démontré que le groupe d'étudiants ayant eu une documentation sur les MIS sont parvenus à identifier plus de structures à contraindre et à spécifier en moyenne plus de contraintes OCL en comparaison avec le groupe d'étudiants n'ayant pas eu de documentation sur les MIS. Bien que l'étude soit réalisée sur les métamodèles State Machine et Activités d'UML seulement, les résultats obtenus sont encourageants. Pour conclure, l'étude menée dans cette section vise à mesurer l'utilité des MIS en tant concepts. Afin de mieux cerner cela, nous procédons dans la section suivante à une étude des performances d'un outil recherchant automatiquement et exhaustivement toutes les occurrences de MIS.

6.4 Évaluation de la recherche automatique des MIS

6.4.1 Question de Recherche et Approche de Validation

QR : *Est-ce qu'un outil de recherche automatique d'instances de MIS serait utile pour des concepteurs de métamodèles ?*

Cette question de recherche vise à déterminer si un outil qui recherche automatiquement les occurrences de MIS dans un métamodèle donné aide le concepteur. Puisque le but principal de la recherche automatique est de retourner un ensemble exhaustif d'instances de MIS présentes dans le métamodèle, ce qui peut donner un nombre conséquent d'instances MIS à étudier par le concepteur, notre objectif est de vérifier si cela pourrait aider le concepteur à identifier plus de structures à contraindre, et ainsi de rendre son métamodèle plus précis.

Pour cette question de recherche, nous avons formulé une hypothèse comme suit :

$H1$: Les occurrences de MIS retournées par la recherche automatique requière tout le temps des contraintes OCL, ce qui prouve qu'elle est utile pour assister les concepteurs de métamodèles. À partir de cette hypothèse, nous formulons une hypothèse nulle et une hypothèse alternative :

— $H1.1$: Les occurrences de MIS retournées par la recherche automatique ne requièrent pas de contraintes OCL.

— $H1.2$: Les occurrences de MIS retournées par la recherche automatique requièrent souvent des contraintes OCL.

6.4.2 Méthode et données

Afin d'obtenir toutes les occurrences des MIS, nous avons développé un outil basé sur la logique du premier ordre. Pour la recherche de MIS, nous avons opté pour l'utilisation du moteur d'inférence Prolog. La raison principale de ce choix est la capacité d'étendre ou modifier l'ensemble de MIS, sans avoir à modifier le code source, en modifiant exclusivement la base de connaissances. De plus, le langage Prolog est très performant et largement utilisé quand il s'agit de problèmes de requêtage.

Construction d'un programme Prolog

La construction d'un programme Prolog se fait en deux étapes : 1) la création de la base de connaissances et 2) l'interrogation du moteur d'inférence. La création de la base de connaissances (étape 1) est une description du problème à résoudre à travers un ensemble de clauses. Une clause représente un fait ou une règle. Le fait regroupe des connaissances décrivant des objets sous forme de prédicats (énoncés). La règle regroupe les relations à respecter entre les différents faits. Dans un deuxième temps, l'utilisateur formule une requête au moteur d'inférence (interrogation du moteur d'inférence) qui opère sur les faits et les règles et tente de trouver des solutions (jeu d'instructions) pour répondre à la requête.

Utilisation de Prolog pour la recherche de MIS

La figure 6.6 illustre les différentes étapes nécessaires à l'utilisation de Prolog pour rechercher les MIS. D'abord, le programme Java prend en entrée le métamodèle ".ecore" puis le transforme en base de connaissances Prolog. Pour une telle transformation, nous

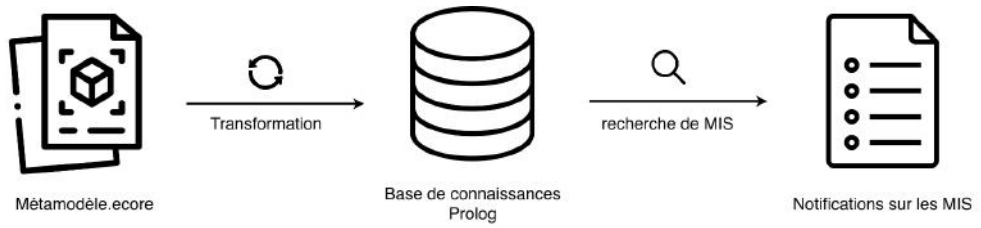

FIGURE 6.6 – Processus simplifié de recherche automatique MIS

nous sommes inspirés de la représentation d'un métamodèle sous Prolog qui a été proposée par [140], où chaque élément du métamodèle (classe, attribut, association, etc..) est représentée par un prédicat Prolog. Après l'obtention de la base de connaissances, nous présentons chaque MIS sous forme de requête Prolog. Celle-ci est analysée par le moteur d'inférence qui recherche toutes les valeurs qui satisfont la requête. Ainsi, le moteur d'inférence retourne un fichier contenant toutes les occurrences trouvées pour chaque MIS.

Génération d'une base de connaissances à partir d'un métamodèle en Ecore
Pour transformer un métamodèle ecore vers une base de connaissances, il est nécessaire de décrire chaque élément du métamodèle sous la forme de faits. La figure 6.7 représente le méta-métamodèle Ecore. Par conséquent, chaque méta-métaclasse du métamodèle ecore est représentée en Prolog suivant ces règles :

- ***EPackageversFait***
  Chaque package d'un métamodèle est traduit par le *fait package(idPackage, nompackage, superPackage)*, où l'*idPackage* désigne un identifiant unique pour le package, *nompackage* correspond au nom du package et *superPackage* au nom du super package s'il existe. Seul le super package direct est précisé dans ce *fait*. *superPackage* est remplacé par "_" pour le package racine.

- ***EClassversFait***
  Chaque classe du méta-modèle est transformée en un fait sous la forme de *class(idClass, nomClasse, nomPackage, estAbstraite)* en Prolog. L'élément *idClass* définit l'identifiant unique de la classe, *nomClasse* définit le nom de la classe et *nomPackage* le package auquel elle appartient. L'élément *estAbstraite* prend la valeur *vrai* ou *faux* qui exprime respectivement que la classe est abstraite ou non.

- ***EAttributversFait***

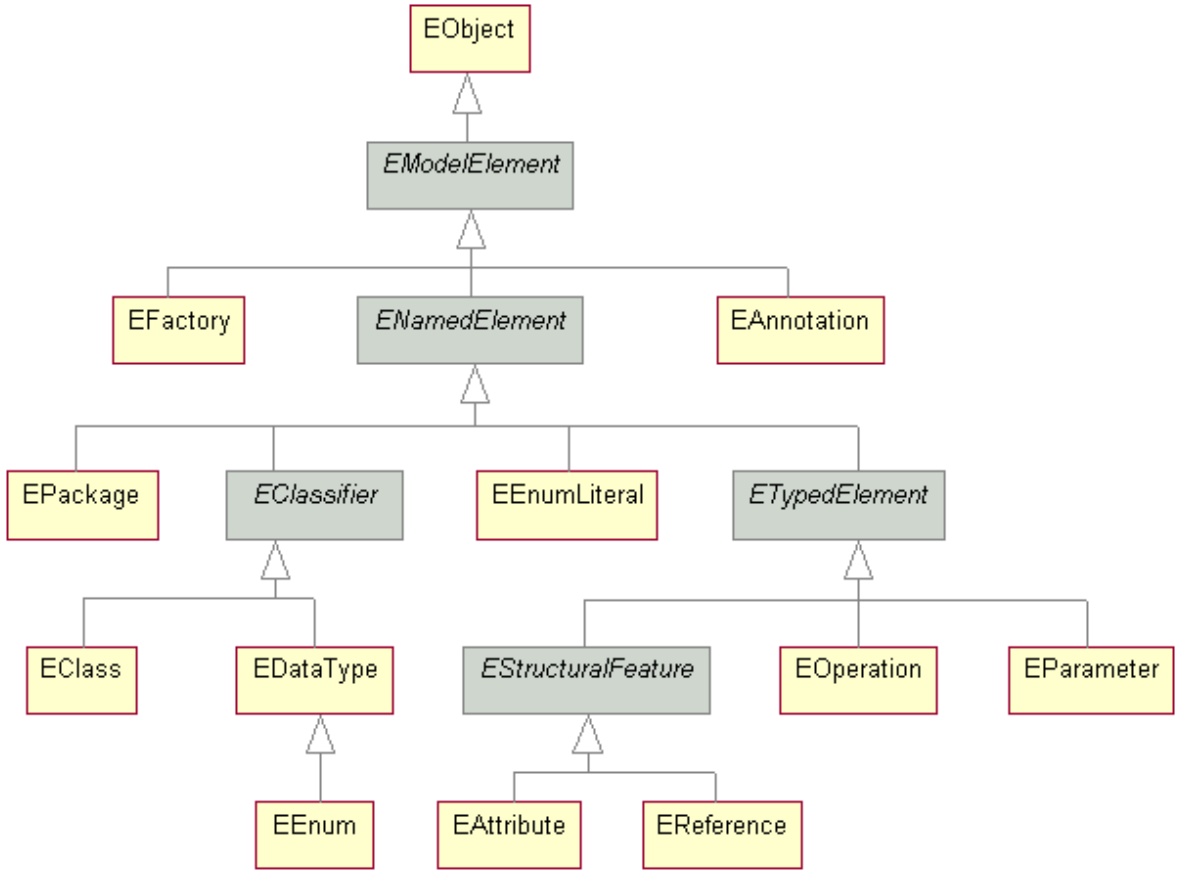

FIGURE 6.7 – Le méta-métamodèle Ecore

Un attribut se traduit par *attribute(idAttribut, nomAttribut, typeAttribut, nomClasse, borneInf, borneSup, valeurInit)*, où l'*idAttribut* et *nomAttribut* indiquent l'identifiant et le nom de l'attribut, *typeAttribut* définit son type et *nomClasse* sa classe. Les attributs sont caractérisés par une cardinalité inférieure (*borneInf*) et une cardinalité supérieure (*borneSup*).

- ***EOperationversFait***

  Chaque opération est traduite vers un fait sous la forme de *opération(idOpération, nomOpération, typeOpération, nomClasse, borneInf, borneSup)*, où *idOpération* désigne l'identifiant unique de l'opération, *typeOpération* et *nomClasse* désignent le type d'éléments retournés par l'opération et la classe contenant l'opération. Les cardinalités inférieure et supérieure de l'opération sont traduites comme *borneInf* et *borneSup* respectivement.

- ***EParameterversFait***

  Chaque paramètre d'opération est traduit par *paramètre(idParamètre, nomParamètre, typeParamètre, nomOpération, nomClasse)*. *idParamètre* désigne l'identifiant du paramètre. Les éléments *nomParamètre* et *typeParamètre* indiquent respectivement le nom du paramètre et son type. L'élément *nomOpération* définit le nom de l'opération contenante et l'élément *nomClasse* sa classe contenante.

- ***EReferenceversFait***

  Chaque référence (ou association), est traduite par *référence(idRéférence, nomRéférence nomClasse, classeCible, borneInf, borneSup, référenceOpposée)*. *idRéférence* désigne l'identifiant de la référence. *nomRéférence* indique le nom de la référence, la classe source (*nomClasse*) et la classe de destination (*classeCible*), sa cardinalité inférieure (*borneInf*) et supérieure (*borneSup*). Une référence peut contenir une référence opposée (*référenceOpposée*). Ainsi, une référence bidirectionnelle est représentée comme deux faits, chacun désigne une référence unidirectionnelle.

- ***HéritageversFait***

  Bien que le lien d'héritage n'apparaît pas dans le méta-métamodèle ecore de la figure 6.7, celui-ci est représenté comme *héritage(sous-classe, super-classe)*, où *sous-classe* décrit le nom de la classe fille et *super-classe* le nom de la classe mère.

- ***EEnumversFait***

  Chaque énumération dans le métamodèle est traduite à *énumeration(idÉnumération, nomÉnumération)*, tel que *idÉnumération* et *nomÉnumération* définissent l'identifiant et le nom de l'énumération.

- ***EEnumLiteralToFact***

  Chaque littéral est traduit comme *littéral(nomÉnumération, idLittéral, nomLittéral)* tel que *idÉnumération* et *idLittéral* désignent le nom de l'énumération ainsi que l'identifiant du littéral, tandis que *nomLittéral* représente le nom du littéral.

- ***EDataTypeversFait***
  Chaque type de données est traduit comme *type(idType,nomType)*, tel que *idType* et *dataTypeName* definissent l'identifiant et le nom du type de données.

Nous n'avons pas traduit les annotations(*EAnnotation* dans la figure 6.7), EFactories (*EFactory*) et la méta-métaclasse *EObject*, étant donné que ces concepts ne sont pas nécessaires à notre étude.

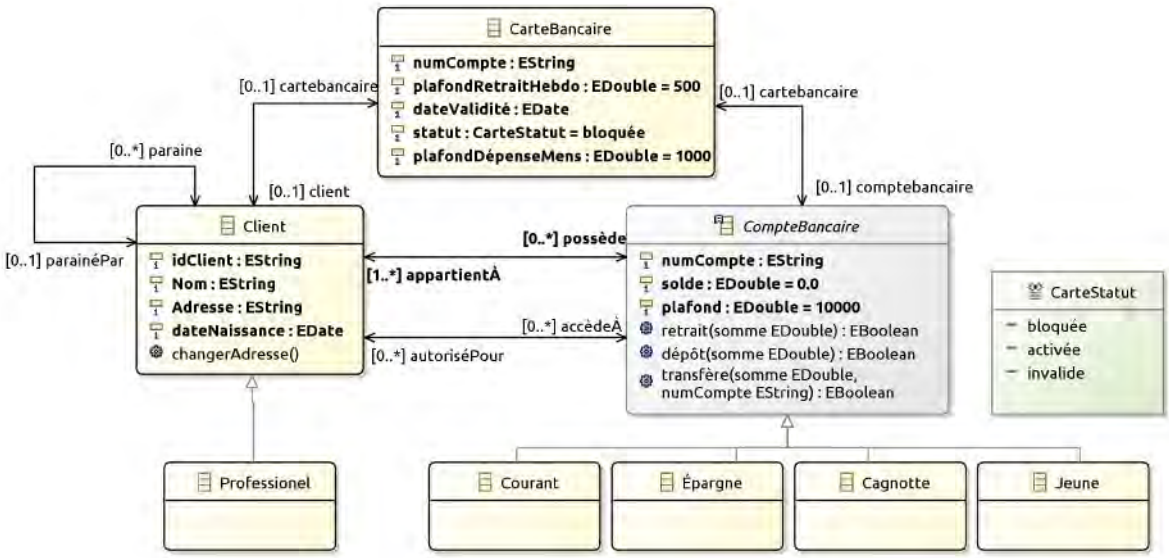

FIGURE 6.8 – Exemple d'un métamodèle qui représente une banque

Pour illustrer l'approche de génération d'une base de connaissances à partir d'un métamodèle ecore, nous nous basons sur le métamodèle de la figure 6.8. Ce dernier est une représentation simplifiée d'une banque. À partir de ce métamodèle, la base de connaissance suivante est générée. Nous présentons seulement quelques exemples pour chaque concept.

```
package ( banque , _ ).
classe ( c1 , CarteBancaire , banque , faux ).
classe ( c2 , Client , banque , faux ).
classe ( c3 , CompteBancaire , banque , vrai ).
classe ( c4 , Courant , banque , faux ).
    ...
```

```
attribut ( atr1 , idClient , EString , Client , 1 , 1 , _ ).
attribut ( atr2 , solde , EDouble , CompteBancaire , 1 , 1 , 0.0 ).
attribut ( atr3 , dateNaissance , EDate , Client , 1 , 1 , _ ).
    ...
référence ( ref1 , possède , Client , CompteBancaire , 0 , * , appartientÀ ).
référence ( ref2 , appartientÀ , CompteBancaire , Client , 1 , * , possède ).
    ...
héritage ( Courant , CompteBancaire ).
    ...
énumération ( enum1 , CarteStatut ).
    ...
littéral ( CarteStatut , lit1 , bloquée ).
    ...
opération ( op1 , retrait , EBoolean , CompteBancaire , 0 , 1 ).
    ...
paramètre ( par1 , somme , EDouble , retrait , CompteBancaire ).
    ...
type ( type1 , EDate ).
```

Représentation des MIS sous forme de requêtes

Dans le but de rechercher les occurrences des MIS dans la base de connaissances, chaque MIS est représenté comme étant une requête. Celle-ci est traitée par le moteur d'inférence qui recherche toutes les situations pour lesquelles la requête est valide.

L'exemple suivant illustre la représentation des deux MIS "Cycle" et "Restriction de la multiplicité d'une association héritée". Par exemple, le MIS *"Restriction de la multiplicité d'une association héritée"* est formulé par la requête suivante :

```
:- classe(_,C1,_,_), classe(_,C2,_,_), classe(_,C3,_,_),
   héritage(C3,C1), référence(IdRef,Ref,C1,C2,Bi,Bs,_), Bs > Bi.
```

Dans cette requête, le symbole ' :- ' désigne une condition "si", et la virgule entre les différents prédicats désigne l'opérateur logique "et". La règle précédente est ainsi interprétée comme suit :

Interprétation de la requête

S'il existe une classe C1 et une classe C2 et une classe C3, et un lien d'héritage entre les classes C1 et C3, et une référence dans la classe C1 qui cible la classe C2, et que la borne supérieure de la multiplicité de cette référence soit strictement supérieure a sa borne inférieure, retourner le vecteur de valeurs.

Un vecteur de valeur correspond à un ensemble ordonné de valeurs que le moteur d'inférence a donné à chaque variable pour satisfaire la requête. Pour chaque MIS, chaque vecteur de valeurs désigne une solution, et ainsi une occurrence du MIS. Nous expliquons dans ce qui suit la démarche suivie pour la recherche de chaque MIS.

- Restriction de la valeur d'attributs : pour rechercher ce MIS, il suffit de rechercher tous les attributs figurant sur le métamodèle.
- Restriction des littéraux d'énumérations : pour rechercher ce MIS, il faut vérifier s'il existe deux attributs ayant comme type la même énumération.
- Restriction de la valeur d'un attribut hérité : pour rechercher ce MIS, il faut rechercher tous les attributs hérités.
- Restriction d'un attribut optionnel : parmi les attributs retournés par la recherche du MIS précédent, on sélectionne les attributs optionnels.
- Restriction de la multiplicité d'une association héritée : pour rechercher ce MIS, il faut rechercher toutes les associations héritées.
- Restriction de la valeur d'une opération héritée : pour ce MIS, il faut rechercher toutes les opérations héritées.
- Relation de types : pour ce MIS, il faut rechercher deux classes liées par une association unidirectionnelle, où au moins une des classes est héritée. Par conséquent, si l'association qui lie les deux classes, le MIS est compté deux fois.
- Relation de types avec énumération : recherche une association qui lie deux classes, une des ces classes contient un attribut de type énumération, et l'autre classe est héritée.
- Cycles : pour contrôler la recherche de ce MIS, nous l'avons décomposé en plusieurs variants :

  1. Cycles Directes : recherche toutes les associations réflexives, mais aussi les opérations qui sont spécifiées dans une classe A, et qui retournent une collection

d'éléments du même type (classe A). Pour les associations réflexives, si l'association est bidirectionnelle, le MIS est compté deux fois.

2. Cycles indirects de taille 2 : recherche toutes les navigations de taille 2 comme suit : (Classe1 -> Navigation1 -> ClasseInter -> Navigation2 -> Classe2). Si une classe contient une association bidirectionnelle, la navigation par le biais de cette association puis par l'association opposée n'est pas considérée comme un cycle de taille 2.

— Chemins : pour la recherche de chemins, nous l'avons décomposé comme suit :

1. Deux navigations de taille 1 qui ont les mêmes classes source et cible. Cela s'illustre par le schéma suivant :
   Chemin A : Classe1 -> Navigation1 -> Classe2
   Chemin B : Classe1 -> Navigation2 -> Classe2

2. Deux navigations de taille 2 au plus et qui ont les mêmes classes source et cible.
   Cas 1 : les chemins ont une taille différente
   Chemin A : Classe1 -> Navigation1 -> Classe2
   Chemin B : Classe1 -> Navigation2.1 -> ClasseInter -> Navigation2.2 -> Classe2
   Cas 2 : les chemins ont la même taille Chemin A : Classe1 -> Navigation1.1 -> ClasseInter1 -> Navigation1.2 -> Classe2
   Chemin B : Classe1 -> Navigation2.2 -> ClasseInter2 -> Navigation2.2 -> Classe2

Nous avons appliqué l'approche de recherche automatique sur 8 métamodèles comme illustré dans le tableau 6.6. Ces métamodèles représentent un sous-ensemble des métamodèles utilisés pour la validation quantitative des MIS, en plus du métamodèle UML 2.5. Pour chacun, le nombre d'éléments de modélisations est détaillé (classes, attributs, héritages, etc...), ainsi que le nombre de contraintes OCL, et le taux des contraintes liées aux MIS.

Métamodèle	Packages	Classes	Asso	Attr	Oper	Énum	Héritage	Nb-COCL	Nb-CMIS	Total éléments
CWM	38	251	343	236	16	16	256	96	42 (43%)	1156
DD	2	39	48	54	16	20	27	16	15 (93%)	206
SAD3	1	41	102	35	0	0	0	8	2 (25%)	179
CPFS	1	18	25	17	0	3	7	27	23 (85%)	71
EP2RE	1	18	68	3	7	0	3	59	35 (56%)	100
SAM	1	48	56	9	43	5	56	82	14 (17%)	218
RBAC	1	11	46	29	22	0	0	34	10 (29%)	109
UML	17	242	478	112	588	62	282	450	240 (53%)	2231

TABLE 6.6 – Nombre d'éléments pour chaque métamodèle

Afin d'évaluer l'utilité de la recherche automatique, nous comparons le nombre d'instances de chaque MIS avec le nombre de contraintes de ce MIS qui sont présents dans la spécification du métamodèle, et qui sont présentés dans le tableau 6.7. Ceci nous permet de vérifier si un outil automatique qui recherche toutes les instances de chaque MIS peut être considéré comme une valeur ajoutée. La question ne vise pas à vérifier les performances de la recherche étant donné qu'elle est exhaustive, mais plutôt d'évaluer le ratio bénéfice/temps. Ainsi, conclure si cela vaut la peine.

MIS	CFPST	CWM	DD	ER2RE	RBAC	SAD3	SAM	UML	Total
Attributs	6	1	15	13	0	2	7	7	51
attributs d'énum	0	0	0	0	0	0	0	2	2
Relation de types	0	13	0	0	0	0	0	31	44
Attrib hérités (optio)	4(0)	1(0)	0	0	0	0	0	12(3)	17(3)
Association héritée	1	6	0	0	0	0	0	60	67
Opération héritée	0	0	0	0	0	0	1	23	24
Cycles 1	0	10	0	0	3	0	0	11	24
Cycle 2	0	7	0	0	0	0	0	7	14
Total Cycles	0	17	0	0	3	0	0	18	38
Chemins 1-1	0	3	0	3	3	0	0	29	38
Chemins 2-1	12	0	0	4	4	0	0	5	5
Chemins 2-2	0	0	0	15	0	0	6	15	36
Chemins >2-2	0	1	0	0	0	0	0	35	36
Total Chemins	12	4	0	22	7	0	6	84	135

TABLE 6.7 – Nombre de contraintes liées aux MIS par métamodèle

6.4.3 Résultats et discussion

Dans cette section, nous détaillons les résultats de la recherche automatique de MIS qui a été menée par notre outil. les résultats de la recherche automatique de MIS sont présentés dans le tableau 6.8. Ces résultats sont très variables d'un métamodèle à un autre, mais surtout d'un MIS à un autre. Par exemple, le MIS ayant engendré le plus de notifications est "chemins", plus particulièrement dans les métamodèles de grande taille, tel qu'UML et CWM. Au total, la recherche de ce MIS par le moteur d'inférence prolog a retourné 2101, 39743, et 463091 pour les chemins de taille 1-1, de taille 2-1, et de taille 2-2 respectivement. De l'autre côté, les MIS ayant été le moins trouvé est celui des énumérations avec 7 occurrences au total. Ceci est dû au fait que les énumérations ne sont pas autant présentes que les autres éléments de modélisation. Aussi, les MIS liés à l'héritage dépendent du métamodèle. Par exemple, les métamodèles ER2RE, RBAC et SAD3 ne présentent que peu de liens d'héritage, tandis que les métamodèles UML et CWM présentent beaucoup d'héritage.

Afin de mieux interpréter les résultats du tableau 6.8, nous avons créé le tableau 6.10 qui résume la précision des résultats. Étant donné que la recherche automatique de MIS retourne toutes les occurrences, la valeur du rappel est toujours de 1 étant donné que toutes les structures contraintes figurent parmi la liste des MIS que retourne la recherche. Pour calculer la précision, nous avons souligné trois cas de figure pour le calcul de précision.

1. S'il existe des contraintes d'un MIS, la précision est calculée en divisant le nombre de contraintes du MIS $NB{-}CMIS$ par le nombre d'occurrences $NB{-}Occurrences$ comme décrit par l'équation suivante :

$$Précision = NB - Occurrences/NB - CMIS$$

2. Si un MIS n'a pas de contraintes OCL, et que l'outil retourne des occurrences de ce MIS, la valeur de la précision prend 0 car ces occurrences de MIS que le concepteur de métamodèle devra étudier vont augmenter le temps nécessaire à l'étude des MIS.
3. Si un MIS n'a pas de contraintes OCL, et que la recherche ne retourne aucun MIS. Dans ce cas de figure, nous ne sommes pas apte à mesurer la précision de la recherche automatique et la case est barrée avec '/'.

MIS	CFPST	CWM	DD	ER2RE	RBAC	SAD3	SAM	UML	Total
Attributs	17	236	54	3	29	35	9	113	496
attributs d'énum	0	1	0	0	0	0	0	6	7
Relation de types	12	182	14	0	0	0	22	310	540
Attrib hérités (optio)	7 (0)	64 (12)	1 (0)	1 (0)	0	0	4 (4)	41(11)	118(27)
Association héritée	7	92	12	0	0	0	8	207	326
Opération héritée	0	12	0	0	0	0	23	231	266
Cycles 1	1	18	2	0	34	0	6	63	124
Cycle 2	12	16	5	0	8	0	10	206	257
Chemins 1-1	1	479	22	0	11	0	100	1488	2101
Chemins 2-1	23	9437	135	54	80	35	1195	28784	39743
Chemins 2-2	124	56975	536	420	228	82	5058	399792	463091

TABLE 6.8 – Résultats de la recherche automatique de MIS

MIS	CFPST	CWM	DD	ER2RE	RBAC	SAD3	SAM	UML	Total
Attributs	35%	0.4%	27%	100%	0%	5.7%	77%	6.2%	10.3%
attributs d'énum	/	0%	/	/	/	/	/	33.3%	28.6%
Relation de types	0%	7%	0%	/	/	/	0%	10%	8.1%
Attrib hérités (optio)	57%(/)	1.5%(/)	0%	0%	/	/	0%	29%(27%)	14.4%(11.1%)
Association héritée	14%	6.5%	0%	/	/	/	0%	29%	20.55%
Opération héritée	/	0%	/	/	/	/	4.3%	9.9%	9%
Cycles 1	0%	55%	0%	/	8.8%	/	0%	17.4%	19.3%
Cycle 2	0%	43.7%	0%	/	0%	/	0%	3.3%	5.44%
Chemins 1-1	0%	0.6%	0%	0%	27%	/	0%	1.9%	1.8%
Chemins 2-1	52%	0%	0%	7.4%	5%	0%	0%	0.0018	0.01%
Chemins 2-2	0%	0%	0%	3.5%	0%	0%	0.1%	0.003%	0.007%
Total									

TABLE 6.9 – Précision de la recherche automatique par MIS et par métamodèle

MIS	CFPST	CWM	DD	ER2RE	RBAC	SAD3	SAM	UML	Total
Attributes	35%	0.4%	27%	100%	0%	5.7%	77%	6.2%	10.3%
Enum Attribs	/	0%	/	/	/	/	/	33.3%	28.6%
Type Relation	0%	7%	0%	/	/	/	0%	10%	8.1%
Inh Attribs (opt)	57%(/)	1.5%(/)	0%	0%	/	/	0%	29%(27%)	14.4%(11.1%)
Inh Assoc	14%	6.5%	0%	/	/	/	0%	29%	20.55%
Inh Oper	/	0%	/	/	/	/	4.3%	9.9%	9%
Cycles 1	0%	55%	0%	/	8.8%	/	0%	17.4%	19.3%
Cycle 2	0%	43.7%	0%	/	0%	/	0%	3.3%	5.44%
Paths 1-1	0%	0.6%	0%	0%	27%	/	0%	1.9%	1.8%
Paths 2-1	52%	0%	0%	7.4%	5%	0%	0%	0.0018	0.01%
Paths 2-2	0%	0%	0%	3.5%	0%	0%	0.1%	0.003%	0.007%

TABLE 6.10 – Précision de la recherche automatique par MIS et par métamodèle

Suivant le tableau 6.10, les résultats montrent que la précision varie entre 0% et 100% en fonction du métamodèle et du MIS. Par exemple, dans le métamodèle ER2RE, la précision relative au MIS "Restriction de la valeur d'un attribut" est de 100%. Ceci s'explique par le fait que le métamodèle ne contient que trois attributs, et que pour ces derniers, les concepteurs ont spécifié 13 contraintes OCL. De l'autre côté, nous retrouvons la précision de 0% dans 31 cas de figure. Ce chiffre s'explique par la présence de la structure, mais l'absence de contraintes pour cette même structure. Au final, le MIS ayant la précision la plus élevée est "attributs d'énumération" avec 28.6%, ce qui représente à peu près une occurrence à contraindre parmi quatre occurrences retournées. En revanche, le MIS "chemins" a obtenu la précision la plus faible avec 0.007%. Ceci s'explique par l'explosion combinatoire que la recherche de ce MIS provoque surtout en recherchant des chemins de plus grande taille. La recherche automatique de ce MIS n'est pas utile puisque pour 14286 occurrences, il existerait une seule occurrence ayant besoin de contraintes OCL.

Intervalle	Nombre de cas
Entre 100% et 50%	5
Entre 50% et 25%	8
7 heightEntre 25% et 10%	7
6 5 4 3 heightEntre 10% et 05%	10
9 8 7 heightEntre 05% et 00 exclu%	14
13 12 11 height00%	31
Cas indéterminés	24

TABLE 6.11 – Précision de la recherche automatique de MIS regroupée par intervalles

En se basant sur les résultats, nous remarquons que la recherche automatique peut être intéressante certains cas. Nous regroupons les précisions du tableau 6.10 par intervalles dans le tableau 6.11 afin de mieux visualiser les performances de la recherche automatique. Les données du tableau montrent que la précision se situe entre 50% et 100% dans cinq cas, 8 cas entre 50% et 25%, 7 cas entre 25% et 10%, 10 cas entre 10% et 05%, 14 cas entre 05% et 00% exclu, 31 cas avec 0% de cas, et 24 cas indéterminés. Selon le MIS et le métamodèle, la recherche peut être très utile ou médiocre. En se basant sur ces résultats, nous pouvons confirmer que la recherche exhaustive de MIS présente un nombre trop important de faux positifs pour être exploitée par un concepteur de métamodèles en quête de spécification de contraintes OCL. Cependant, nous ne sommes pas en mesure d'exclure cette approche, surtout dans certains domaines où l'exhaustivité prends le dessus sur le temps d'étude. Aussi, il est possible que l'approche de recherche automatique soit avantageuse si elle est

effectuée au cours de la création du métamodèle. Ainsi, à chaque modification apportée au métamodèle, des notifications relatifs à cette modification sont affichées. Il est aussi possible de construire un système d'assistance intelligent qui permet au concepteur de choisir les MIS qu'il souhaiterait étudier.

Suivant les résultats, nous sommes en mesure de confirmer l'hypothèse $H1.2$ et d'écarter l'hypothèse $H1.1$

6.4.4 Conclusion

Dans cette section, nous avons évalué l'intérêt d'utiliser un outil de recherche automatique de MIS pour assister les concepteurs de métamodèles pendant la phase de spécification de contraintes OCL à trouver plus de structures à contraindre, et ainsi obtenir plus de contraintes OCL. Pour cela, nous avons construit un outil basé sur prolog et la logique du premier ordre pour chercher automatiquement les MIS, puis nous l'avons utilisé sur huit métamodèles. Nous avons procédé à l'évaluation des performances de la recherche automatique en comparant le nombre d'occurrences de chaque MIS que l'outil retourne, avec le nombre de contraintes OCL que les concepteurs ont écrit pour ce MIS. Les résultats ont montré que la recherche automatique provoque énormément de faux positifs, c.à.d des occurrences de MIS que la recherche retourne mais qui n'ont pas besoin de contraintes OCL. Néanmoins, pour certains MIS dans certains métamodèles, l'étude s'est avérée très utile proposant de bonnes performances. Bien que l'approche donne souvent de mauvais résultats, nous sommes persuadés qu'il serait possible de l'améliorer afin d'en tirer profit, et de proposer une approche d'assistance à la création de métamodèles précis en améliorant cette approche.

6.5 Évaluation sur l'identification de MIS suite à l'évolution du métamodèle

Dans cette partie, nous évaluons les performances de notre approche d'assistance à la coévolution de contraintes OCL. Pour cela, nous avons mené une étude de cas afin de fournir des réponses aux questions de recherche suivantes.

6.5.1 Questions de recherche et approche de validation

— **QR1. Quelle est la performance de notre méthode dans la co-évolution des contraintes OCL ?**

Cette question évalue la capacité de notre approche à faire co-évoluer correctement les contraintes OCL pendant l'évolution de leur méta-modèle. Pour cela, nous définissons les hypothèses suivantes : Hypothèse 1 : l'approche de coévolution de contraintes OCL est performante en terme de nombre de co-évolutions à proposer. Cette hypothèse est détaillée en une hypothèse nulle et une alternative comme suit :

1. Hypothèse 1.1 : l'approche permet de proposer toutes les contraintes à co-évoluer dans le cas d'étude.
2. Hypothèse 1.2 : l'approche permet de proposer quelques coévolutions seulement.

— **QR2. Quelle est la performance de notre méthode dans la notification de nouvelles contraintes potentielles liées au MIS ?**

Cette question vise à étudier la capacité de notre méthode à notifier les structures qui peuvent potentiellement nécessiter de nouvelles contraintes, et ainsi compléter la coévolution des contraintes OCL au cours de l'évolution d'un métamodèle. Les résultats de cette question permettront de mettre en évidence l'originalité de notre méthode par rapport aux méthodes de l'état de l'art. Pour répondre à cette question, nous avons formulé les hypothèses suivantes : Hypothèse 2 : l'approche de notification de MIS permet d'identifier de nouvelles structures introduites par l'évolution du métamodèle et qui nécessitent obligatoirement des contraintes OCL. Cette hypothèse est divisée en deux hypothèses nulle et alternative comme suit :

1. Hypothèse 2.1 : l'approche de notification de MIS suite à l'évolution du métamodèle ne permet pas d'identifier de structures ayant obligatoirement besoin de contraintes OCL.
2. Hypothèse 2.2 : l'approche de notification de MIS suite à l'évolution du métamodèle permet d'identifier de structures ayant obligatoirement besoin de contraintes OCL.

6.5.1.1 Cas d'étude sélectionné

Nous avons choisi d'évaluer notre méthode sur le métamodèle State Machine qui a évolué de la version 1.5 (Fig. 6.9) à la version 2.0 (Fig. 6.10). Ce choix est guidé par :

1. l'accessibilité du métamodèle et de ses contraintes OCL avec une documentation complète ;
2. le nombre d'opérations d'évolution significatif (73 opérations qui ont été nécessaires pour faire évoluer le métamodèle de 1.5 à 2.0), cela couvre de nombreux opérateurs d'évolution (19), ce qui nous permet de valider la coévolution des contraintes telles que les approches existantes dans la littérature. De plus, cela permet d'identifier de nouvelles contraintes qui sont nécessaires pour raffiner la nouvelle version du métamodèle ;
3. le métamodèle est largement utilisé dans la pratique ce qui prouve sa bonne construction ;
4. la maniabilité : le métamodèle est concis comparé à l'ensemble du métamodèle UML. Dans ce qui suit, nous présentons le métamodèle StateMachine basé sur sa version 2.0 (Fig. refuml20).

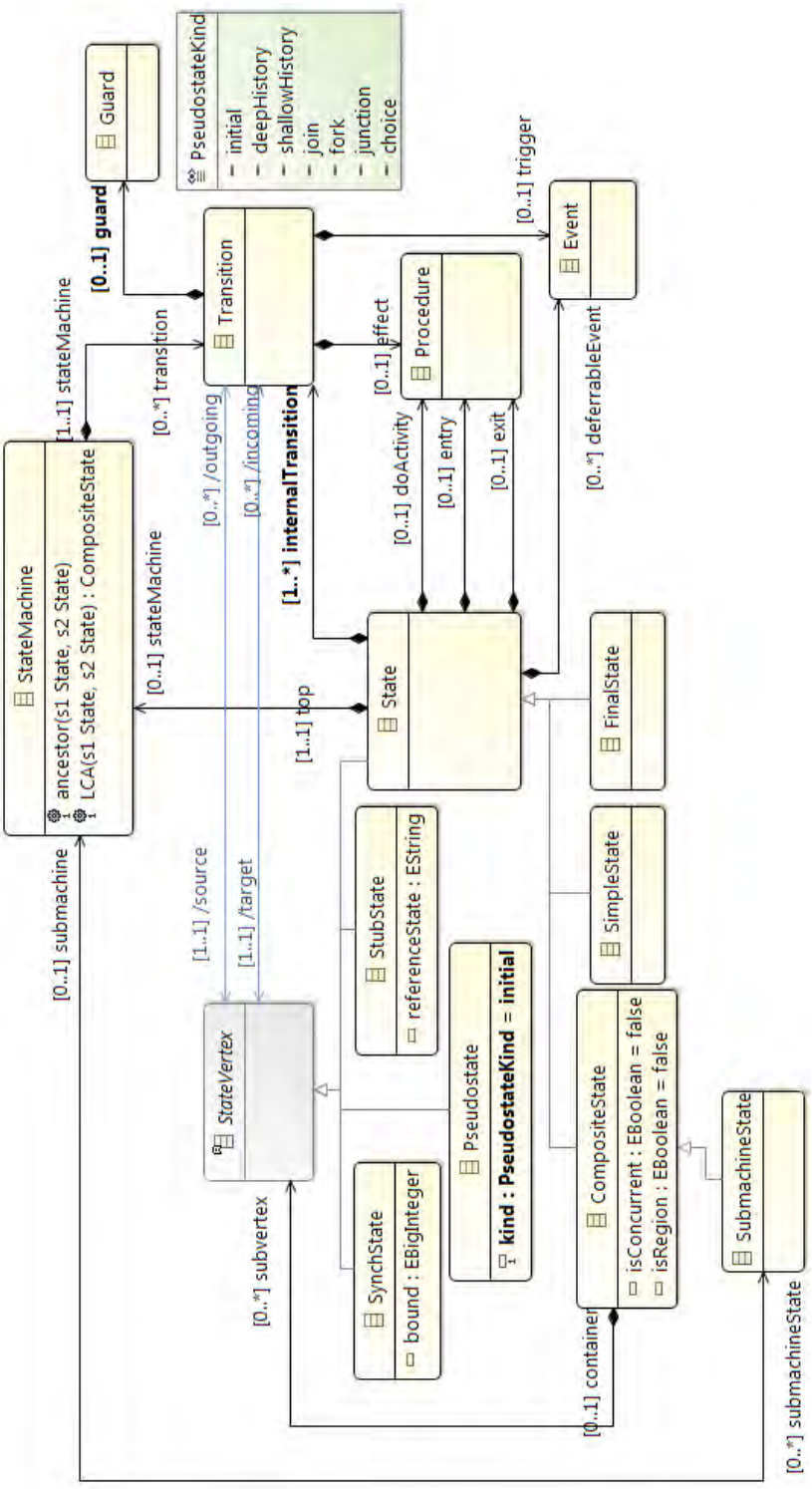

FIGURE 6.9 – La version 1.5 du métamodèle StateMachine (depuis [141])

Les machines à états offrent une gamme de concepts permettant de modéliser le comportement discret d'un système sous la forme d'un graphe de sommets et de transitions. Les sommets sont de différents types : états, pseudo-états et références de points de connexion (Fig. 6.10). Les états décrivent une situation dans laquelle une condition est vérifiée [142]. Les états peuvent être des états simples, composites, sous-machines ou finaux. Contrairement à un état simple, un état composite peut contenir d'autres états et transitions souvent regroupés en régions. Un état composite, contenant au moins une région, peut contenir plusieurs régions. Dans ce cas, l'état est dit orthogonal car les régions peuvent fonctionner en parallèle. Un état final est le dernier état à parcourir lors de l'exécution d'une machine à états. Les pseudo-états sont des états transitoires (l'exécution ne s'arrête jamais dans ces états). Le diagramme de l'automate à états offre 10 pseudo-états (entryPoint, exitPoint, initial, deepHistory, shallowHistory, join, fork, junction, terminate, ou choice). D'autre part, le point de connexion caractérise les points d'entrée et de sortie d'une sous-machine.

Une transition est un arc allant d'un sommet source à un sommet cible. Une transition est déclenchée à la réception d'un événement donné et/ou si sa garde associée est vraie. Dans la version 2.0 [142], trois types de transitions ont été définis : interne, locale et externe. Une transition interne est une transition qui relie un état à elle-même et qui ne provoque pas de changement d'état pendant son exécution. Une transition locale est une transition qui ne sort pas d'un état composite, elle est toujours exécutée dans l'état composite. En revanche, l'exécution d'une transition externe entraîne la sortie de l'état composite dont elle est issue.

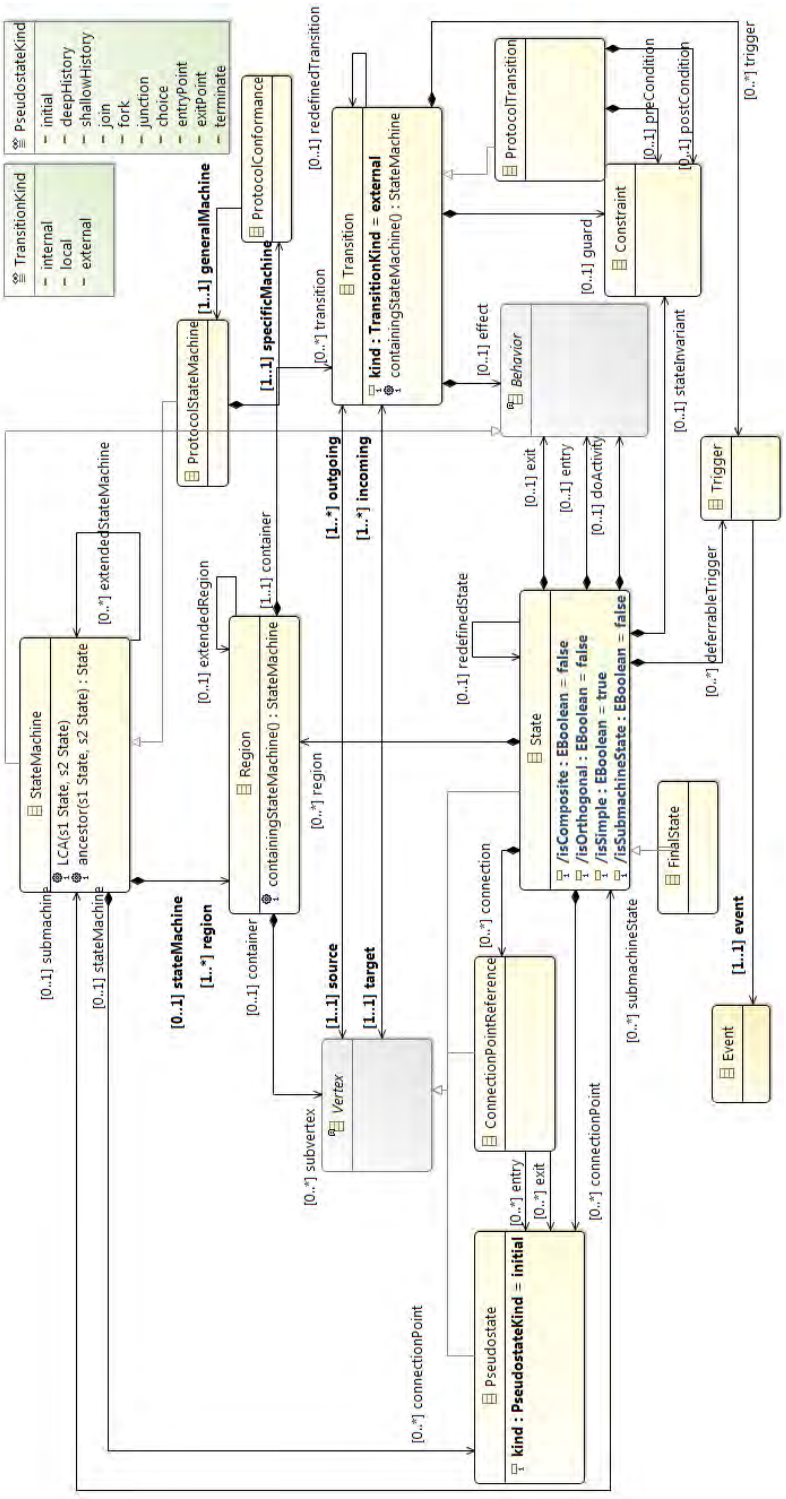

FIGURE 6.10 – La version 2.0 du métamodèle StateMachine (depuis [142])

6.5.2 Données

Nous avons étudié l'évolution du métamodèle State Machine de la version 1.5 à 2.0. Pour les deux versions, nous avons utilisé la spécification officielle du métamodèle fournie dans [14]. Ensuite, pour les deux versions, nous avons construit un métamodèle ECORE et un fichier ".ocl" contenant les contraintes OCL associées à ce métamodèle.

Le métamodèle 1.5 (Table 6.12) est composé de 47 éléments de modélisation : 14 classes, 1 énumération, 5 attributs, 2 opérations et 25 associations. Sur ce métamodèle, 30 contraintes OCL ont été spécifiées [141]. Sur ces 30 contraintes, nous n'avons pu en utiliser que 25. Les 5 autres contraintes ont été supprimées soit parce qu'elles ne sont pas pu être parsées (elles contiennent des erreurs), soit parce qu'elles concernent des éléments de modélisation contenus dans d'autres packages du métamodèle UML qui n'ont pas été pris en compte dans cette étude de cas. D'autre part, le métamodèle 2.0 (Tableau 6.12) contient 69 éléments de modélisation et 49 contraintes OCL [142]. Seules 38 des 49 contraintes ont été utilisées dans cette étude. 11 contraintes OCL ont été exclues pour les mêmes raisons que celles mentionnées ci-dessus.

TABLE 6.12 – Caractéristiques de StateMachine pour ses deux versions 1.5 et 2.0

	Metamodel						Ocl	
StateMachine	Classe	énumération	Attribut	Opération	Association	**Total**	Correctes	Incorrec
Version 1.5	14	1	5	2	25	**47**	25	5
Version 2.0	15	2	6	6	40	**69**	38	11

Une fois les métamodèles et les contraintes OCL collectés, nous avons identifié l'ensemble des opérations d'évolution qui font évoluer le métamodèle Statemachine 1.5 vers 2.0. Nous avons comparé les deux versions du métamodèle et extrait toutes les opérations d'évolution. Nous avons identifié 73 opérations d'évolution (Tableau 6.13). 63% (46 sur 73) de ces opérations visent à ajouter de nouveaux éléments de modélisation dans le métamodèle. 14% (10 sur 73) sont des opérations de suppression et 10% (7 sur 73) des opérations de renommage. D'autres opérations (13%) concernent la manipulation de propriétés telles que pullUp attribut/propriété, pushDown attribut/propriété, déplacer la propriété, etc.

TABLE 6.13 – Liste des opérations d'évolution

Ajouter class	4	Retirer association	4
Ajouter association	23	Restreindre propriété	2
Ajouter attribut	4	Généraliser propriété	1
Ajouter opération	4	Extraire classe	1
Ajouter énumération	1	déplacer propriété	2
Ajouter littéral	6	PushDown Association	1
Ajouter généralisation	4	PullUp attribut	1
Retirer classe	4	PullUp association	2
Retirer attribut	1	Renommer Élément	7
Retirer généralisation	1	**Total**	**73**

6.5.3 Traitement des données & méthode

Pour exécuter automatiquement ces 73 opérations sur notre outil, nous avons développé un programme Java. Ce programme charge le métamodèle 1.5 avec ses 25 contraintes et applique les opérations d'évolution une par une. L'application de chaque opération renvoie trois résultats : la modification du métamodèle en intégrant les changements générés par l'opération, l'adaptation des contraintes OCL après l'application de l'opération, et enfin, la génération d'un ensemble de notifications liées aux occurrences de MIS. Après l'application de toutes les 73 opérations sur le métamodèle 1.5, nous avons obtenu :

1. un métamodèle évolué correspondant à la version 2.0 ;
2. un ensemble de contraintes adaptées ;
3. un ensemble de notifications pour les nouvelles contraintes potentielles (correspondant aux MIS identifiés).

Pour répondre à la première question de recherche, nous avons calculé le nombre de contraintes OCL co-évoluées automatiquement ($COEVOL$) par notre outil, et les avons comparées aux contraintes présentes dans la version 2.0 (coévolution faite par les concepteurs UML). Comme résultat de cette comparaison, nous avons obtenu 4 ensembles : les contraintes co-évoluées par notre méthode et par les concepteurs UML ($COEVOL_{TP}$) ; contraintes co-évoluées par notre outil et non co-évoluées par les concepteurs UML ($COEVOL_{FP}$), contraintes non co-évoluées par notre outil et non co-évoluées par les concepteurs UML ($COEVOL_{TN}$), et contraintes non co-évoluées par notre outil mais co-évoluées par les concepteurs UML ($COEVOL_{FN}$). Avec ces quatre ensembles, nous avons calculé la précision et le rappel de notre approche, comme suit :

$$Precision_{COEVOL} = \frac{COEVOL_{TP}}{COEVOL_{TP} + COEVOL_{FP}} \tag{6.1}$$

$$Recall_{COEVOL} = \frac{COEVOL_{TP}}{COEVOL_{TP} + COEVOL_{FN}} \tag{6.2}$$

Pour répondre à la deuxième question de recherche, nous avons calculé le nombre d'occurrences de MIS notifiées générées par notre outil. Ensuite, pour chaque notification, nous avons recherché une contrainte OCL qui a été ajoutée ou supprimée dans la version 2.0, et qui évite l'occurrence MIS indiquée par notre outil. En effet, les notifications générées par notre outil émettent un soupçon qu'une contrainte doit être ajoutée ou supprimée après l'application d'une opération d'évolution. Cette comparaison a donné lieu à trois ensembles : 1) un ensemble de MIS pour lesquels des contraintes OCL ont été ajoutées par les concepteurs UML dans la version 2.0 (MIS_{TP}) ; 2) un ensemble de MIS pour lesquels des contraintes n'ont pas été ajoutées par les concepteurs UML (MIS_{FP}) ; 3) un ensemble de contraintes ajoutées par les concepteurs UML qui ne correspondent pas aux MIS identifiés (MIS_{FN}). Avec ces trois ensembles, nous avons calculé la précision et le rappel de notre méthode.

$$Precision_{MIS} = \frac{MIS_{TP}}{MIS_{TP} + MIS_{FP}} \tag{6.3}$$

$$Recall_{MIS} = \frac{MIS_{TP}}{MIS_{TP} + MIS_{FN}} \tag{6.4}$$

6.5.4 Résultats

6.5.4.1 QR1. Quelles sont les performances de notre approche dans la coévolution des contraintes OCL suite à l'évolution de leur métamodèle ?

L'application des 73 opérations d'évolution sur le métamodèle StateMachine 1.5 a permis d'adapter 15 contraintes OCL à partir des 25 contraintes existantes. Nous avons obtenu une précision de 1 et un rappel de 1. La précision montre que les contraintes coévoluées par notre outil ont bien été co-évoluées manuellement par les concepteurs UML. Ce résultat est très intéressant car la coévolution de ces 15 contraintes OCL n'a pris que quelques secondes, ce qui représente un gain de temps considérable par rapport à une coévolution manuelle. Ainsi, nous sommes en mesure de valider l'hypothèse 1.1 et ainsi

de rejeter l'hypothèse 1.2.

Au cours de cette expérience, nous avons remarqué que les concepteurs UML ont adopté d'autres coévolutions que celles proposées par notre outil. Par exemple, lors de la suppression d'un élément de métamodélisation, notre outil propose la suppression des contraintes liées à cet élément. Cependant, dans certains cas, les concepteurs UML ne suppriment pas les contraintes mais les réécrivent. Prenons l'exemple de la contrainte suivante.

context Transition
inv inv :(self.stateMachine->notEmpty() **and not**
oclIsKindOf(self.stateMachine, ActivityGraph)) **implies**
self.target**.oclIsKindOf**(Pseudostate) **implies**
((self.target**.oclAsType**(Pseudostate).kind = #join) **implies**
(self.source**.oclIsKindOf**(State)))

Notre outil proposait de supprimer cette contrainte car elle fait référence à l'association *stateMachine* entre la classe *Transition* et la classe *StateMachine*, qui a été supprimée dans la version 2.0 du métamodèle StateMachine. Toutefois, les concepteurs UML ont préféré la modifier plutôt que de la supprimer comme suit.

context Transition
inv inv : ((self.target**.oclAsType**(Pseudostate).kind = #join)
and (self.source**.oclIsKindOf**(State)))

D'autre part, la coévolution automatique a pu détecter des contraintes qui devraient être co-évoluées et qui ont été omises par les concepteurs UML. En effet, lors de l'application de l'opération d'évolution *Supprimer l'association "top" de la classe "StateMachine"*, notre outil a supprimé la contrainte suivante.

context Transition
inv inv : self.source**.oclIsKindOf** (Pseudostate) **implies**
(self.source**.oclAsType**(Pseudostate).kind = #initial)
implies (self.source.container = self.stateMachine.top)
implies ((self.trigger-> isEmpty()) **or**

(self.trigger.stereotype.name =' create')

Cependant, cette contrainte n'a pas été supprimée dans la version 2.0 de StateMachine et elle fait partie des 11 contraintes non analysées. Il est également surprenant de constater que cette contrainte apparaît dans toutes les autres versions de StateMachine (de 2.0, 2.1.2, 2.2, 2.3 et 2.4) sans être corrigée. Elle a été corrigée par les concepteurs d'UML dans la dernière version d'UML, à savoir la version 2.5. Il a fallu 10 ans aux concepteurs UML pour corriger cette contrainte alors que notre outil l'a immédiatement détectée. Ces résultats montrent que la coévolution manuelle des contraintes OCL est un processus long et source d'erreurs. L'assistance des concepteurs pendant l'évolution des métamodèles par la coévolution automatique des contraintes OCL est donc un processus très utile.

6.5.4.2 RQ2. Quelles sont les performances dans la notification de nouvelles contraintes potentielles liées au MIS ?

En ce qui concerne les MIS, notre outil a généré 103 notifications de MIS potentiels. Sur ces 103 notifications ($MIS_{TP} + MIS_{FP}$), 20 (MIS_{TP}) concernaient en fait des contraintes qui ont été ajoutées dans la version 2.0 par les concepteurs UML. Avec ces performances, nous avons obtenu une précision de 0.20 et un rappel de 0.62 ($MIS_{FN} = 12$). A la lumière de ces résultats, il nous semble important d'améliorer les performances de notre approche notamment en ce qui concerne le nombre de faux positifs générés (réduction du nombre de notifications). En effet, nous avons constaté que certaines notifications se chevauchaient et étaient donc comptées deux fois. Cependant, en regardant les résultats plus en détail, d'autres facteurs ont également eu un impact sur les résultats.

Les faux négatifs (les contraintes qui ne correspondent pas au MIS) ne sont pas vraiment des faux négatifs, car notre approche ne recherche que le MIS et non toutes les contraintes ajoutées dans la version 2.0. En effet, les 12 contraintes ajoutées par les experts et qui ne correspondent pas au MIS, sont des contraintes spécifiques et fortement dépendantes du domaine métier et non liées au MIS. Si nous considérons les faux négatifs comme toutes les contraintes MIS que notre outil n'a pas pu notifier, alors nous obtenons un rappel de 1. Notre outil peut identifier tous les MIS liés à une opération d'évolution.

Nous avons également constaté que certains faux positifs (notification de MIS mais aucune contrainte écrite pour ce MIS) peuvent être omis par les concepteurs UML. En effet,

nous avons constaté que 3 MIS identifiés par notre outil ont été complétés dans la version 2.5 par 3 contraintes OCL. Ces MIS concernaient l'ajout de l'énumération *Kind* dans la classe *Transition*. Nous sommes convaincus que d'autres MIS nécessitent des contraintes OCL mais que les experts les ont oubliées. Par exemple, l'opération *Ajouter l'association "connectionPoint" de State à "Pseudostate"* a généré un MIS potentiel qui souligne la nécessité d'ajouter une contrainte OCL afin de restreindre la multiplicité de l'association *connectionPoint* pour la sous-classe *FinalState*. En d'autres termes, peut-être que *FinalState* ne devrait pas avoir de Points de connexion. Bien que cela semble logique, nous n'avons trouvé aucune contrainte OCL dans la spécification qui réponde à ce MIS. Cependant, nous sommes convaincus qu'une partie de ce MIS nécessite une contrainte.

Finalement, nous avons constaté que les concepteurs d'UML ont ajouté 23 nouvelles contraintes OCL entre la version 1.5 et la version 2.0. Sur ces 23, notre outil a été capable de notifier 11 contraintes, ce qui correspond à 48% des nouvelles contraintes ajoutées. Sachant cela, nous sommes convaincus que certaines contraintes ont été omises dans la spécification. Compte-tenu des performances de notre approche de notification de MIS, l'hypothèse 2.2 est validée et l'hypothèse 2.1 est rejetée.

6.5.5 Menaces à la validité

Dans cette section, nous discutons ci-dessous des stratégies adoptées pour éviter certaines menaces à la validité des résultats et assurer une certaine qualité de notre étude. Selon [143], les menaces à la validité d'une étude de cas sont : la validité de construction, la validité interne, la validité externe et la fiabilité.

6.5.5.1 La validité de construction

La validité de construction garantit que les mesures effectuées permettent de mesurer ce que l'on a voulu mesurer. Au niveau de la coévolution des contraintes OCL, il n'y a pas de menaces potentielles de ce type. En effet, nous avons comparé les contraintes OCL. Le langage OCL étant formel, il n'y a pas de risque de confusion ou d'ambiguïté. Cependant, concernant la notification des MIS, un risque peut émerger car nous avons comparé les contraintes existantes, écrites en OCL, avec les notifications (forme textuelle). Pour éviter ce risque, nous avons défini un modèle de contrainte pour chaque MIS qui nous permet de l'éviter. Nous avons ensuite comparé ces modèles avec les contraintes écrites par les experts.

6.5.5.2 La validité interne

La validité interne concerne la cohérence des conclusions tirées sur les causes et les effets entre les résultats. Notre étude est moins concernée par ce type de validité.

6.5.5.3 La validité externe

La validité externe fait référence à la généralisation des conclusions à d'autres données, domaines, etc. Nous avons étudié cette validité selon deux axes : la coévolution OCL et la notification MIS. Sur l'aspect de la coévolution OCL, nous pouvons dire que les résultats obtenus peuvent être généralisés à d'autres métamodèles. Par contre, pour ce qui est de la notification MIS, les résultats ne peuvent pas être généralisés, car ils dépendent fortement du type d'opérations d'évolution appliquées. En effet, dans [144], des études ont montré que le nombre de MIS dans un métamodèle dépend fortement de ses éléments de métamodélisation et donc du type d'opération d'évolution appliquée. Pour éviter cette menace, nous avons choisi une étude de cas avec un ensemble large et varié d'opérations d'évolution. En fait, l'étude de cas nous a permis d'appliquer 18 types d'opérations d'évolution parmi les 31 mentionnées précédemment.

6.5.5.4 La fiabilité

La fiabilité de l'étude consiste en la capacité des résultats à être reproduits et répliqués par d'autres chercheurs. Pour augmenter la fiabilité de notre étude, nous avons choisi d'utiliser un méta-modèle connu par la communauté et toute la documentation de cette étude de cas est librement accessible via Internet. Nous avons également fourni dans cet article, l'ensemble des opérations évolutives et leur impact sur le MIS.

6.5.6 Conclusion

Dans cette section, nous avons réalisé une étude de cas avec le métamodèle State Machine de UML en comparant la version 1.5 avec la version 2.0. Le but de cette étude de cas est d'évaluer les performances de notre approche pour identifier les MIS en utilisant l'ensemble des opérateurs d'évolution du métamodèle. Les résultats ont montré que sur les 23 nouvelles contraintes OCL, notre approche a été capable de notifier le besoin de définir 48% d'entre elles, ce qui est un résultat prometteur. Les performances de notre approche sont encore à améliorer, notamment pour réduire le nombre de faux positifs. En effet, nous avons effectué une recherche MIS exhaustive sur l'ensemble du métamodèle, sans utiliser d'autre technique pour réduire les faux positifs sans perdre en précision.

6.6 En résumé

Dans le présent chapitre, nous avons présenté les différentes évaluations réalisées dans l'objectif de valider nos contributions de thèse. En effet, nous avons opté pour une étude approfondie pour la validation de nos MIS comportant à la fois une évaluation quantitative et qualitative. Nous avons également mené une étude pour évaluer les performances de nos contributions portant sur la recherche automatique ainsi que sur l'identification de nos MIS. Nous avons présenté les résultats obtenus ainsi que les différentes limites que présentent nos études. Celles-ci seront d'avantage détaillées dans le chapitre suivant, section perspectives, afin de présenter quelques améliorations possibles.

Quatrième partie

Conclusion et Perspectives

Chapitre 7

Conclusion et perspectives

Dans ce dernier chapitre, nous discutons les contributions présentées dans notre dissertation, les différentes limites, ainsi que les perspectives de notre travail.

7.1 Contributions majeures

7.1.1 Étude empirique sur l'investigation des structures imprécises dans les métamodèles

Nous avons réalisé une étude empirique sur le métamodèle UML 2.5 (tous les métamodèles qui le composent) avec ces 450 contraintes OCL. L'objectif était d'étudier les structures du métamodèle qui étaient précisées avec des contraintes OCL dans l'optique d'identifier les structures récurrentes qui présentent des faiblesses au niveau de leur sémantique, et qui nécessitent souvent des contraintes OCL. Ceci a permis d'identifier 10 structures que nous avons appelées MIS (*Metamodel Inacurrate Structures*) . Ainsi, pour chaque MIS, nous avons défini :

1. Le(s) problème(s) qui est(sont) à l'origine du manque de précision de ces structures, et pour lequel/lesquels les contraintes sont spécifiées et illustrées par un exemple.
2. une ou plusieurs questions pour chaque MIS. Ces dernières sont utiles pour un concepteur débutant souhaitant étudier son métamodèle pour spécifier les contraintes OCL.
3. Un ou plusieurs patterns OCL par MIS, qui peuvent être utilisés pour préciser la sémantique statique de chaque MIS lorsqu'il s'avère vrai.

Après la proposition de l'ensemble des MIS, nous avons réalisé trois évaluations :

1. *Une évaluation quantitative de l'ensemble des MIS* : l'objectif était de reproduire le même processus expérimental qui a permis d'obtenir les MIS sur un autre en-

semble de données. Cette démarche visait à connaître si l'ensemble de MIS identifié dans l'étude du métamodèle UML apparaissait dans d'autres métamodèles avec des contraintes OCL. Dans ce cas, ces structures ont le mérite d'être identifiés comme étant des structures souvent problématiques. Les résultats de cette évaluation ont montré que 48% des contraintes de l'ensemble des métamodèles de validation étaient liées aux MIS. Aussi, l'étude a permis de retrouver 7 des 9 MIS accompagnés de contraintes OCL.

2. *Une évaluation qualitative de l'ensemble des MIS* : l'objectif était d'évaluer l'utilité des MIS pour les concepteurs de métamodèles. Pour se faire, nous avons réalisé une expérimentation contrôlée sous la forme de projets avec des étudiants en 4ème année informatique de l'école polytechnique de Montpellier. Les étudiants devaient étudier deux métamodèles Activités et State Machine afin d'identifier des parties du métamodèles à contraindre, puis d'écrire les contraitnes OCL nécessaires aux métamodèles. Nous avons réparti les étudiants en deux groupes :
   — Un premier groupe d'étudiants ayant le métamodèle à étudier, une spécification du métamodèle avec les détails conceptuels de chaque métamodèle ainsi que la syntaxe concrète de chaque métaclasse si possible.
   — Un deuxième groupe ayant les mêmes informations que le premier groupe, mais avec une documentation sur l'ensemble des MIS. Celle-ci contient la définition de chaque MIS avec des exemples illustratifs.

   Les résultats ont montré que le groupe avec la documentation des MIS avait trouvé plus de structures à contraindre, et a ainsi écrit plus de contraintes OCL. En moyenne, les étudiants du deuxième groupe ont obtenu un score supérieur aux autres étudiants sans documentation. Ceci nous a permis de conclure que la connaissance des MIS a été utile pour ce groupe d'individus.

3. *Une évaluation de l'utilité d'un outil faisant la recherche automatique des MIS* : suite à l'évaluation de l'utilité des MIS, nous avons évalué les performances d'un outil d'assistance qui recherche automatiquement toutes les occurrences de MIS. Pour cela, nous avons proposé un outil basé sur la logique du premier ordre. Ce dernier transforme le métamodèle en une base de connaissances Prolog, puis recherche toutes les occurrences des MIS décrits sous forme de requêtes. Cette validation a été appliquée sur 8 métamodèles de tailles différentes et de différentes prove-

nances. Les résultats ont montré que selon la taille du métamodèle et les éléments de modélisation qui le composent, les résultats étaient très différents. Le principal inconvénient de la recherche automatique est le nombre de faux positifs. En effet, l'outil fait une recherche exhaustive de toutes les instances de MIS. Ensuite, le concepteur devra parcourir chacune des instances retournées pour vérifier si celle-ci nécessite une contrainte OCL. Sur l'ensemble des métamodèles étudiés, le taux d'instances de MIS ayant nécessité une contrainte OCL était au plus y%, ce qui est assez faible. Ainsi, cette étude nécessite des améliorations qui sont détaillées dans les perspectives ci-dessous.

7.1.2 Identification de MIS pendant la coévolution de contraintes OCL

À l'aide de l'ensemble des MIS que nous avons identifié, nous avons proposé une approche complémentaire aux approches existantes de coévolution de contraintes OCL. En effet, nous avons proposé une approche d'identification de MIS pendant la coévolution de contraintes OCL, engendrée par l'évolution de leur métamodèle. Pour cela, nous avons étudié chacun des opérateurs d'évolution de métamodèles dans l'optique de déterminer l'ensemble des MIS que celui-ci peut engendrer. Cette étude nous a permis de proposer des relations causales entre les opérateurs d'évolution et les MIS. Suite à cette proposition, nous avons évalué les performances de notre approche combinée à une approche de coévolution de contraintes OCL. L'évaluation a été appliquée sur le métamodèle State Machine d'UML en raison du nombre important des opérateurs d'évolution de métamodèle qui lui ont été appliqués. D'après les résultats obtenus, notre approche a permis de co-évoluer 60% des contraintes présentes dans l'ancienne version du métamodèle. En plus, notre approche d'identification de MIS a contribué à identifier parmi 103 notifications de nouveaux MIS, 20 nouvelles contraintes OCL rajoutées dans la nouvelle version du métamodèle. Pour résumer, nous avons obtenu 20% de précision et 63% de rappel pour cette approche. Bien que l'approche nous ait permis d'identifier 48% des nouvelles contraintes OCL spécifiées sur la nouvelle version du métamodèle, nous estimons qu'elle souffre du même problème que nous avons mentionné pour la recherche automatique de MIS : celui du nombre élevé de faux positifs.

7.2 Perspectives

Les contributions de la thèse et les limites observées ouvrent la voie à diverses améliorations et perspectives. Dans ce qui suit, nous donnons les différentes pistes que nous envisageons pour le futur.

7.2.1 Amélioration à apporter à l'ensemble des MIS

Concernant l'étude des MIS, un de nos objectifs serait d'utiliser des techniques d'apprentissage automatique afin de compléter l'ensemble que nous avons proposé dans cette dissertation. En effet, l'idée serait de faire un apprentissage sur un ensemble conséquent de métamodèles et de contraintes OCL bien formées afin d'identifier d'autres structures que nous n'avons pas pu obtenir avec notre recherche manuelle sur le dataset étudié. Contrairement à l'étude manuelle menée dans notre première contribution, une étude automatique permettrait d'être reproduite facilement.

7.2.2 Classification de métamodèles

Un autre avantage de l'apprentissage automatique est la classification des métamodèles en familles. En effet, nous avons constaté qu'il existait plusieurs façons possibles de spécifier un métamodèle selon les préférences du concepteur. Par exemple, suite à l'application de métriques à l'aide de l'outil [145] sur certains des métamodèles étudiés précédemment, les données ont mis en évidence une différence entre ces derniers. Par exemple, le métamodèle UML qui est très factorisé et qui contient très peu de redondances a une profondeur d'héritage très importante (Profondeur de l'arbre d'héritage (DIT) = 10), tandis que d'autres métamodèles tels que RBAC contient beaucoup de propriétés redondantes mais dans lequel l'héritage n'a pas été utilisé. De l'autre côté, les contraintes OCL liées à l'héritage représentent 53% du total des contraintes liées aux MIS du métamodèle UML, alors que le métamodèle RBAC contient plus de contraintes liées aux cycles et aux chemins du fait qu'il soit moins modulaire qu'UML. D'après ces analyses préliminaires, nous suspectons l'existence de familles de métamodèles qui seraient utiles à intégrer dans notre étude. Par conséquent, nous envisageons de proposer une classification de métamodèle en fonction de leur structure. Ce découpage pourra aussi nous permettre de guider la recherche automatique de MIS dans les métamodèles. A titre d'exemple, l'approche proposée se base sur une recherche exhaustive de MIS ce qui est à la fois coûteux en termes de

ressources logicielles mais aussi donne lieu à un nombre conséquent de faux positifs. Ainsi, il serait intéressant de focaliser la recherche sur certains MIS pour une famille particulière de métamodèle et d'en restreindre d'autres pour la même famille.

7.2.3 Validation des MIS

Afin de mieux étudier l'utilité de l'ensemble des MIS, il serait intéressant de les valider avec des concepteurs seniors ayant une connaissance plus approfondie des MIS. En effet, une étude sous forme de sondage dans lequel les concepteurs décrivent leur perception du problème causé par chaque MIS serait d'une grande utilité. D'abord, il nous serait possible suite à ce travail de proposer des coefficients qui décrivent l'impact de chaque MIS, sa gravité, ainsi que sa fréquence d'apparition dans les métamodèles. Ces différents coefficients pourraient ensuite être utilisées pour l'amélioration de notre outil de recherche automatique des instances de MIS. Plus précisément, au lieu d'effectuer une recherche exhaustive de chaque instance de MIS, la recherche automatique sera guidée par des heuristiques liées à la gravité de chaque MIS, où les MIS ayant le plus d'impact négatif seront recherchés exhaustivement (ou en premier), tandis que les autres MIS seront désavantagés.

7.2.4 Utilisation des MIS pour enseigner les langages MOF et OCL

En plus des deux utilisations possibles des MIS (pendant la création du métamodèle puis son évolution), leur utilisation à des fins pédagogiques pourrait faciliter l'enseignement des langages MOF et OCL. En effet, un moyen d'inciter les étudiants à écrire des contraintes OCL serait de leur montrer les limites du langage MOF. Les MIS sont un exemple montrant des faiblesses du langage MOF. Sensibiliser les étudiants sur chaque MIS avec les différents problèmes qu'il pourrait engendrer au niveau des instances de modèles s'il n'est pas contraint pourrait mettre en avant le rôle du langage OCL. De plus, il serait intéressant de mettre en place une check-list qui englobe toutes les structures ayant éventuellement besoin de contraintes OCL, notamment l'ensemble de MIS. Ainsi, les étudiants auront à suivre la liste de structures à étudier pour préciser la sémantique statique du métamodèle.

7.2.5 Amélioration de la recherche automatique de MIS

En plus de tous les points abordés précédemment qui pourraient être exploités pour superviser la recherche automatique de MIS, nous pourrons envisager de mettre en place deux stratégies pouvant réduire le nombre de faux positifs lors de la recherche de MIS, et ainsi de proposer un outil d'assistance exploitable sur le terrain. En effet, dans [19] les auteurs ont cherché à étudier la distribution des contraintes OCL sur les métamodèles. Les résultats trouvés ont montré que pour 80% des métamodèles étudiés, 80% des contraintes OCL qui leur sont spécifiés ne couvrent que 25% de l'ensemble des classes. Par conséquent, il serait intéressant de donner la main au concepteur pour procéder à un découpage de son métamodèle selon des critères. À titre d'exemple, comme illustré dans la figure 7.1, le découpage est réalisé en fonction de l'importance des classes du métamodèle (figure de droite), ou bien en fonction du lien sémantique qu'il y a entre les différentes classes (figure gauche).

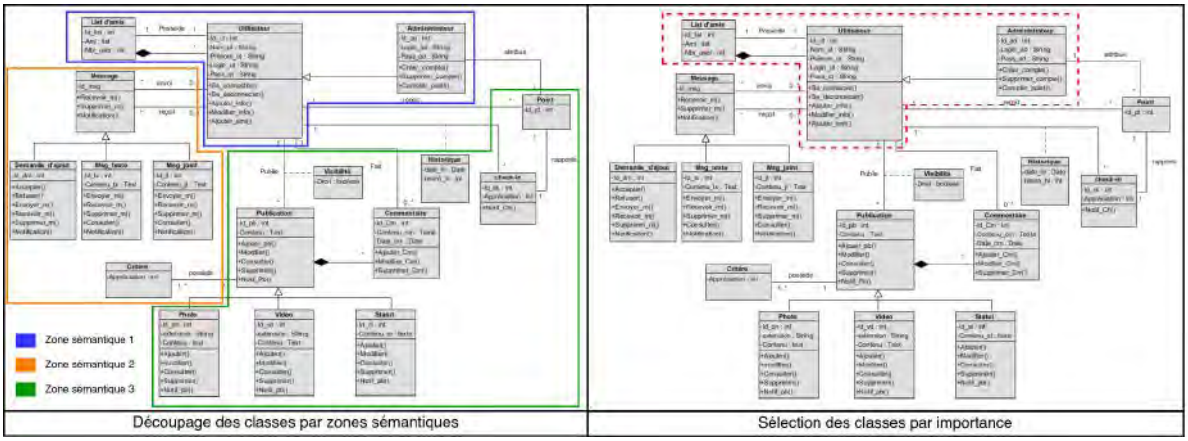

FIGURE 7.1 – Exemples du découpage de métamodèle

7.2.6 Identification de MIS pendant la coévolution de contraintes OCL

Les performances de cette approche reposent sur la complétude de l'ensemble des MIS. Ainsi, l'amélioration de l'ensemble de MIS et la façon de procéder à leur recherche seront des facteurs très importants dans l'obtention de meilleurs résultats. De plus, bien que les résultats de la validation de l'approche soient prometteurs, nous envisageons de valider l'approche sur un ensemble de métamodèles plus large.

Compte-tenu du nombre important de faux positifs retourné par notre outil d'identification de MIS pendant l'évolution du métamodèle, nous jugeons intéressant de réaliser une étude approfondie sur différentes versions de métamodèles dans lesquelles nous nous intéresserons particulièrement aux contraintes rajoutées suite à l'évolution du métamodèle. Effectivement, au niveau code, il existe une multitude d'études qui concernent l'analyse des différentes versions de code dans le but d'étudier l'introduction de vulnérabilités suite à un commit. Cependant, on ne retrouve aucun travail qui s'intéresse à l'introduction de structures imprécises suite à l'évolution du métamodèle. Ainsi, il serait intéressant d'étudier particulièrement les nouvelles contraintes OCL qui sont introduites à la suite d'une évolution de métamodèle. Cette étude pourra permettre de raffiner certains MIS qui sont propres à l'évolution, mais aussi de permettre de désigner des poids pour chaque MIS afin de guider l'identification automatique et ainsi réduire le taux de faux positifs.

Les travaux présentés au cours de cette thèse ont fait l'objet de deux papiers acceptés et un papier en cours de soumission :

1. Cherfa, E., Kesraoui, S., Tibermacine, C., Fleurquin, R., & Sadou, S. (2020, March). On investigating metamodel inaccurate structures. In Proceedings of the 35th Annual ACM Symposium on Applied Computing (pp. 1642-1649).
2. Cherfa, E., Mesli-Kesraoui, S., Tibermacine, C., Fleurquin, R., & Sadou, S. Identifying Metamodel Inaccurate Structures During Metamodel/Constraint Co-Evolution. In Proceedings of the 24th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (En cours de publication en ligne)
3. Cherfa, E., Kesraoui, S., Tibermacine, C., Fleurquin, R., & Sadou, S. Metamodel Inaccurate Structures. Journal of Systems and Software (en cours de soumission)

BIBLIOGRAPHIE

[1] Adrian MACKENZIE. *Cutting code : Software and sociality*. T. 30. Peter Lang, 2006.

[2] James M KELLER. « Reading up books : Software engineering—The critical component for computing : The goal of software engineering is to provide structures and techniques that will lead to the production of high-quality programs ». In : *IEEE Potentials* 2.December (1983), p. 33-35.

[3] Steven J VAUGHAN-NICHOLS. « Building better software with better tools ». In : *Computer* 36.9 (2003), p. 12-14.

[4] Antonio CICCHETTI et al. « On the concurrent versioning of metamodels and models : challenges and possible solutions ». In : *Proceedings of the 2nd International Workshop on Model Comparison in Practice*. 2011, p. 16-25.

[5] Anneke G KLEPPE et al. *MDA explained : the model driven architecture : practice and promise*. Addison-Wesley Professional, 2003.

[6] Stephen J MELLOR et al. *MDA distilled : principles of model-driven architecture*. Addison-Wesley Professional, 2004.

[7] Gonzalo GÉNOVA, María Cruz VALIENTE et Mónica MARRERO. « On the difference between analysis and design, and why it is relevant for the interpretation of models in Model Driven Engineering. » In : *J. Object Technol.* 8.1 (2009), p. 107-127.

[8] Juha-Pekka TOLVANEN et Steven KELLY. « MetaEdit+ defining and using integrated domain-specific modeling languages ». In : *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. 2009, p. 819-820.

[9] Steven KELLY et Juha-Pekka TOLVANEN. *Domain-specific modeling : enabling full code generation*. John Wiley & Sons, 2008.

[10] Anneke KLEPPE. *Software language engineering : creating domain-specific languages using metamodels*. Pearson Education, 2008.

[11] Marjan Mernik, Jan Heering et Anthony M Sloane. « When and how to develop domain-specific languages ». In : *ACM computing surveys (CSUR)* 37.4 (2005), p. 316-344.

[12] Object Management Group. *Meta-Object Facility 2.5.1.* `https://www.omg.org/spec/MOF/2.5.1/`. 2016.

[13] Object Management Group. *Object Constraint Language 2.4.* `https://www.omg.org/spec/OCL/2.4/`. 2014.

[14] Object Management Group. `https://www.omg.org/`.

[15] Martin Faunes et al. « Automatically searching for metamodel well-formedness rules in examples and counter-examples ». In : *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2013, p. 187-202.

[16] Duc-Hanh Dang et Jordi Cabot. « Automating inference of ocl business rules from user scenarios ». In : *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*. T. 1. IEEE. 2013, p. 156-163.

[17] Michael Wahler, Jana Koehler et Achim D Brucker. « Model-driven constraint engineering ». In : *Electronic Communications of the EASST* 5 (2007).

[18] Dan Chiorean, Vladiela Petraşcu et Ileana Ober. « Testing-oriented improvements of OCL specification patterns ». In : *2010 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*. T. 2. IEEE. 2010, p. 1-6.

[19] Juan Cadavid, Benoit Combemale et Benoit Baudry. « Ten years of Meta-Object Facility : an analysis of metamodeling practices ». Thèse de doct. INRIA, 2012.

[20] Dan Chiorean, Vladiela Petrascu et Ileana Ober. « MDE-driven OCL Specification Patterns ». In : *Journal of Control Engineering and Applied Informatics* 14.1 (2012), p. 83-92.

[21] Ali Hamie. « Constraint specifications using patterns in OCL ». In : *International Journal on Computer Science and Information Systems* 8.1 (2013).

[22] Muhammad Hammad et al. « IOCL : An interactive tool for specifying, validating and evaluating OCL constraints ». In : *Science of Computer Programming* 149 (2017), p. 3-8.

[23] Alan W Brown. « Model driven architecture : Principles and practice ». In : *Software and systems modeling* 3.4 (2004), p. 314-327.

[24] Douglas C Schmidt. « Model-driven engineering ». In : *Computer-IEEE Computer Society-* 39.2 (2006), p. 25.

[25] Manny M Lehman. « Laws of software evolution revisited ». In : *European Workshop on Software Process Technology*. Springer. 1996, p. 108-124.

[26] John Hutchinson et al. « Empirical assessment of MDE in industry ». In : *Proceedings of the 33rd international conference on software engineering*. 2011, p. 471-480.

[27] Jochen Ludewig. « Models in software engineering–an introduction ». In : *Software and Systems Modeling* 2.1 (2003), p. 5-14.

[28] Jean-Marie Favre. « Megamodelling and etymology ». In : *Dagstuhl Seminar Proceedings*. 2006.

[29] Thomas Kühne. « Matters of (meta-) modeling ». In : *Software & Systems Modeling* 5.4 (2006), p. 369-385.

[30] Jean-Marie Favre. « Megamodeling and etymology-a story of words : From MED to MDE via MODEL in five milleniums ». In : *In Dagstuhl Seminar on Transformation Techniques in Software Engineering, number 05161 in DROPS 04101. IFBI*. Citeseer. 2005.

[31] E. Seidewitz. « What models mean ». In : *IEEE Software* 20.5 (2003), p. 26-32. doi : 10.1109/MS.2003.1231147.

[32] B. Selic. « The pragmatics of model-driven development ». In : *IEEE Software* 20.5 (2003), p. 19-25. doi : 10.1109/MS.2003.1231146.

[33] Jean Bézivin et Olivier Gerbé. « Towards a precise definition of the OMG/MDA framework ». In : *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. IEEE. 2001, p. 273-280.

[34] Franck Fleurey. « Langage et méthode pour une ingénierie des modèles fiable ». Theses. Université Rennes 1, oct. 2006. url : https://tel.archives-ouvertes.fr/tel-00538288.

[35] Mark Harman, S Afshin Mansouri et Yuanyuan Zhang. « Search-based software engineering : Trends, techniques and applications ». In : *ACM Computing Surveys (CSUR)* 45.1 (2012), p. 1-61.

[36] James R Williams. « A novel representation for search-based model-driven engineering ». Thèse de doct. University of York, 2013.

[37] Shane Sendall et Wojtek Kozaczynski. « Model transformation : The heart and soul of model-driven software development ». In : *IEEE software* 20.5 (2003), p. 42-45.

[38] Eric Umuhoza et al. « Automatic code generation for cross-platform, multi-device mobile apps : Some reflections from an industrial experience ». In : *Proceedings of the 3rd International Workshop on Mobile Development Lifecycle*. 2015, p. 37-44.

[39] Bran Selic. « The pragmatics of model-driven development ». In : *IEEE software* 20.5 (2003), p. 19-25.

[40] Bran Selic. « What will it take ? A view on adoption of model-based methods in practice ». In : *Software & Systems Modeling* 11.4 (2012), p. 513-526.

[41] John Hutchinson, Mark Rouncefield et Jon Whittle. « Model-driven engineering practices in industry ». In : *Proceedings of the 33rd International Conference on Software Engineering*. 2011, p. 633-642.

[42] Jordi Cabot et Martin Gogolla. « Object constraint language (OCL) : a definitive guide ». In : *International school on formal methods for the design of computer, communication and software systems*. Springer. 2012, p. 58-90.

[43] Jean Bézivin. « On the unification power of models ». In : *Software & Systems Modeling* 4.2 (2005), p. 171-188.

[44] Artur Boronat et José Meseguer. « An algebraic semantics for MOF ». In : *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2008, p. 377-391.

[45] Thomas Kühne et al. « Explicit transformation modeling ». In : *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2009, p. 240-255.

[46] Object Management Group (OMG). *Meta Object Facility Specification Version 2.4.1*. 2021. URL : `https://www.omg.org/spec/MOF/2.4.1/`.

[47] Object Management Group (OMG). *Unified Modeling Language (OMG UML), Superstructure v2.4.1*. 2021. URL : http://www.omg.org/spec/UML/2.4.1/.

[48] Jos B Warmer et Anneke G Kleppe. *The object constraint language : getting your models ready for MDA*. Addison-Wesley Professional, 2003.

[49] Charles Ashbacher. *The Object Constraint Language Second Edition, Getting Your Models Ready for MDA*. 2003.

[50] Martin Gogolla, Fabian Büttner et Mark Richters. « USE : A UML-based specification environment for validating UML and OCL ». In : *Science of Computer Programming* 69.1-3 (2007), p. 27-34.

[51] Martin Gogolla, Jörn Bohling et Mark Richters. « Validating UML and OCL models in USE by automatic snapshot generation ». In : *Software & Systems Modeling* 4.4 (2005), p. 386-398.

[52] Mathias Soeken et al. « Verifying UML/OCL models using Boolean satisfiability ». In : *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE. 2010, p. 1341-1344.

[53] Object Management Group. *Unified Modeling Language 2.5*. https://www.omg.org/spec/UML/2.5/, year = 2015.

[54] Juan José Cadavid Gómez. « Assistance à la méta-modélisation précise ». Thèse de doct. Rennes 1, 2012.

[55] Jean-Marie Favre. « Towards a basic theory to model model driven engineering ». In : *3rd workshop in software model engineering, wisme*. Citeseer. 2004, p. 262-271.

[56] Massimo Tisi et al. « On the use of higher-order model transformations ». In : *European Conference on Model Driven Architecture-Foundations and Applications*. Springer. 2009, p. 18-33.

[57] Keith H Bennett et Václav T Rajlich. « Software maintenance and evolution : a roadmap ». In : *Proceedings of the Conference on the Future of Software Engineering*. 2000, p. 73-87.

[58] Guido Wachsmuth. « Metamodel adaptation and model co-adaptation ». In : *European Conference on Object-Oriented Programming*. Springer. 2007, p. 600-624.

[59] Kelly Garcés et al. « Adaptation of models to evolving metamodels ». Thèse de doct. INRIA, 2008.

[60] Kelly Garcés et al. « Managing model adaptation by precise detection of metamodel changes ». In : *European Conference on Model Driven Architecture-Foundations and Applications*. Springer. 2009, p. 34-49.

[61] Norman F. Schneidewind. « The state of software maintenance ». In : *IEEE Transactions on Software Engineering* 3 (1987), p. 303-310.

[62] Mark Vigder. « The evolution, maintenance, and management of component-based systems ». In : *Component-Based Software Engineering : Putting the Pieces Together* (2001), p. 527-539.

[63] Ned Chapin et al. « Types of software evolution and software maintenance ». In : *Journal of software maintenance and evolution : Research and Practice* 13.1 (2001), p. 3-30.

[64] Michael W Godfrey et Daniel M German. « The past, present, and future of software evolution ». In : *2008 Frontiers of Software Maintenance*. IEEE. 2008, p. 129-138.

[65] William F Opdyke. « Refactoring object-oriented frameworks ». In : (1992).

[66] Robert S Arnold. « Software restructuring ». In : *Proceedings of the IEEE* 77.4 (1989), p. 607-617.

[67] Tom Mens et Tom Tourwé. « A survey of software refactoring ». In : *IEEE Transactions on software engineering* 30.2 (2004), p. 126-139.

[68] Gerson Sunyé et al. « Refactoring UML models ». In : *International Conference on the Unified Modeling Language*. Springer. 2001, p. 134-148.

[69] James R Williams, Richard F Paige et Fiona AC Polack. « Searching for model migration strategies ». In : *Proceedings of the 6th International Workshop on Models and Evolution*. 2012, p. 39-44.

[70] Louis M Rose et al. « Epsilon Flock : a model migration language ». In : *Software & Systems Modeling* 13.2 (2014), p. 735-755.

[71] Georg Hinkel et al. « An empirical study on the perception of metamodel quality ». In : *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. 2016, p. 145-152.

[72] Haohai Ma et al. « Applying OO Metrics to Assess UML Meta-models ». In : *«UML» 2004 — The Unified Modeling Language. Modeling Languages and Applications*. Sous la dir. de Thomas Baar et al. Berlin, Heidelberg : Springer Berlin Heidelberg, 2004, p. 12-26. isbn : 978-3-540-30187-5.

[73] Sandeep Purao et Vijay Vaishnavi. « Product metrics for object-oriented systems ». In : *ACM Computing Surveys (CSUR)* 35.2 (2003), p. 191-221.

[74] Shyam R Chidamber et Chris F Kemerer. « A metrics suite for object oriented design ». In : *IEEE Transactions on software engineering* 20.6 (1994), p. 476-493.

[75] Misha Strittmatter et al. « Challenges in the evolution of metamodels : Smells and anti-patterns of a historically-grown metamodel ». In : (2016).

[76] Ralf H Reussner et al. *Modeling and simulating software architectures : The Palladio approach*. MIT Press, 2016.

[77] Josh GM Mengerink et al. « Automated analyses of model-driven artifacts : obtaining insights into industrial application of MDE ». In : *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*. 2017, p. 116-121.

[78] Juri Di Rocco et al. « Mining Metrics for Understanding Metamodel Characteristics ». In : *Proceedings of the 6th International Workshop on Modeling in Software Engineering*. MiSE 2014. Hyderabad, India : Association for Computing Machinery, 2014, 55–60. isbn : 9781450328494. doi : 10.1145/2593770.2593774. url : https://doi.org/10.1145/2593770.2593774.

[79] Taciana Novo Kudo, Renato F. Bulcão Neto et Auri M. R. Vincenzi. « Toward a Metamodel Quality Evaluation Framework : Requirements, Model, Measures, and Process ». In : *Proceedings of the 34th Brazilian Symposium on Software Engineering*. SBES '20. Natal, Brazil : Association for Computing Machinery, 2020, 102–107. isbn : 9781450387538. doi : 10.1145/3422392.3422461. url : https://doi.org/10.1145/3422392.3422461.

[80] Jesús J López-Fernández, Esther Guerra et Juan De Lara. « Assessing the Quality of Meta-models. » In : *MoDeVVa@ MoDELS*. Citeseer. 2014, p. 3-12.

[81] Bruce Horn. « Constraint patterns as a basis for object oriented programming ». In : *Proc. ACM OOPSLA*. 1992, p. 218-233.

[82] Elita Miliauskaite et Lina Nemuraite. « Taxonomy of integrity constraints in conceptual models ». In : *IADIS virtual multi conference on computer science and information systems*. 2005, p. 247-254.

[83] Elita Miliauskaitė et Lina Nemuraitė. « Representation of integrity constraints in conceptual models ». In : *Information technology and control* 34.4 (2005).

[84] Dolors Costal et al. « Facilitating the Definition of General Constraints in UML ». In : *Model Driven Engineering Languages and Systems*. Sous la dir. d'Oscar Nierstrasz et al. Berlin, Heidelberg : Springer Berlin Heidelberg, 2006, p. 260-274. isbn : 978-3-540-45773-2.

[85] Michael S Wahler. « Using patterns to develop consistent design constraints ». Thèse de doct. ETH Zurich, 2008.

[86] Edouard Batot et Houari Sahraoui. « Injecting Social Diversity in Multi-objective Genetic Programming : The Case of Model Well-Formedness Rule Learning ». In : *International Symposium on Search Based Software Engineering*. Springer. 2018, p. 166-181.

[87] Duc-Hanh Dang et Jordi Cabot. « On Automating Inference of OCL Constraints from Counterexamples and Examples ». In : *Knowledge and Systems Engineering*. Springer, 2015, p. 219-231.

[88] I. S. Bajwa, B. Bordbar et M. G. Lee. « OCL Constraints Generation from Natural Language Specification ». In : *2010 14th IEEE International Enterprise Distributed Object Computing Conference*. Oct. 2010, p. 204-213. doi : 10.1109/EDOC.2010.33.

[89] Imran S Bajwa et Mark G Lee. « Transformation rules for translating business rules to OCL constraints ». In : *European Conference on Modelling Foundations and Applications*. Springer. 2011, p. 132-143.

[90] Object Management Group. *Semantics Of Business Vocabulary and Rules 1.4*. https://www.omg.org/spec/SBVR/1.4/. 2017.

[91] Louis M Rose et al. « An analysis of approaches to model migration ». In : *Proc. Joint MoDSE-MCCM Workshop*. 2009, p. 6-15.

[92] Boris Gruschko, Dimitrios Kolovos et Richard Paige. « Towards synchronizing models with evolving metamodels ». In : *Proceedings of the International Workshop on Model-Driven Software Evolution*. Amsterdam, The Netherlands. 2007, p. 3.

[93] Markus Herrmannsdoerfer, Sebastian Benz, Elmar Juergens et al. « COPE : A language for the coupled evolution of metamodels and models ». In : *1st International Workshop on Model Co-Evolution and Consistency Management*. 2008.

[94] Bart Meyers et al. « A generic in-place transformation-based approach to structured model co-evolution ». In : *Electronic Communications of the EASST* 42 (2012).

[95] Florian Mantz, Gabriele Taentzer et Yngve Lamo. « Co-Transformation of Type and Instance Graphs Supporting Merging of Types and Retyping ». In : *Electronic Communications of the EASST* 61 (2013).

[96] Gabriele Taentzer et al. « Customizable model migration schemes for metamodel evolutions with multiplicity changes ». In : *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2013, p. 254-270.

[97] Florian Mantz et al. « Co-evolving meta-models and their instance models : A formal approach based on graph transformation ». In : *Science of Computer Programming* 104 (2015), p. 2-43.

[98] Markus Herrmannsdoerfer, Sander D Vermolen et Guido Wachsmuth. « An extensive catalog of operators for the coupled evolution of metamodels and models ». In : *International Conference on Software Language Engineering*. Springer. 2010, p. 163-182.

[99] Slaviša Marković et Thomas Baar. « Refactoring OCL annotated UML class diagrams ». In : *Software & Systems Modeling* 7.1 (2008), p. 25-47.

[100] K. Hassam et al. « Assistance System for OCL Constraints Adaptation during Metamodel Evolution ». In : *2011 15th European Conference on Software Maintenance and Reengineering*. Mars 2011, p. 151-160. doi : 10.1109/CSMR.2011.21.

[101] Steffen Kruse. *Co-Evolution of Metamodels and Model Transformations : An operator-based, stepwise approach for the impact resolution of metamodel evolution on model transformations.* BoD–Books on Demand, 2015.

[102] Martin Fowler. *Refactoring : improving the design of existing code*. Addison-Wesley Professional, 2018.

[103] Wael Kessentini, Manuel Wimmer et Houari Sahraoui. « Integrating the Designer In-the-Loop for Metamodel/Model Co-Evolution via Interactive Computational Search ». In : *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. MODELS '18. Copenhagen, Denmark : Association for Computing Machinery, 2018, 101–111. ISBN : 9781450349499. DOI : 10.1145/3239372.3239375. URL : https://doi.org/10.1145/3239372.3239375.

[104] Wael Kessentini, Houari Sahraoui et Manuel Wimmer. « Automated metamodel/model co-evolution : A search-based approach ». In : *Information and Software Technology* 106 (2019), p. 49-67. ISSN : 0950-5849. DOI : https://doi.org/10.1016/j.infsof.2018.09.003. URL : https://www.sciencedirect.com/science/article/pii/S0950584918301915.

[105] Min Zhang et al. « Improving the precision of fowler's definitions of bad smells ». In : *2008 32nd Annual IEEE Software Engineering Workshop*. IEEE. 2008, p. 161-166.

[106] Christian Franz Josef Lange. « Assessing and Improving the Quality of Modeling : A series of Empirical Studies about the UML ». In : (2007).

[107] Dave Astels. « Refactoring with UML ». In : *Proc. of International Conference on eXtreme Programming and Flexible Process in Software Engineering 2002*. 2002.

[108] Thorsten Arendt et Gabriele Taentzer. « UML model smells and model refactorings in early software development phases ». In : *Universitat Marburg* (2010).

[109] Lorenzo Bettini et al. « Quality-Driven detection and resolution of metamodel smells ». In : *IEEE Access* 7 (2019), p. 16364-16376.

[110] Jordi Cabot et Ernest Teniente. « Transformation techniques for OCL constraints ». In : *Science of Computer Programming* 68.3 (2007), p. 179-195.

[111] Jesús Sánchez Cuadrado et al. « Deriving OCL optimization patterns from benchmarks ». In : *Electronic Communications of the EASST* 15 (2008).

[112] Jesús Sánchez Cuadrado. « A verified catalogue of OCL optimisations ». In : *Software and Systems Modeling* 19.5 (2020), p. 1139-1161.

[113] Alexandre Correa, Cláudia Werner et Márcio Barros. « An empirical study of the impact of OCL smells and refactorings on the understandability of OCL specifications ». In : *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2007, p. 76-90.

[114] Alexandre Correa et Cláudia Werner. « Refactoring object constraint language specifications ». In : *Software & Systems Modeling* 6.2 (2007), p. 113-138.

[115] Lu Hong et al. « Automated refactoring of OCL constraints with search ». In : *IEEE Transactions on Software Engineering* (2017).

[116] Muhammad Uzair Khan et al. « Aspectocl : using aspects to ease maintenance of evolving constraint specification ». In : *Empirical Software Engineering* 24.4 (2019), p. 2674-2724.

[117] Luis Reynoso et al. « Does object coupling really affect the understanding and modifying of OCL expressions ? » In : *Proceedings of the 2006 ACM symposium on Applied computing*. 2006, p. 1721-1727.

[118] Luis Reynoso et al. « Assessing the influence of import-coupling on OCL expression maintainability : A cognitive theory-based perspective ». In : *Information Sciences* 180.20 (2010), p. 3837-3862.

[119] Martin Gogolla et Timo Stüber. « Metrics for OCL Expressions : Development, Realization, and Applications for Validation ». In : *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems : Companion Proceedings*. MODELS '20. Virtual Event, Canada : Association for Computing Machinery, 2020. isbn : 9781450381352. doi : 10.1145/3417990.3419228. url : https://doi.org/10.1145/3417990.3419228.

[120] *UML-Based Specification Environment*. http://useocl.sourceforge.net/w/index.php/The_UML-based_Specification_Environment.

[121] Josh GM Mengerink et al. « A Case of Industrial vs. Open-source OCL : Not So Different After All. » In : *MODELS (Satellite Events)*. 2017, p. 472-474.

[122] Jeroen Noten, Josh GM Mengerink et Alexander Serebrenik. « A data set of OCL expressions on GitHub ». In : *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE. 2017, p. 531-534.

[123] Josh GM Mengerink, Jeroen Noten et Alexander Serebrenik. « Empowering OCL research : a large-scale corpus of open-source data from GitHub ». In : *Empirical Software Engineering* 24.3 (2019), p. 1574-1609.

[124] Slaviša Marković et Thomas Baar. « Refactoring OCL annotated UML class diagrams ». In : *International Conference On Model Driven Engineering Languages And Systems*. Springer. 2005, p. 280-294.

[125] Object Management Group. *MOF Query/View/Transformation 1.3*. `https://www.omg.org/spec/QVT/1.3/`. 2016.

[126] Kahina Hassam, Salah Sadou et Régis Fleurquin. « Adapting ocl constraints after a refactoring of their model using an mde process ». In : *9th edition of the BElgian-NEtherlands software eVOLution seminar (BENEVOL 2010)*. 2010, p. 16-27.

[127] Andreas Demuth, Roberto E Lopez-Herrejon et Alexander Egyed. « Supporting the co-evolution of metamodels and constraints through incremental constraint management ». In : *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2013, p. 287-303.

[128] Andreas Demuth, Roberto E Lopez-Herrejon et Alexander Egyed. « Automatically generating and adapting model constraints to support co-evolution of design models ». In : *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM. 2012, p. 302-305.

[129] Jordi Cabot et Jordi Conesa. « Automatic integrity constraint evolution due to model subtract operations ». In : *International Conference on Conceptual Modeling*. Springer. 2004, p. 350-362.

[130] Angelika Kusel et al. « A Systematic Taxonomy of Metamodel Evolution Impacts on OCL Expressions. » In : *ME@ MoDELS*. 2014, p. 2-11.

[131] Angelika Kusel et al. « Systematic co-evolution of OCL expressions ». In : *11th APCCM* 27 (2015), p. 30.

[132] Djamel Eddine Khelladi et al. « Detecting complex changes and refactorings during (meta) model evolution ». In : *Information Systems* 62 (2016), p. 220-241.

[133] Djamel Eddine Khelladi et al. « Metamodel and constraints co-evolution : A semi automatic maintenance of ocl constraints ». In : *International Conference on Software Reuse*. Springer. 2016, p. 333-349.

[134] E. Batot et al. « Heuristic-Based Recommendation for Metamodel — OCL Coevolution ». In : *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Sept. 2017, p. 210-220. doi : 10.1109/MODELS.2017.25.

[135] ISA. *5.1 Instrumentation Symbols and Identification*. 1992.

[136] Thomas McAvinew et Raymond Mulley. *Control System Documentation : Applying Symbols and Identification*. ISA, 2004.

[137] github. *GitHub*. 2021. url : https://github.com/.

[138] James Skene et Wolfgang Emmerich. « Specifications, not meta-models ». In : *Proceedings of the 2006 international workshop on Global integrated model management*. ACM. 2006, p. 47-54.

[139] Object Management Group. *Unified Modeling Language 2.4*. https://www.omg.org/spec/UML/2.4/, year = 2015.

[140] Harald Störrle. « A PROLOG-based Approach to Representing and Querying Software Engineering Models. » In : *VLL* 274 (2007), p. 71-83.

[141] Object Management Group. *Unified Modeling Language 1.5*. https://www.omg.org/spec/UML/1.5/, year = 2003.

[142] Object Management Group. *Unified Modeling Language 2.0*. https://www.omg.org/spec/UML/2.0/, year = 2005.

[143] Per Runeson et Martin Höst. « Guidelines for conducting and reporting case study research in software engineering ». In : *Empirical software engineering* 14.2 (2009), p. 131.

[144] Elyes Cherfa et al. « On Investigating Metamodel Inaccurate Structures ». In : *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. SAC '20. Brno, Czech Republic : Association for Computing Machinery, 2020, 1642–1649. isbn : 9781450368667. doi : 10.1145/3341105.3374035. url : https://doi.org/10.1145/3341105.3374035.

[145] Jürgen Wüst. *SD Metrics*. https://www.sdmetrics.com/.

Titre : Assistance à la spécification de contraintes OCL dans les métamodèles

Mot clés : OCL, MOF, Ingénierie Dirigée par les Modèles (IDM), Structures Imprécises dans les Métamodèles, Assistance à la définition de contraintes OCL, Assistance à la co-évolution de contraintes OCL

Résumé : Un métamodèle permet de capturer la connaissance du domaine par la définition de la structure du domaine (concepts et relations entre eux) et des contraintes (expressions logiques) souvent écrites en OCL pour préciser la sémantique statique. Les contraintes OCL ajoutées à un métamodèle sont de deux types : 1) les contraintes liées au domaine, qui diffèrent d'un domaine à un autre et qui sont exprimées sur la base des connaissances des experts ; 2) celles qui sont ajoutées à la majorité des métamodèles pour préciser certaines structures imprécises qui peuvent poser des problèmes lors de l'instanciation des modèles. Nous appelons ces structures les structures imprecises des métamodèles (MIS). Malheureusement, la tâche de spécifier les contraintes OCL est souvent négligée. En effet, on peut trouver dans la littérature de nombreux métamodèles avec un ensemble incomplet de contraintes OCL, ou même sans contraintes. Par conséquent, en partant d'un métamodèle avec un ensemble incomplet de contraintes OCL, il est possible de créer des modèles qui sont conformes au métamodèle, mais qui ne représentent pas le domaine visé. Dans la littérature, de nombreux travaux ont été réalisés concernant le langage OCL, plus particulièrement sur l'aide à la définition et à la génération automatique de contraintes OCL pendant la phase de conception du métamodèle, ou encore sur la co-évolution des contraintes OCL existantes après l'évolution de leur métamodèle. Dans cette thèse de doctorat, nous nous concentrons principalement sur les contraintes liées à la structure. Nous commençons par étudier "où" les contraintes OCL sont souvent définies dans le métamodèle, et "pourquoi" elles sont définies. Pour ce faire, nous réalisons une étude empirique sur les métamodèles et leurs contraintes OCL. Cette étude empirique a donné lieu à un ensemble de structures inexactes de métamodèles (MIS). Nous validons les résultats quantitativement et qualitativement. La deuxième contribution concerne la co-évolution des contraintes OCL après l'évolution de leur métamodèle. Dans ce contexte, nous avons proposé une approche basée sur les MIS pour compléter les approches de co-évolution existantes car elles se concentrent sur la co-évolution de l'ensemble des contraintes OCL, mais elles ne prennent pas en compte les nouveaux concepts qui peuvent être ajoutés dans la version évoluée du métamodèle. L'approche a été testée sur une étude de cas qui a donné des résultats encourageants.

Title: Assisting the creation of OCL constraints

Keywords: Object-Constraint Language, Meta-Object Facility, Model-Driven Engineering, Metamodel Inaccurate Structures, Assisting OCL Constraints Creation, Assisting OCL Constraints

Co-Evolution

Abstract: Metamodeling allows capturing domain knowledge through the definition of a domain structure and well-formedness rules often written in OCL to precise the static semantics. The OCL constraints added to a metamodel are of two types: 1) domain-related constraints, which differ from one domain to another and are expressed based on the experts' knowledge; 2) those that are added to the majority of metamodels to precise some inaccurate structures that may cause problems when instantiating models. We call these structures Metamodel Inaccurate Structures (MIS). Unfortunately, one may find in the literature many metamodels with an incomplete set of OCL constraints, or even without constraints. Consequently, starting from a metamodel with an incomplete set of OCL constraints it is possible to create models that are conforming to the metamodel, but that does not represent the intended domain. In the literature, many works have been achieved regarding the OCL language, more particularly about assisting the definition and the automatic generation of OCL constraints during the metamodel design phase, or even the co-evolution of existing OCL constraints after the evolution of their metamodel. In this Ph.D. thesis, we focus on structure-related constraints. We start by studying "where" OCL constraints are often defined within the metamodel, and "why" are they defined. To do that we perform an empirical study over metamodels and their OCL constraints, which resulted in a set of Metamodel Inaccurate Structures (MIS). We validate the findings quantitatively and qualitatively. The second contribution is regarding the co-evolution of OCL constraints after the evolution of their metamodel. In this context, we have proposed an approach based on MIS to complete the existing co-evolution approaches since they focus on co-evolving the set of OCL constraints, but they do not take into consideration the new concepts that can be added in the metamodel evolved version. The approach has been tested on a case study which gave encouraging results.