THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Institut National Polytechnique de Toulouse (INP Toulouse)*

Présentée et soutenue le *11/07/2013* **par :**

Mounira KEZADRI

Assistance à la validation et vérification de systèmes critiques : Ontologies et Intégration de composants.

JURY

M. Antoine BEUGNARD	Professeur, Telecom Bretagne	Rapporteur
Mme Catherine DUBOIS	Professeur d'Université, ENSIIE	Rapporteur
M. Yamine AIT AMEUR	Professeur d'Université, INPT	Directeur de thèse
M. Marc PANTEL	Maître de conférences, INPT	Directeur de thèse
M. Franck BARBIER	Professeur d'Université, Pau	Président du Jury
M. Benoit COMBEMALE	Maître de conférences, IRISA	Examinateur

École doctorale et spécialité :
MITT : Domaine STIC : Sureté de logiciel et calcul de haute performance

Unité de Recherche :
IRIT (UMR 5505)

Directeur(s) de Thèse :
M. Marc PANTEL et *M. Yamine AIT AMEUR*

Rapporteurs :
M. Antoine BEUGNARD et *Mme Catherine DUBOIS*

Mounira KEZADRI

Assistance à la validation et vérification de systèmes critiques : Ontologies et Intégration de composants

Encadrants : Marc Pantel & Xavier Thirioux

Résumé

Les activités de validation et vérification de modèles sont devenues essentielles dans le développement de systèmes complexes. Les efforts de formalisation de ces activités se sont multipliés récemment étant donné leur importance pour les systèmes embarqués critiques. Notre travail s'inscrit principalement dans cette voie. Nous abordons deux visions complémentaires pour traiter cette problématique. La première est une description syntaxique implicite macroscopique basée sur une ontologie pour aider les concepteurs dans le choix des outils selon leurs exigences. La seconde est une description sémantique explicite microscopique pour faciliter la construction de techniques de vérification compositionnelles.

Nous proposons dans la première partie de cette thèse une ontologie pour expliquer et expliciter les éléments fondateurs du domaine que nous appelons VVO. Cette ontologie pourra avoir plusieurs autres utilisations : une base de connaissance, un outil de formation ou aussi un support pour le choix de la méthode à appliquer et l'inférence de correspondance entre outils.

Nous nous intéressons dans la seconde partie de cette thèse à une formalisation dans un assistant à la preuve de l'introduction de composants dans un langage de modélisation et des liens avec les activités de validation et vérification. Le but est d'étudier la préservation des propriétés par composition : les activités de vérification sont généralement coûteuses en terme de temps et d'effort, les faire d'une façon compositionnelle est très avantageux. Nous partons de l'atelier formel pour l'Ingénierie Dirigée par les Modèles Coq4MDE. Nous suivons la même ligne directrice de développement prouvé pour formaliser des opérateurs de composition et étudier la conservation des propriétés par assemblage. Nous nous intéressons au typage puis à la conformité de modèles par rapport au métamodèle et nous vérifions que les opérateurs définis permettent de conserver ces propriétés. Nous nous focalisons sur l'étude d'opérateurs élémentaires que nous exploitons pour spécifier des opérateurs de plus haut niveau. Les préconditions des opérateurs représentent les activités de vérification non compositionnelles qui doivent être effectuées en plus de la vérification des composants pour assurer la postcondition des opérateurs qui est la propriété souhaitée. Nous concluons en présentant des perspectives pour une formalisation algébrique en théorie des catégories.

Mots clés : formalisation, Vérification et Validation, ontologie, Ingénierie Dirigée par les Modèles, assemblage, assistant à la preuve

---

Institut de Recherche en Informatique de Toulouse - UMR 5505

INPT, Toulouse

Mounira KEZADRI

SUPPORT FOR THE VALIDATION AND VERIFICATION OF CRITICAL SYSTEMS: ONTOLOGIES AND INTEGRATION OF COMPONENTS

Supervisions: Marc Pantel & Xavier Thirioux

Abstract

The validation and verification of models have become essential in the development of complex systems. The formalisation efforts for these activities have increased recently being given their importance for critical embedded systems. We discuss two complementary visions for addressing these issues. The first is a syntactic implicit macroscopic description based on an ontology to help designers in the choice of tools depending on their requirements. The second is a microscopic explicit semantics description aiming to facilitate the construction of compositional verification techniques.

We propose in the first part of this thesis an ontology to explain and clarify the basic elements of the domain of Verification and Validation that we call VVO. This ontology may have several other uses: a knowledge base, a training tool or a support for the choice of the method to be applied and to infer correspondence between tools .

We are interested in the second part of this thesis in a formalisation using a proof assistant for the introduction of components in a modelling language and their links with verification and validation activities. The aim is to study the preservation of properties by the composition activities. The verification are generally expensive in terms of time and efforts, making theme in a compositional way is very advantageous. Starting from the formal framework for Model Driven Engineering COQ4MDE, we follow the same line of though to formalize the composition operators and to study the conservation of properties by composition. We are interested in typing and conformity of models in relation with metamodels and we verify that the defined operators allow to preserve these properties. We focus on the study of elementary operators that we use to specify hight level operators. The preconditions for the operators represent the non-compositional verification activities that should be performed in addition to verification of components to ensure the desired postcondition of the operator. We conclude by studying algebraic formalisation using concepts from category theory.

Key words: formalisation, Verification and Validation, ontology, Model Driven Engineering, composition, proof assistant

---

Institut de Recherche en Informatique de Toulouse - UMR 5505
INPT, Toulouse

À mes parents et à mes grands-parents

« Vouloir prouver des choses qui sont claires d'elles-mêmes, c'est éclairer le jour avec une lampe. »

Aristote

Remerciements

JE remercie tout d'abord celui qui m'a donné la force d'écrire ces lignes et de mener à bout ce travail, je ne le remercie jamais assez sans remercier toutes les personnes qui ont contribué de près ou de loin à l'élaboration de cette thèse.

DURANT mes années d'études j'ai eu la chance d'être toujours encadrée par des personnes vraiment exceptionnelles. Mes très vifs et éternels remerciements vont en premier lieu à Marc Pantel pour m'avoir proposée cette thèse, soutenue, encouragée et orientée tout au long de ces années. Je le remercie et lui adresse mon estime et gratitude surtout pour son humanité et sa compréhension, sa culture générale et scientifique très vaste sans laquelle je ne pourrais jamais aborder de tels sujets.

JE tiens à remercier Catherine Dubois et Antoine Beugnard d'avoir accepté de rapporter sur cette thèse. Merci à Catherine pour ses remarques très pertinentes et l'intérêt qu'elle a porté pour ce travail et même aux détails de l'implémentation en Coq. Je remercie Antoine pour ses remarques complètes, rapides et très pertinentes qui m'ont permise d'améliorer encore la présentation de ce manuscrit.

UN grand merci à Franck Barbier pour l'intérêt qu'il a porté pour ce travail et d'avoir accepté de présider mon jury de thèse. Je remercie Benoit Combemale d'avoir accepté d'être dans le jury ainsi que pour ses conseils et sa collaboration pour la réalisation de ce travail. Je remercie également Jean-Paul Bodeveix pour sa présence et pour ses réponses enrichissantes à mes questions pendant la rédaction de ce manuscrit.

JE remercie Xavier Thirioux pour sa collaboration, pour ses idées et son savoir incontournable en Coq. Je remercie Patrick Sallé et Yamine Ait Ameur d'avoir accepté d'être mes directeurs de thèse, merci à Yamine pour son aide et ses conseils précieux pour la préparation de la présentation.

JE tiens à remercier spécialement Pascaline Parisot pour son amitié, conseils, encouragement et son aide à la préparation de la présentation.

IL m'est très agréable de remercier toutes les personnes qui m'ont formée. Merci à ceux qui ont encadré mon projet de Master, merci à Martin Strecker, Jean-Paul Bodeveix et Mamoun Filali. Merci à mes enseignants à l'Université M'Hamed Bougerra en Algérie qui m'ont initiée à l'informatique, je dois spécialement remercier Mohamed Khalifa d'avoir encadré mon projet de licence et surtout pour sa rigueur. Je remercie également mon professeur Mohamed Mezghiche pour son encouragement et ses conseils. Je remercie tous ceux qui n'ont pas économisé des éfforts pour réussir le système LMD à l'UMBB, je pense spécialement au chef de département informatique et le doyen de la faculté des sciences à Boumerdès que je remercie aussi pour sa confiance et son soutien. Merci aux organisateurs du programme Imageen de l'université M'Hamed Bouguera et aussi de l'Université Paul Saba-

tier à Toulouse.

MA gratitude et remerciements vont à tous mes professeurs : à l'école primère Mustapha Oukil, au collège Dermouche Rabah et au lycée Baaziz Mohamed à Lakhdaria en Algérie.

UN très grand merci à tous les membres de l'équipe Acadie qui m'ont soutenue et encouragée sans exception. Merci à ceux qui m'ont proposé volontairement de me lire ou m'ont vraiment relu (Philipe, Aurélie, Meriem, Andrès, Martin, Ludivine, Malika, Yamine et Zoé), merci pour tous les membres de l'équipe pour leurs remarques et propositions dans mes différentes présentation ainsi que pour la préparation de la soutenance.

JE remercie tous les membres du laboratoire IRIT pour leurs sympathie et spécialement nos deux secrétaires Sylvie Eichen et Sylvie Armengaud pour tous ce qu'elle font tous les jours pour simplifier notre travail. Je n'oublie pas de remercier le responsable du site Enseeiht de l'IRIT André-Luc Beylot pour son encouragement et ses conseils pour la présentation ainsi que tous les membres du service informatique à l'Enseeiht pour leur sympathie. Un grand merci également à l'équipe de restauration à l'Enseeiht pour leur bon humeur, attention et compréhension.

MES vifs remerciements à tous mes amis des deux rives, ceux qui sont en Algérie et qui m'encouragent toujours, ou que j'ai connu ici en France, je cite spécialement mon binôme de licence Fethia, ma chère Amina, Salma, Asmaa, Kamelia, Selma, Ludivine, Hà, Thanh, Faten, Samira, Sondès, Tasnim, Nawel, Caroline, Aurélie, Stéphanie et Malika. Je remercie tous celles avec qui j'ai partagé mon logement (Fatma, Malika, Wahiba), tous ceux avec qui je partage toujours mes repas (Rania, Ahmed, Mustapha, Rawdha, Bouchra), tous ceux et celles avec qui j'ai partagé mon bureau (Nassima, Celia et Paul), aussi ceux et celles qui étaient ici juste pour une petite période et qui m'ont tous encouragée et soutenue.

JE remercie tous les membres de ma famille, je remercie mes très chers parents qui m'ont toujours soutenue et encouragée, d'avoir toujours été à mes cotés même virtuellement, je les remercie surtout pour m'avoir fait confiance et pour tous leurs sacrifices. Je remercie mes sœurs (Karima, Hanane, Yasmine, Nadjate, Aziza et Chahrazed) et mon frère (Sid Ahmed) pour leurs amour et fraternité vraiment exemplaire. Je remercie mes beaux-frères Marouane at AbdelWahab et mes deux nièces Wissam et Manar. J'ai une pensé pour ma grande-mère paternelle qui est toujours ici quand j'ai commencé cette thèse et qui est toujours présente dans mes pensées et mon cœur. Mon éternel merci pour ma grande mère maternelle qui m'encourage et me soutient toujours. Je remercie infiniment ma tante Fatma et toute sa famille pour leur soutien et encouragement. Je remercie mes oncles (Omar, Ali, Hocine, Kamel et Azeddine) et mes tantes (Aziza, Baya qui nous a laissé très tôt et Cherifa) ainsi que tous mes cousins et cousines.

ENFIN , je remercie mon âme sœur, mon très cher tendre mari Adnane pour ses conseils, aide, soutien sans faille et surtout pour sa patience et compréhension. Je remercie aussi tous les membres de sa famille qui m'ont encouragée et spécialement sa chère maman et Leila.

Table des matières

Introduction générale **1**

I Formalisation syntaxique sous forme d'ontologie **5**

1 L'ontologie de V&V (VVO) **7**

1.1 V&V : Validation et Vérification 9
1.2 Ontologie 9
1.2.1 Concepts 10
1.2.2 Instances 10
1.2.3 Relations 11
1.3 L'implémentation de la VVO 12
1.3.1 La méthodologie de construction 12
1.3.1.1 Les principes de conception 13
1.3.1.2 Le choix du langage 15
1.3.1.3 Le choix de l'outil 16
1.3.2 La structure générale de la VVO 16
1.3.3 Une ontologie pour les formalismes de description 17
1.3.4 Une ontologie pour les vues 20
1.3.5 Une ontologie pour les techniques de V&V 21
1.4 étude de cas 25
1.5 D'autres travaux sur les ontologies 27
1.6 Conclusion et perspectives 30

II Formalisation sémantique **33**

2 Concepts **35**

2.1 Une introduction à l'Ingénierie Dirigée par les Modèles 36
2.1.1 Modèles 37
2.1.2 Les métamodèles 37
2.1.3 Le métamétamodèle exploité 38
2.1.4 L'architecture à quatre niveaux 38
2.2 Les approches de composition dans l'IDM 38
2.3 Une introduction à l'assistant à la preuve COQ 39
2.3.1 Les termes de base 41
2.3.2 Les définitions 41
2.3.3 Les théorèmes et les preuves 43
2.3.4 Développement réalisé en COQ 46
2.3.5 Types dépendants 47
2.4 La formalisation de l'ingénierie dirigée par les modèles 48
2.4.1 Spécification algébrique 48
2.4.2 Théorie des types 48
2.4.3 Raffinement et théorie des ensembles 50
2.5 Conclusion 50

3 Opérateurs élémentaires et vérification de propriétés 51

3.1 Coq4MDE 53
3.2 Formalisation de la structure du modèle 55
3.2.1 La structure des modèles 56
3.2.2 La structure des ensembles 57
3.2.3 La structure des graphes 58
3.2.3.1 Des arcs corrects 59
3.2.3.2 Égalité des graphes 59
3.3 Formalisation de la composition de modèles 59
3.3.1 L'opérateur élémentaire d'Union 60
3.3.2 L'opérateur élémentaire de Substitution 61
3.4 Formalisation et vérification de propriétés 63
3.4.1 InstanceOf 64
3.4.2 Conformité dans le standard MOF 64
3.4.3 subClass 65
3.4.4 isAbstract 66
3.4.5 lower & upper 67

3.4.6 isOpposite 69
3.4.7 areComposite 70
3.5 La vérification compositionnelle 71
3.6 Conclusion 72

4 Formalisation d'une composition (Package Merge) 75
4.1 La sémantique du Package Merge 75
4.1.1 Les règles de filtrage 78
4.1.2 Les contraintes 78
4.1.3 Les transformations 79
4.2 Un exemple pour la résolution de con its 81
4.3 Améliorations 84
4.4 Travaux connexes de formalisation pour Package Merge 84
4.5 Conclusion 85

5 Formalisation d'une composition de modèles (ISC) 87
5.1 `ISC` et la formalisation de ses opérateurs 88
5.1.1 L'extension du métamodèle 89
5.1.2 L'extraction et l'élimination de l'interface 90
5.1.3 L'assemblage de composants 92
5.1.3.1 L'opérateur `bind` 92
5.1.3.2 L'opérateur `extend` 93
5.2 Un exemple détaillé 95
5.3 La vérification des propriétés 97
5.3.1 L'opérateur `bind` 98
5.3.1.1 Une vue sur le langage de tactiques 100
5.3.2 L'opérateur `bind` de deux modèles avec plusieurs points de variation 102
5.3.3 L'opérateur `extend` 105
5.3.3.1 L'opérateur `extend` basé sur l'union disjointe 105
5.3.3.2 L'opérateur `extend` basé sur l'union classique 107
5.3.4 L'extraction de l'interface des fragments 110
5.4 Conclusion 112

6 Perspectives : Vers une catégorie de modèles 115
6.1 Une catégorie pour les modèles 116
6.1.1 Une première approche intuitive 118

6.1.2 Une seconde approche 119
6.2 Proposition d'interprétation des opérateurs algébriques 122
6.3 Conclusion 127

Conclusion générale et perspectives **129**
6.4 Bilan des travaux présentés 129
6.4.1 La formalisation syntaxique 129
6.4.2 La formalisation sémantique 130
6.5 Perspectives 130
6.5.1 Compatibilité de composants à base d'ontologies 130
6.5.2 Compatibilité comportementale lors de l'assemblage 131
6.5.3 Modélisation d'autres opérateurs 132
6.5.4 Opérateurs sur les ontologies 132

III Annexes **135**

A Une extension pour la bibliothèque Uniset de COQ **137**
A.1 L'union 137
A.2 Les ensembles disjoints 137
A.3 L'intersection 138
A.4 La différence 138
A.5 La substitution 139

B Calcul des cardinaux **141**
B.1 Les itérateurs de graphes 142
B.1.1 Les itérateurs des nœuds et des liens 142
B.1.2 Le module Forall 143
B.1.3 Le module `Fold` 144
B.1.4 Le module Exists 144
B.2 Les itérateurs sur les modèles 145
B.3 Exemples de descriptions de propriétés 147
B.3.1 La définition de la propriété *lower* 147
B.3.2 La définition de la propriété *subClass* 147

C La preuve InstanceOf de l'opérateur Substitution **149**

D Package Merge **151**

D.1 Préliminaires 151
D.2 L'Union 154
D.3 Les preuves d'équivalence sémantique 154
D.4 Exemple 154

E Les chiers Coq **157**
E.1 Les définitions et les propriétés 157
E.2 Les preuves 158

Bibliographie **159**

Liste des gures **171**

Index **173**

Glossary **177**

Introduction générale

La construction de systèmes qui respectent les standards de certification (e.g. DO-178 [RTC12a] en aéronautique) et de qualification (e.g. DO-330 [RTC12b] en aéronautique) est une exigence essentielle dans le domaine des systèmes critiques. Cette exigence se concrétise actuellement à travers le développement dirigé par les modèles (MDD (Model Driven Development), MDE (Model Driven Engineering), MDA (Model Driven Architecture)) et la mise en œuvre de technologies de Validation et Vérification (V&V) tout au long du développement. Les méthodes formelles jouent un rôle important dans ce cadre. De plus, la réutilisation des éléments qui composent le système et des outils de développement est une exigence de plus en plus forte. L'objectif de cette thèse est d'étudier les perspectives en terme de composants pré-qualifiés et pré-certifiés pour le développement de systèmes critiques. La notion de qualification concerne les outils exploités pour développer des systèmes critiques lorsque l'on souhaite s'appuyer sur la qualité des outils pour réduire les activités de V&V effectuées sur le système. La notion de certification est liée au système lui même. Les activités de validation et vérification y jouent un rôle fondamental.

Nous proposons dans cette thèse deux visions pour cette problématique. La première est une description syntaxique implicite macroscopique basée sur une ontologie de domaine pour les technologies de modélisation, validation et vérification. L'objectif est de faciliter la réutilisation d'outils qualifiés pour construire des chaînes de développement de systèmes critiques. La seconde est une description sémantique explicite microscopique basée sur une formalisation des mécanismes de composition dans les langages de modélisation et des activités de vérification compositionnelle. Cette formalisation est effectuée et validée avec l'assistant de preuve Coq.

Ce travail a été initié dans le cadre du projet CESAR[1] (Cost-Efficient methods and processes for SAfety Relevant embedded systems) qui regroupe des partenaires de plusieurs domaines critiques (l'aéronautique, l'automatisation industrielle, l'automobile, le ferroviaire et le spatial) qui ont comme exigence commune le développement de systèmes embarqués sûrs et fiables. Pour ce type de système, les activités de validation et vérification sont primordiales car toute erreur peut causer des catastrophes humaines, économiques et environnementales. Deux approches jouent actuellement un rôle fondamental pour assurer cette correction. D'une part, la construction de modèles des différents aspects du système tout au long de son développement permet de formaliser les documents de spécification et de conception, puis de mettre en place des activités de V&V sans attendre que l'implémen-

[1] http://www.cesarproject.eu/

tation du système soit disponible. D'autre part, l'utilisation de méthodes formelles pour la spécification et la vérification permet de mesurer la qualité des activités de vérification et d'augmenter l'assurance obtenue habituellement par tests et relecture. L'approche macroscopique améliore la compréhension de ce domaine et la formalisation microscopique contribue à la construction de systèmes corrects par assemblage de composants corrects. Cette thèse s'est déroulée au sein de l'équipe ACADIE[2] et profite de l'expérience de ses membres dans le domaine de la Validation et Vérification (V&V) des logiciels critiques ainsi que de leur expertise dans le domaine des méthodes formelles et de l'Ingénierie Dirigée par les Modèles (IDM).

Dans notre démarche macroscopique, nous construisons une ontologie de domaine pour la V&V. La construction d'ontologies n'est plus seulement une question philosophique ou linguistique. Elles permettent de classer les informations et de rendre les connaissances plus faciles à assimiler, diffuser et partager. L'ontologie telle qu'elle est utilisée dans l'informatique moderne et d'autres domaines est une approche pour l'organisation formelle des informations. Une ontologie de domaine contribue largement à réduire ou éliminer les confusions terminologiques ou conceptuelles entre les membres d'une communauté d'utilisateurs qui peuvent partager des documents et des informations de différents types. Les fondements d'une proposition d'ontologie de domaine pour la Modélisation, la Validation et la Vérification de systèmes logiciels (VVO) sont présentés dans cette thèse.

Dans notre démarche microscopique, nous proposons une formalisation de l'introduction de composants dans un langage de modélisation et du lien avec la vérification compositionnelle. Cette formalisation, réalisée avec l'assistant de preuve COQ, s'appuie sur les concepts de l'Ingénierie Dirigée par les Modèles. L'IDM s'est donnée comme objectif la maîtrise de la complexité croissante de la construction de systèmes logiciels. Le principe est d'utiliser les modèles ou les composants comme les pierres de base. Le développement orienté composant est une idée très ancienne, qui était présente depuis les premiers travaux sur la programmation structurelle et modulaire [Mci68] mais sa formalisation et la vérification de la composition est encore un sujet de recherche en constante évolution. Nous définissons des opérateurs élémentaires de composition pour lesquels nous vérifions des propriétés de conformité en relation avec le métamétamodèle EMOF (Essential Meta-Object Facility). Les propriétés vérifiées s'appliquent à tous les composants qui peuvent se décrire dans des DSMLs (Domain-Specific Modeling Languages) conformes à EMOF. Ceci rend les opérateurs vérifiés génériques. Ces opérateurs sont par la suite utilisés pour implémenter des opérateurs de plus haut niveau vérifiés par composition.

Résumé des contributions

Cette thèse traite de la problématique de l'assemblage de composants pré-validés, pré-vérifiés, pré-certifiés et pré-qualifiés. Nous pouvons résumer les contributions apportées en quelques points :

▷ Une formalisation syntaxique sous forme d'une ontologie. Cette démarche abstraite

[2] http://www.irit.fr/-Equipe-ACADIE-?lang=fr

nous a permis de construire une ontologie de domaine pour la V&V qui permet d'assister les utilisateurs dans la conception d'une chaîne d'outils qualifiée pour le développement de systèmes critiques.

- ▷ Une formalisation sémantique concrète dans un assistant de preuve de l'ajout de la notion de composants dans un langage de modélisation sous la forme d'opérateurs élémentaires génériques d'assemblage.
- ▷ La vérification de la préservation des propriétés d'intérêt par les opérateurs d'assemblage en introduisant les préconditions minimales nécessaires pour obtenir la compositionalité.
- ▷ L'utilisation des opérateurs élémentaires vérifiés pour implémenter des opérateurs de plus haut niveau.

Visite guidée

Cette thèse est organisée principalement en deux parties. La première partie concerne les points clés pour le développement de l'ontologie de domaine pour la V&V (VVO) tandis que la deuxième partie concerne la formalisation d'opérateurs d'assemblage vérifiés en assistant de preuves. Étant donné les différences d'échelle entre les approches macroscopique et microscopique suivies dans cette thèse, il n'était pas judicieux de réaliser un état de l'art unique comme cela est effectué habituellement. Nous avons donc réparti celui-ci dans les différents chapitres (sections 1.5, 2.4 et 4.4). Ceci permet également une lecture autonome des différentes parties. Nous résumons ci-dessous le contenu des différents chapitres présentés au long du manuscrit.

Introduction générale : ce chapitre présente une introduction aux différentes problématiques abordées dans le cadre de cette thèse.

Chapitre 1 : il décrit les fondements de la VVO. Les concepts de V&V et d'ontologie sont présentés en premier lieu. Par la suite, les points clés de la conceptualisation générale de cette ontologie ainsi que son implémentation sont expliqués. Une étude de cas pour l'exploitation de la VVO et des travaux connexes sont exposés. Le chapitre se termine par des conclusions et des perspectives pour l'enrichissement et l'utilisation de la VVO.

Chapitre 2 : celui-ci introduit les concepts nécessaires à la compréhension de la formalisation sémantique. Il commence par l'introduction des concepts de l'IDM en définissant les notions de modèle, métamodèle et métamétamodèle ainsi que leurs relations. Dans une seconde section le chapitre aborde le monde de l'assistant de preuve Coq en introduisant ses concepts à l'aide d'une première preuve.

Chapitre 3 : il présente en premier lieu un atelier formel basé sur les concepts de l'IDM et l'assistant de preuve Coq appelé `Coq4MDE`. Par la suite, cet atelier est enrichi par des opé-

rateurs élémentaires de composition et des propriétés sur ces opérateurs. Enfin, ce chapitre présente les théorèmes exprimant la préservation des propriétés ainsi que les liens vers leurs preuves de correction en COQ.

Chapitre 4 : celui-ci introduit l'opérateur de fusion de paquetages (`Package Merge`). Le chapitre commence par une présentation de la sémantique du `Package Merge` à l'aide d'exemples. Par la suite, cet opérateur est exprimé comme une composition des opérateurs élémentaires présentés dans le chapitre 3. Enfin des travaux connexes pour la formalisation de la fusion de paquetages et une conclusion sont présentés.

Chapitre 5 : il décrit une formalisation d'une composition de modèles selon la méthode `ISC` (Invasive Software Composition). Dans une première section la méthode `ISC` est introduite et une formalisation des opérateurs s'inspirant de son style est présentée. Par la suite, un exemple détaillé est expliqué. Enfin, les propriétés de correction sont vérifiées.

Chapitre 6 : ce chapitre décrit nos expériences préliminaires pour définir une catégorie de modèles et les opérateurs algébriques associés. Ce cadre doit permettre de formaliser nos opérateurs déjà définis avec ces nouveaux opérateurs algébriques.

Conclusion générale et perspectives : ce chapitre présente une conclusion générale sous forme de bilan des travaux présentés et des perspectives.

Une bibliographie : elle présente les différentes ressources bibliographiques référencées dans le manuscrit.

Annexe A : elle présente une extension de la bibliothèque Uniset de la librairie COQ avec des opérateurs et des propriétés utilisés dans nos développements.

Annexe B : elle décrit les définitions permettant d'itérer et calculer les cardinaux pour la structure des modèles.

Annexe C : elle développe une preuve en utilisant le format mathématique habituel pour la préservation de la propriété $instanceOf$ pour l'opérateur de substitution réalisée en COQ.

Annexe D : elle présente une formalisation de l'opérateur `Package Merge` en se basant sur un opérateur Union décrit au niveau métamodèle.

Annexe E : elle donne les liens hypertextes ainsi qu'un court résumé des fichiers de développement COQ.

Première partie

Formalisation syntaxique sous forme d'ontologie

1 L'ontologie de V&V (VVO)

Table des matières

1.1 V&V : Validation et Vérification . . . 9
1.2 Ontologie . . . 9
1.2.1 Concepts . . . 10
1.2.2 Instances . . . 10
1.2.3 Relations . . . 11
1.3 L'implémentation de la VVO . . . 12
1.3.1 La méthodologie de construction . . . 12
1.3.1.1 Les principes de conception . . . 13
1.3.1.2 Le choix du langage . . . 15
1.3.1.3 Le choix de l'outil . . . 16
1.3.2 La structure générale de la VVO . . . 16
1.3.3 Une ontologie pour les formalismes de description . . . 17
1.3.4 Une ontologie pour les vues . . . 20
1.3.5 Une ontologie pour les techniques de V&V . . . 21
1.4 étude de cas . . . 25
1.5 D'autres travaux sur les ontologies . . . 27
1.6 Conclusion et perspectives . . . 30

Ce chapitre présente les éléments clés de la VVO (Verification and Validation Ontology) [KP12] [KP10a] ; il décrit la conceptualisation (structure) générale de cette ontologie et justifie les principaux choix de modélisation. La VVO formalise une partie des connaissances sur le domaine de la modélisation et les techniques de V&V correspondantes en décrivant les concepts qui existent dans ce domaine d'intérêt et les relations entre ces concepts.

Le but initial de la VVO est de partager et rendre réutilisables les connaissances du do-

maine de la modélisation de systèmes et les techniques de V&V associées et par la suite d'utiliser celle-ci comme un moyen pour choisir et appliquer d'une manière automatique la technique de V&V la plus adaptée dans le but de simplifier le développement de systèmes critiques sûrs.

La VVO propose une classification des différents formalismes de modélisation et des méthodes de V&V. Dans la VVO, un système est représenté selon plusieurs vues qui sont conformes à certains langages de modélisation. L'ontologie VVO définit les méthodes de V&V qui peuvent être appliquées sur un système dont le comportement est défini en utilisant un langage de modélisation. Les exigences associées aux systèmes exprimées sous la forme de propriétés peuvent être vérifiées avec des techniques de V&V. La VVO décrit ainsi les relations sémantiques entre les outils de V&V, les formalismes et les vues.

La VVO est une base de connaissance obtenue en alimentant des classes de l'ontologie avec les informations et les relations appropriées.

La classification et la population initiale de l'ontologie VVO sont principalement basées sur l'état de l'art des projets CESAR[1] et Ptolemy[2]. La VVO peut être utilisée comme un langage de communication, comme une fondation pour d'autres ontologies et plus tard pour constituer une plateforme globale pour un ensemble de techniques de V&V. Cette base de connaissance sera exploitée pour guider le choix de l'outil de V&V en relation avec le formalisme utilisé pour modéliser le système et les exigences en terme de propriétés du système.

La version actuelle de la VVO considère trois domaines : les vues, les formalismes et les techniques de V&V. Les principales contributions de cette version sont : 1) la classification des formalismes de description de systèmes, 2) la classification des méthodes de V&V, 3) la définition des relations entre les techniques de V&V, les langages de description de comportement et les langages de description de propriétés.

La version actuelle de la VVO est une première étape qui présente une conceptualisation générale du domaine. Elle peut être largement améliorée au niveau du contenu et enrichie avec d'autres outils et méthodes du domaine de la V&V. Elle ne présente qu'une interprétation particulière du domaine qui devra être élargie pour améliorer la rigueur et la précision de la représentation.

La VVO est publiquement disponible sur http://www.irit.fr/~Mounira.Kezadri/Ontologies/VVO.owl pour être validée, étendue et réutilisée par la communauté.

Ce chapitre est organisé comme suit : le concept de V&V est présenté dans une première section. La deuxième section définit le concept d'ontologie et d'ontologie de domaine ainsi que son utilisation pour la spécification formelle. L'implémentation détaillée est présentée dans une troisième section qui donne en premier lieu la méthodologie de construction, puis le choix du langage et de l'outil et détaille enfin les principes de conception et la structure générale de l'ontologie avec la description des trois sous-ontologies et de leurs composantes (concepts et relations). La quatrième section donne un cas d'utilisation de la VVO. La cinquième section présente des éléments de bibliographie et certains travaux connexes sur les

[1] https://cesarproject.eu/
[2] http://ptolemy.eecs.berkeley.edu/

ontologies. Le chapitre se termine par des conclusions résumant des perspectives sur cette première approche.

1.1 V&V : Validation et Véri cation

La Vérification et la Validation (V&V) sont les activités dans le processus de développement assurant que le système satisfait ses exigences (qu'il se comporte correctement par rapport aux attentes des utilisateurs et à sa spécification). La vérification concerne l'implémentation du système et sa spécification explicite tandis que la validation relie un système et les contraintes implicites de l'utilisateur final. La vérification est souvent qualifiée de « bien faire les choses » (do the thing right) et la validation de « faire les bonnes choses » (do the right thing). Si les contraintes sont spécifiées, le système est vérifié par rapport à cette spécification et la spécification est validée par rapport aux besoins des utilisateurs finaux.

Étant donné le grand nombre de méthodes et outils de V&V, une description formelle du domaine de V&V est indispensable pour faciliter le choix de la technologie la plus adaptée lorsque un développeur a besoin de vérifier et/ou valider des parties du système qu'il est en train de construire.

1.2 Ontologie

Une ontologie est une spécification formelle explicite d'une conceptualisation partagée [G+93]. L'ontologie doit être formelle dans le sens où elle doit être exploitable par une machine. Elle est explicite car les types de ses concepts et contraintes doivent être explicitement définis. Elle est partagée dans le sens où ses données ne sont pas réservées à une personne particulière mais acceptées par un groupe. C'est une conceptualisation dans le sens où elle est un modèle abstrait d'éléments ou phénomènes qui existent dans le monde réel et elle identifie les concepts pertinents de cet élément/phénomène.

Les ontologies permettent de :

- ▷ partager une compréhension commune de la structure de l'information.
- ▷ permettre la réutilisation des connaissances du domaine afin de standardiser ces disciplines pour permettre l'interopérabilité.
- ▷ formuler explicitement les hypothèses du domaine permettant de rendre plus facile le changement, la compréhension et la mise à jour de ce modèle de connaissance.
- ▷ ne plus avoir besoin de construire des connaissances à partir de zéro mais en s'appuyant sur les connaissances déjà construites.

Une ontologie constitue une terminologie commune de concepts et relations entre concepts. Le vocabulaire peut être conçu comme une structure de graphe composée de termes (qui forment les nœuds du graphe correspondant) liés à l'aide d'arcs nommés relations. La VVO est constituée de plusieurs sous-ontologies, ces dernières sont organisées

à l'aide de différents types de relations. Les relations à l'intérieur des sous-ontologies sont en général des relations de type `is_a` (sorte de). D'autres relations sont définies et seront présentées par la suite. Elles sont utilisées dans les sous-ontologies et entre les différentes sous-ontologies .

La construction d'une nouvelle ontologie peut s'appuyer sur d'autres ontologies préexistantes. Elle permet dans ce cas d'avoir une interopérabilité sémantique plus forte. Les entreprises et les individus doivent converger vers une ontologie standard pour leur domaine et développer des liens entre leur ontologie et les autres ontologies.

Nous nous sommes intéressée dans ce travail à une ontologie de domaine pour la V&V. Une ontologie de domaine signifie une ontologie qui est restreinte à un domaine d'intérêt particulier. Elle définit les termes de base et les relations entre les termes du domaine, ainsi que les règles pour combiner les termes et les relations permettant de définir des extensions du vocabulaire.

1.2.1 Concepts

Les concepts constituent les éléments de base de la terminologie commune d'un domaine particulier. Ils représentent les classes d'un monde réel ou abstrait possédant une spécificité commune ou partagée. Le terme classe est utilisé pour référencer ce qui est général, il peut être appelé également type ou genre ou aussi catégorie. Les classes dans la VVO sont en premier lieu les termes les plus généraux intervenant dans le domaine par exemple de la V&V.

Les membres d'une classe sont appelés instances du concept correspondant à la classe. Formellement, les concepts sont interprétés comme des ensembles sémantiques d'éléments. Un concept peut être un sous-concept de plusieurs (éventuellement aucun) super-concepts directs tel que spécifié avec la relation `is_a`. Formellement, ceci déclare une relation entre les instances du concept et ses super-concepts : chaque instance des sous-concepts est considérée aussi comme une instance du super-concept. Par conséquent, les sous-concepts précisent en quelque sorte leurs super-concepts en ajoutant un sens spécifique à ces derniers ce qui permet de distinguer les instances des sous-concepts des instances des super-concepts.

Dans la VVO chaque concept peut avoir un nombre fini de sous-concepts qui sont à leur tour des super-concepts d'un certain nombre de concepts. être un sous-concept d'un autre concept en particulier signifie que le concept hérite du type et des éléments définis dans ce super-concept et des contraintes correspondantes. Des exemples de hiérarchie de quelques concepts de la VVO sont donnés par la suite.

1.2.2 Instances

Un concept représente un ensemble d'objets d'un monde réel ou abstrait caractérisé par une propriété spécifique partagée. Les objets eux-mêmes sont appelés instances ou individus.

La description des instances d'un concept suit un schéma commun qui est la signature imposée par la définition du concept. Par exemple, dans la VVO , l'individus `aiT`[3]

[3] http://www.absint.com/ait/index.htm

```
<!-- http://www.irit.fr/~Mounira.Kezadri/Ontologies/VVO.owl#SymbolicSimulation -->

<owl:Class rdf:about="&VVO;SymbolicSimulation">
    <rdfs:subClassOf rdf:resource="&VVO;Simulation"/>
    <rdfs:comment rdf:datatype="&xsd;string">Is a formal verification technique, which combines the flexibil
ity of conventional simulation with powerful symbolic methods. For each input in the design, a symbolic variable
 is introduced which represents all values this input may take. The symbolic simulator computes symbolic express
ions for each output in terms of the input variables. These expressions implicitly represent the values of the o
utputs for all possible input assignments, increasing the probability of finding design errors by orders of magn
itude.</rdfs:comment>
    <ace_lexicon:sg rdf:datatype="&xsd;string">SymbolicSimulation</ace_lexicon:sg>
    <ace_lexicon:pl rdf:datatype="&xsd;string">SymbolicSimulations</ace_lexicon:pl>
</owl:Class>
-u:**  VVO.owl        89% L5382  (XML Fill)----4:42PM 0.64-----------------------------------------------------
```

Figure 1.1 — La classe SymbolicSimulation et sa relation avec la classe Simulation dans le fichier OWL

[FH04] (un outil développé dans le cadre du projet DAEDALUS[4]) est une instance de WCETAnalysis (outils d'analyse du pire cas d'exécution).

1.2.3 Relations

Les relations sont utilisées pour modéliser les dépendances entre plusieurs concepts et/ou les relations entre les instances de ces concepts. La cardinalité de la relation n'est pas limitée.

Les dépendances entre les classes et entre les instances correspondantes ont fait l'objet de nombreux travaux sur les logiques de description [CDGL+05] [BHS05] [H+02].

Dans le cadre de la VVO, nous nous concentrons sur des relations assez génériques. Ces relations sont définies par la suite.

En général, plusieurs types de relations peuvent être distingués :

Des relations entre deux classes Comme la relation *subClassOf* entre la classe Simulation [KL00] et la classe SymbolicSimulation [RG05] illustrée sur la figure 1.1 comme exprimée dans le fichier VVO.owl .

Des relations entre une classe et une instance Par exemple, la relation de typage (instance_of) entre l'instance CBMC [RG05] et la classe ModelChecking [BK+08].

Ceci est présenté avec l'interface graphique de l'outil Protégé[5] comme illustré sur la figure 1.2.

Des relations entre deux instances sont possibles dans une ontologie mais ce type de relations n'est pas utilisé dans la VVO et est remplacée par des relations entre les classes de plus bas niveau et les instances . Par exemple, la relation verifies entre ModelChecking et Automata signifie aussi qu'il y a une relation verifies entre CBMC et Automata comme il existe une relation instanceOf entre ModelChecking et CBMC.

[4] http://www.absint.com/projects.htm
[5] http://protege.stanford.edu/

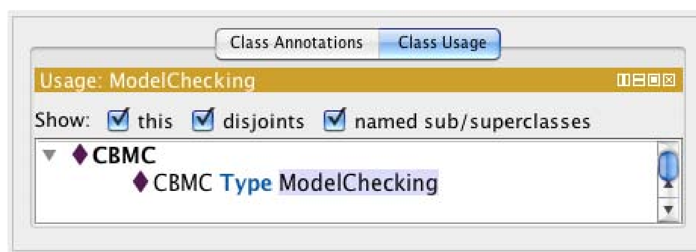


Figure 1.2 — La relation entre la classe CBMC et ModelChecking présentée dans protégé

1.3 L'implémentation de la VVO

Dans cette section, nous commençons par présenter la méthodologie de construction de l'ontologie, puis les choix du langage et de l'outil pour son développement et enfin son architecture générale et ses trois sous-ontologies avec leurs principaux concepts et relations.

Cette ontologie n'est bien sûr pas complète. Ce n'était pas l'objectif de ces travaux. Mais elle constitue une base solide pour contacter les différentes communautés et construire une ontologie coopérative.

1.3.1 La méthodologie de construction

Un modèle d'un monde réel complexe comme la V&V peut être explicitement représenté avec des objets existants dans le domaine et des relations entre ces objets.

Comme dans la majorité des domaines, le choix de ces éléments est une tâche complexe. Comme montré dans [FDH11], une construction rigoureuse d'une ontologie nécessite l'utilisation de méthodologies et plates-formes de développement.

Plusieurs méthodes de modélisation d'ontologies sont proposées dans la littérature [GPFLC91] [FL99] [JBCV98] [BCC06] ainsi que des efforts pour les unifier [Usc96].

Une méthodologie décrit principalement les lignes directrices pour la spécification, la conceptualisation, la formalisation et la mise en œuvre de l'ontologie.

La phase de spécification définit les objectifs et les rôles de l'ontologie, ainsi que les personnes qui peuvent l'utiliser. Pendant la phase de conceptualisation d'une ontologie, une représentation générale du domaine est construite à partir de la connaissance du domaine et enrichi par la suite à partir de différentes sources de littérature (différents ouvrages et différentes ressources sur internet) comme l'état de l'art des projets CESAR[6] IEEE 1471 [Hil00].

Dans sa représentation simple, une ontologie conceptuelle est un graphe dont les sommets sont les objets, les concepts et les entités du domaine. Les arcs sont des lignes d'interconnexion entre les sommets représentants les relations entre les composantes du domaine. Dans notre cas, le schéma de la Figure 1.4 est la représentation de l'architecture générale de la VVO comme un graphe conceptuel. Nous avons exploité les principes de conceptions suivants pour construire cette représentation.

[6] `https://cesarproject.eu/` et Ptolemy[7] et le standard

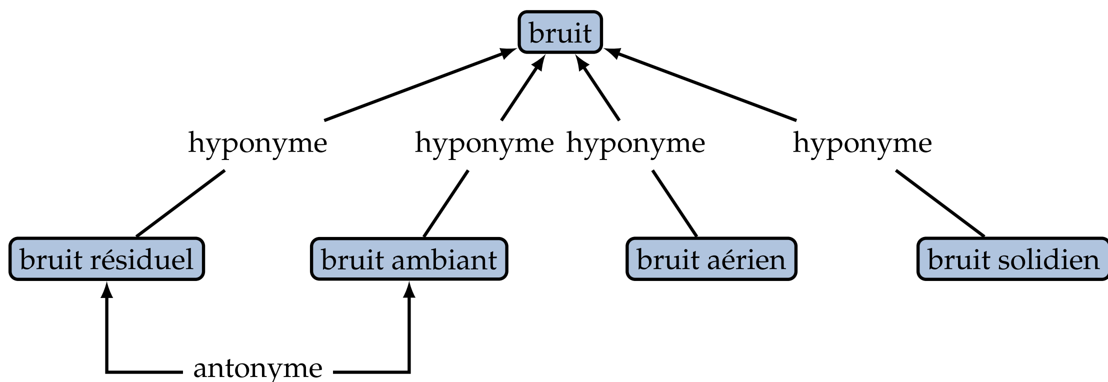


Figure 1.3 — Un réseau terminologique partiel associé au terme bruit

1.3.1.1 Les principes de conception

Contrairement aux terminologies simples qui se concentrent exclusivement sur les aspects syntaxiques (un réseau terminologique est un ensemble de termes candidats reliés entre eux par des relations lexicales [BAG03]), une relation lexicale est une relation entre termes (hyponymie, hyperonymie, holonymie, méronymie, synonymie, antonymie, causalité, etc.) [Cru86]. Les ontologies peuvent en outre fournir des définitions formelles qui sont manipulables par une machine et permettent ainsi à d'autres applications d'utiliser leurs significations réelles. Un exemple d'une terminologie simple tirée de [DS+08] est présenté sur la figure 1.3.

De nombreuses solutions sont en général possibles lors de la conception d'une ontologie. Pour guider et évaluer celle-ci, des critères objectifs basés sur l'objectif de l'ontologie sont nécessaires. La VVO est basée sur les principes de conception suivants (inspirés largement de WSMO [RKL+05] et de [G+95]) :

Clarté Les termes de l'ontologie doivent être suffisamment clairs pour exprimer le sens. Même si les concepts sont forcément définis initialement dans un contexte spécifique comme le domaine de V&V, cette définition doit être aussi indépendante du contexte que possible.

Notre expérience dans la construction d'ontologies nous a permis de constater qu'il y a confusion entre les concepts. Nous avons noté que les difficultés proviennent du fait que, même si nous avons partagé une compréhension générale pour les constructions choisies, il y avait de nombreuses différences subtiles dans nos conceptualisations individuelles qui constituaient un obstacle à la compréhension mutuelle. En outre, nous avons découvert que certains termes ont plusieurs significations, toutes également valides, mais que nous n'avions pas suffisamment différenciées, ce qui conduit à une confusion. Ces confusions sont résolues à travers l'introduction des définitions semi-formelles comme des commentaires dans l'ontologie. Donc, toutes les définitions présentées dans la VVO sont dans la mesure du possible complètes et documentées à l'aide de commentaires en langage naturel.

Cohérence L'ontologie doit assurer que les conclusions sont compatibles avec les définitions et les assertions (axiomes) définies. La description formelle du langage OWL 2 permet d'assurer la cohérence. Elle permet aussi de raisonner sur les concepts, le principe est d'étudier les assertions proposées et d'en déduire de nouvelles propriétés à l'aide d'un raisonneur ou moteur d'inférence comme Pellet[8] [SPG+07] et Racer[9] [HM01]. La cohérence doit également s'appliquer à des concepts qui sont définis d'une façon informelle, tels que ceux décrits dans la documentation en langage naturel et les exemples. Si un axiome (hypothèse sur le contexte) est incohérent (contradictoire) avec un exemple ou une définition, l'ontologie est incohérente. Toutes les définitions des concepts et des relations entre concepts dans la VVO sont définies d'une manière cohérente. Dans le cadre de la VVO, nous avons utilisé les deux moteurs d'inférences FacT++[10] [Tsa] et HermiT[11] [SMH08] qui implémentent des algorithmes permettant de vérifier la cohérence d'une ontologie, identifient les contraintes sur les relations entre les classes et permettent ainsi de structurer les ontologies.

Extensibilité Une ontologie doit être conçue pour anticiper les usages du vocabulaire commun. Elle doit offrir un fondement conceptuel pour un ensemble de tâches prévues et la représentation doit être conçue de telle sorte que l'on puisse l'étendre et la spécialiser. Les concepts clés de la VVO peuvent être étendus comme expliqué par la suite pour enrichir la VVO et pour construire d'autres ontologies. Plusieurs outils peuvent gérer l'importation d'ontologies comme l'outil Protégé que nous avons utilisé. Une fois l'ontologie importée, elle peut être modifiée ou étendue avec d'autres concepts, relations et individus.

Biais de codage minimal La conceptualisation doit spécifier les connaissances sans dépendre d'un niveau particulier de codage symbolique (un langage d'implémentation particulier). Un biais de codage peut apparaître quand les choix de la représentation sont faits purement pour des convenances de notation ou d'implémentation. La VVO est écrite d'une manière générique dans le langage formel OWL 2, ce qui permet d'utiliser et partager les concepts de cette ontologie par des applications utilisant des langages différents.

Engagements ontologiques minimaux Une ontologie doit exiger un engagement minimal suffisant pour permettre de partager les connaissances du domaine mais ne doit pas contenir des éléments supplémentaires. La VVO fait le minimum possible d'hypothèses sur le monde de la V&V, permettant de spécialiser et d'instancier l'ontologie si nécessaire. L'engagement ontologique est basé sur la pertinence de l'utilisation du vocabulaire, il peut être minimisé en spécifiant la théorie minimale qui permet de supporter le monde modélisé en définissant seulement les termes qui sont essentiels pour la communication des informations pertinentes dans cette théorie.

Respect des normes du web La VVO utilise le concept d'URI (Universal Resource Identifier) pour identifier d'une manière unique les ressources comme un principe de conception

[8] http://clarkparsia.com/pellet/
[9] http://www.sts.tu-harburg.de/~r.f.moeller/racer/
[10] http://owl.man.ac.uk/factplusplus/
[11] http://hermit-reasoner.com/

essentiel. La VVO adopte le concept d'espace de nom pour annoter l'espace des informations, supporte XML et les autres recommandations du W3C comme la décentralisation des ressources. Par exemple l'URI du concept `Formalism` de la VVO est http://www.irit.fr/~Mounira.Kezadri/ontologies/VVO.owl#Formalism et l'URI du formalisme `AutomataBased` est http://www.irit.fr/~Mounira.Kezadri/Ontologies/VVO.owl#AutomataBased.

La description versus l'implémentation La VVO fait la distinction entre les descriptions des éléments (les méthodes) et des technologies exécutables (les outils). Ceci nécessite un cadre de la description basé sur les formalismes appropriés et les outils existants et émergents qui sont décrits comme des individus. La VVO vise à fournir un modèle de description ontologique approprié pour le domaine et d'être compatible avec les techniques existantes et émergentes.

Pendant la phase de mise en œuvre de l'ontologie, celle-ci est formellement représentée à l'aide d'un des langages du Web sémantique en exploitant une des plates-formes d'édition d'ontologies.

1.3.1.2 Le choix du langage

Lorsqu'elles sont utilisées sur le Web, les ontologies précisent des conditions standards et des définitions lisibles par la machine.

L'architecture du Web sémantique est une architecture fonctionnelle évolutive. Berners Lee a défini trois niveaux distincts qui introduisent progressivement des primitives expressives : la couche métadonnées, la couche schéma et la couche logique.

RDF[12] (Resource Description Framework) et le schéma RDF[13] sont des langages généraux pour la représentation des métadonnées sur le Web et sont positionnés sur la couche de métadonnées et la couche schéma. Le langage d'ontologie pour le web (OWL (Web Ontology Language) [MVH+04]) (positionné sur la couche logique) est un langage de balisage sémantique conçu pour être utilisé par des applications qui ont besoin de traiter le contenu de l'information au lieu de présenter simplement les informations à l'être humain. OWL facilite plus d'intelligibilité du contenu Web que celui assuré par XML, RDF et RDFS en fournissant un vocabulaire supplémentaire avec une sémantique formelle [DGD06].

OWL est développé par le W3C[14] (Web Ontology Working Group) et publié en 2004. C'est un langage qui implémente des mécanismes pour l'égalité des individus, des classes, et des propriétés. Deux classes peuvent être déclarées comme équivalentes à travers `equivalentClass`, deux classes équivalentes ont les mêmes instances. Deux propriétés peuvent être déclarées équivalentes en utilisant `equivalentProperty`. L'égalité peut être utilisée pour définir des relations synonymes. Deux individus peuvent être déclarés comme représentant la même chose avec la relation `sameAs`. Des restrictions sur les propriétés (comme : le domaine et la cible de propriétés, la transitivité, le cardinal... etc) peuvent être

[12] http://www.w3.org/RDF/
[13] http://www.w3.org/TR/rdf-schema/
[14] http://www.w3.org/

exploitées.

La spécification OWL peut être considérée comme un modèle (la notion de modèle sera présentée dans la section 2.1) du langage de modélisation OWL, il est donc un métamodèle (la notion de métamodèle sera présentée dans la section 2.1).

Afin de permettre la vérification syntaxique automatique de la validité d'ontologie, ce métamodèle linguistique de OWL doit être spécifié en utilisant un langage de métamodélisation. C'est le rôle du langage RDF Schéma qui est une partie de la spécification officielle de OWL. Ce schéma peut être considéré comme un métamodèle linguistique du langage OWL. Pour importer des ontologies OWL à la MDA (Model Driven Architecture), un métamodèle linguistique de OWL a besoin d'être représenté par un langage MDA. C'est le but de l'initiative ODM [OMG09], à savoir, utiliser MOF pour développer un méta-modèle pour des ontologies.

Nous avons choisi le langage OWL 2 (une extension du OWL) qui est mature, standardisé, pour lequel existent des outils pour la conception et l'entretien et qui est suffisamment expressif pour ce domaine. Le langage OWL 2 est un langage d'ontologies pour le web sémantique avec une sémantique bien définie, il est une recommandation du W3C [MPSP+09] depuis 2009.

1.3.1.3 Le choix de l'outil

Protégé[15] [HKR+04] (version 4.1 Alpha) a été choisi parmi les outils existants de développement d'ontologies exploitant OWL. Il s'agit d'une plate-forme open-source, mature, évolutive et extensible qui est largement utilisée dans le monde. Protégé fournit une interface graphique interactive pour la conception, l'affichage et la manipulation d'ontologies. Sa structure interne représente les éléments de l'ontologie : les classes, les propriétés, les contraintes et les instances. Protégé peut être utilisé pour définir des ontologies en OWL 2 et dans différents autres formats comme RDF[16].

1.3.2 La structure générale de la VVO

La première phase de la définition de la VVO est l'énumération des termes et relations importantes dans le domaine de la V&V. La Figure 1.4 montre la structure globale de l'ontologie, elle représente les éléments de base et les relations entre eux.

Nous avons choisi de mettre en valeur les concepts fondamentaux suivants : `Formalism`, `System`, `Properties`, V&V, `SystemAbstraction`, `View` et `ViewPoint`. La plupart figurent dans le standard IEEE 1471 [Hil00] (`System`,`View` et `ViewPoint`) ou font partie de l'objectif de cette ontologie (V&V, `Model`, `Formalism`, `SystemAbstraction` et `Requirement`). Les relations principales spécifiques pour le domaine qui sont utilisées pour lier ces concepts sont : `describes`, `verifies`, `concerns`, `checks`, `specifies`, `refines`, `abstracts` et `conformsTo`.

Comme la figure 1.4 le montre, un système peut être décrit à l'aide de différentes abstrac-

[15] http://protege.stanford.edu/
[16] http://www.w3.org/RDF/

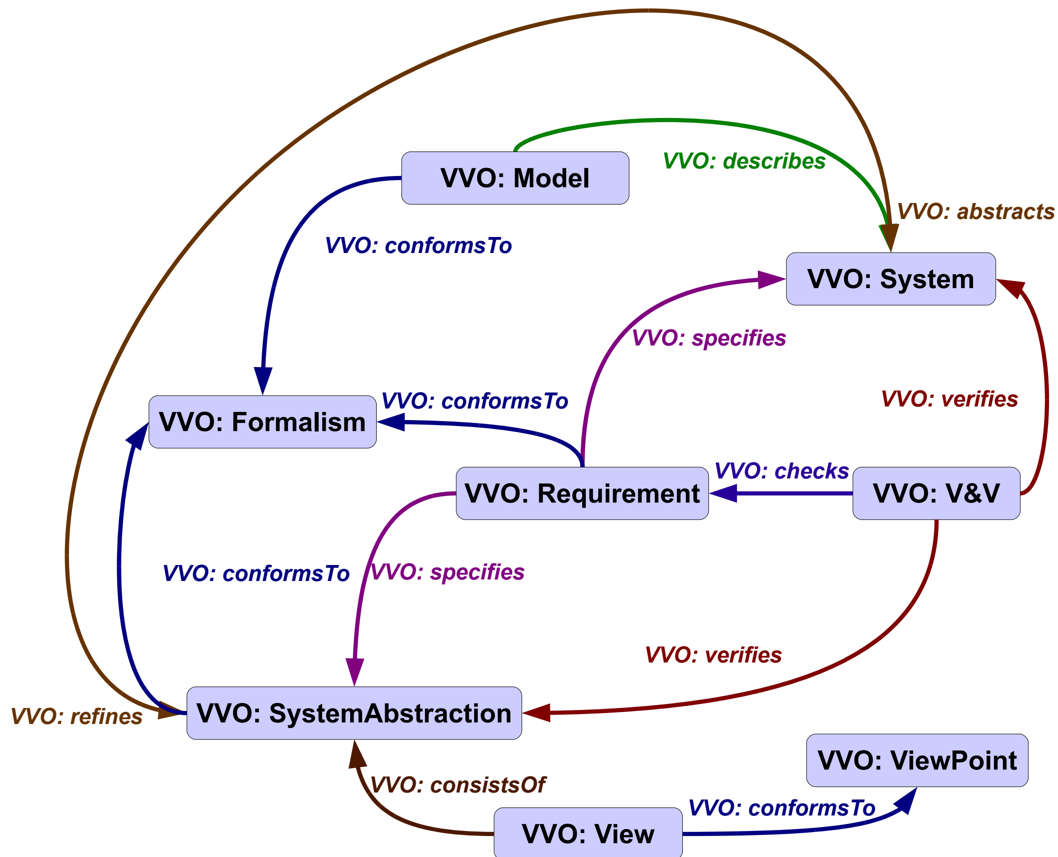


Figure 1.4 — L'architecture globale du VVO

tions. Chaque abstraction est exprimée à l'aide d'un langage de modélisation (`Formalism`) en conformité avec une certaine vue (`View`). Nous pouvons associer des propriétés aux systèmes ou aux abstractions de systèmes. L'abstraction est une technique qui permet de réduire la complexité d'un système en conservant une partie de son comportement, de sorte que le système simplifié soit plus accessible à des outils d'analyse et reste suffisant pour exprimer certaines propriétés de sûreté [Tiw08]. Un système avec ou sans propriétés associées peut être vérifié en utilisant une des techniques de V&V.

Dans les sous-sections qui suivent, nous présentons la définition et la hiérarchie des concepts de base de cette ontologie de domaine. La VVO est constituée principalement de trois sous-ontologies : une ontologie pour les vues, une ontologie pour les formalismes, et une ontologie pour les techniques de V&V.

1.3.3 Une ontologie pour les formalismes de description

Le but de cette sous-ontologie est d'introduire quelques concepts centraux de la méthodologie pour la modélisation, elle est considérée comme la base pour structurer les moyens de modélisation dédiés. Les principaux concepts liés à cette sous-ontologie sont :

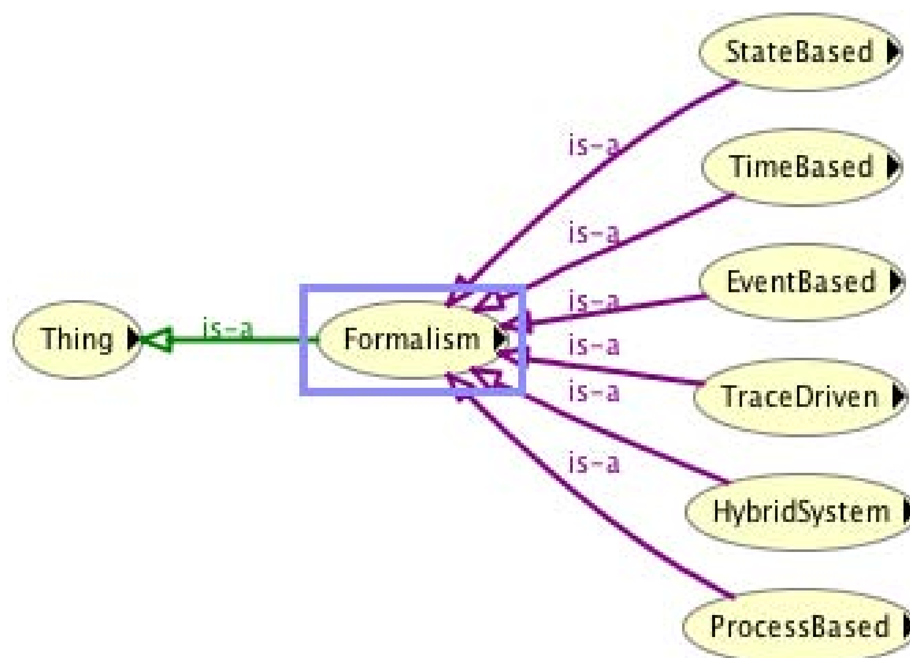


Figure 1.5 — Une partie du premier niveau de la hiérarchie des formalismes

Le concept `Formalism` Un formalisme est un support de modélisation qui permet la description de la structure et/ou la dynamique d'un système. La structure décrit les relations statiques entre les entités du système. La dynamique décrit le comportement des entités du système et son évolution temporelle. Ces deux points de vue d'un même système ont donné naissance à une multitude d'outils. L'ontologie des formalismes collecte un grand nombre de formalismes largement utilisés pour la modélisation du comportement des systèmes. Comme le nombre de formalismes est très grand, une classification de ces formalismes est proposée. Une petite partie de cette classification est présentée dans la Figure 1.5.

Nous présentons sur la Figure 1.6 une partie de la hiérarchie du formalisme `Automata`. `Automata` [HMU79] est une sous-classe du formalisme `state based`, il y a parmi ses concepts : `hybrid automata` [ACHH93], `Büchi automata` [B6̈0], `hierarchical automata` [MLS97], `finite automata` [Law05], `Muller automata` [Per04], `cellular automata` [CFG10], `timed automata` [BY03] et `stochastic automata` [DK05]. Chaque type de formalisme possède ses propres caractéristiques et peut avoir sa propre hiérarchie. Des combinaisons entre les classes sont aussi possibles, par exemple, la classe `AutomataBased` appartient en même temps à la hiérarchie de la classe `Automata` (qui fait partie de la hiérarchie de `StateBased`) et la hiérarchie de `TimeBased`.

à chaque classe, nous associons des annotations qui sont la définition du concept et son origine. Le concept `FiniteAutomata` par exemple est défini dans la VVO comme illustré sur la figure 1.7.

Le concept `FormalismElement` Ce concept est défini pour regrouper les différents éléments pouvant intervenir dans la définition de formalismes. Une partie de la hiérarchie du concept `FormalismComponent` est montrée pour l'illustration sur la figure 1.8. Sa hiérar-

Figure 1.6 — Une partie de la hiérarchie du formalisme Automata

Figure 1.7 — La définition du concept `FiniteAutomata`

chie est accessible avec l'ontologie, sa représentation peut être largement améliorée en terme de population et de relations entre concepts.

La relation `describes` Elle a comme relation inverse `isDecribedBy`. C'est une propriété primitive entre la classe `Formalism` et les deux classes `System` et

Figure 1.8 — Une partie de la hiérarchie du FormalismElement

SystemAbstraction pour spécifier le fait qu'un formalisme est utilisé pour décrire un système ou une abstraction de système. Cette relation peut être spécifiée par des sous-relations qui peuvent porter sur les instances de chacune des deux classes.

La relation concerns C'est une relation entre les instances des classes FormalismComponent et Formalism. Elle est utilisée pour décrire le fait qu'un élément est utilisé dans un formalisme particulier. Elle a comme relation inverse isConcernedBy. Un exemple est la relation entre le formalisme Automata et les composants de formalismes State et Transition.

1.3.4 Une ontologie pour les vues

Les vues sont des représentations de l'architecture globale qui sont significatives pour un ou plusieurs intervenants dans le système. Le développement de systèmes complexes exige la prise en compte de plusieurs vues [RW11].

La difficulté est de combiner, relier et gérer les abstractions du système en relation avec les vues.

Le but de cette sous-ontologie est de décrire le domaine des vues. Ses concepts sont principalement dérivés du standard IEEE 1471 [Hil00] et concernent les vues de description d'architecture.

Le concept System Un système est un assemblage de composants organisés pour accomplir une ou un ensemble de fonctions spécifiques. Barros définit dans [Bar97] un système comme un modèle abstrait qui décrit comment les entités se comportent dans le temps. Il décrit les sorties selon les entrées et les informations sur l'état du système. Un système peut être décrit selon plusieurs vues exprimées dans plusieurs formalismes de description (la relation describes dans la Figure 1.4). Enfin, le système est vérifié en utilisant certaines techniques de V&V (la relation verifies dans la Figure 1.4).

Le concept SystemAbstraction Les systèmes actuels sont généralement très complexes et plusieurs abstractions sont souvent nécessaires pour gérer leur description. Une abstraction est utilisée dans la description d'une certaine vue et conforme à un certain formalisme, elle peut être caractérisée par des propriétés qui doivent être vérifiées à l'aide de certaines techniques de V&V.

Le concept ViewPoint Il s'agit d'une spécification des conventions utilisées pour la construction et l'exploitation des vues (Views). C'est aussi un pattern à partir duquel des vues individuelles peuvent être développées. Il peut être vu comme une abstraction obtenue en sélectionnant un ensemble de constructions et des règles dans le but de se concentrer sur des préoccupations particulières au sein d'un système.

La Figure 1.9 montre une partie de la classification des ViewPoints dans la VVO, les définitions des ViewPoints présentées sont tirés de [RW11]. Cette classification est non exhaustive et peut être liée avec des éléments de la hiérarchie des vues.

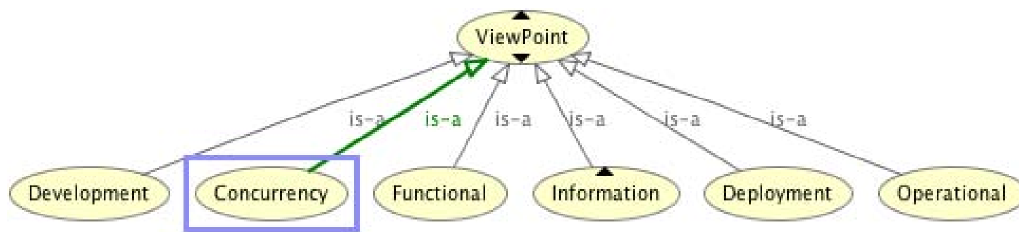


Figure 1.9 — Une partie de la hiérarchie des points de vue

CLASS EDITOR for VVO:Concurrency (instance of owl:Class)

For Class: http://www.irit.fr/~Mounira.Kezadri/Ontologies/VVO.owl#Concurrency — Inferred View

Annotations

Property	Value	Lang
rdfs:comment	Describes the concurrency structure of the system and maps functional elements to concurrency units to clearly identify the parts of the system that can execute concurrently and how this is coordinated and controlled. This entails the creation of models that show the process and thread structures that the system will use and the interprocess communication mechanisms used to coordinate their operation.	
rdfs:isDefinedBy	Book: Software systems architecture: working with stakeholders using viewpoints and perspectives Chapter: INTRODUCTION TO THE VIEWPOINT CATALOG Rozanski, N. and Woods, E.	

Figure 1.10 — La définition du concept concurrency

La définition des points de vue actuelle est préliminaire et doit être précisée et détaillée.

La définition du concept de `Concurrency` est présentée dans la figure 1.10 ainsi que sa source de définition dans la VVO.

Le concept `View` Il s'agit d'une représentation d'un système complet selon un `ViewPoint`. Selon le standard, une vue doit être conforme à exactement un seul `ViewPoint` et doit être conforme à la spécification du point de vue correspondant. Ceci est représenté par la relation `conformsTo` dans la figure 1.4. Par conséquent, les éléments de la hiérarchie des points de vues doivent être liés à des éléments de la hiérarchie des vues.

La relation `conformsTo` C'est une relation entre les concepts `View` et `ViewPoint` et aussi entre chacun des concepts `SystemAbstraction`, `Requirement` et `Model` et le concept `Formalism`. Elle est utilisée pour définir le fait qu'une vue est conforme à un certain point de vue et qu'une abstraction d'un système est conforme à une certaine vue. Des sous-relations entre les instances des trois concepts peuvent être définies.

1.3.5 Une ontologie pour les techniques de V&V

La correction d'un système vis-à-vis du comportement désiré est vérifiée avec une certaine méthode de V&V, qui détermine si le modèle considéré du système satisfait une formule décrivant ce comportement appelée `Property`. La plus grande partie de ce travail est la classification des techniques de V&V.

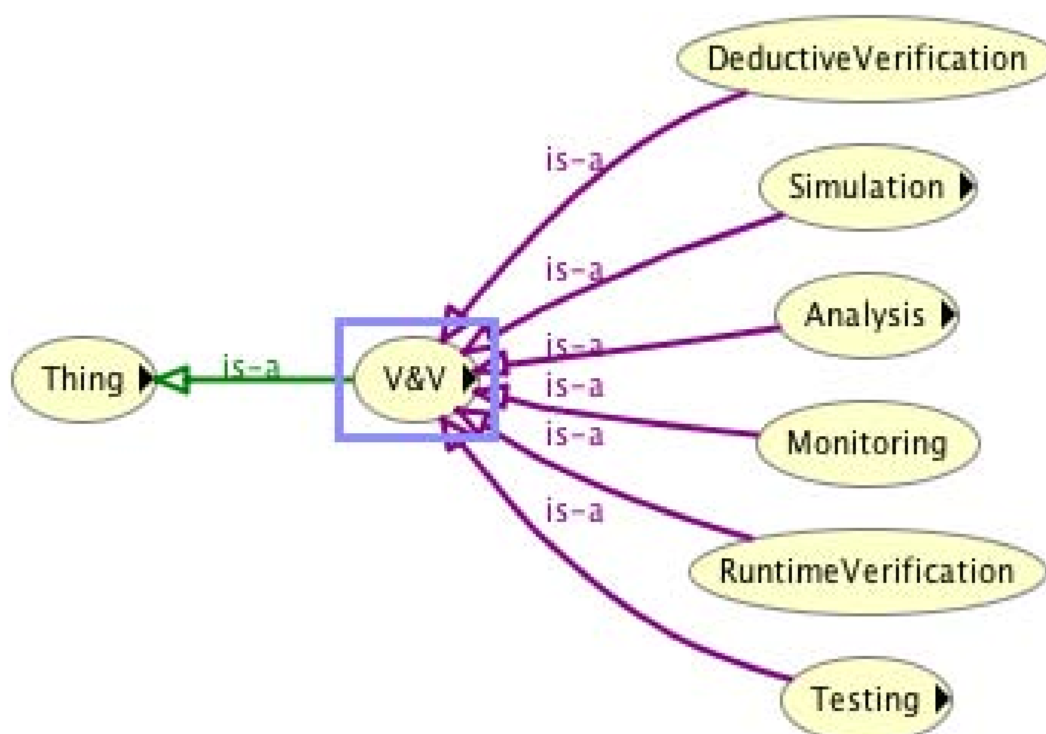


Figure 1.11 — Une partie de la hiérarchie des techniques de V&V

Le concept V&V Une grande variété de stratégies et techniques de V&V sont disponibles. Une technique de V&V peut être appliquée sur une ou plusieurs abstractions d'un système selon le formalisme utilisé pour la description du système et la propriété qui doit être vérifiée.

Par technique de vérification, nous entendons une technique permettant de déterminer si un système satisfait certaines exigences explicites (spécification de systèmes) et donc que le système est conforme à sa spécification. Par technique de validation, nous entendons une technique qui permet d'assurer que le modèle est correct par rapport aux besoins implicites de l'utilisateur futur du système et donc que le système est conforme aux besoins de l'utilisateur (c'est exactement ce que voulait exprimer l'utilisateur). L'objectif est d'augmenter la confiance que nous avons sur le système développé. Cela peut être fait avec différentes approches.

La figure 1.11 montre une petite partie de la hiérarchie proposée dans la VVO pour les techniques de V&V. Cette hiérarchie est organisée sur plusieurs niveaux. Le concept `Analysis` par exemple possède 60 sous-catégories.

Les techniques de V&V illustrées sur la Figure 1.11 sont utilisées pour la V&V matérielle et logicielle. Baier, C. et al. dans [BK+08] montrent que le `PeerReview` et `Testing` sont les techniques de vérification logicielle les plus utilisées en pratique. Le `PeerReview` est de préférence effectué par une équipe d'ingénieurs qui n'étaient pas impliqués dans le développement du logiciel. Selon cette technique, le code source n'est pas exécuté mais examiné statiquement. Les études montrent que cette technique est efficace et permet de dé-

CLASS EDITOR for VVO:PeerReview (instance of owl:Class)

For Class: http://www.irit.fr/~Mounira.Kezadri/Ontologies/VVO.owl#PeerReview — Inferred View

Annotations

Property	Value	Lang
rdfs:comment	a process or meeting during which a software product is presented to project personnel, managers, users, customers, user representatives, or other interested parties for comment or approval.	
rdfs:isDefinedBy	IEEE 1028-1997	

Figure 1.12 — La définition du `PeerReview`

CLASS EDITOR for VVO:Testing (instance of owl:Class)

For Class: http://www.irit.fr/~Mounira.Kezadri/Ontologies/VVO.owl#Testing — Inferred View

Annotations

Property	Value	Lang
rdfs:comment	Testing is the process of executing a program with the intent of finding errors.	
rdfs:isDefinedBy	Book: The art of software testing. Glenford J. Myers, Tom Badgett, Todd M. Thomas, Corey Sandler	

Figure 1.13 — La définition du `Testing`

tecter entre 31% et 93% des défauts. `PeerReview` est définie dans [GRL07] et dans la VVO comme illustré sur la figure 1.12.

Le test logiciel constitue aussi une partie significative de n'importe quel projet. Entre 30% et 50% du coût du projet lui est consacré [BK+08]. Contrairement au `PeerReview` qui permet d'analyser le code sans l'exécuter, le test est une technique dynamique qui exécute le code.

Le test permet pour le code compilé d'une partie du logiciel avec des paramètres appelées jeux de tests de déterminer la correction de l'ensemble de chemins d'exécution traversés. En se basant sur l'observation durant l'exécution, les résultats du logiciel sont comparées aux résultats attendus selon la spécification

L'avantage principal du test est qu'il peut être appliqué à tous les types de logiciels ou de systèmes (des applications, des compilateurs ou des systèmes d'exploitation).

Éviter les erreurs dans la conception matériel est vital. Le matériel a généralement un coût de fabrication important ; corriger les défauts puis transmettre une mise à jour aux utilisateurs est difficile au contraire des erreurs logiciels. Corriger les erreurs matérielles après la livraison nécessite souvent la refabrication et le redistribution, ce qui a des conséquences économiques énormes. Par exemple le remplacement du processeur Pentium II a coûté à Intel environ 475 millions $. Des études ont montré que plus de 50% de tous les ASICs (Application-Specific Integrated Circuits) ne fonctionnent pas correctement après le design initial et la fabrication. Ce n'est pas étonnant que les constructeurs invertissent beaucoup de leur temps pour assurer que leur design est correct. La conception dans un matériel typique prend environ 27% du temps total ; le reste du temps est consacré à la détection des erreurs.

L'émulation (Figure 1.14), la simulation et l'analyse structurelle sont les techniques majoritairement utilisées dans la vérification matérielle. L'analyse englobe aussi plusieurs tech-

CLASS EDITOR for VVO:Emulation (instance of owl:Class)

For Class: http://www.irit.fr/~Mounira.Kezadri/Ontologies/VVO.owl#Emulation Inferred View

Annotations

Property	Value	Lang
rdfs:comment	Builds a version of system using programmable logic. It can be used, for example, to boot the operating system on a processor.	
rdfs:isDefinedBy	WIKIPEDIA	

Figure 1.14 — La définition de `Emulation`

CLASS EDITOR for VVO:Simulation (instance of owl:Class)

For Class: http://www.irit.fr/~Mounira.Kezadri/Ontologies/VVO.owl#Simulation Inferred View

Annotations

Property	Value	Lang
rdfs:comment	Ensure that a finite number of user-defined system trajectories meet the desired specification.	
rdfs:isDefinedBy	Verification using Simulation Antoine Girard and George J. Pappas	

Figure 1.15 — La définition de `Simulation`

niques spécifiques comme la synthèse, l'analyse temporelle et le test d'équivalence. L'émulation est un type de test ; dans cette technique un système matériel reconfigurable est paramétré pour se comporter comme le circuit considéré. Comme le test logiciel, l'émulation exploite un ensemble de stimuli du circuit et compare les sorties générées avec celle prévues dans la spécification. Pour tester intégralement le circuit, toutes les combinaisons d'entrés possibles dans chaque système doivent être examinées. Ceci n'est pas pratique et le nombre de tests doit être réduit d'une manière significative, ce qui peut mener à des erreurs non découvertes.

Avec la simulation (Figure 1.15), un modèle est décrit typiquement avec un langage de description de matériel comme Verilog [TM02] ou VHDL [Ash08] qui sont standardisés par l'IEEE ; en se basant sur des stimuli, les chemins d'exécution sont examinés en utilisant le simulateur. Les stimuli peuvent être données par un utilisateur ou automatiquement par un générateur. Une non-correspondance entre la sortie des simulateurs et la sortie décrite dans la spécification détermine la présence des erreurs [BK+08].

Le concept `Requirement` La correction d'un système par rapport à un comportement spécifique est contrôlée en vérifiant que le modèle satisfait une formule décrivant le comportement désiré appelé `Property`. Pour décrire une propriété, nous pouvons utiliser plusieurs langages de description de propriétés (PDL). A titre d'exemple, la figure 1.16 présente une partie de la hiérarchie des logiques temporelles [Pri03] que nous utilisons dans le cas d'étude dans la Section 1.4.

Nous pouvons distinguer plusieurs logiques temporelles, la logique temporelle linéaire (LTL) [Pnu77], "Computational Tree Logic" (CTL) [EH86] [CE08] et "State Event-LTL" (SE-LTL) [CCO+04] sont des sous-classes de la logique temporelle présentée sur la Figure 1.16.

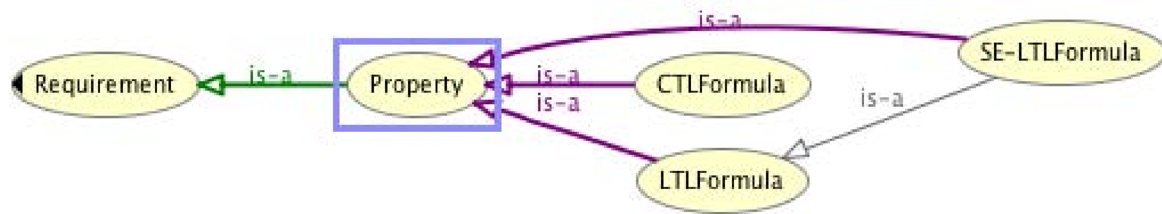


Figure 1.16 — la hiérarchie de TemporalLogic

Une formule SE-LTL peut être associée à un certain modèle décrit dans le formalisme réseaux de Petri (la hiérarchie des réseaux de Petri est illustrée sur la Figure 1.17). Ce type de propriétés associé aux réseaux de Petri peut être vérifié en utilisant la technique du model checking. Le model checking [BK+08] est une technique initiée dans deux travaux indépendants dans les années quatre-vingt, celui de Clarke et Emerson [EC80] (qui ont inventé eux mêmes le terme model checking) et celui de Queille et Sifakis [QS82].

La relation `verifies` C'est une relation qui peut être utilisée entre les instances de V&V (les méthodes et outils de Vérification et Validation) et les instances de chacun des concepts `Property`, `System` et `SystemAbstraction`. Elle a comme relation inverse `isVerifiedBy`.

La relation `specifies` C'est une relation entre le concept `Property` et le concept `System`. Elle est utilisée pour décrire le fait qu'une propriété est utilisée pour spécifier des contraintes sur un système particulier. Elle a comme relation inverse `isSpecifiedBy`.

1.4 étude de cas

Nous présentons l'exemple de la vérification de réseaux de Petri, aussi appelés automates état/transition. Les réseaux de Petri qui font partie des formalismes classés dans la VVO, sont composés de plusieurs sous-classes, une partie de la hiérarchie des réseaux de Petri est présentée sur la figure 1.17.

Les éléments que nous souhaitons réutiliser ici sont les outils de V&V. Ce sont les composants d'une chaîne de développement.

Nous instancions les sous-concepts du concept V&V de l'ontologie avec des outils de V&V, comme l'outil TINA[17] [BRV04] utilisé dans cette étude de cas.

Protégé OWL offre la possibilité d'interrogation d'ontologies à travers l'onglet DL_Query[18]. La syntaxe de requêtes d'interrogation est in uencée par la syntaxe abstraite de OWL et celle de la logique de description. Cette syntaxe utilise le symbole de la quantification universelle et existentielle. Cette syntaxe est beaucoup moins verbeuse et plus facile à écrire et à comprendre. La syntaxe utilise des mots clés comme `some`, `only` et `not` avec une notation infixée.

[17] `http://projects.laas.fr/tina/`
[18] `http://protegewiki.stanford.edu/wiki/DL$_$Query`

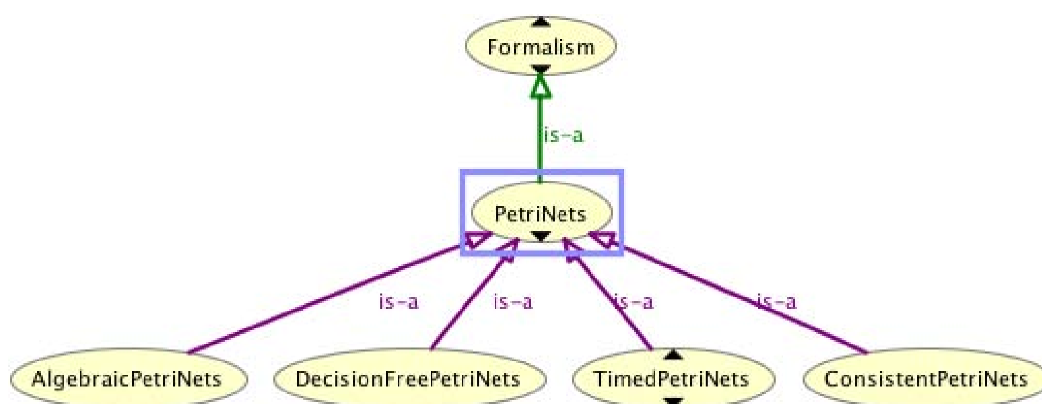


Figure 1.17 — la hiérarchie des PetriNets

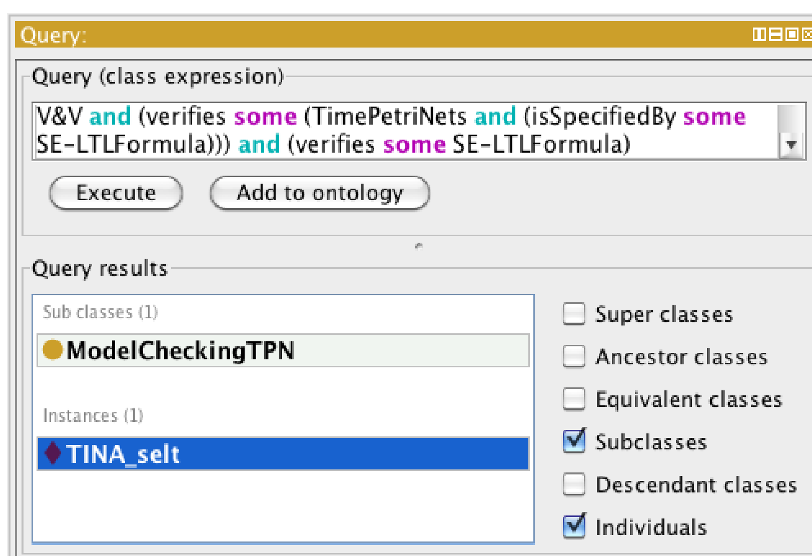


Figure 1.18 — Un exemple de requête

Grâce à DL_Query, faire des requêtes sur la VVO est rendu simple. Par exemple, si nous voulons vérifier une formule SE-LTL (State/Event LTL) sur un système décrit à l'aide d'un réseau de Petri temporisé, la requête et son résultat sont présentés sur la figure 1.18.

La requête exige que la technique de V&V supporte le formalisme réseaux de Petri et les formules SE-LTL et qu'une propriété sur le formalisme réseaux de Petri peut être définie dans la logique SE-LTL comme présenté sur la figure 1.18. Le résultat est l'outil TINA-Selt, spécialisé dans la vérification des formules SE-LTL sur des systèmes spécifiés sous la forme de réseaux de Petri.

1.5 D'autres travaux sur les ontologies

Beaucoup d'efforts visent à la construction d'ontologies universelles (comme cyc[19]) dans le but de permettre aux communautés de converger à travers les ontologies mais les concepts liés à la V&V sont peu développés.

Un certain nombre d'ontologies de domaine sont disponibles sur internet couvrant plusieurs domaines comme la médecine (Ménélas [Zwe94]), le e-gouvernement et les systèmes juridiques [VBC98], le biomédical [BB05], la gestion de l'information économique et financière [CFL+04], et les virus biologiques (BVCO). Le projet SWEET[20] (Semantic Web for Earth and Environmental Terminology) fournit un cadre sémantique commun pour un ensemble d'initiatives de représentation des sciences de la terre. Les ontologies permettent de décrire la structure sémantique de ces domaines.

À notre connaissance, il n'existe pas d'ontologie pour le domaine de la V&V, mais une ontologie du test logiciel a été proposé dans [HZG03]. Nous avons inclus ses termes en relation avec le concept Test des techniques de V&V dans la VVO.

Une autre ontologie en relation avec le domaine de V&V est l'ontologie VV&A (Verification, Validation, and Accreditation) [Bla08] qui, contrairement à nos travaux, se focalise principalement sur la gestion de la documentation liée à la vérification, la validation et l'accréditation. Le but de cette ontologie est de faciliter la recherche des informations au sein des documents de VV&A. Ceci est effectué en précisant le contenu et la structure des documents standardisés de VV&A. Cette ontologie est développée également à l'aide de l'outil Protégé et le langage OWL. Par contre, les classes ne semblent pas être peuplées avec des instances actuellement. Nous nous focalisons principalement sur les méthodes et outils de modélisation et vérification. Il serait pertinent de combiner les deux approches pour positionner les résultats de vérification issu des outils dans les documents décrits par la VV&A.

L'ontologie la plus proche de la VVO est une ontologie pour les cours de MMISS (MultiMedia Instruction in Safe Systems) qui sont approuvés par IFIP WG1.3 (Foundations of System Specification) comme une bibliothèque pour les méthodes formelles.

Le but du projet MMISS[21] était de mettre en œuvre un système multi-media adaptatif sur internet pour l'éducation pour le domaine des méthodes formelles. Il s'agit de développer un système pour les cours sur le domaine de la sûreté logicielle. Le système constitue un atelier formel pour l'intégration des matériaux de cours et les lier en suivant une structure sémantique. La structure sémantique est particulièrement importante pour la compréhension du domaine des méthodes formelles, pour exprimer les différences et les similarités entre les techniques formelles et savoir quelle technique est adéquate pour une application particulière.

Pour la correspondance de concepts entre la VVO et l'ontologie de Wirsing, les concepts `Formalism`, `Verification` et `Analysis Techniques` de l'ontologie de Wirsing sont équivalents aux concepts `Formalism`, V&V et `Analysis` de la VVO.

[19] www.cyc.com
[20] http://sweet.jpl.nasa.gov/
[21] http://www.mmiss.de/

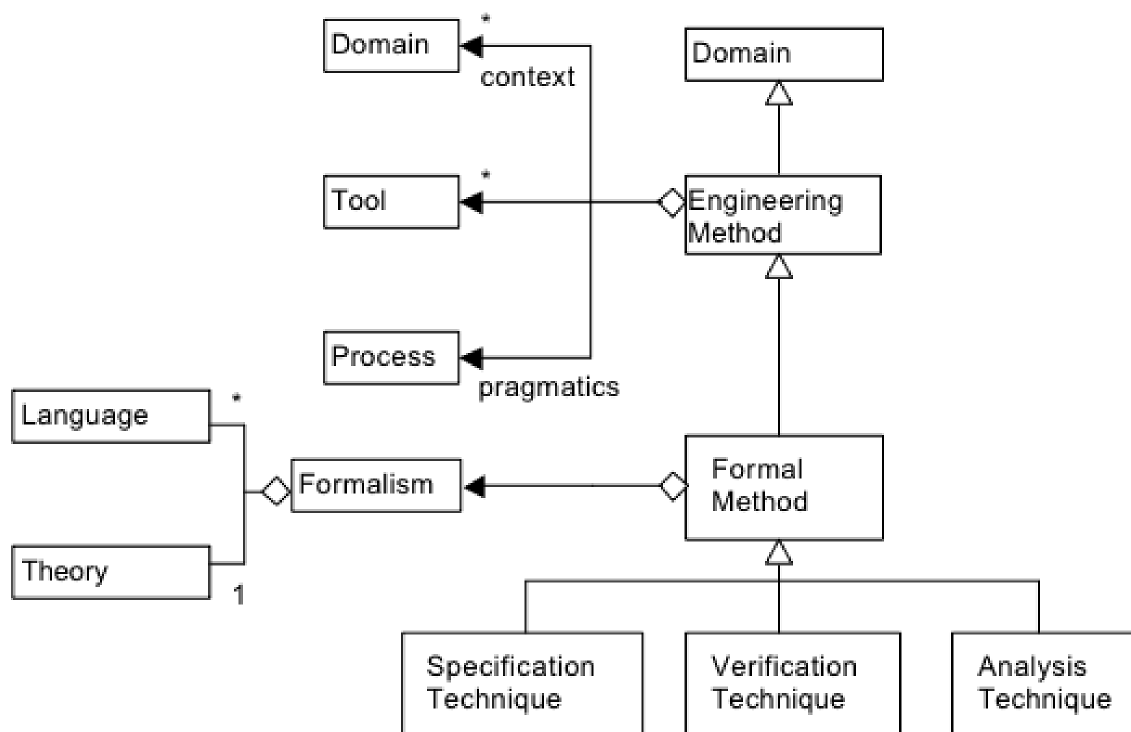


Figure 1.19 — Les concepts de base pour l'ontologie des méthodes formelles

L'ontologie de Wirsing [BW03] s'appuie sur plusieurs approches pour la classification et la définition des thèmes reliés aux méthodes formelles, comme le schéma de classification ACM [CCS98], le schéma des méthodes et techniques formelles de Clarke et Wing [CW96] et l'atelier de Steffen [SMB97] pour les outils des méthodes formelles.

L'ontologie est décrite en utilisant le langage UML. Pour la description, les diagrammes de classes et d'instances sont utilisés. Par exemple, les thèmes de l'informatique sont des instances du domaine.

La notion la plus générale pour la description d'un thème de recherche ou d'enseignement est la notion `Domain`. La classe `Engineering Method` est une spécialisation de la classe `Domain`.

Une `Engineering Method` est établie dans le contexte de `Domain` (aucun, un ou plusieurs), elle a des outils qui peuvent supporter la `Method` et des `Process` pour l'application de l'`Engineering Method`. La classe `Formal Method` est une spécialisation de `Engineering Method` avec la caractéristique particulière que chaque instance de `Formal Method` est basée sur un `Formalism`. Les `Formal Methods` sont classifiées comme `Specification`, `Verification` et `Analysis Techniques`. Un `Formalism` a un ou plusieurs `Languages` et une `Theory` est constituée de définitions et de théorèmes.

Une initiative de Renault pour une ontologie de système et sécurité a été proposé dans [CTG+11] dans le but d'assurer la cohérence du processus de conception tout en respectant la norme ISO 26262. Pour cette dernière ontologie, la prise en compte des exigences de la

norme ISO 26262 concernant l'utilisation des méthodes formelles comme des techniques de vérification est présentée comme une perspective.

Dans le cadre du projet RECOMP[22], la société Validas a élaboré en se basant sur les exigences de l'ISO-26262 [ISO12] pour le développement des applications critiques pour l'automobile, un modèle pour la qualification et la classification d'outils qui sont utilisés dans le processus de développement. Le travail considère cette problématique pour les plugins Eclipse. En prenant en compte les standards (ISO, EN, DO, IEC), le but final est de produire un kit pour la qualification des outils Eclipse par rapport aux standards ISO-26262 et DO-330 [RTC12b]. Le travail est intéressant pour enrichir les concepts `Exigence` et `View` de notre VVO.

Dans notre contexte, le DO-330 est le standard le plus pertinent. Il fait parti de la famille de standards de sûreté pour les logiciels embarqués en aéronautique DO-178 [RTC12a]. DO-330 concerne la qualification des outils utilisés pour le développement du logiciel soumis au DO-178. Il définit les règles de développement pour ces outils. Il s'agit d'une réécrire du DO-178 spécialisée pour les outils. Il a été conçu pour être applicable à tous les domaines exploitant du logiciel critique et a été accepté plus ou moins formellement par ces autres domaines. Il définit les niveaux de qualification pour les outils TQL (Tool Qualification Level, de 1 à 5) en fonction de leur rôle dans le développement des systèmes logiciels critiques selon le niveau de criticité du système considéré DAL (Design Assurance Level, de A à E). En fonction du niveau souhaité pour l'outil, le DO-330 définit des exigences à satisfaire dans son développement. L'intégration de ces informations dans la VVO ne pose aucun soucis majeur. Plusieurs approches plus ou moins détaillées peuvent être suivies. Soit nous ajoutons uniquement un attribut TQL avec la bonne relation d'ordre sur les valeurs (un outil de niveau n couvre les exigences de tous les niveaux supérieurs) et l'utilisateur fait des requêtes avec des valeurs de TQL. Soit nous ajoutons également un attribut DAL avec la bonne relation d'ordre sur les valeurs (un logiciel développé au niveau n couvre les exigences de tous les niveaux supérieurs) ainsi que la logique reliant les DAL et les TQL, et l'utilisateur fait des requêtes avec des valeurs de DAL ou de TQL. Et enfin, nous pouvons introduire toutes les exigences prescrites par le DO-330, et la logique qui permet de déduire la TQL obtenue. La VVO permet alors en fonction des informations sur le développement d'un outil d'en déduire automatiquement son TQL. Les deux premières approches s'intègrent dans la VVO sans difficulté. La dernière approche est beaucoup plus intéressante en terme de modélisation. Mais, d'une part, elle a déjà été abordée par la société Validas et d'autre part, les standards ne sont que des prescriptions qui sont ensuite interprétées par les utilisateurs et les autorités de certification en fonction de chaque produit certifié. Il aurait donc été nécessaire de faire participer à ces activités ces deux catégories d'utilisateurs ce qui n'a pas été possible car la DO-330 n'a été standardisée qu'en 2012. Nous avons donc choisi de ne pas pousser plus en avant ces travaux.

[22] http://www.recomp-project.eu/

1.6 Conclusion et perspectives

Ce chapitre avait pour but de proposer une conceptualisation générale de l'ontologie de V&V. La VVO contient les fondations pour les formalismes et les techniques de V&V, elle est utilisée actuellement dans un but de partage de connaissances. Sa structure générale ainsi que des éléments de sa population sont présentés avec plus ou moins de détails. L'ontologie complète est librement accessible. La version actuelle de la VVO contient : 259 classes, 14 relations hors héritage entre classes et 19 individus à titre d'exemples.

Cette ontologie peut être développée dans plusieurs directions. Dans le futur proche, la hiérarchie du concept `System` doit être développée avec plusieurs types de systèmes comme les systèmes temps-réel et les systèmes concurrents, en plus des liens avec les formalismes (les relations comme : un système peut être décrit dans le formalisme `Pi-Calculus` et un système temps réel peut être décrit avec le formalisme `Timed Automata`). Les opérations acceptées pour chaque type de système doivent être décrites. Par exemple, le formalisme `Automata` peut supporter la composition hiérarchique et parallèle des états. Enfin, nous pouvons développer les aspects liés à la qualification en relation avec les travaux réalisés par la société Validas.

Pour la validation de l'ontologie, nous pouvons noter qu'il n'y a pas qu'une seule ontologie correcte pour un domaine particulier. Comme la conception d'ontologies est un processus créatif ouvert, des ontologies spécifiées par différentes personnes pour un même domaine peuvent être différentes. La validation sémantique d'une ontologie de domaine nécessite la relecture par des experts de domaine ou l'utilisation d'un système expert pour générer le rapport de validation sémantique [Héo10].

La VVO résultante doit donc être validée en la reliant avec d'autres outils de V&V existants et en l'expérimentant avec d'autres systèmes. Le travail planifié est important et dépasse le cadre d'une thèse car il requiert la coopération de l'ensemble de la communauté pour enrichir à la fois les concepts et leurs instances et surtout valider la proposition initiale effectuée dans cette thèse. Une approche possible est d'établir des contacts avec les associations SIF[23], EAPLS[24], IEEE[25], ACM[26] ou les groupes de recherche tel le GDR GPL[27] pour constituer un groupe de travail dans ce but.

Un point important est que nous ne parlons que de validation de cette ontologie, c'est-à-dire de confrontations avec les connaissances d'autres intervenants du domaine. En effet, nous avons travaillé de manière usuelle dans le monde de l'ontologie, c'est-à-dire que nous avons formalisé un ensemble de connaissances dont la sémantique est l'ensemble des connaissances qui peuvent en être dérivées. Il s'agit d'une sémantique implicite liée au vocabulaire utilisé pour nommer les classes, les attributs, les relations et qui comporte de nombreuses informations sous la forme de texte en langage naturel. Pour compléter ces activités de validation par relecture, il serait utile de faire des activités de vérification formelle. Actuellement, les seules activités qui peuvent être effectuées sont liés à la déduction de

[23] http://www.societe-informatique-de-france.fr/
[24] http://eapls.org/
[25] http://www.ieeefrance.org/
[26] http://www.acm.org/
[27] http://gdr-gpl.cnrs.fr/

connaissances. Il est possible de faire des tests que l'ontologie permet, ou pas, de déduire certaines connaissances particulières. Il est également possible de vérifier la consistance de l'ontologie, c'est-à-dire qu'elle ne permet de déduire deux informations incohérentes. Dans l'hypothèse du monde ouvert, il n'est pas possible de vérifier la complétude de l'ontologie c'est-à-dire que tous les outils et les informations associées de présents. Dans tous les cas, il n'est pas possible de vérifier qu'une connaissance exprimée est correcte car l'ontologie ne contient pas une sémantique explicite qui permettrait, par exemple, de vérifier la correction des relations entre les classes. Dans la seconde partie de cette thèse, nous allons nous focaliser sur la vérification en nous appuyant sur une définition explicite des activités de vérification. L'objectif consiste à formaliser l'introduction de composants dans un langage de modélisation comportant des activités de vérification et à étudier la compositionnalité de ces activités. Dans ce but, nous ne nous appuierons plus sur OWL comme moyen de formalisation mais sur un plongement de l'IDM (Ingénierie Dirigée par les Modèles) dans l'assistant de preuve COQ.

Deuxième partie

Formalisation sémantique

2 Concepts

Table des matières

2.1 Une introduction à l'Ingénierie Dirigée par les Modèles 36
    2.1.1 Modèles 37
    2.1.2 Les métamodèles 37
    2.1.3 Le métamétamodèle exploité 38
    2.1.4 L'architecture à quatre niveaux 38
2.2 Les approches de composition dans l'IDM 38
2.3 Une introduction à l'assistant à la preuve COQ 39
    2.3.1 Les termes de base 41
    2.3.2 Les définitions 41
    2.3.3 Les théorèmes et les preuves 43
    2.3.4 Développement réalisé en COQ 46
    2.3.5 Types dépendants 47
2.4 La formalisation de l'ingénierie dirigée par les modèles 48
    2.4.1 Spécification algébrique 48
    2.4.2 Théorie des types 48
    2.4.3 Raffinement et théorie des ensembles 50
2.5 Conclusion 50

Ce chapitre présente les concepts fondamentaux essentiels à la compréhension du reste du manuscrit ainsi que quelques travaux connexes à la formalisation de l'IDM. La première section présente certains concepts de l'IDM et la deuxième section introduit l'utilisation de l'assistant à la preuve COQ pour la spécification et la vérification formelle. La troisième section présente quelques travaux sur la spécification formelle des concepts de l'IDM.

2.1 Une introduction à l'Ingénierie Dirigée par les Modèles

La modélisation est une activité importante dans plusieurs domaines scientifiques, elle permet de représenter un phénomène en reproduisant son fonctionnement pour pouvoir analyser et prédire son comportement. Parmi les exemples les plus connus, le modèle de Bohr [Boh13] en physique qui vise à comprendre la constitution des atomes, le modèle moléculaire en double hélice de Watson [WC53] pour la structure de l'ADN en biologie et le modèle de M. Minsky [MH88] pour le domaine des sciences cognitives et l'intelligence artificielle qui présente une modélisation de l'esprit comme une architecture d'agents élémentaires hiérarchisés.

L'Ingénierie Dirigée par les Modèles (IDM) est une initiative de la communauté du génie logiciel développée pour instaurer les règles de la modélisation dans ce domaine. Cette initiative suggère que l'on devrait d'abord développer un modèle du système à l'étude, qui est ensuite transformé en une implémentation réelle (c'est-à-dire une entité logicielle exécutable) [GDD09].

L'IDM a évolué en tant que changement du paradigme de la technologie orientée objet dans laquelle le principe est que tout est objet. Le paradigme de l'ingénierie des modèles est basé sur le principe que tout est un modèle [Béz05]. La technologie à base d'objet repose sur les notions de classe et d'objet, et les principales relations sont l'instanciation (un objet est une instance d'une classe) et l'héritage (une classe hérite d'une autre classe) comme illustré sur la figure 2.1. D'autre types de relations peuvent exister entre les objets comme les liens de composition.

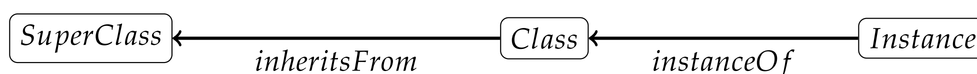


Figure 2.1 — Les notions de base de la technologie objet

L'IDM manipule les modèles composés d'éléments et de liens entre ces éléments, mais aussi les relations entre le modèle du système à l'étude, le métamodèle et les transformations de modèles. D'une façon similaire aux technologies objets, l'IDM peut être caractérisée par deux relations principales, à savoir, la représentation (un modèle représente un élément du monde réel, ou du monde virtuel du logiciel) et la conformité (un modèle est conforme à un métamodèle) comme illustré sur la figure 2.2.

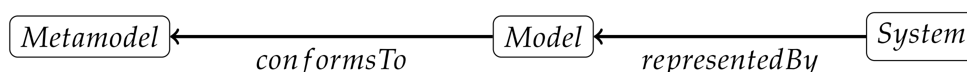


Figure 2.2 — Les notions de base de l'IDM

2.1.1 Modèles

Les modèles jouent un rôle majeur dans l'IDM. La définition la plus générale dit qu'un modèle est une représentation simplifiée de la réalité [Sel03]. En fait, on peut dire qu'un modèle est un ensemble d'éléments formels (bien définis) qui décrivent quelque chose. Il est développé dans un but précis et qui peut être analysé à l'aide de diverses méthodes [MCF03]. Un modèle dans l'ingénierie doit satisfaire quelques caractéristiques [Sel03] :

- **Abstraction :** Un modèle est toujours une simplification du système qu'il représente.
- **La compréhensibilité :** Il ne suffit pas de donner l'abstraction, il faut aussi présenter tout ce qui concerne le modèle et permet de le comprendre (par exemple, en utilisant une notation).
- **Correction** et précision Un modèle doit fournir une représentation fidèle du système modélisé et un niveau de détail adéquat pour l'exploitation souhaitée.
- **Prédictibilité :** Nous devrions être en mesure d'utiliser un modèle pour prédire correctement les propriétés intéressantes mais non triviales du système modélisé, que ce soit par l'expérimentation (comme par la simulation (exécution)) ou à travers un certain type d'analyses formelles.
- **Peu coûteux :** Un modèle doit être nettement moins cher à construire et à analyser que le système modélisé.

Les modèles respectent des contraintes décrites par un métamodèle. Nous présentons dans ce qui suit la notion du métamodèle.

2.1.2 Les métamodèles

Les métamodèles sont les éléments de structuration de modèles. Ils permettent de définir de façon précise les différents formalismes qui permettent d'élaborer des modèles. C'est par leur intermédiaire que la pérennité des modèles est assurée. Conscient de la difficulté inhérente à la définition de formalismes de modélisation, l'OMG a en premier lieu défini le standard `MOF` (Meta Object Facility), qui apporte le support de définition des formalismes de modélisation sous la forme de métamodèles [BS11] qu'il a par la suite enrichi avec le standard OCL [OMG12] qui permet d'exprimer des contraintes sur les modèles comme des prédicats logiques. Un métamodèle définit la structure, les entités ainsi que les propriétés de leurs relations et de leurs règles de cohérence que doit avoir tout modèle conforme à ce métamodèle. Autrement dit, tout modèle doit respecter la structure définie par son métamodèle. Par exemple, le métamodèle UML[1] [PMV03] spécifie que les modèles UML peuvent contenir des paquetages, les paquetages des classes, les classes des attributs et des opérations, etc [BS11].

Un métamodèle respecte des contraintes décrites dans un métamétamodèle.

[1] http://www.uml.org/

2.1.3 Le métamétamodèle exploité

Notons que la relation qui existe entre le standard MOF et les métamodèles est exactement la même que celle qui existe entre un métamodèle et ses modèles instances. MOF définit la structure que doit avoir tout métamodèle, il est naturel de vouloir représenter un métamodèle sous forme de diagramme de classes. Son diagramme de classes est appelé aussi bien métamétamodèle que modèle MOF [BS11].

2.1.4 L'architecture à quatre niveaux

À partir de ces définitions, nous pouvons représenter l'architecture à quatre niveaux MDA (Model Driven Architecture) proposée par le standard OMG. La figure 2.3 illustre cette architecture. Elle comporte : le niveau M0 contenant les entités à modéliser, ici les applications informatiques, le niveau M1 contenant les différents modèles de l'application informatique, le niveau M2 contenant les différents métamodèles qui ont été utilisés, et le niveau M3 contenant le métamétamodèle qui a permis de définir uniformément les métamodèles [BS11]. Pour rendre l'IDM plus populaire dans la communauté du génie logiciel, la théorie sous-

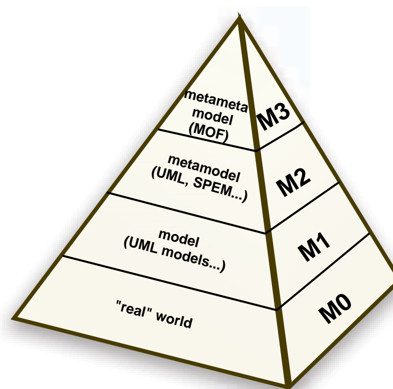


Figure 2.3 — L'architecture à quatre niveaux du MDA

jacente doit être précise et facile à comprendre. Par conséquent, des définitions précises des modèles sont nécessaires et doivent être faites dans un cadre théorique cohérent. Cela implique un développement de la théorie de l'IDM qui doit être suffisamment complète pour répondre à tous les phénomènes liés aux langages et métamodèles utilisés dans l'IDM (par exemple, OWL[2], XML[3] et UML[4]).

2.2 Les approches de composition dans l'IDM

Les modèles sont des abstractions des différentes facettes d'un système qui doivent être composés pour construire le système final comme dans la programmation par aspect (AOP)

[2] http://www.w3.org/TR/owl-features/
[3] http://www.w3.org/XML/
[4] http://www.omg.org/spec/UML/

[KLM+97]. Des outils et des approches sont proposés pour automatiser la composition. Ce problème concerne plusieurs domaines de modélisation et fait intervenir plusieurs techniques. Plusieurs méthodes de composition sont collectées dans [Jea08]. La majorité de ces méthodes sont intéressantes pour l'implémentation de l'opérateur de fusion en utilisant des correspondances entre les modèles comme Rational Software Architect[5] [Let05], le modèle de données de Bernstein et al. [BHP00], Atlas Model Weaver [6] [DDFV09], Epsilon[7] [KRPP10], Theme/UML[8] [Cla02] et EMF Facet[9] [MD10].

Nous sommes intéressée à un ensemble d'opérateurs élémentaires qui peuvent servir pour l'implémentation des opérateurs de haut niveau comme le Package Merge d'UML (chapitre 4) et le style de composition ISC (chapitre 5).

2.3 Une introduction à l'assistant à la preuve COQ

La preuve assistée par ordinateur de théorèmes est devenue très importante dans l'informatique théorique moderne. Plusieurs assistants à la preuve sont populaires et des problèmes de taille réaliste peuvent être traités que ce soit des outils informatiques (comme le compilateur Compcert[10]) ou mathématique (comme le théorème des 4 couleurs [Gon] et le théorème de Feit et Thompson [GAA+13]).

Les avantages des preuves assistées par ordinateurs sont bien connues[11]. Une preuve mathématique sur papier malgré tous les efforts de relecture peut contenir des erreurs. Cela provient du fait que qu'il n'est pas raisonnable de mettre tous les détails dans une preuve papier car ce niveau de dépliage rend la preuve trop longue et son écriture très peu lisible. C'est dans les parties implicites d'une preuve que des erreurs peuvent apparaître.

Une preuve faite avec un assistant à la preuve est une preuve formelle vérifiée. Le principe est de réduire la preuve en étapes élémentaires qu'un ordinateur peut vérifier et qui ne nécessitent pas l'intervention de l'intuition humaine. Les règles que l'assistant à la preuve peut appliquer reposent sur un noyau d'une très petite taille. Ceci donne certainement une preuve plus longue mais qui possède la garantie d'absence d'erreurs. En plus, ce noyau est utilisé pour prouver l'assistant à la preuve lui-même. Bien sûr la démarche d'utilisation d'assistant à la preuve contribue à la certification de la preuve mais ne résout pas la problématique de trouver la preuve.

COQ fait partie des assistants de preuves les plus utilisés. Il est similaire à HOL[12] [GM93], Agda[13] [Coq01], LEGO[14] [LP92], Isabelle[15] [NPW02] et d'autres assistants de preuves interac-

[5] http://www-306.ibm.com/software/awdtools/architect/swarchitect/
[6] http://www.eclipse.org/gmt/amw/
[7] http://www.eclipse.org/gmt/epsilon/
[8] http://www.dsg.cs.tcd.ie/aspects/themeUML
[9] www.eclipse.org/proposals/emf-facet/
[10] http://compcert.inria.fr/
[11] http://images.math.cnrs.fr/Coq-et-caracteres.html#nb6
[12] http://hol.sourceforge.net/
[13] http://wiki.portal.chalmers.se/agda/pmwiki.php
[14] http://www.dcs.ed.ac.uk/home/lego/
[15] http://www.cl.cam.ac.uk/research/hvg/Isabelle/

tifs. Ces systèmes ont un point commun qui est le fait que les fonctions sont des programmes qui peuvent être calculés et non pas seulement des relations comme en mathématiques. À la différence de HOL, COQ est basé sur la théorie des types intuitionniste. La logique intuitionniste est une logique constructive, elle a été introduite par Luitzen Egbertus Jan Brouwer en 1907. Le point de vue intuitionniste s'oppose au logicisme de la logique classique de Bertrand Russel, Georg Canter et au formalisme de David Hilbert. Ce point de vue mathématique rejette deux concepts qui sont : le tiers-exclu et l'existentiel non constructif. Cette logique remplace le "il existe tel objet" par "nous pouvons construire tel objet". Du point de vue intuitionniste, une formule logique ou mathématique est valide si et seulement si une preuve de celle-ci existe [VD86].

COQ est basé sur la théorie des types [ML75] et le calcul des constructions inductives [CH+86]. Une caractéristique intéressante des systèmes de preuves basés sur la théorie des types comme COQ est qu'ils contiennent un langage de programmation purement fonctionnel avec une notion de type qui est suffisamment riche pour exprimer des propriétés sur les programmes sous forme de spécifications formelles. Une deuxième caractéristique est le traitement similaire des types de donnés et des propositions logiques (suivant l'isomorphisme de Curry-Howard qui permet de relier les formules aux types et les preuves aux termes). Donner une preuve d'un théorème est la construction d'un terme dont le type correspond à ce théorème. Le système COQ fournit un ensemble de tactiques pour aider l'utilisateur à construire la preuve d'une manière interactive (partiellement automatisée). La preuve correspond à un terme du CIC (Calculus of Inductive Constructions). La correction de cette preuve correspond au fait que le terme est bien typé. Le noyau de vérification interne de COQ est relativement simple car il ne s'agit « que » d'un mécanisme de typage pour un système de type très expressif.

COQ est complètement écrit en Objective Caml, le système complet comporte de l'ordre de 120 000 lignes de code et environ 500 fichiers. Le site officiel de COQ[16] contient les distributions officielles, le manuel de référence et les librairies et les contributions des utilisateurs. Le langage et l'environnement sont enrichis en permanence : extraction de programmes [Let08], coercions [Saï97], modules [Chr03], types co-inductifs [GC05], langage de tactiques [Del00], etc.

Si le but de cette section est une initiation rapide à COQ, nous recommandons d'expérimenter directement avec les exemples en utilisant l'interface graphique `CoqIde`[17] [Tea12] ou `proof-general`[18] [Asp00] dans `emacs`[19] [Sta81]. Il est aussi possible d'utiliser la commande `coqtop` qui permet de lancer l'interprète COQ dans un environnement textuel ou compiler le fichier COQ avec la commande `coqc` mais il est plus agréable de voir l'avancement des preuves sur une interface graphique. Les deux environnements graphiques proposent des fonctionnalités similaires à travers une fenêtre principale qui contient les commandes COQ exécutées d'une manière séquentielle. Lorsque le mode preuve est lancé, une autre fenêtre apparaît affichant l'avancement de la preuve (le contexte et les sous-buts restant à prouver).

[16] coq.inria.fr
[17] http://coq.inria.fr/cocorico/CoqIde
[18] http://proofgeneral.inf.ed.ac.uk/
[19] http://www.gnu.org/software/emacs/

Dans ce qui suit, nous présentons d'une manière concise le langage GALLINA qui est le langage de spécification de COQ et à l'aide d'exemple le langage de commandes Vernaculaire. Cette présentation permet de comprendre les éléments présentés dans le manuscrit et de se pencher rapidement sur le développement[20] fait en COQ. Nous conseillons de consulter pour plus d'information à titre non exclusif l'introduction à COQ sur le site officiel [21], ce document sur la vérification logicielle [PM12] et aussi le Coq'Art [BC11].

2.3.1 Les termes de base

Le langage utilisé dans COQ est une extension du λ-calcul simplement typé, les constructions de base de cette famille de langages sont : la variable, l'abstraction et l'application. Dans ce langage les types sont aussi des termes qui sont eux-mêmes typés, leurs types sont appelés Sortes. COQ fait la distinction entre deux catégories de types : le type Prop pour les propositions (formules logiques) et le type Set pour les types de données. Ces deux types sont aussi typés et leurs type commun est nommé Type.

Le tableau 2.1 présente un résumé de la syntaxe de COQ pour les propositions logiques et les quantificateurs. La première ligne du tableau présente la notation mathématique et la deuxième présente son écriture en COQ. En COQ, la èche $\rightarrow$ représente l'implication.

$\perp$	$\top$	$t = u$	$t \neq u$	$\neg p$	$p \wedge q$	$p \rightarrow q$	$p \leftrightarrow q$	$\forall x,\ p$	$\exists x,\ p$
False	True	t = u	$t <> u$	˜p	p /\ q	p -> q	p <-> q	forall x, p	exists x, p

Tableau 2.1 — La syntaxe de COQ pour les propositions et les quantificateurs

Nous allons décrire les aspects du langage à l'aide des exemples simples utilisés dans notre code. Nous distinguons dans ce qui suit de deux couleurs différentes deux types de listings. Le premier représente la fenêtre des définitions et des commandes COQ (en bleu/-gris) et la deuxième correspond à la fenêtre des résultats (contexte et buts restant à prouver en rose).

2.3.2 Les dé nitions

Une définition inductive est spécifiée en donnant le nom et le type de la famille des données définies et les noms et les types de ses constructeurs. La forme générale d'une définition inductive est la suivante :

```
Inductive ident binder1 ...binderk : sort :=
  ident1                  : type1
| ...
| identn                  : typen
```

Le nom *ident* est le nom du type défini inductivement et *sort* est son univers de définition. Les noms $ident_1$, ..., $ident_n$ sont les noms des constructeurs et $type_1$, .., $type_n$ sont

[20] http://www.irit.fr/~Mounira.Kezadri/FormalAssembly.html
[21] http://coq.inria.fr/a-short-introduction-to-coq

leurs types respectifs. La définition inductive doit être bien formée pour ne pas introduire d'inconsistances et ceci par la définition d'un ensemble de restrictions syntaxiques [22]. Dans le cas d'une définition bien formée, les constantes $ident_1$, ..., $ident_n$ sont ajoutées à l'environnement avec leurs types respectifs. Selon l'univers de types, COQ produit le schéma d'induction pour *ident*. Ces règles sont nommées : $ident_{ind}$, $ident_{rec}$ et $ident_{rect}$. Si les paramètres $binder_1$ ...$binder_k$ (optionnels) sont présents, le type est dit type inductif paramétrique (des types polymorphes comme les types de données algébriques dans ML ou Haskell). Dans le cas d'un type `paramétrique`, les conclusions de chaque constructeur $term_i$ invoquent le type inductif avec les mêmes valeurs de paramètres que sa spécification.

On peut présenter la définition d'un type inductif à l'aide d'un système d'inférence et c'est ce qui sera fait dans la section 3.2.3.

Le type `uniset` que nous utilisons dans le chapitre 3 pour la représentation des ensembles et par la suite dans la représentation des graphes est défini dans la bibliothèque `Uniset`[23] comme un type inductif paramétrique à l'aide du constructeur `Charac`. Ce constructeur a comme paramètre une fonction associant à chaque élément de type `A` un booléen qui permet de savoir si l'élément existe ou pas dans l'ensemble.

```
Inductive uniset (A : Set) : Set :=
 Charac : (A → bool) → uniset A
```

Dans ce cas, le type `uniset` et le constructeur `Charac` sont ajoutés à l'environnement ainsi que les trois règles `uniset_rect`, `uniset_ind` et `uniset_rec`.

```
uniset is defined
uniset_rect is defined
uniset_ind is defined
uniset_rec is defined
```

Un objet dans l'environnement COQ a un nom et un type. La commande `Check` prend en paramètre un terme et permet de tester son bon typage et d'afficher son type. Nous invoquons `Check` pour afficher la définition du *uniset_ind*.

```
Check uniset_ind.
```

Le principe d'induction pour `uniset` (`uniset_ind`) est le suivant :

```
uniset_ind
     : forall (A : Set) (P : uniset A → Prop),
       (forall b : A → bool, P (Uniset.Charac b)) ->
       forall u : uniset A, P u
```

D'un point de vue mathématique ceci assure que les objets satisfaisants le prédicat `uniset` sont exactement les objets satisfaisants un des constructeurs (dans notre cas le prédicat `Charac`). Par exemple l'ensemble singleton 0 peut être représenté comme suit[24] :

```
Definition singleton0: uniset nat := Charac (fun i => beq_nat i 0).
```

Pour faire une preuve pour le type `uniset`, il suffit de faire la preuve pour le constructeur `Charac`.

[22] http://coq.inria.fr/doc/Reference-Manual006.html#Positivity
[23] http://coq.inria.fr/distrib/8.4/stdlib/Coq.Sets.Uniset.html
[24] beq_nat est défini dans la bibliothèse EqNat de COQ

Le type `bool` est exprimé dans la bibliothèque COQ comme un type inductif simple avec deux constructeurs `true` et `false` comme suit :

```
Inductive bool : Set := true : bool | false : bool
```

Nous expliquons dans ce qui suit l'implémentation et la preuve d'une propriété de la fonction d'intersection des `unisets` (complètement accessibles dans l'annexe A). Un élément appartient à l'intersection de deux ensembles s'il appartient aux deux ensembles. Ceci est exprimé en COQ comme illustré sur le listing suivant à l'aide de la fonction `inter` définie en utilisant le mot clé `Definition`. La fonction `andb` est définie en COQ comme l'opérateur de conjonction des booléens[25].

```
Variable typ : Set.

Definition inter (s s' : uniset typ) :=
  Charac (fun v => andb (charac s v) (charac s' v)).
```

La fonction `charac` qui décide si un élément a appartient à une fonction caractéristique s utilise pour sa définition la fonction définie par le constructeur de s. Elle est définie ainsi :

```
Definition charac := fun (A : Set) (s : uniset A) (a : A) => let (f) := s in f a.
```

Un autre aspect pour la simplification et l'amélioration de la lisibilité des termes et des théorèmes dans le code est obtenu à travers l'utilisation des notations. Une notation est une abréviation permettant d'exprimer un terme ou une expression. Ceci modifie la façon dont COQ analyse et affiche les termes. Nous pouvons utiliser par exemple le symbole $\cap$ pour remplacer la fonction d'intersection :

```
Notation "a ∩ b" := (@inter a b) (at level 99).
```

L'utilisation de plusieurs notations symboliques peut causer des ambiguïtés. Pour éviter ce problème COQ utilise des niveaux de priorité de 0 à 100 (et aussi un niveau supplémentaire 200) en plus des règles d'associativité. Nous avons choisi le niveau 99 dans l'exemple pour l'intersection.

2.3.3 Les théorèmes et les preuves

Pour déclarer une proposition comme étant vraie, nous devons produire une preuve. Une preuve se fait par l'application d'un ensemble de tactiques. Une tactique appliquée à un certain but le transforme en un ensemble de sous-buts tel que la preuve de ces sous-buts suffit pour prouver le but.

Un théorème intéressant et simple à démontrer est `inter_intro` qui indique que si un élément appartient à un ensemble s et à un ensemble s', alors ce même élément appartient à l'intersection des deux ensembles. Pour exprimer ce théorème, nous avons besoin d'une définition pour la fonction d'appartenance à un ensemble. La fonction `In` est définie dans COQ comme suit :

```
Definition In (s:uniset) (a:A) : Prop := charac s a = true.
```

[25] http://coq.inria.fr/V8.4/stdlib/Coq.Init.Datatypes.html#andb

Nous déclarons une notation pour la fonction `In` avec le symbole ∈ :

```
Notation "x ∈ vs " := (In vs x) (at level 100).
```

Ceci est décrit à l'aide du théorème `inter_intro`. La preuve est la séquence de tactiques décrites entre `Proof` et `Qed` (nous pouvons remplacer `Qed` par `Defined`, la différence est que dans le cas de `Defined`, la définition de la fonction peut être explicitée et utilisée avec les tactiques de conversion).

```
Lemma inter_intro :
forall v (s s' : uniset typ), v ∈ s -> v ∈ s' -> v ∈ (s ∩ s').
Proof.
  intros v s s' Hs Hs'.
  unfold inter.
  unfold In in *; simpl in *.
  rewrite Hs; rewrite Hs'; simpl in *.
  reflexivity.
Qed.
```

Le théorème en notation règle d'inférence est le suivant :

$$\frac{(v \ \in \ s) \qquad (v \ \in \ s')}{(v \ \in \ (s \cap s'))} \ inter_intro$$

Nous essayons dans ce qui suit de détailler cette preuve pour donner une idée de la démarche de preuve en utilisant COQ. Nous commençons par la définition du théorème, dans notre cas nous utilisons le mot clé `Lemma`, nous pouvons utiliser d'une manière similaire les mots clé `Theorem`, `Fact` ou `Remark`.

```
Lemma inter_intro :
forall v (s s' : uniset typ), (v ∈ s) -> (v ∈ s') -> (v ∈ (s ∩ s')).
```

Une fois cette dernière définition évaluée, le mode preuve est automatiquement déclenché. La commande `Proof` (éventuelle et introduite dans un but de faciliter la lecture des scripts de preuves) annonce le début du mode preuve. Le nombre de sous-buts restant à prouver, le contexte (les hypothèses) ainsi que les sous-buts à prouver sont affichés. Les hypothèses sont nommées et représentées sur la partie supérieure séparée par une ligne de double traits du sous-but courant. Ici *typ* et *eq_dec* sont les hypothèses définies dans la même section.

```
1 subgoal
 typ : Set
 eq_dec : decidable typ
 ============================
 forall (v: typ) (s s': uniset typ), (v ∈ s) → (v ∈ s') → (v ∈ s ∩ s')
```

Des nouvelles commandes sont disponibles dans ce mode, comme par exemple les tactiques qui sont des combinaisons de preuves primitives. Une tactique agit sur le contexte et le sous-but courant pour essayer de le résoudre. La tactique `intro` correspond à la règle de l'introduction du *forall* et de l'implication. La tactique peut prendre en paramètre un nom qui sera associé à la nouvelle hypothèse. Nous utilisons une version spéciale de cette tactique appelée `intros` qui prend en paramètres plusieurs noms d'hypothèses à introduire, elle est équivalente à plusieurs applications de la tactique `intro`. Dans le cas d'absence de paramètres `intros` introduit toutes les hypothèses possibles avec des noms choisis par

un algorithme de nommage interne au système jusqu'à obtention d'un sous-but atomique (échec de la tactique `intro`).

```
intros v s s' Hs Hs'.
```

La tactique `intros` appliquée au sous-but précédent remonte v, s et s' et les deux hypothèses Hs et Hs' et permet de transformer le sous-but ainsi :

```
1 subgoal
  typ : Set
  eq_dec : decidable typ
  v : typ
  s : uniset typ
  s' : uniset typ
  Hs : v ∈ s
  Hs' : v ∈ s'
  ============================
   v ∈ s ∩ s'
```

Nous avons besoin à ce niveau de déplier les définitions de l'inclusion et l'intersection. Nous commençons à titre d'exemple par la fonction d'intersection. Nous appliquons la tactique `unfold` qui permet de déplier la définition de son paramètre qui doit être une constante transparente. Une constante est transparente si nous avons accès à son type et son algorithme de calcul (tous les détails sont importants), dans ce cas, elle est complètement définie avec `Definition` ou sa preuve est terminée par `Defined`. La tactique `unfold` permet de remplacer chaque occurrence de son paramètre dans le but par sa forme normale.

```
  unfold inter.
```

L'application de la tactiques `unfold` avec comme paramètre la fonction $\cap$ transforme le sous-but ainsi :

```
1 subgoal
  typ : Set
  eq_dec : decidable typ
  v : typ
  s : uniset typ
  s' : uniset typ
  Hs : v ∈ s
  Hs' : v ∈ s'
  ============================
   v ∈ Charac (fun v0 : typ => charac s v0 && charac s' v0)
```

La syntaxe du langage de commandes de COQ permet la composition de tactiques. Par exemple, si tac_1 et tac_2 sont deux tactiques, la séquence tac_1 ; tac_2 applique la tactique tac_1 au but courant et tac_2 à chacun de ses sous-buts. Il est possible aussi d'enchainer plusieurs tactiques en suivant le même principe. Ici nous invoquons encore une fois la tactique `unfold` pour déplier la définition de $\in$, par la suite la tactique `simpl` est invoquée pour effectuer des simplifications sur toutes les hypothèses et les sous-buts résultants.

```
unfold In in *; simpl in *.
```

Le contexte et le sous-but courant sont transformés ainsi :

```
1 subgoal
  typ : Set
  eq_dec : decidable typ
  v : typ
  s : uniset typ
  s' : uniset typ
  Hs : charac s v = true
  Hs' : charac s' v = true
  ============================
   charac s v && charac s' v = true
```

Nous utilisons la tactique `rewrite` pour replacer `charac s v` et `charac s  v` par leur correspondant dans les hypothèses `Hs` et `Hs` .

```
  rewrite Hs; rewrite Hs'.
```

L'application de la séquence de tactiques transforme les hypothèses ainsi que le but courant comme suit :

```
1 subgoal
  typ : Set
  eq_dec : decidable typ
  v : typ
  CharacS : typ → bool
  CharacS' : typ → bool
  Hs : CharacS v = true
  Hs' : CharacS' v = true
  ============================
   true && true = true
```

La tactique `reflexivity` essaye d'appliquer la règle de ré exivité sur le sous-but courant.

```
reflexivity.
```

Cette dernière tactique permet de réussir la preuve et l'espéré `Proof completed` est affiché.

```
Proof completed.
```

Cette preuve n'est pas optimisée, par exemple l'utilisation de la tactique `intuition` permet de conclure la preuve juste après le dépliage des fonctions $\cup$ et $\in$.

Les définitions COQ peuvent être enregistrées dans des fichiers avec l'extension .v, ces fichiers peuvent être compilés avec la commande `coqc` et par la suite importés pour pouvoir utiliser les définitions qu'ils contiennent par la commande `Require Import` (aussi utilisée pour importer les bibliothèques standards de COQ). Nous pouvons aussi extraire du code fonctionnel vérifié dans le langage ML [MTHM97], Haskell[26] ou Lisp [McC60].

2.3.4 Développement réalisé en COQ

La taille de nos développement en COQ présentés dans cette thèse dépasse les 15000 lignes de code. D'après notre expérience d'utilisation de cet assistant à la preuve sans connaissance

[26] http://www.haskell.org/haskellwiki/Haskell

profonde de ses mécanismes internes, nous pouvons aborder quelques points dans ce qui suit :

- **Les axiomes de COQ :** Il y a plusieurs axiomes non constructifs typiques utilisés dans COQ indépendamment du CIC et qui sont ajoutés à la librairie standard de COQ sans inconvénients. Ces axiomes font partie de la librairie Logic. Ces axiomes ont été utilisés généralement dans nos preuves de propriétés, sous une forme qui n'a pas d'impact sur la construction des fonctions dans notre développements. Les plus utilisés sont :
  - Excluded-middle : $\forall A : Prop, A \vee \neg A$
  - Proof-irrelevance : $\forall A : Prop,\ \forall p_1\ p_2 : A, p_1 = p_2$
  - Extensionality of functions : $\forall f\ g : A \rightarrow B, (\forall x, f(x) = g(x)) \rightarrow f = g$
- **Comparaison des types :** Concernant la comparaison des types, COQ ne réussit pas à déduire l'égalité entre types dans tous les cas. La solution est de forcer l'affichage complet de types (*Set Printing All*) et par la suite d'aider COQ par des conversions explicites de type à l'aide de la tactique $unfold$ pour introduire la définition des types.
- **La modularité de la spéci cation :** Notre code COQ utilise des modules pour définir les types qui représentent les nœuds et les relations des modèles. Ces modules sont paramétrés par les types des nœuds et des relations et définissent plusieurs opérations génériques sur ces types. Les modèles sont définis génériquement comme des types dépendant de leurs ensembles de nœuds et arcs, donc nous pouvons décrire différents types de modèles.
- **Dé nition des tactiques :** Nous avons utilisé dans notre développement un aspect important de l'assistant à la preuve COQ qui est la possibilité de définir des schémas de preuves (nous abordons ce point dans la section 5.3.1.1).

2.3.5 Types dépendants

Nos développements COQ s'appuient largement sur l'utilisation de types dépendants dont les détails ne sont pas visibles dans le manuscrit. Un type dépendant contient des propriétés dans sa définition. Voici un exemple simple du type dépendant *ilist* (liste indexée par sa longueur) tiré de [Chl11].

```
Section ilist.
  Variable A : Set.
  Inductive ilist : nat → Set :=
  | Nil : ilist O
  | Cons : ∀ n, A → ilist n → ilist (S n).
```

Le type *ilist* comporte lui même l'information sur le nombre d'éléments de la liste. L'argument *nat* de *ilist* montre la longueur de la liste. Les constructeurs montrent qu'une liste *Nil* a comme longueur 0 et qu'une liste *Cons* a comme longueur le successeur de la longueur de sa queue.

Nous nous basons sur COQ4MDE [TCCG07] pour les travaux que nous présentons dans les chapitres suivants pour la vérification de composition de modèles. COQ4MDE est un

« framework » de l'IDM qui s'appuie sur l'assistant à la preuve COQ. Nous présentons dans la section suivante d'autres travaux de formalisation des concepts de l'IDM.

2.4 La formalisation de l'ingénierie dirigée par les modèles

L'ingénierie dirigée par les modèles a été l'objet de nombreux travaux et de plusieurs tentatives de formalisation.

2.4.1 Spéci cation algébrique

Nous faisons le point dans ce qui suit sur deux travaux s'appuyant sur le système algébrique `Maude`[27] [CDE+02]. Le premier est MoMENT (MOdel manageMENT) [BM10] qui est un atelier pour la gestion de modèles basé sur des expériences sur les transformations formelles de modèles et la migration de données. Il fournit un ensemble d'opérateurs génériques pour la manipulation de modèles. MoMENT utilise le langage `Maude`[28] [CDE+02] comme un formalisme algébrique. Dans cet atelier, les modèles sont représentés comme des spécifications algébriques et les opérateurs sont définis indépendamment du métamodèle. Pour être utilisé, un opérateur doit être spécifié dans un module appelé `signature` qui décrit les contraintes du métamodèle. L'approche est implémentée dans un outil[29] qui permet aussi la transformation automatique à partir des métamodèles EMF.

Le second est un travail de Vallecillo et al. [RRRDV07] qui ont réalisé un codage différent des notions de métamodèle et modèle ainsi que des transformations de modèles comme un plug-in Eclipse [TV10] en utilisant le système `Maude`.

Dans les deux travaux cités, la modélisation est superficielle (shallow embedding), [BGG+92] c'est-à-dire qu'elle est reliée à la structure proposée dans `Maude` et utilise directement les éléments similaires, les objets par exemple, pour modéliser les éléments de modèles. Les transformations de modèles sont implémentées en utilisant le mécanisme de réécriture des objets proposés par `Maude`.

2.4.2 Théorie des types

Poernomo dans [Poe06] a proposé un codage des métamodèles en utilisant la théorie des types dans le but de permettre le développement correct par construction de transformations de modèles. Son codage repose sur l'utilisation d'enregistrements éventuellement récursifs. Celui-ci est présenté de manière plus détaillée dans [Poe08] et illustré en avec des métamodèles simples sans cycle. Ces travaux ont ensuite été implantés par Terrel en utilisant l'assistant à la preuve COQ [PT10]. La structure récursive des enregistrements doit alors être représentée par des types inductifs et co-inductifs. Les exemples considérés dans ces travaux restent relativement simple et ne se heurtent pas aux contraintes structurelles de garde imposées par COQ pour assurer la correction sémantique des définitions mixtes inductives/co-

[27] http://maude.cs.uiuc.edu/
[28] http://maude.cs.uiuc.edu/
[29] http://moment.dsic.upv.es/

inductives (voir [PM11]). Fernández et Terrell dans [FT13] reprennent les travaux précédents en ajoutant la spécification de transformations de modèles composées hiérarchiquement et les preuves de ces transformations à partir de l'assemblage des constituants. Les auteurs n'indiquent pas explicitement comment coder des graphes ce qui est nécessaire en IDM pour représenter les métamodèles. Il s'agit dans leurs travaux de reprendre le codage précédent sans indication particulière concernant les contraintes de garde. Calegari et al dans [CLST11] ont également proposé un codage en COQ pour modéliser des transformations ATL mais dans ce travail ils ne peuvent pas représenter n'importe quel métamodèle à cause des liens cycliques, ils sont obligés de couper les cycles. Une structure générale de modèles en graphes peut être implémentée en utilisant des types co-inductifs. Mais, comme montré dans [PM11] par Picard et Matthes, le codage est complexe car COQ impose des contraintes structurelles de garde pour la combinaison des types inductifs et co-inductifs qui interdisent le codage naturel proposé par Poernomo et al dans [Poe06, PT10]. M. Giorgino et al. dans [GSMP11] utilisent l'arbre couvrant associé au graphe combiné avec des liens supplémentaires pour surmonter cette contrainte en utilisant l'assistant à la preuve ISABELLE[30] [NPW02]. Cela permet de développer une transformation de modèles en s'appuyant sur des preuves inductives adaptées aux liens de partage, puis extraire les implémentations. Ces plongements sont tous superficiels : ils s'appuient sur une structure de données similaire sophistiquée pour représenter les éléments du modèle et méta-modèles.

Une autre formalisation en COQ des concepts de l'IDM par Franck Barbier et al est accessible[31] [BCCLG13], cette représentation s'attache à la preuve des propriétés présentées dans [Küh06] (sur les relations d'instantion et sur des transformations de modèles). Cette dernière formalisation diffère de la notre par sa représentation détaillée des différentes composantes des modèles et métamodèles en se basant sur les concepts du MOF. La formalisation COQ4MDE a l'avantage d'être plus générique et minimale par l'usage des modules pour la représentation de ces concepts et son support de tous types de propriétés par la description de la conformité par un prédicat dans le métamodèle.

Le système HOL/OCL[32] [BW06] est un environnement de développement interactif pour UML et OCL développé dans ISABELLE/HOL. L'atelier peut être utilisé pour prouver par exemple les invariants de classes.

Notre travail est un plongement profond (deep embedding), chaque concept dans les modèles ou les métamodèles est codé en utilisant des constructions élémentaires au lieu de s'appuyer sur des éléments similaires dans COQ. La contribution de cette thèse n'est pas de mettre en œuvre des transformations de modèles en utilisant des outils vérifiant la correction par construction, mais de donner une sémantique dénotationnelle pour modéliser les concepts de l'Ingénierie Dirigée par les Modèles pour fournir une meilleure compréhension et permettre la validation formelle des différentes technologies de mise en œuvre.

[30] http://www.cl.cam.ac.uk/research/hvg/Isabelle/
[31] http://web.univ-pau.fr/~barbier/Coq/
[32] http://www.brucker.ch/projects/hol-ocl/

2.4.3 Raffinement et théorie des ensembles

UML2Alloy[33] [Ana12] a comme but de formaliser UML à l'aide du langage Alloy[34] [Jac12]. Le langage Alloy a été aussi utilisé dans [ZD06] pour élaborer une formalisation de ces concepts dans le but de raisonner sur l'opérateur de fusion, mais l'analyseur comme il est expliqué dans la section 4.4 a des limites lorsqu'il s'agit d'un grand nombre d'instances du modèle.

UML−B[35] [SB06] est un atelier qui vise à combiner les atouts du langage UML et ceux du langage B. En effet UML offre une représentation graphique pratique mais ne dispose pas d'une expressivité sémantique précise formelle et le langage B[36] [ALN+91] qui offre la précision, l'animation et la vérification rigoureuse de modèles mais a une notation relativement difficile à visualiser et à communiquer. Selon [OJ04], cette approche présente l'inconvénient de fournir une spécification UML difficile à lire pour les non connaisseurs du langage B à cause des stéréotypes B utilisés pour spécialiser le sens des entités UML et l'interprétation des dépendances entre les paquetages. Elle pose alors le problème de la sémantique objet de ces stéréotypes.

Les deux ateliers (UML−B et HOL/OCL) permettent de raisonner sur des modèles UML avec une sémantique formelle. L'objectif de nos travaux, en plus de raisonner sur la vérification de modèles, est de spécifier des opérateurs de composition et de vérifier leur correction par rapport à certaines propriétés. Il est possible de coupler notre atelier avec HOL/OCL en définissant une sémantique commune : HOL/OCL permet alors de raisonner sur des invariants de modèle, nos travaux sur la composition de modèles. Il serait également possible de respécifier COQ4MDE dans ISABELLE/HOL ou B par exemple, le codage que nous avons choisi ne dépend pas des spécificités de COQ.

2.5 Conclusion

Nous avons introduit dans ce chapitre des concepts de l'Ingénierie Dirigée par les Modèles et des concepts liés à l'utilisation de l'assistant à la preuve COQ. Nous avons fait brièvement le point sur des travaux de formalisation de l'IDM et quelques approches de composition. Nous introduisons dans le chapitre suivant un cadre formel d'IDM en COQ, ce « framework » combine les atouts de l'IDM et la sûreté des méthodes formelles. Nous avons choisi l'assistant à la preuve COQ car nous partions de travaux pré-existants en COQ pour ce « framework ». Nous aurions pu utiliser ISABELLE, PVS[37] [ORS92] ou Focalize[38] [HPWD09]. Ce même « framework » est utilisé dans les chapitres 3, 4 et 5 pour exprimer et vérifier des opérateurs de composition.

33 http://www.cs.bham.ac.uk/~bxb/UML2Alloy/index.php
34 http://alloy.mit.edu/alloy/
35 http://users.ecs.soton.ac.uk/cfs/umlb.html
36 http://www.methode-b.com/
37 http://pvs.csl.sri.com/
38 http://focalize.inria.fr/

3 Opérateurs élémentaires et vérification de propriétés

Table des matières

3.1 Coq4MDE . . . 53
3.2 Formalisation de la structure du modèle . . . 55
3.2.1 La structure des modèles . . . 56
3.2.2 La structure des ensembles . . . 57
3.2.3 La structure des graphes . . . 58
3.2.3.1 Des arcs corrects . . . 59
3.2.3.2 Égalité des graphes . . . 59
3.3 Formalisation de la composition de modèles . . . 59
3.3.1 L'opérateur élémentaire d'Union . . . 60
3.3.2 L'opérateur élémentaire de Substitution . . . 61
3.4 Formalisation et vérification de propriétés . . . 63
3.4.1 InstanceOf . . . 64
3.4.2 Conformité dans le standard MOF . . . 64
3.4.3 subClass . . . 65
3.4.4 isAbstract . . . 66
3.4.5 lower & upper . . . 67
3.4.6 isOpposite . . . 69
3.4.7 areComposite . . . 70
3.5 La vérification compositionnelle . . . 71
3.6 Conclusion . . . 72

L'Ingénierie Dirigée par les Modèles (IDM) joue un rôle très important dans le développe-

ment des systèmes critiques car elle permet la validation et la vérification de modèles du système dès les premières étapes de développement. Pour assurer la correction de ces mécanismes de V&V, il est nécessaire de spécifier formellement les notions de l'IDM et les mécanismes de V&V. Dans ce but, un cadre formel a été proposé dans [TCCG07]. L'approche consiste à séparer le niveau des données du niveau des types en définissant des structures différentes pour caractériser les deux natures de modèle. Un modèle (`Model`) correspond au niveau des données et une famille de modèles (`MetaModel`) correspond au niveau des types et au langage de modélisation à partir duquel nous pouvons créer une famille de modèles (Figure 3.1). La famille des modèles définit aussi bien la structure que la sémantique du modèle qu'elle permet de décrire. Un modèle est défini comme un multigraphe dont les nœuds représentent les objets du modèle et les arcs représentent les liens entre ces objets. Les attributs d'un objet sont également représentés comme des nœuds du type de l'attribut lié à l'objet par un lien du nom de l'attribut. Un multigraphe est un graphe qui permet d'avoir plusieurs arcs entre deux nœuds. Une famille de modèles est également définie comme un multigraphe dont les nœuds représentent les classes et les arcs représentent les références. Elle est complétée par des propriétés sémantiques comme les multiplicités. La conformité est définie comme la vérification, étant donné un modèle `M` et une famille de modèles `MM`, que le modèle `M` respecte les contraintes contenues dans `MM`.

Le but de ce « framework » appelé COQ4MDE est de fournir un cadre bien fondé mathématiquement pour l'étude, la vérification et la validation des technologies de l'IDM. Le choix d'une logique constructive et de la théorie des types comme langage de spécification formelle permet d'extraire des prototypes et des outils de spécification exécutables qui peuvent être utilisés pour valider la spécification elle-même par rapport aux implémentations existantes des outils de l'IDM (par exemple : le projet Eclipse Modeling Framework[1]). Nous reprenons pour notre formalisation les notions de `Model`, `MetaModel` ainsi que la relation de conformité entre un modèle et un métamodèle. Une première étude sur la formalisation des modèles par rapport à leurs différentes fonctions dans un environnement global comme l'IDM était proposée dans [JB06], les problèmes de cette formalisation sont discutés dans [TCCG07].

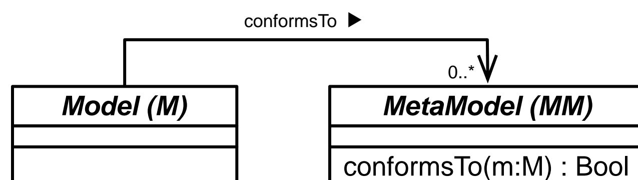


Figure 3.1 — Les définitions de `Model` et `MetaModel` par un diagramme de classe

Remarque 3.1. *Dans ce manuscrit, nous avons choisi d exprimer nos dé nitions, théorèmes et preuves en notation mathématique (usuelle) plus accessible que le langage GALLINA de* COQ*. Les dé nitions en GALLINA sont accessibles sur internet en version hypertexte générée par Coqdoc à l adresse :* `http://www.irit.fr/~Mounira.Kezadri/FormalAssembly.html`*. Les dé nitions* COQ *correspondantes sont parfois plus complexes d une part car la logique constructive exploitée l impose et d autre part pour pouvoir générer des fonctions calculatoires.*

[1] `http://www.eclipse.org/modeling/emf`

Nous commençons par définir plus précisément les notions générales du « framework » `coq4MDE`, les notions de `MetaModel` et `Model` correspondent aux notions de `Formalism` et `Model` de la VVO, nous décrivons ces notions explicitement avec plus de détails dans la section 3.1. Nous décrivons par la suite quelques détails de l'implémentation dans 3.2. Nous présentons dans 3.3 l'extension du « framework » par des opérateurs de composition élémentaires. La formalisation et les vérifications des propriétés EMOF sont présentées dans la section 3.4. Des travaux sur la vérification compositionnelle sont présentés dans la section 3.5. La dernière section 3.6 présente les conclusions et les perspectives.

3.1 Coq4MDE

Dans le « framework » COQ4MDE, le concept de `MetaModel` n'est pas une spécialisation de `Model`. Ils sont formellement définis de la manière suivante. Considérons deux ensembles : `Classes`, respectivement `References`, représente l'ensemble de toutes les étiquettes (noms/identificateurs) de classes, respectivement toutes les étiquettes (noms/identificateurs) des références possibles. Nous considérons aussi les instances de telles classes, l'ensemble `Objects` des étiquettes d'objets. L'ensemble des références contient une étiquette d'une relation spécifique *inh* utilisée pour spécifier la relation d'héritage.

Remarque 3.2. *Dans ce qui suit, on ne parle pas d étiquettes mais de classes, références et objets.*

Dé nition 1 (Model). *Soit* $\mathscr{C} \subseteq \texttt{Classes}$ *un ensemble de classes.*
Soit $\mathscr{R} \subseteq \{\langle c_1, r, c_2\rangle \mid c_1, c_2 \in \mathscr{C}, r \in \texttt{References}\}$ *un ensemble de références entre les classes. Un modèle sur* $\mathscr{C}$ *et* $\mathscr{R}$*, écrit* $\langle MV, ME\rangle \in \texttt{Model}(\mathscr{C}, \mathscr{R})$ *est un multigraphe dé ni sur un ensemble ni MV de nœuds d objets typés et un ensemble ME d arcs de références tel que :*

$$MV \subseteq \{\langle o, c\rangle \mid o \in \texttt{Objects}, c \in \mathscr{C}\}$$
$$ME \subseteq \left\{ \langle\langle o_1, c_1\rangle, r, \langle o_2, c_2\rangle\rangle \;\middle|\; \langle o_1, c_1\rangle, \langle o_2, c_2\rangle \in MV, \langle c_1, r, c_2\rangle \in \mathscr{R} \right\}$$

Notons que, dans le cas d'héritage, le même objet peut être utilisé plusieurs fois dans le même graphe de modèle, associé à différentes classes pour construire des nœuds différents. La réutilisation des objets est liée au polymorphisme d'héritage qui est un aspect important dans la majorité des langages à objets. L'héritage est représenté avec une relation spéciale notée *inh* (usuellement définie dans les langages de métamodélisation tel que `MOF` [OMG11a]).

Remarque 3.3. *inh ne doit pas être utilisée dans les modèles et métamodèles comme une référence classique et ne permet pas par hypothèse les cycles.*

Exemple 3.1. *La gure 3.2 montre un exemple de modèle dans la notation « diagramme à objets ». Cet exemple illustre qu un même objet peut avoir plusieurs types. Nous appelons ces objets des duplicatas. Dans l exemple, les ensembles de nœuds et d arcs pour le modèle peuvent être dé nis pour* $\mathscr{C} = \{Person, Employee, Job, Departement\}$ *et* $\mathscr{R} = \{(Employee, inh, Person), (Employee, worksAs, Job), (Employee, worksAt, Departement)\}$ *comme suit :*

$$MV = \{\langle Mounira, Person\rangle, \langle Mounira, Employee\rangle, \langle Researcher, Job\rangle, \langle IRIT, Departement\rangle\}$$

$$ME = \{(\langle Mounira, Employee\rangle, inh, \langle Mounira, Person\rangle),$$
$$(\langle Mounira, Employee\rangle, worksAs, \langle Researcher, Job\rangle),$$
$$(\langle Mounira, Employee\rangle, worksAt, \langle IRIT, Departement\rangle)\}$$

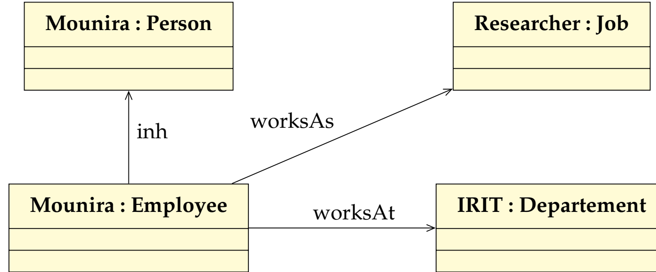


Figure 3.2 — Un exemple de modèle en notation diagramme d'objets

Le prédicat suivant exprime qu'un objet o de type c_1 a un duplicata de type c_2, avec c_2 sous-type de c_1.

$$hasSub(o \in \texttt{Objects}, c_1, c_2 \in \texttt{Classes}, \langle MV, ME\rangle) \triangleq$$
$$c_1 = c_2 \vee \exists c_3 \in \texttt{Classes}, \langle\langle o, c_2\rangle, inh, \langle o, c_3\rangle\rangle \in ME$$
$$\wedge\ hasSub(o, c_1, c_3, \langle MV, ME\rangle)$$

La notion d'héritage standard est exprimée à l'aide du prédicat `standardInheritance`. Le premier prédicat exprime que la relation d'héritage se transmet à travers les objets duplicatas. Le deuxième exprime que chaque ensemble d'objets dupliqués a un objet d'origine associé à la classe qui a permis de le créer. Les autres classes sont les ancêtres de la classe de création jusqu'aux racines du graphe.

$$standardInheritance(\langle MV, ME\rangle) \triangleq$$
$$\forall\langle\langle o, c_1\rangle, inh, \langle o, c_2\rangle\rangle \in ME$$
$$\wedge\forall\langle o, c_1\rangle, \langle o, c_2\rangle \in MV, \exists c \in \texttt{Classes},$$
$$hasSub(o, c, c_1, \langle MV, ME\rangle)$$
$$\wedge\ hasSub(o, c, c_2, \langle MV, ME\rangle)$$

Enfin, le prédicat `subClass`[2] exprime que c_2 est une sous-classe directe de c_1.

$$subClass(c_1, c_2 \in \texttt{Classes}, \langle MV, ME\rangle) \triangleq$$
$$\forall o \in \texttt{Objects}, \langle o, c_2\rangle \in MV \Rightarrow \langle\langle o, c_2\rangle, inh, \langle o, c_1\rangle\rangle \in ME$$

Les classes abstraites sont spécifiées dans le métamodèle en utilisant le prédicat `isAbstract`. Les classes abstraites ne peuvent pas être instanciées, elles servent comme patrons pour construire des classes concrètes qui sont généralement plus spécialisées et qui peuvent être utilisées pour construire des instances. Les classes abstraites sont utilisées pour la représentation des concepts abstraits partagés entre plusieurs classes filles. Donc, généralement dans un graphe d'héritage, les feuilles sont concrètes et les racines sont abstraites.

$$isAsbstract(c_1 \in \texttt{Classes}, \langle MV, ME\rangle) \triangleq$$
$$\forall o \in \texttt{Objects}, \langle o, c_1\rangle \in MV \Rightarrow \exists c_2 \in \texttt{Classes}, \langle\langle o, c_2\rangle, inh, \langle o, c_1\rangle\rangle \in ME$$

[2] $\langle o, c_1\rangle \in MV$ est assuré par $\langle\langle o, c_2\rangle, inh, \langle o, c_1\rangle\rangle \in ME$ car il s'agit d'une structure de graphe.

Dé nition 2 (MetaModel)**.** *Un métamodèle est un multigraphe représentant les classes et les références ainsi que les relations sémantiques qui doivent être satisfaites entre les instances de classes et de références. Il est représenté comme une paire composée d un multigraphe* (MMV, MME) *construit sur un ensemble ni de nœuds* MMV *correspondant à des classes, un ensemble ni d arcs* MME *étiquetés par des références et d un prédicat sur les modèles représentant les propriétés sémantiques.*

Un métamodèle est une paire $\langle (MMV, MME), conformsTo \rangle \in \texttt{MetaModel}$ *tel que :*

$$\begin{aligned} & MMV \subseteq \texttt{Classes} \\ & MME \subseteq \{\langle c_1, r, c_2 \rangle \mid c_1, c_2 \in MMV, r \in \texttt{References}\} \\ & conformsTo : \texttt{Model}(MMV, MME) \rightarrow Bool \end{aligned}$$

Étant donné un modèle M et un métamodèle MM, nous pouvons vérifier la conformité de M par rapport à MM. Le prédicat $conformsTo$ défini dans MM permet d'atteindre ce but. Il définit l'ensemble des modèles conformes par rapport au métamodèle.

Dé nition 3 (Conformance)**.** *La conformité véri e sur le modèle M que :*

1. *chaque objet o dans M est l instance d une classe C dans MM.*
2. *pour chaque relation entre deux objets, il existe dans MM, une relation entre deux classes typant les deux éléments. Dans ce qui suit ces liens sont dits instances des références entre les classes dans MM.*
3. *en n, chaque propriété sémantique dé nie dans MM est satisfaite sur M. Les multiplicités dé nies sur les références entre les concepts sont un exemple de propriété sémantique considérée. Une multiplicité peut être dé nie comme suit :*

$$multplicity(\texttt{c}_1 \in \texttt{MMV}, \texttt{r}_1 \in \texttt{MME}, \langle \texttt{MV}, \texttt{ME} \rangle) = |\{\texttt{m}_2 \in \texttt{MV} \mid \langle \langle \texttt{o}, \texttt{c}_1 \rangle, \texttt{r}_1, \texttt{m}_2 \rangle \in \texttt{ME}\}|$$

Cette notion de conformité est illustrée sur la figure 3.1 par une relation de dépendance entre un M et un MM qui lui est conforme. Les propriétés sémantiques associées au métamodèle sont implémentées dans le prédicat $conformsTo$.

Une version simplifiée du prédicat $conformsTo$ (cf. Section 5.3) est notée $instanceOf$. Cette version prend en compte les deux propriétés 1 et 2. La vérification de la propriété $instanceOf$ sera prouvée compositionnelle en relation avec les opérateurs de base de composition définis par la suite dans ce chapitre. La vérification de $conformsTo$ impose des contraintes sémantiques sur les opérations d'assemblage pour que la vérification soit compositionnelle. Ces contraintes prendront la forme de préconditions sur les paramètres des opérations de composition. La postcondition sera alors la propriété attendue en terme de vérification compositionnelle.

3.2 Formalisation de la structure du modèle

Nous présentons dans cette section l'approche que nous adoptons pour la formalisation de la structure des modèles.

3.2.1 La structure des modèles

Nous pouvons différencier plusieurs approches pour la formalisation de la structure des modèles (présentés principalement comme des multigraphes). Nous adoptons une version proche de l'approche dite classique. Cette approche consiste à représenter le multigraphe comme un ensemble de nœuds et d'arcs. Notre représentation comporte des propriétés assurant la bonne formation du multigraphe. Elle est construite à l'aide d'un type inductif avec trois constructeurs permettant ainsi d'avoir une structure pour les modèles. C'est l'approche qui était choisie pour COQ4MDE initialement et que nous n'avons pas remise en cause dans ce travail, mais, dans ce qui suit, nous présentons ses avantages et ses inconvénients. Parmi les avantages de cette représentation :

- ▷ Premièrement, elle s'appuie sur les ensembles ce qui rend la représentation assez simple et naturelle.
- ▷ Ensuite, elle permet toutes les opérations qui ont été utiles sur les modèles dont le parcours des éléments (nœuds et arcs) du multigraphe en utilisant des itérateurs.
- ▷ Aussi, elle contient des informations sur la construction du multigraphe, donc elle permet la construction d'un ordre et de chemins de parcours.
- ▷ Et enfin, elle peut être adaptée à de nombreuses autres techniques formelles qui offrent une modélisation des ensembles.

Ce que nous pouvons citer comme inconvénients est que l'approche choisie n'est pas la plus efficace pour la construction d'ordre de parcours pour le multigraphe et ne permet pas la construction de graphes infinis. Ces deux dernières propriétés ne sont pas les plus importantes dans le contexte de manipulation de modèles.

- ▷ Premièrement, concernant le parcours du multigraphe, nous disposons d'opérateurs de parcours qui en s'appuyant sur la structure de multigraphe permettent d'appliquer génériquement une fonction ou de tester une condition particulière, une partie de ces opérateurs sont présentés dans l'annexe B. Les approches basées sur la représentation comme des arbres sont plus adaptées au parcours de graphes et permettent plus de facilité pour la navigation (garder les traces des nœuds visités plus facilement). Dans la représentation en arbre, on pourrait avoir un arbre pour les relations d'héritage qui permettrait d'obtenir facilement (formellement et algorithmiquement) toutes les sous-classes d'une classe ou de déplacer une classe pour en hériter d'une autre. Les autres relations seraient alors ajoutées dans cet arbre (des relations de composition) ou représentées par d'autres structures de données comme des listes de relations d'association, cette structure est inductive et bien typée mais elle est plus complexe et l'ordre induit ne peut pas vraiment se justifier dans un contexte de modèles.
- ▷ Deuxièmement, concernant la non finitude, c'est une propriété qui n'est pas nécessaire dans notre contexte car nous considérons et manipulons des modèles qui sont toujours finis. Des expérimentations pour la définition et la manipulation de modèles infinis étaient proposés dans [CTB12] par la définition d'un opérateur *coiterate* pour l'itération sur des modèles infinis.

La structure du modèle telle que nous la présentons est implémentée sur trois niveaux hiérarchiques : le niveau des ensembles de nœuds et de liens, le niveau du graphe et enfin le niveau du modèle.

Pour chaque niveau, des propriétés spécifiques doivent être vérifiées et font intervenir des preuves pour des théorèmes partiellement présentés dans ce qui suit et dont les preuves sont accessibles avec le code COQ sur la page web http://www.irit.fr/~Mounira.Kezadri/FormalAssembly.html.

3.2.2 La structure des ensembles

Un ensemble fini est une structure qui permet de stocker certains éléments, sans aucun ordre, et sans valeurs répétées. Les ensembles des sommets et des arêtes dans notre cas et, plus généralement, souvent dans la théorie des types, ils sont représentés par leurs fonctions caractéristiques (aussi appelées fonctions indicatrices) qui contrairement aux listes assurent par définition que chaque élément est unique. Le concept de fonction caractéristique (notée $\chi_A(x)$) est implémenté dans la librairie `Uniset`[3] dans le standard de COQ comme suit : $\chi_A(x)$ prend la valeur *true* si x est un membre de l'ensemble A, et prend la valeur $false$ sinon.

$$\chi_A(x) = \begin{cases} true\ si\ x \in A \\ false\ si\ x \notin A \end{cases}$$

Remarque 3.4. *Les ensembles des nœuds et des liens sont structurés dans deux modules E et V respectivement (la structure du module est implémentée en* COQ *depuis sa version 7.4 [Chr03]). Ces modules génériques englobent les opérateurs et les propriétés spéci ques pour chaque ensemble in uençant par la suite la structure du graphe et celle du modèle. Nous présentons ici seulement les opérations et preuves de propriétés que nous utilisons par la suite.*

Nous avons enrichi le module *Uniset* de la bibliothèque standard avec de nouveaux opérateurs ensemblistes et lemmes présentés dans l'annexe A. Ceux-ci sont utilisés pour implémenter les opérations sur les ensembles de nœuds et d'arcs. L'opération *add* d'ajout d'un élément dans un ensemble est la suivante :

$$\chi_{add\ v\ A}(x) = \begin{cases} true\ si\ x = v \\ \chi_A(x)\ sinon \end{cases}$$

La fonction *add* concerne d'une manière similaire l'ensemble de nœuds et celui de liens. Le lemme 1 est très utile pour nos preuves, il montre qu'ajouter un élément v à un ensemble s d'éléments d'un certain type construit un ensemble qui contient cet élément.

Lemme 1. (addInO) $\forall\ s\ v, v \in (add\ s\ v)$.

Le type d'élément peut être `Classes`, `References` ou n'importe quel autre type. Le lemme 2 montre que pour deux éléments v et v', si un ensemble s contient v, alors l'ensemble construit en ajoutant v' à s contient aussi v.

[3] http://coq.inria.fr/stdlib/Coq.Sets.Uniset.html

Lemme 2. (addInS) $\forall\ s\ v\ v', v \in s \rightarrow v \in (add\ s\ v')$

Le lemme 3 montre que pour deux éléments v et v' et un ensemble s. Si v appartient à l'ensemble construit en ajoutant v' à s, alors soit v appartient à s ou il est égal à l'élément ajouté v'.

Lemme 3. (addElim) $\forall\ s\ v\ v', v \in (add\ s\ v') \rightarrow v \in s \vee v = v'$

Le lemme 4 montre que le résultat de l'ajout d'un élément v déjà existant dans un ensemble s produit le même ensemble. La preuve de ce lemme utilise le principe de l'extensionnalité fonctionnelle[4].

Lemme 4. (addCancel) $\forall\ s\ v, v \in s \rightarrow (add\ s\ v) = s$

3.2.3 La structure des graphes

Un graphe est défini en COQ comme un type inductif dépendant avec trois constructeurs : `Nil`[5], `AddV` et `AddA`.

$$\frac{}{graph(EmptySet, EmptySet)}\ Nil$$

$$\frac{graph(vs, es) \wedge (v \notin vs)}{(graph((add\ v\ vs), es)))}\ AddV$$

$$\frac{graph(vs, es) \wedge (v_1 \in vs) \wedge (v_2 \in vs) \wedge ((v_1, e, v_2) \notin es)}{(graph(vs, (add\ (v_1, e, v_2)\ es)))}\ AddA$$

Le type inductif `graph` dépend des deux ensembles de nœuds et d'arcs. Les nœuds et les arcs sont définis génériquement en utilisant des modules COQ [Chr03] qui peuvent être paramétrés avec différents types.

Le lemme 5 montre qu'ajouter un nœud v à un graphe g produit aussi un graphe. La preuve peut se faire en distinguant les cas d'inclusion de v dans l'ensemble vs et par la suite en utilisant le lemme 4 dans le premier cas et le second cas est trivial.

Lemme 5. $(graph_{AddV})\ \forall\ vs\ es\ v\ (g : graph(vs, es)), graph(add\ vs\ v, es)$.

Le lemme 6 montre qu'ajouter un arc (v_1, a, v_2) à un graphe g produit aussi un graphe. La preuve est faite d'une manière similaire que celle du théorème précédent.

Lemme 6. $(graph_{AddV})\ \forall\ vs\ es\ v_1\ v_2\ a\ (g : graph(vs, es)), graph(vs, add\ es\ (v_1, a, v_2))$.

Remarque 3.5. *Les lemmes 5 et 6 correspondent aux dé nitions par la preuves des fonctions de construction des graphes correspondants.*

Nous définissons dans ce qui suit des propriétés utiles sur les graphes qui seront exploitées dans nos preuves.

[4] http://coq.inria.fr/stdlib/Coq.Logic.FunctionalExtensionality.html

[5] EmptySet dans la définition représente l'ensemble vide.

3.2.3.1 Des arcs corrects

La propriété de correction des arcs est décrite à l'aide du lemme 7. Tous les arcs du graphe doivent être corrects. Un arc appartenant à l'ensemble des arcs est dit correct si ses deux extrémités appartiennent aussi à l'ensemble des nœuds. La preuve est faite par induction sur la structure du graphe.

Lemme 7. (correctEdges) $\forall\ vs\ es\ (g : graph(vs, es))\ v_1\ v_2\ a,$
$(v_1, a, v_2) \in es \rightarrow (v_1 \in vs) \wedge (v_2 \in vs)$

3.2.3.2 Égalité des graphes

Un graphe dans `Coq4mde` est défini comme un type dépendant. L'égalité entre graphes notée Eq_{graph} est aussi définie comme une égalité dépendant[6] des ensembles de nœuds et d'arcs. L'égalité telle qu'elle est définie assure les propriétés de ré exivité, symétrie et transitivité exprimées successivement dans les trois lemmes 8, 9 et 10.

Lemme 8. (eqGraphRe) $\forall\ VE\ (g : graph\ VE), Eq_{graph}\ g\ g.$

Lemme 9. (eqGraphSym) $\forall\ VE\ VE'\ (g : graph\ VE)\ (g' : graph\ VE'),$
$Eq_{graph}\ g\ g' \rightarrow Eq_{graph}\ g'\ g.$

Lemme 10. (eqGraphTrans) $\forall\ VE\ VE'\ VE''\ (g : graph\ VE)\ (g' : graph\ VE')$
$(g'' : graph\ VE''), Eq_{graph}\ g\ g' \wedge Eq_{graph}\ g'\ g'' \rightarrow Eq_{graph}\ g'\ g''$

Le lemme 11 indique que l'ajout du même nœud v à deux graphes égaux g et g' produit deux graphes qui sont aussi égaux.

Lemme 11. (eqGraphAddV) $\forall\ VE\ VE'\ (g : graph\ VE)\ (g' : graph\ VE')\ v,$
$Eq_{graph}\ g\ g' \rightarrow Eq_{graph}\ (graph_{AddV}\ g\ v)\ (graph_{AddV}\ g'\ v).$

Le lemme 12 montre que si les deux graphes g et g' sont égaux, alors le graphe construit en ajoutant un arc a au graphe g est égal au graphe construit en ajoutant a au graphe g'.

Lemme 12. (eqGraphAddA) $\forall\ VE\ VE'\ (g : graph\ VE)\ (g' : graph\ VE')\ a, Eq_{graph}\ g\ g'$
$\rightarrow Eq_{graph}\ (graph_{AddA}\ g\ a)\ (graph_{AddA}\ g'\ a).$

3.3 Formalisation de la composition de modèles

Chronologiquement , nous avons d'abord proposé dans [KCP+11] une extension du « framework » `Coq4MDE` pour offrir le style de composition `ISC`. Ces travaux comportaient une formalisation des opérateurs de base `ISC` ainsi que la vérification de la préservation du bon typage et des propriétés sémantiques du métamodèle par ceux-ci. A l'issu de ces travaux, pour factoriser les définitions et preuves réalisées, nous avons défini des opérateurs élémentaires (primitifs) de composition ont été définis. Nous présenterons ceux-ci dans la section suivante. Ces opérateurs sont suffisants pour l'implémentation des opérateurs `ISC` que

[6] http://coq.inria.fr/stdlib/Coq.Logic.Eqdep.html

nous présentons dans le chapitre 5 ainsi que d'autres opérateurs de composition classiques comme le `Package Merge` d'UML et EMOF que nous présenterons dans le chapitre 4.

Les opérateurs élémentaires doivent être décrits sur les trois niveaux hiérarchiques des modèles (le niveau ensemble, le niveau graphe et le niveau modèle). L'algèbre des ensembles considère de nombreux opérateurs primitifs. Nous sommes intéressés par deux opérateurs élémentaires qui ont été suffisants selon nos expériences pour implémenter la plupart des opérateurs de composition de modèles.

3.3.1 L'opérateur élémentaire d'Union

Nous utilisons l'union ensembliste définie dans le module `Uniset` de la bibliothèque standard que nous notons $\cup$. L'union est implémentée en utilisant les booléens tel que, x est un membre de l'ensemble $A_1 \cup A_2$ si la valeur de $\chi_{A_1}(x) \vee \chi_{A_2}(x)$ est *true*, x n'appartient pas à l'union dans le cas contraire.

$$\chi_{A_1 \cup A_2}(x) = \chi_{A_1}(x) \vee \chi_{A_2}(x)$$

L'union de deux graphes est aussi un graphe, l'ensemble des nœuds est l'union des deux ensembles de nœuds et l'ensemble des arcs est l'union des deux ensembles d'arcs des deux graphes. Le lemme 13 définit la propriété que les deux ensembles forment bien un graphe. La preuve est faite par induction sur la structure du graphe[7].

Lemme 13. (graphUnion) $\forall\ vs_1\ es_1\ vs_2\ es_2\ (g_1 : graph(vs_1, es_1))\ (g_2 : graph(vs_2, es_2)),$ $graph((Union\ vs_1\ vs_2), (Union\ es_1\ es_2))$

Remarque 3.6. *Le lemme 13 correspond à la dé nition par la preuve du* $graph((Union\ vs_1\ vs_2), (Union\ es_1\ es_2))$ *que nous notons sous la forme* $graph_{union}\ g_1\ g_2$.

Remarque 3.7. *L union de deux modèles* $\langle vs_1, es_1\rangle$ *et* $\langle vs_2, es_2\rangle$ *est l union de leurs deux ensembles de nœuds et d arcs en plus de la preuve du lemme 13. L union de ces deux modèles peut être notée :* $\langle vs_1 \cup vs_2, es_1 \cup es_2\rangle$ *ou* $(Union\langle vs_1, es_1\rangle\langle vs_2, es_2\rangle)$.

Exemple 3.2. *Nous illustrons la dé nition de l opérateur union sur l exemple des deux modèles* M_1 *présenté sur la gure 3.3 et* M_2 *présenté sur la gure 3.4.*

Figure 3.3 — Le modèle M_1

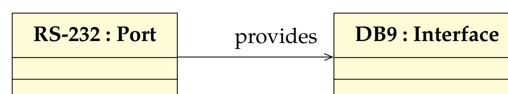


Figure 3.4 — Le modèle M_2

Le modèle M_1 *indique que le* `Port RS-232` *a comme interface* `DB25` *et le modèle* M_2 *indique que le* `Port RS-232` *a comme interface* `DB9`*. Le résultat de l application de l union est le modèle présenté sur la gure 3.5.*

[7] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Graph.html#elements

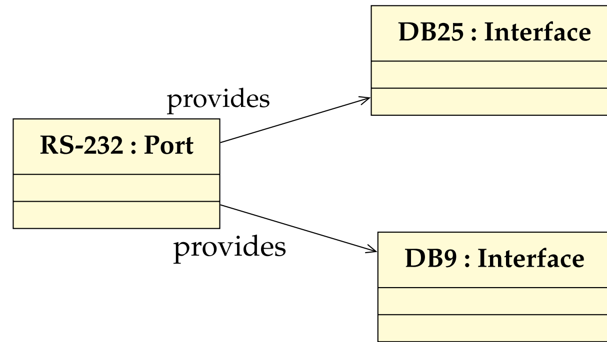


Figure 3.5 — L'union des deux modèles M_1 et M_2

Propriétés de l'union de graphes La propriété de congruence de l'égalité pour l'union de graphes est expliquée dans le lemme 14. La preuve est faite par application des règles de l'égalité dépendante.

Lemme 14. (congruenceEqGraph) $\forall\ vs_1\ es_1\ (g_1 : graph(vs_1, es_1))$
$vs'_1\ es'_1\ (g'_1 : graph(vs'_1, es'_1))\ vs_2\ es_2\ (g_2 : graph(vs_2, es_2))\ vs'_2\ es'_2\ (g'_2 : graph(vs'_2, es'_2))$,
$Eq_{graph}\ g_1\ g'_1 \rightarrow Eq_{graph}\ g_2\ g'_2 \rightarrow Eq_{graph}\ (graph_{union}\ g_1\ g_2)\ (graph_{union}\ g'_1\ g'_2)$.

Le résultat de l'union d'un graphe avec un graphe vide est le même graphe.

Lemme 15. (elementsNil) $\forall\ vs_2\ es_2\ (g_2 : graph(vs_2, es_2)), Eq_{graph}\ (graph_{union}\ nil\ g_2)\ g_2$.

Le lemme 16 montre qu'ajouter un nœud au résultat de l'union de deux graphes produit un graphe égal à l'union du résultat de l'ajout d'un nœud au premier graphe avec le deuxième graphe.

Lemme 16. (elementsAddV) $\forall\ vs_1\ es_1\ (g_1 : graph(vs_1, es_1))\ v\ vs_2\ es_2\ (g_2 : graph(vs_2, es_2))$,
$Eq_{graph}\ (graph_{union}\ (graph_{AddV}\ v\ g_1)\ g_2)\ (graph_{AddV}\ v\ (graph_{union}\ g_1\ g_2))$.

Le lemme 17 montre qu'ajouter un arc (v_1, a, v_2) au résultat de l'union de deux graphes produit un graphe égal à l'union du résultat de l'ajout de l'arc (v_1, a, v_2) au premier graphe avec le deuxième graphe.

Lemme 17. (elementsAddA) $\forall\ vs_1\ es_1\ (g_1 : graph(vs_1, es_1))\ v_1\ v_2\ a$
$vs_2\ es_2\ (g_2 : graph(vs_2, es_2))$,
$Eq_{graph}\ (graph_{union}\ (addA\ (v_1, a, v_2)\ g_1)\ g_2)\ (addA\ (v_1, a, v_2)\ (graph_{union}\ g_1\ g_2))$.

3.3.2 L'opérateur élémentaire de Substitution

Les modèles sont des graphes dont les éléments sont des objets typés. Par exemple (o, c) est un élément de modèle dont le type est c et le nom est o. L'opérateur de substitution a comme but de remplacer le nom d'un élément de modèle par un autre nom. Cette opération comme l'union doit être décrite sur les trois niveaux hiérarchiques. Substituer un élément de modèle dont le nom est *src* par un élément de modèle dont le nom est *dst* dans l'ensemble de nœuds et d'arcs est implémenté en utilisant un opérateur *map* générique construit à partir

de trois opérations : *mapv* , *mapa* et *mape* qui s'appliquent respectivement sur les éléments du modèle, les références et les arcs ainsi qu'une fonction *image* qui permet de construire l'image du graphe.

Remarque 3.8. *Dans les fonctions qui suivent l objet src, l objet dst et le graphe g sont des paramètres implicites.*

Dans l'opérateur *Substitution*, la fonction *mapv* est utilisée pour remplacer *src* par *dst* dans l'ensemble des nœuds. Elle est définie comme suit :

$$mapv\ (o, c) = \begin{cases} (dst, c)\ si\ o = src \\ (o, c)\ sinon \end{cases}$$

La fonction *mapa* définit l'image des références par l'opérateur de substitution, elle est définie pour l'opérateur *Substitution* comme une fonction identité, sa présence vient du fait que l'implémentation s'appuie sur une fonction *map* plus générique qui peut permettre de remplacer également les références. Un arc est défini avec ses deux extrémités (les éléments de modèle) et sa référence. Par exemple, (v_1, a, v_2) est l'arc entre v_1 et v_2 dont la référence est a. La fonction *mape* remplace les noms des éléments de modèle dans les arcs tel que :

$$mape\ (v_1, a, v_2) = (mapv\ v_1, mapa\ a, mapv\ v_2)$$

Nous supposons que *mapv* et *mapa* sont injectives, nous pouvons donc prouver que *mape* l'est aussi en utilisant le lemme 18 $mape_{inj}$.

Lemme 18. $(mape_{inj})\ \ \forall\, e_1\, e_2,\ mape\ e = mape\ e' \to e = e'$

L'image du graphe est obtenue à partir des images des ensembles de nœuds et d'arcs. Pour un graphe *g* avec *vs* l'ensemble des nœuds et *es* l'ensemble d'arcs, l'image des nœuds par la fonction de *Substitution* est : $(V.image\ mapv\ (vs, es)\ g)$ et l'image des arcs par la fonction de *Substitution* est : $(E.image\ mapv\ mapa\ g)$.

Remarque 3.9. *Les notations V.image et E.image viennent du fait que ces deux fonctions sont codées séparément dans le module V pour les nœuds et le module E pour les arcs.*

Représenter les ensembles comme des fonctions caractéristiques est efficace pour décider si un élément appartient ou pas à un ensemble. Mais, ce codage est moins adapté pour itérer les éléments de l'ensemble ce qui est nécessaire par exemple pour définir *V.image* qui applique *mapv* sur tous les éléments de l'ensemble des nœuds.

Une fonction *fold* (présentée dans l'annexe B) pour itérer sur les graphes est utilisée pour l'implémentation de *V.image* et *E.image* (la fonction qui applique *mape* sur tous les éléments de l'ensemble des arcs). La fonction *fold* applique *mapv* ou *mape* sur chaque nœud ou arc du graphe et l'ensemble des images de nœuds et l'ensemble des images d'arcs. Enfin, l'image du graphe est constituée des images des deux ensembles. La définition de la fonction *graphSubstitution* est décrite en utilisant le lemme 19 et prouvée par induction sur la structure du graphe[8].

[8] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Subst_Verif.html#elements

Lemme 19. (graphSubstitution) $\forall$ *vs es* $(g : graph(vs, es))$,
$graph\ (V.image\ mapv\ g)\ (E.image\ mapv\ mapa\ g)$

Le modèle qui résulte de la substitution d'un nom d'élément de modèle x_1 par un autre nom x_2 dans les ensembles des nœuds et d'arcs associés à l'image du graphe par substitution (Lemme 19) forme le modèle substitué. Le modèle résultat de la substitution peut être noté : $\langle SubstV\ x_1\ x_2\ vs, SubstE\ x_1\ x_2\ es\rangle$.

Exemple 3.3. *Le résultat de* $(Substitution\ DB25\ DC37\ M_1)$ *du modèle présenté sur la gure 3.3 est illustré sur la gure 3.6 :*

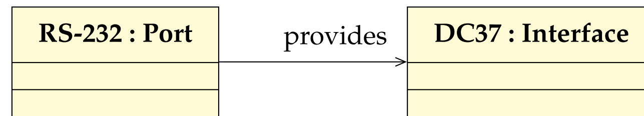


Figure 3.6 — Le modèle résultat de la substitution

Propriétés de l'image du graphe par substitution Le lemme 20 montre que le résultat de la substitution d'un nom de nœud par un autre dans un graphe (g) après l'ajout d'un nœud (v) est équivalent à l'ajout du nœud ($mapv\ v$) au résultat de la substitution du graphe initial ($graphSubstitution\ (vs, es)\ g$).

Lemme 20. (elementsAddV) $\forall$ *vs es* $(g : graph(vs, es))\ v$,

$$Eq_{graph}\quad (graph_{Substitution}\ ((add\ vs\ v), es)\ (graph_{AddV}\ v\ g))$$
$$(graph_{AddV}\ (mapv\ v)\ (graph_{Substitution}\ (vs, es)\ g)).$$

Le lemme 21 montre que la substitution dans un graphe (g) résultant de l'ajout d'un arc (src, a, dst) est équivalente à l'ajout de l'arc $((mapv\ src), (mapa\ a), (mapv\ dst))$ résultant de l'application des fonctions *mapv* et *mapa* au résultat de la substitution du graphe initial $(graph_{Substitution}\ (vs, es)\ g)$.

Lemme 21. (elementsAddE) $\forall$ *vs es src dst a* $(g : graph(vs, es))$,

$$Eq_{graph}\quad (graph_{Substitution}\ (vs, (add\ es\ (src, a, dst)))\ (graph_{AddA}\ (vs, es)\ src\ dst\ a\ g))$$
$$(graph_{AddA}\ (mapv\ src)\ (mapv\ dst)\ (mapa\ a)\ (graph_{Substitution}\ (vs, es)\ g)).$$

3.4 Formalisation et véri cation de propriétés

Les opérateurs `Substitution` et `Union` sont définis pour préserver les propriétés de bon typage. Les deux opérateurs comme tous les concepts, théorèmes et preuves présentés sont codés dans l'assistant de preuve COQ et sont complètement accessibles sur une page web spécifique[9]. Le but de cette formalisation est de vérifier quelques propriétés sur le modèle composite et de fournir les bases pour la spécification et la preuve de correction des techniques de vérification compositionelles. La première propriété considérée est le bon typage d'un modèle par rapport à un métamodèle (`instanceOf`).

[9] http://www.irit.fr/~Mounira.Kezadri/FormalAssembly.html

3.4.1 InstanceOf

Cette propriété est liée à la relation de conformité définie dans 3.1. Elle vérifie pour un modèle M et un métamodèle MM que chaque objet de M est une instance d'une classe dans MM et chaque lien de M est une instance d'une relation dans MM. Pour prouver que cette vérification est compositionelle, nous avons besoin de prouver que la composition de deux modèles instances d'un même métamodèle est aussi instance du même métamodèle. Nous définissons le premier critère de correction d'une fonction de composition. Ce critère est défini comme un prédicat d'ordre supérieur qui vérifie le bon typage pour une certaine fonction. La fonction `instanceOf` est utilisée pour ce but, elle vérifie que tous les objets et liens d'un modèle sont des instances de classes et références d'un métamodèle.

$$\begin{aligned}
&InstanceOf(\langle\langle \mathtt{MV}, \mathtt{ME}\rangle, \langle\langle \mathtt{MMV}, \mathtt{MME}, conformsTo\rangle\rangle\rangle) \triangleq \\
&\quad \forall \langle \mathtt{o}, \mathtt{c}\rangle \in \mathtt{MV}, \mathtt{c} \in \mathtt{MMV} \\
&\wedge \forall \langle\langle \mathtt{o}, \mathtt{c}\rangle, \mathtt{r}, \langle \mathtt{o}', \mathtt{c}'\rangle\rangle \in \mathtt{ME}, \langle \mathtt{c}, \mathtt{r}, \mathtt{c}'\rangle \in \mathtt{MME}
\end{aligned}$$

Ce prédicat est ensuite utilisé pour vérifier le bon typage pour le résultat de l'opérateur *Substitution* en s'inspirant du style de preuves présenté dans [KP10b] pour la vérification de transformations de modèles. La propriété du bon typage pour la substitution est exprimée à l'aide du théorème 1.

Théorème 1. (ValidSubstitution) $\forall\, \mathtt{M} \in Model,\ \mathtt{MM} \in MetaModel,$
$InstanceOf\ (\mathtt{M}, \mathtt{MM}) \rightarrow InstanceOf\ ((Substitution\ \mathtt{o}_1\ \mathtt{o}_2\ \mathtt{M}), \mathtt{MM})$

Nous utilisons aussi ce prédicat pour vérifier que sur deux composants instances de MM, le composant résultant de l'application de l'opérateur `Union` est aussi instance de MM.

Théorème 2. (ValidUnion) $\forall\, \mathtt{M}_1\ \mathtt{M}_2 \in Model,\ \mathtt{MM} \in MetaModel,$
$InstanceOf\ (\mathtt{M}_1, \mathtt{MM})\ \wedge\ InstanceOf\ (\mathtt{M}_2, \mathtt{MM}) \rightarrow InstanceOf\ ((Union\ \mathtt{M}_1\ \mathtt{M}_2), \mathtt{MM})$

3.4.2 Conformité dans le standard **MOF**

Nous avons abordé le métamétamodèle EMOF dans le chapitre 2. à partir de sa version 1.2 standardisée en 2006, un noyau du EMOF appelé EMOF est extrait à partir de la version complète de EMOF (CMOF). Selon la spécification, EMOF contient un ensemble minimal d'éléments nécessaires pour modéliser les systèmes à base d'objet. La figure 3.7 montre les concepts de base dans EMOF spécifiés comme un diagramme de classe UML. Le métamodèle spécifie un ensemble de `Types` qui sont soit des `Datatypes` ou des `Classes`. Le concept principal dans EMOF est `Class` pour définir les métaclasses. Chaque métaclasse représente un concept ou une construction dans le langage de modélisation. Les métaclasses peuvent hériter des caractéristiques à travers le concept de sous-classe. L'héritage est exprimé à l'aide de la référence `superClass`. Une classe peut être abstraite (`isAbstract`). Les classes ont un nombre arbitraire d'attributs (`ownedAttributes`), caractérisés par une propriété `lower` et `upper` définies par `MultiplicityElement`. Dans EMOF, une caractéristique est définie comme `Property`. Une `Property` peut avoir une `Property opposite`. Pour vérifier que les opérateurs élémentaires de composition préservent la conformité par rapport aux propriétés sémantiques du métamétamodèle (autre

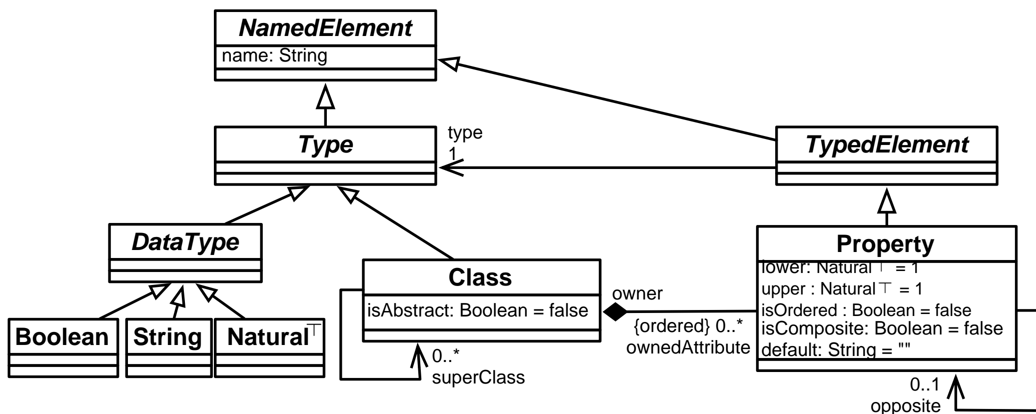


Figure 3.7 — Les concepts de base de EMOF

que le bon typage), nous avons élaboré une manière élégante pour prendre en compte les contraintes sémantiques du métamétamodèle EMOF. Cette approche nous évite d'essayer d'extraire les propriétés à partir du métamodèle initial ce qui est complexe car *conformsTo* est définie d'une manière générique pour supporter tout type de propriétés sur le métamodèle. La solution est de vérifier que chaque propriété élémentaire vérifiée sur le modèle initial est aussi vérifiée sur le modèle résultat de l'application d'un opérateur élémentaire de composition. Donc, si les modèles initiaux sont conformes à un métamodèle, le modèle résultant est conforme au même métamodèle comme illustré sur la figure 3.8. Les propriétés sémantiques élémentaires considérées pour EMOF sont : l'héritage (`subClass`), les classes abstraites (`isAbstract`), les multiplicités (`lower`, `upper`), les références opposites (`isOpposite`) et les références composites (`areComposite`).

Dans ce qui suit, nous allons présenter pour chaque propriété sémantique élémentaire : sa formalisation, le lemme qui prouve sa préservation par les opérateurs élémentaires `Substitution` et `Union` de base et les liens vers les preuves complètes en COQ. Nous indiquerons pour chaque propriété les préconditions supplémentaires nécessaires sur les opérateurs de composition pour obtenir une vérification compositionnelle.

3.4.3 subClass

La formalisation de la propriété `subClass` est présentée dans la section 3.1. Le théorème 3 montre[10] que l'héritage est préservé par l'opérateur `Substitution` de base. Ainsi, pour toutes classes c_1, c_2 et pour tous objets o_1, o_2, si c_1 est une *subClass* de c_2 dans un modèle M, alors c_1 est aussi une *subClass* de c_2 dans le modèle (*Substitution* o_1 o_2 M).

Théorème 3. (SubstSubClassPreserved)
$\forall\, \mathtt{M} \in \mathit{Model},\ \mathtt{c}_1\ \mathtt{c}_2 \in \mathit{Classes},\ \mathtt{o}_1\ \mathtt{o}_2 \in \mathit{Objects},$
$\mathit{subClass}\ \mathtt{c}_1\ \mathtt{c}_2\ \mathtt{M} \rightarrow \mathit{subClass}\ \mathtt{c}_1\ \mathtt{c}_2\ (\mathit{Substitution}\ \mathtt{o}_1\ \mathtt{o}_2\ \mathtt{M})$

Le théorème 4 montre[11] que la propriété `subClass` est préservée par l'opérateur `Union`

[10] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Subst_Verif.html#SubstSCP
[11] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Union_Verif.html#SCUP

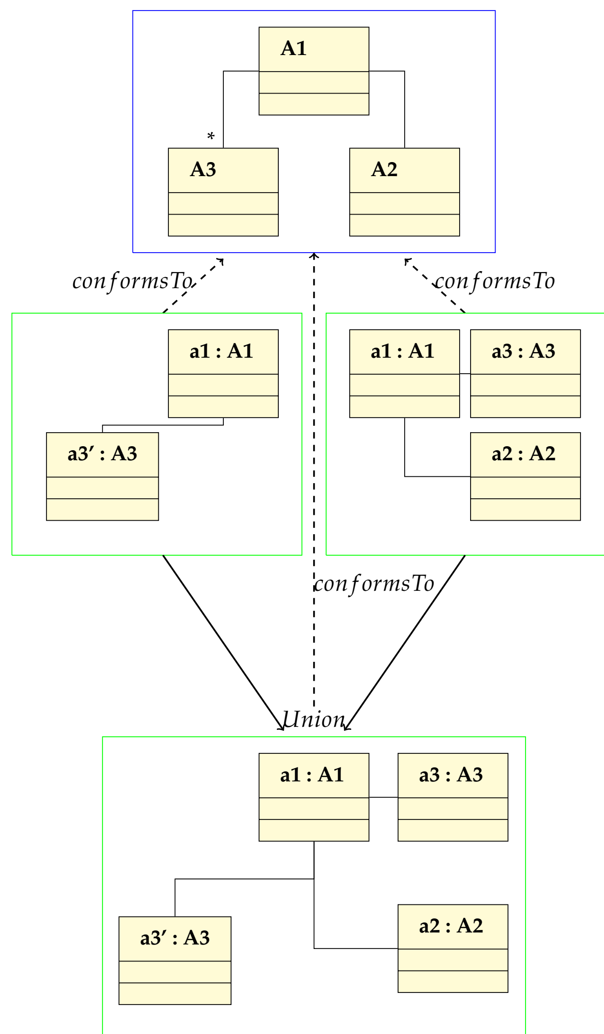


Figure 3.8 — La vérification du *conformsTo*

de base. Ainsi, pour toutes classes c_1 c_2 typant les éléments du modèle, si c_1 est une *subClass* de c_2 dans chacun des deux modèles M_1 et M_2, alors c_1 est aussi une *subClass* de c_2 dans le modèle résultat de l'opération $(Union\ M_1\ M_2)$

Théorème 4. (subClassUnionPreserved)
$\forall\, \mathtt{M}_1\ \mathtt{M}_2 \in \mathtt{Model},\ c_1\ c_2 \in \mathtt{Classes},$
$subClass\ c_1\ c_2\ \mathtt{M}_1\ \wedge subClass\ c_1\ c_2\ \mathtt{M}_2 \rightarrow subClass\ c_1\ c_2\ (Union\ \mathtt{M}_1\ \mathtt{M}_2).$

Il n'y a donc aucune précondition supplémentaire sur les opérateurs de composition pour que la vérification de la relation `subClass` soit compositionnelle.

3.4.4 isAbstract

La propriété `isAbstract` est également formalisée dans la section 3.1. La préservation de cette propriété par l'opérateur `Substitution` de base est prouvée en utilisant le théo-

rème 5. Ce théorème montre[12] que toutes les classes abstraites dans un modèle sont aussi abstraites dans le modèle après substitution.

Théorème 5. (SubstIsAbstractPreserved)
$\forall\, \mathtt{M} \in \mathit{Model},\ \mathtt{c} \in \mathit{Classes},\ \mathtt{o_1}\ \mathtt{o_2} \in \mathit{Objects},$
$\mathit{isAbstract}\ \mathtt{c}\ \mathtt{M} \rightarrow \mathit{isAbstract}\ \mathtt{c}\ (\mathit{Substitution}\ \mathtt{o_1}\ \mathtt{o_2}\ \mathtt{M})$

Le théorème 6 montre[13] que la propriété `isAbstract` est préservée par l'opérateur `Union`. Le théorème vérifie que chaque classe abstraite dans les deux modèles est aussi abstraite dans le modèle résultat de l'union de ces deux modèles.

Théorème 6. (isAbstractUnionPreserved)
$\forall\, \mathtt{M_1}\ \mathtt{M_2} \in \mathtt{Model},\ c \in \mathtt{Classes},$
$\mathit{isAbstract}\ c\ \mathtt{M_1} \wedge \mathit{isAbstract}\ c\ \mathtt{M_2} \rightarrow \mathit{isAbstract}\ c\ (\mathit{Union}\ \mathtt{M_1}\ \mathtt{M_2}).$

Il n'y a donc aucune précondition supplémentaire sur les opérateurs de composition pour que la vérification de la relation `isAbstract` soit compositionnelle.

3.4.5 lower & upper

Pour un attribut ou une référence, le nombre minimum et maximum d'instances du concept cible peut être défini en utilisant les attributs `lower` et `upper`. Cette paire est usuellement appelée multiplicité. Un intervalle non borné pour `upper` est modélisé à l'aide de la valeur $\top$. Pour cela le type $Natural^{\top} = \mathbb{N} \cup \{\top\}$ où $\forall e \in \mathbb{N}, e < \top$.

$$\begin{aligned} &\mathit{lower}(\mathtt{c_1} \in \mathtt{MMV}, \mathtt{r_1} \in \mathtt{MME}, \mathtt{n} \in \mathit{Natural}^{\top}, \langle \mathtt{MV}, \mathtt{ME} \rangle) \triangleq \\ &\quad \forall\, o \in \mathtt{Objects}, \langle \mathtt{o}, \mathtt{c_1} \rangle \in \mathtt{MV} \Rightarrow |\{\mathtt{m_2} \in \mathtt{MV} \mid \langle \langle \mathtt{o}, \mathtt{c_1} \rangle, \mathtt{r_1}, \mathtt{m_2} \rangle \in \mathtt{ME}\}| \geq \mathtt{n} \end{aligned}$$

Une formalisation analogue est définie pour la propriété `upper` en remplaçant $\geq$ par $\leq$.

Le théorème `SubstLowerPreserved` montre[14] que la propriété `lower` est préservée par l'opérateur de `Substitution`. Cette propriété impose que la substitution soit injective et préserve la distinction entre les éléments dans le modèle résultant. Ceci est assuré si l'élément de modèle o_2 n'appartient pas au modèle substitué, dans ce cas, l'opérateur `Substitution` n'ajoute pas un élément qui est déjà dans le modèle. Donc, les bornes `lower` sont préservées.

Théorème 7. (SubstLowerPreserved)
$\forall\, \langle \mathtt{MV}, \mathtt{ME} \rangle \in \mathit{Model},\ \mathtt{c} \in \mathit{Classes},\ \mathtt{r} \in\ \mathit{References},$
$\mathtt{n} \in \mathit{Natural}^{\top},\ \mathtt{o_1}\ \mathtt{o_2} \in \mathit{Objects},\ \mathtt{c_1}\ \mathtt{c_2} \in \mathit{Classes},$
$\mathtt{c_1} = \mathtt{c_2} \wedge ((o_2, c) \notin \mathtt{MV}) \wedge \mathit{Injectif\ Substitution}$
$\wedge\ (\mathit{lower}\ \mathtt{c}\ \mathtt{r}\ \mathtt{n}\ \langle \mathtt{MV}, \mathtt{ME} \rangle) \rightarrow (\mathit{lower}\ \mathtt{c}\ \mathtt{r}\ \mathtt{n}\ (\mathit{Substitution}\ (\mathtt{o_1}, \mathtt{c_1})\ (\mathtt{o_2}, \mathtt{c_2})\ \langle \mathtt{MV}, \mathtt{ME} \rangle)).$

Remarque 3.10. *La propriété Injectif Substitution est dé nie dans le code* COQ *comme suit :* $\forall o_1\ o_2\ v\ v',\ mapv\ o_1\ o_2\ v = mapv\ o_1\ o_2\ v' \rightarrow v = v'$

[12] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Subst_Verif.html#SubstIAP
[13] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Union_Verif.html#IAUP
[14] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Subst_Verif.html#SubstLP

La préservation de la propriété `lower` pour l'opérateur `Union` est vérifiée[15] à l'aide du théorème 8. Le théorème vérifie que la borne minimale n pour une relation r avec comme source une classe c qui est vérifiée dans les deux modèles est aussi vérifiée dans le modèle obtenu par union des deux modèles.

Théorème 8. (lowerUnionPreserved)
$\forall \langle \mathtt{MV_1,ME_1} \rangle \langle \mathtt{MV_2,ME_2} \rangle \in \mathtt{Model},\ c \in \mathtt{Classes}, r \in \mathtt{References},\ n \in \mathtt{Natural},$
$\mathtt{MM} \in \mathtt{MetaModel}, (lowerCond\ c\ r\ n\ \langle \mathtt{MV_1,ME_1} \rangle \langle \mathtt{MV_2,ME_2} \rangle)$
$\wedge (lower\ c\ r\ n\ \langle \mathtt{MV_1,ME_1} \rangle) \wedge (lower\ c\ r\ n\ \langle \mathtt{MV_2,ME_2} \rangle)$
$\rightarrow (lower\ c\ r\ n\ (Union\ \langle \mathtt{MV_1,ME_1} \rangle \langle \mathtt{MV_2,ME_2} \rangle\ \mathtt{MM})).$

La preuve nécessite une hypothèse particulière sur les paramètres de l'opérateur de composition : le cardinal n doit être supérieur ou égal au cardinal dans le modèle obtenu par intersection des deux modèles. La condition est exprimée comme suit :

$$lowerCond(\mathtt{c},\ \mathtt{r},\ \mathtt{n},\ \langle \mathtt{MV_1,ME_1} \rangle \langle \mathtt{MV_2,ME_2} \rangle) \triangleq$$
$$\mathtt{n} \geq |\{\mathtt{o_2} \in (\mathtt{MV_1} \cap \mathtt{MV_2}) \mid \langle \langle \mathtt{o,c} \rangle, \mathtt{r}, \mathtt{o_2} \rangle \in (\mathtt{ME_1} \cap \mathtt{ME_2})\}|$$

La préservation de la propriété `upper` par l'opérateur `Substitution` est décrite[16] en utilisant le théorème 9 qui est similaire au théorème 7 sur la propriété `lower`. Le modèle ne doit pas contenir un élément dont le nom est o_2.

Théorème 9. (SubstUpperPreserved)
$\forall \langle \mathtt{MV,ME} \rangle \in Model,\ \mathtt{c} \in Classes,\ \mathtt{r} \in\ References,$
$\mathtt{n} \in Natural^{\top},\ \mathtt{o_1}\ \mathtt{o_2} \in Objects,\ \mathtt{c_1}\ \mathtt{c_2} \in Classes,$
$\mathtt{c_1} = \mathtt{c_2} \wedge ((o_2, c) \notin \mathtt{MV}) \wedge Injectif\ Substitution$
$\wedge (upper\ \mathtt{c}\ \mathtt{r}\ \mathtt{n}\ \langle \mathtt{MV,ME} \rangle) \rightarrow (upper\ \mathtt{c}\ \mathtt{r}\ \mathtt{n} (Substitution\ (\mathtt{o_1,c_1})\ (\mathtt{o_2,c_2})\ \langle \mathtt{MV,ME} \rangle)).$

Le théorème 10 vérifie[17] que la propriété `upper` est préservée par l'opérateur `Union`.

Théorème 10. (upperUnionPreserved)
$\forall \langle \mathtt{MV_1,ME_1} \rangle \langle \mathtt{MV_2,ME_2} \rangle \in \mathtt{Model},\ c \in \mathtt{Classes},\ r \in \mathtt{References},\ n \in \mathtt{Natural},$
$\mathtt{MM} \in \mathtt{MetaModel},\ (upperCond\ c\ r\ n\ \langle \mathtt{MV_1,ME_1} \rangle \langle \mathtt{MV_2,ME_2} \rangle)$
$\wedge(upper\ c\ r\ n\ \langle \mathtt{MV_1,ME_1} \rangle) \wedge (upper\ c\ r\ n\ \langle \mathtt{MV_2,ME_2} \rangle)$
$\rightarrow (upper\ c\ r\ n\ (Union\ \langle \mathtt{MV_1,ME_1} \rangle \langle \mathtt{MV_2,ME_2} \rangle)).$

L'hypothèse assurant la vérification est que le cardinal n associé à la relation r avec comme cible la classe c doit être supérieur à la somme de ses cardinaux dans les deux modèles réduite de son cardinal dans l'intersection des deux modèles. Donc, le cardinal n doit satisfaire la condition `upperCond` :

$$upperCond(\mathtt{c},\ r,\ \mathtt{n},\ \langle \mathtt{MV_1,ME_1} \rangle \langle \mathtt{MV_2,ME_2} \rangle) \triangleq$$
$$\mathtt{n} > |\{\mathtt{o_2} \in \mathtt{MV_1} \mid \langle \langle \mathtt{o,c} \rangle, \mathtt{r}, \mathtt{o_2} \rangle \in \mathtt{ME_1}\}| + |\{\mathtt{o_2} \in \mathtt{MV_2} \mid \langle \langle \mathtt{o,c} \rangle, \mathtt{r}, \mathtt{o_2} \rangle \in \mathtt{ME_2}\}|$$
$$-|\{\mathtt{o_2} \in (\mathtt{MV_1} \cap \mathtt{MV_2}) \mid \langle \langle \mathtt{o,c} \rangle, \mathtt{r}, \mathtt{o_2} \rangle \in (\mathtt{ME_1} \cap \mathtt{ME_2})\}|$$

[15] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Union_Verif.html#LUP
[16] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Subst_Verif.html#SubstUP
[17] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Union_Verif.html#UUP

Pour illustrer la signification et l'utilité des conditions pour les multiplicités nous présentons un exemple simplifié d'un métamodèle dans la figure 3.9.

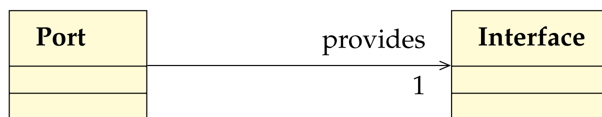


Figure 3.9 — Un extrait du métamodèle des ports

Exemple 3.4. *La multiplicité* 1 *pour la relation* `provides` *pour la classe* `Port` *signi e que dans un modèle conforme à ce métamodèle, un* `Port` *particulier peut avoir exactement une interface. Les deux modèles M_1 (sur la gure 3.3) et M_2 (sur la gure 3.4) sont conformes à ce dernier métamodèle. Par contre, le résultat de l application d une union simple est le modèle présenté sur la gure 3.5, le résultat est non conforme au même métamodèle.*

Ce problème est contourné grâce aux préconditions supplémentaires exprimées sur les opérateurs de composition et dans ce cas l hypothèse `upperCond`*. Véri ons la condition de la multiplicité* `upper` *par rapport au* `Port RS-232` *en relation avec la propriété* `provides` *et les objets de type interface (égal à* `1` *à partir du métamodèle 3.9). Le cardinal de cette relation dans le modèle M_1 est* `1`*, le cardinal de la relation dans le modèle M_2 est* `1` *et sa multiplicité dans le modèle construit à partir de l intersection de ces deux modèles est* `0`*. Nous remarquons que la condition* `upperCond` *n est pas satisfaite et l union de ces deux modèles n est pas un modèle conforme au métamodèle de base.*

Il est donc nécessaire d'introduire des préconditions supplémentaires sur les opérateurs pour que la vérification des propriétés `lower` et `upper` soit compositionnelle.

3.4.6 isOpposite

Une référence peut être associée à une référence *opposite*. Ceci implique que, dans les modèles instances du métamodèle qui défini la référence, pour chaque lien de ce type entre deux objets instances de cette référence, un lien doit exister dans la direction inverse entre ces deux mêmes objets.

$$\begin{aligned} &isOpposite(\mathtt{r_1}, \mathtt{r_2} \in \mathtt{MME}, \langle \mathtt{MV}, \mathtt{ME} \rangle) \triangleq \\ &\quad \forall\, \mathtt{m_1}, \mathtt{m_2} \in \mathtt{MV}, \langle \mathtt{m_1}, \mathtt{r_1}, \mathtt{m_2} \rangle \in \mathtt{ME} \Leftrightarrow \langle \mathtt{m_2}, \mathtt{r_2}, \mathtt{m_1} \rangle \in \mathtt{ME} \end{aligned}$$

Le théorème 11 montre[18] que toutes les références opposites dans un modèle restent opposites dans le modèle après substitution. Donc, la propriété `isOpposite` est préservée.

Théorème 11. (SubstIsOppositePreserved)
$\forall\, \mathtt{M} \in Model,\ \mathtt{r_1}\ \mathtt{r_2} \in References,\ \mathtt{o_1}\ \mathtt{o_2} \in Objects,$
$(isOpposite\ \mathtt{r_1}\ \mathtt{r_2}\ \mathtt{M}) \rightarrow (isOpposite\ \mathtt{r_1}\ \mathtt{r_2}\ (Substitution\ \mathtt{o_1}\ \mathtt{o_2}\ \mathtt{M})).$

[18] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Subst_Verif.html#SubstIOP

La préservation de la propriété `isOpposite` est décrite à l'aide du théorème 12. Le théorème montre[19] que toutes les relations qui satisfont cette propriété dans les deux modèles satisfont aussi la propriété dans l'union de modèles.

Théorème 12. (isOppositeUnionPreserved) $\forall\, \mathtt{M}_1\, \mathtt{M}_2 \in \mathtt{Model},\ r_1\, r_2 \in \mathtt{References}$, $(isOpposite\ r_1\ r_2\ \mathtt{M}_1) \wedge (isOpposite\ r_1\ r_2\ \mathtt{M}_2) \rightarrow (isOpposite\ r_1\ r_2\ (Union\ \mathtt{M}_1\ \mathtt{M}_2)))$.

Il n'est donc pas nécessaire d'ajouter des préconditions supplémentaires sur les opérateurs de composition pour que la vérification de la relation `isOpposite` soit compositionnelle.

3.4.7 areComposite

Une référence peut être *composite*. Une instance d'un concept ne peut être cible au plus que d'une instance d'une référence composite. La formalisation de cette propriété nécessite de considérer l'ensemble R des références composites qui est donné par le prédicat `areComposite` pour les objets instances d'une classe c_1.

$$areComposite(\mathtt{c}_1 \in \mathtt{MMV}, R \subseteq \mathtt{MME}, \langle \mathtt{MV},\mathtt{ME}\rangle) \triangleq$$
$$\forall\, o \in \mathtt{Objects}, |\{\mathtt{m}_1 \in \mathtt{MV} \mid \langle \mathtt{m}_1,\mathtt{r},\langle \mathtt{o},\mathtt{c}_1\rangle\rangle \in \mathtt{ME}, \mathtt{r} \in R\}| \leq 1$$

Le théorème 13 montre[20] que toutes les références composites dans un modèle sont aussi composites dans ce modèle après substitution. Ce théorème suppose que la substitution est injective et exige que le modèle substitué ne contienne pas un élément dont le nom est o_2.

Théorème 13. (SubstAreCompositeSubsPreserved)
$\forall\, \langle \mathtt{MV},\mathtt{ME}\rangle \in Model,\ \mathtt{c} \in Classes,\ \mathtt{R} \subset References,\ \mathtt{r} \in \mathtt{R},\ \mathtt{o}_1\, \mathtt{o}_2 \in Objects,\ \mathtt{c}_1\, \mathtt{c}_2 \in Classes,$
$\mathtt{c}_1 = \mathtt{c}_2 \wedge ((o_2, c) \notin \mathtt{MV}) \wedge (\mathtt{Injectif}\ Substitution)$
$\wedge\, (areComposite\ \mathtt{c}\ \mathtt{r}\ \langle \mathtt{MV},\mathtt{ME}\rangle)$
$\rightarrow (areComposite\ \mathtt{c}\ \mathtt{r}\ (Substitution\ (\mathtt{o}_1,\mathtt{c}_1)\ (\mathtt{o}_2,\mathtt{c}_2)\ \langle \mathtt{MV},\mathtt{ME}\rangle))$

Le théorème 14 prouve[21] que la propriété `areComposite` est préservée par l'opérateur `Union`. Une condition similaire à `upperCond` 3.4.5 en remplaçant n par la valeur 1 est nécessaire pour vérifier cette propriété.

$$areCompositeCond(\mathtt{c},\ r,\ \langle \mathtt{MV}_1,\mathtt{ME}_1\rangle\ \langle \mathtt{MV}_2,\mathtt{ME}_2\rangle) \triangleq$$
$$1 > |\{\mathtt{o}_2 \in \mathtt{MV}_1 \mid \langle\langle \mathtt{o},\mathtt{c}\rangle,\mathtt{r},\mathtt{o}_2\rangle \in \mathtt{ME}_1\}| + |\{\mathtt{o}_2 \in \mathtt{MV}_2 \mid \langle\langle \mathtt{o},\mathtt{c}\rangle,\mathtt{r},\mathtt{o}_2\rangle \in \mathtt{ME}_2\}|$$
$$-|\{\mathtt{o}_2 \in (\mathtt{MV}_1 \cap \mathtt{MV}_2) \mid \langle\langle \mathtt{o},\mathtt{c}\rangle,\mathtt{r},\mathtt{o}_2\rangle \in (\mathtt{ME}_1 \cap \mathtt{ME}_2)\}|$$

La condition signifie que pour une instance d'une référence composite, le concept cible appartient à une seule instance du concept source qui doit être la même dans les deux modèles.

Théorème 14. (areCompositeUnionPreserved)
$\forall\, \mathtt{M}_1\, \mathtt{M}_2 \in \mathtt{Model},\ c \in \mathtt{Classes},\ r \subset \mathtt{References},$
$(areCompositeCond\ c\ r\ n\ \langle \mathtt{MV}_1,\mathtt{ME}_1\rangle\ \langle \mathtt{MV}_2,\mathtt{ME}_2\rangle)$
$\wedge (areComposite\ c\ r\ \mathtt{M}_1) \wedge (areComposite\ c\ r\ \mathtt{M}_2)$
$\rightarrow (areComposite\ c\ r\ (Union\ \mathtt{M}_1\ \mathtt{M}_2))$.

[19] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Union_Verif.html#IOECP
[20] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Subst_Verif.html#SubstACP
[21] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Union_Verif.html#ACECP

Il est donc nécessaire d'ajouter des préconditions supplémentaires sur les opérateurs de compositions pour que la vérification de la relation `areComposite` soit compositionnelle.

Nous avons explicité notre démarche pour la vérification compositionnelle de propriétés de modèles. Nous discutons brièvement dans la section suivante de la vérification compositionnelle et quelques travaux autour de cette problématique.

3.5 La véri cation compositionnelle

Pour développer des systèmes critiques sûrs, des méthodes sont nécessaires qui permettent non seulement la réutilisation des composants mais aussi de leurs propriétés pour assurer les propriétés globales du système à partir des propriétés de ses composants. Nguyen a proposé dans [BBNS10] une approche de vérification compositionnelle pour les propriétés des systèmes décrits dans le langage BIP (Behavior - Interaction - Priority) [BBS06]. Une autre approche assurant la vérification de systèmes par composition de composants déjà vérifiés a été proposée dans [XB03]. Cette approche réduit la complexité de la vérification des systèmes basés sur des composants en utilisant leur structure compositionnelle. Dans cette approche, les propriétés temporelles des composants logiciels sont spécifiées, vérifiées et englobées avec le composant. La sélection d'un composant pour la réutilisation considère aussi ses propriétés temporelles. Le projet Ptolemy[22] [LJ99] propose une théorie compositionnelle pour les systèmes concurrents, temps-réel et embarqués. Il utilise des modèles de calcul bien définis et un atelier mathématique unifié pour relier plusieurs modèles de calcul en exploitant une sémantique dénotationnelle vers le Tagged Signal Model [LSV96].

Tous les ateliers cités sont spécifiques à des modèles particuliers. Dans notre travail, nous avons adopté une méthode de composition générique basée sur des opérateurs élémentaires qui préservent les propriétés du métamodèle. Pour chaque forme de vérification, nous définissons pour chaque opérateur de composition les préconditions nécessaires pour prouver ensuite que la vérification est compositionnelle.

Liu et al dans [LYX09] modélisent les patrons de conception comme une liste de propositions, au lieu d'une conjonction de propositions. La composition est alors décrite comme la concaténation des listes. Le problème de la vérification de la correction de la composition peut être résolu en vérifiant certaines propriétés sur ces listes. à partir de ce principe, les théorèmes sont formalisés et prouvés en Coq. La propriété vérifiée est que aucun composant ne peut perdre une propriété après la composition et aucune nouvelle propriété n'est introduite après composition.

Notre approche est plus large, elle permet d'étudier toute forme de propriété que nous pouvons exprimer comme un prédicat sur un modèle. Elle permet ainsi la vérification de la propriété de conformité pour des opérateurs de composition appliqués d'une manière similaire à des modèles et des métamodèles comme des modèles conformes à EMOF.

[22] http://ptolemy.eecs.berkeley.edu/

3.6 Conclusion

En partant du « framework » COQ4MDE, nous avons défini des opérateurs élémentaires de composition pour lesquels nous avons réussi à prouver la préservation du bon typage et des propriétés sémantiques en relation avec le métamétamodèle EMOF en ajoutant certaines préconditions supplémentaires pour les opérateurs de composition dans le cas de la vérification des relations `lower`, `upper` et `areComposite`. L'utilisation des propriétés en relation avec le métamétamodèle EMOF permet d'exploiter les éléments de vérification pour chaque langage qui peut se décrire comme un modèle conforme à un métamodèle. Les opérateurs sont aussi spécifiés d'une manière indépendante du métamodèle. Tous les théorèmes et les notions utilisés pour les définir sont complètement codés dans l'assistant de preuve COQ. Les opérateurs vérifiés peuvent être utilisés pour formaliser des opérateurs de plus haut niveau qui doivent aussi préserver les propriétés déjà prouvées.

Cette notion de vérification pour les opérateurs peut être définie d'une manière abstraite paramétrée par des pré/postconditions sur les modèles, ceci permet de formaliser les activités de vérification complètement compositionnelle et celles qui nécessitent des vérifications supplémentaires. Pour vérifier une propriété Φ sur un modèle, la postcondition de l'opération de composition doit être Φ, les préconditions supplémentaires minimales sont Φ sur chaque composant renforcée éventuellement par une précondition supplémentaire ψ définie sur tous les composants paramètres de l'opérateur. Le tableau 3.1 résume les pré et postconditions pour chaque opérateur étudié dans ce chapitre. Dans le tableau 3.1 : $M = \langle \mathtt{MV}, \mathtt{ME} \rangle$ et $M_i = \langle \mathtt{MV_i}, \mathtt{ME_i} \rangle$.

Nous allons dans les chapitres suivants étudier des opérateurs plus abstraits que nous décrirons à partir des opérateurs élémentaires pour ré-utiliser les preuves réalisées. Dans une première étape, nous traitons de l'opérateur de fusion de paquetage (Package Merge) défini par l'OMG pour UML et EMOF.

La version COQ4MDE dont on n'est partie avant de commencer cette thèse comporte environ 1107 lignes de code la version actuelle comporte environ 15000 avec 300 Lemmes et théorèmes et 210 définitions.

	M_r=Substitution $((\mathtt{o_1}, \mathtt{c_1}), (\mathtt{o_2}, \mathtt{c_2}), \mathrm{M})$	M_r=Union (M_1, M_2)
instanceOf	$\psi(M)$=True $\Phi(M_r) = instanceOf(M, MM)$	$\psi(M_{i\in\{1,2\}})$=True $\Phi(M_r) = instanceOf(M_r, MM)$
subClass	$\psi(M)$=True $\Phi(M_r) = subClass\ \mathtt{c_1}\ \mathtt{c_2}\ \mathtt{M}$	$\psi(M_{i\in\{1,2\}})$=True $\Phi(M_r) = subClass\ \mathtt{c_1}\ \mathtt{c_2}\ \mathtt{M_r}$
isAbstract	$\psi(M)$=True $\Phi(M_r) = isAbstract(c, M)$	$\psi(M_{i\in\{1,2\}})$=True $\Phi(M_r) = isAbstract(c, M_r)$
lower	$\psi(M) = (\mathtt{c_1} = \mathtt{c_2}) \wedge ((o_2, c) \notin \mathtt{MV})$ $\wedge$*Injectif Substitition* $\Phi(M_r) = lower(c, r, n, M_r)$	$\psi(M)$=$\mathtt{n} \geq$ $\|\{\mathtt{o_2} \in (\mathtt{MV_1} \cap \mathtt{MV_2})$ $\| \langle\langle \mathtt{o}, \mathtt{c}\rangle, \mathtt{r}, \mathtt{o_2}\rangle \in (\mathtt{ME_1} \cap \mathtt{ME_2})\}\|$ $\Phi(M_r) = lower(c, r, n, M_r)$
upper	$\psi(M) = (\mathtt{c_1} = \mathtt{c_2}) \wedge ((o_2, c) \notin \mathtt{MV})$ $\wedge$*Injectif Substitition* $\Phi(M_r) = upper(c, r, n, M_r)$	$\psi(M)$=$\mathtt{n} >$ $\|\{\mathtt{o_2} \in \mathtt{MV_1} \mid \langle\langle \mathtt{o}, \mathtt{c}\rangle, \mathtt{r}, \mathtt{o_2}\rangle \in \mathtt{ME_1}\}\|$ $+\|\{\mathtt{o_2} \in \mathtt{MV_2} \mid \langle\langle \mathtt{o}, \mathtt{c}\rangle, \mathtt{r}, \mathtt{o_2}\rangle \in \mathtt{ME_2}\}\|$ $-\|\{\mathtt{o_2} \in (\mathtt{MV_1} \cap \mathtt{MV_2}) \mid$ $\langle\langle \mathtt{o}, \mathtt{c}\rangle, \mathtt{r}, \mathtt{o_2}\rangle \in (\mathtt{ME_1} \cap \mathtt{ME_2})\}\|$ $\Phi(M_r) = upper(\mathtt{c},\ r,\ \mathtt{n},\ M_r))$
isOpposite	$\psi(M)$=True $\Phi(M_r) = isOpposite(r_1, r_2, M_r)$	$\psi(M_{i\in\{1,2\}})$=True $\Phi(M_r) = isOpposite(r_1, r_2, M_r)$
areComposite	$\psi(M) = (\mathtt{c_1} = \mathtt{c_2}) \wedge ((o_2, c) \notin \mathtt{MV})$ $\wedge$*Injectif Substitition* $\Phi(M_r) = areComposite(c, r, M_r)$	$\psi(M) = 1 >$ $\|\{\mathtt{o_2} \in \mathtt{MV_1} \mid \langle\langle \mathtt{o}, \mathtt{c}\rangle, \mathtt{r}, \mathtt{o_2}\rangle \in \mathtt{ME_1}\}\|$ $+\|\{\mathtt{o_2} \in \mathtt{MV_2} \mid \langle\langle \mathtt{o}, \mathtt{c}\rangle, \mathtt{r}, \mathtt{o_2}\rangle \in \mathtt{ME_2}\}\|$ $-\|\{\mathtt{o_2} \in (\mathtt{MV_1} \cap \mathtt{MV_2}) \mid$ $\langle\langle \mathtt{o}, \mathtt{c}\rangle, \mathtt{r}, \mathtt{o_2}\rangle \in (\mathtt{ME_1} \cap \mathtt{ME_2})\}\|$ $\Phi(M_r) = areComposite(\mathtt{c},\ r,\ M_r)$

***Tableau 3.1* —** pré et postconditions pour les opérateurs Union et Substitution

4 Formalisation d'une composition (Package Merge)

Table des matières

4.1 La sémantique du Package Merge 75
4.1.1 Les règles de filtrage 78
4.1.2 Les contraintes 78
4.1.3 Les transformations 79
4.2 Un exemple pour la résolution de con its 81
4.3 Améliorations 84
4.4 Travaux connexes de formalisation pour Package Merge 84
4.5 Conclusion 85

LE but de ce chapitre est de présenter un exemple d'utilisation des opérateurs étudiés dans le chapitre 3 (`Union` et `Substitution`) pour la formalisation d'un opérateur de plus haut niveau : la fusion des paquetages d'`UML` et `MOF` (`Package Merge`).

Remarque 4.1. *Dans ce chapitre, on représente les métamodèles comme des modèles conformes à* `EMOF`*.*

4.1 La sémantique du Package Merge

Un paquetage peut être défini comme un groupement d'éléments et une manière de structurer les diagrammes. Package Merge est une relation dirigée entre deux paquetages. Elle indique que les contenus des deux paquetages doivent être composés par fusion des éléments de même nom. Elle peut être vue comme une opération qui prend le contenu de deux paquetages et produit un nouveau paquetage qui fusionne le contenu des paquetages intervenants [OMG11b]. C'est une opération très similaire à la généralisation dans le sens ou l'élément de base ajoute d'une manière conceptuelle les caractéristiques du paquetage cible

à ses caractéristiques ce qui donne lieu à un paquetage résultat qui combine les caractéristiques des deux paquetages [OMG07].

La procédure `merge` est simple : tous les éléments (par exemple : les classes, les associations et les opérations) dans le modèle cible qui n'ont pas un correspondant avec le même nom dans le modèle source sont simplement copiés dans le paquetage source. Si deux éléments ont le même nom, ils sont agrégés pour construire un seul élément et leurs caractéristiques sont combinées récursivement [Zit06]. Le mécanisme (`merge`) doit être utilisé lorsque les éléments définis dans plusieurs paquetages ont le même nom et sont censés représenter le même concept. Ceci est souvent utilisé pour donner plusieurs définitions d'un même concept pour différents objectifs à partir d'une même définition de base. Cela permet d'exprimer des vues/points de vue par exemple. Un concept de base est étendu par incrément, chaque incrément défini dans un paquetage est ensuite fusionné séparément. En sélectionnant quels incréments fusionner, il est possible d'obtenir une définition paramétrée d'un concept particulier dans un but spécifique. `Package Merge` est particulièrement utile dans le domaine de la méta-modélisation. Par exemple, il est largement utilisé pour la définition du métamodèle d'`UML` et spécialement pour la définition des niveaux de conformité.

La terminologie utilisée est basée sur une vue conceptuelle du `Package Merge` présentée par le diagramme de la figure 4.1 tirée de la spécification `UML` [OMG11b].

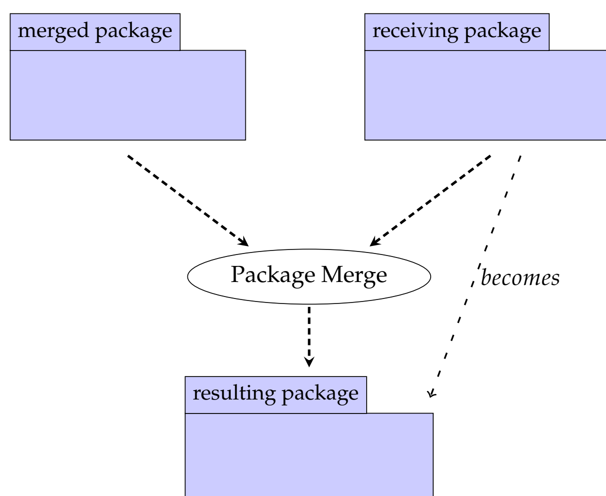


Figure 4.1 — Une vue conceptuelle de la sémantique du Package Merge

Une application du `Package Merge` sur deux paquetages implique un ensemble de transformations qui décrivent comment le contenu du paquetage `merged` doit être fusionné avec le paquetage `receiving` pour produire le paquetage `resulting`. Dans le cas où certains éléments dans les deux paquetages représentent une même entité, leurs contenus sont fusionnés d'une manière conceptuelle dans une seule entité résultante en relation avec les règles formelles précisées par la suite [OMG11b].

Pour comprendre les règles de construction du `Package Merge`, il est nécessaire de faire clairement la distinction entre les trois entités distinctes présentées dans l'ordre sur la figure 4.1 (les paquetages `merged`, `receiving` et `resulting`). Nous appuyant sur un

exemple d'une fusion simple sans con it.

Le paquetage `merged` est le premier intervenant dans l'opération de fusion, c'est le paquetage qui doit être fusionné dans le paquetage `receiving`. Considérons le modèle sur la figure 4.2 comme le paquetage `merged`.

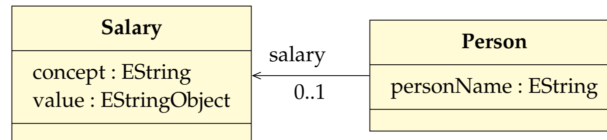


Figure 4.2 — Le paquetage `merged`

Le paquetage `receiving` est le deuxième intervenant dans la fusion, qui est conceptuellement le paquetage qui va contenir le résultat de la fusion. Supposons que le modèle sur la figure 4.3 soit le paquetage `receiving`.

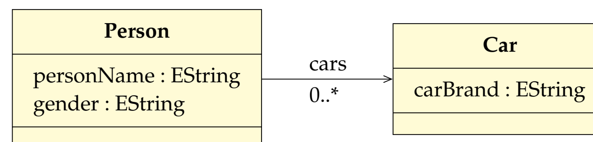


Figure 4.3 — Le paquetage `receiving`

Le paquetage `resulting` est le paquetage qui contient conceptuellement le résultat de la fusion. Le paquetage `receiving` est exactement le même que le paquetage `receiving`, mais ce terme particulier est utilisé pour se référer au paquetage et son contenu après l'application de la fusion. Le paquetage `resulting` à partir de la fusion des deux modèles précédents est présenté sur la figure 4.4.

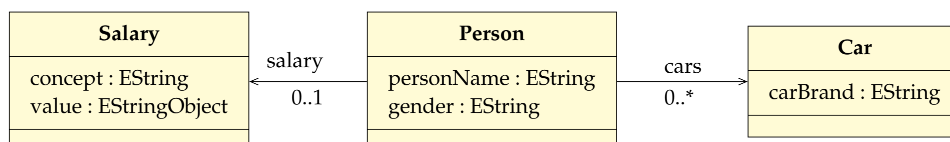


Figure 4.4 — Le paquetage `resulting`

La sémantique du `Package Merge` est définie à l'aide d'un ensemble de contraintes et de transformations. Les contraintes spécifient les pré-conditions pour une fusion valide. Les transformations spécifient les conséquences de sa sémantique (c-à-d. les post-conditions). Si une des contraintes est violée, le `Package Merge` est mal formé et le modèle `resulting` est invalide. Les différents metatypes (types `EMOF`) ont différentes sémantiques, mais le principe général est toujours le même : l'élément `resulting` doit être substituable avec chacun des éléments fusionnés. Cela signifie par exemple que les multiplicités d'un élément

dans le modèle `receiving` ne doivent pas être réduites dans le résultat de la fusion. Une des conséquences principales est que les éléments dans le paquetage `resulting` sont des extensions compatibles (ne changent pas les propriétés sémantiques) des éléments correspondants dans les deux paquetages. Cette propriété est intéressante dans la définition des niveaux de conformité pour que chaque niveau soit compatible avec le niveau précédent.

Dans la spécification, les transformations sont explicitement définies pour des metatypes généraux qui sont récurrents dans les langages de métamodélisation (`Packages`, `Classes`, `Associations`, `Properties`, etc). Les sémantiques de la fusion pour d'autres metatypes (comme state machines et interactions) sont complexes et spécifiques à leurs domaines. Les éléments de tous les autres metatypes sont transformés selon une règle générale : ils sont simplement copiés profondément (avec leurs contenus) dans le paquetage `resulting`.

Zito dans [Zit06] distingue trois groupes de règles : filtrage (match), contraintes et transformations. Nous expliquons dans les sections suivantes comment nous pouvons implémenter ces règles à l'aide des opérateurs élémentaires. La démarche peut être généralisée par la considération de prédicats `matches` plus génériques.

4.1.1 Les règles de ltrage

Celles-ci définissent l'égalité entre éléments des paquetages `merged` et `receiving` ; si deux éléments sont en correspondance, ils sont considérés comme étant le même élément, et sont combinés pour former un seul élément dans le paquetage `resulting` : par exemple la classe `Job` dans le paquetage `BasicEmployee` sur la figure 4.7 correspond à la classe `Job` dans le paquetage `EmployeeLocation` sur la figure 4.8, ces deux classes sont combinées récursivement pour produire la classe `Job` dans le modèle résultat. Les éléments dans le paquetage `merged` qui n'ont pas de correspondants dans le paquetage `receiving` sont simplement copiés.

Les modèles sont décrits dans COQ4MDE comme des structures plates où les éléments du modèle sont des objets typés du même niveau (classes, attributs de classes, etc). Ce premier type de filtrage est assuré par l'opérateur `Union` de modèles décrit dans le chapitre 3. Ceci permet d'avoir dans son modèle résultat une seule copie avec le contenu des deux modèles (attributs, opérations, etc) pour tout élément existant dans les deux modèles. Le résultat de l'`Union` contient aussi tous les éléments existants uniquement dans un des deux modèles.

4.1.2 Les contraintes

Celles-ci définissent les pré-conditions de la fusion ; si une paire de paquetages (`receiving`/`merged`) viole une des contraintes, la fusion est dite invalide. En général, la fusion est invalide seulement si les éléments correspondants dans les modèles `receiving` et `merged` sont fondamentalement incompatibles. Le fait que les attributs ont des types complètement différents peut montrer que même s'ils avaient le même nom, ils ne représentent pas le même concept. Nous avons choisi de résoudre ce con it en renommant les attributs en utilisant l'opérateur de substitution où un élément peut être remplacé par un autre s'ils sont du même type.

Nous présentons dans ce chapitre un exemple de déroulement étape par étape du `Package Merge`. Dans le cas d'une exécution automatique, des vérifications complémentaires sont nécessaires. Celles-ci imposent par exemple que d'autres aspects des deux modèles soient identiques notamment les attributs *areComposite* doivent être les mêmes.

4.1.3 Les transformations

Celles-ci définissent les post-conditions de la fusion ; elles décrivent comment résoudre n'importe quel con it quand les deux éléments correspondants sont récursivement fusionnés (par exemple : si deux classes ont des attributs différents, ou deux attributs correspondants ont des multiplicités différentes).

Dans certains cas de con its (par exemple, le cas où deux classes correspondantes ont des attributs différents), le con it est résolu en combinant les caractéristiques des éléments des modèles `merged` et `receiving` ceci est fait automatiquement à l'aide de l'opérateur `Union`. Pour les autres con its (par exemple, deux attributs correspondants ont des multiplicités différentes), les con its sont résolus en combinant les valeurs d'une certaine manière en utilisant l'opérateur `Substitution`. Il n'y a pas de règle générique pour la résolution d'un con it. Chaque règle de transformation dépend de la sémantique de la propriété.

Pour deux métamodèles conformes à `EMOF`, des con its peuvent être rencontrés au moment de la fusion. Nous montrons des exemples de con its et comment les contraintes de l'opérateur `Union` assurent que la fusion est valide. Une partie du métamétamodèle `EMOF` dans la notation graphique est présentée sur la figure 4.5.

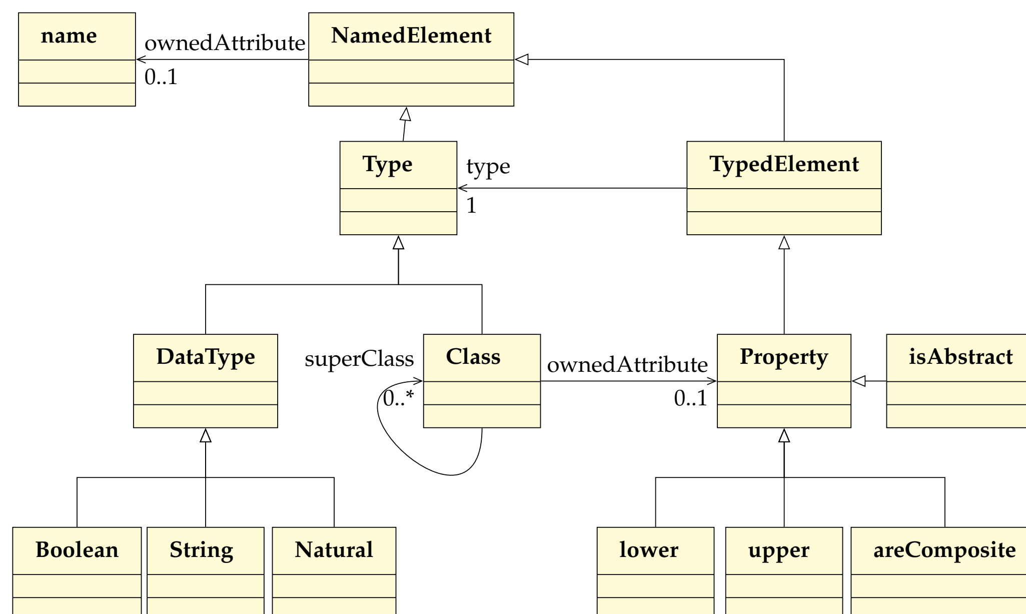


Figure 4.5 — Une partie du métamétamodèle EMOF (notation COQ4MDE)

Les trois métamodèles `merged`, `receiving` et `resulting` présentés sur les figure 4.3, 4.2 et 4.4 sont conformes au métamétamodèle `EMOF` dans le sens ou ils satisfont toutes les contraintes de ce dernier. Un premier exemple de con it est la fusion des deux

modèles présentés sur la figure 4.6. La propriété `upper` (le cardinal maximal) de l'attribut `isAbstract` est égal à 1 comme montré sur la partie du métamétamodèle `EMOF` sur la figure 4.5. En relation avec les contraintes de l'opérateur `Union` présentées dans 3.3.1, la valeur 1 correspond à `n` et doit être supérieure à la somme des cardinaux de l'attribut `isAbstract` dans les deux modèles fusionnés réduite de son cardinal dans l'intersection des deux modèles. Dans cet exemple le cardinal de l'attribut `IsAbstract` pour la classe `c` dans M_{merged} est égal à 1, le cardinal de l'attribut `IsAbstract` dans la classe `c` dans $M_{receiving}$ est aussi égal à 1. Le cardinal dans l'intersection des deux modèles est 0. La somme des deux cardinaux est égal à 2 qui est strictement supérieur à 1. Les préconditions de l'opérateur `Union` ne sont pas satisfaites et la fusion dans ce cas est invalide (le paquetage généré n'est pas une instance du métamodèle `EMOF`). Les deux modèles `merged` et `receiving` présentés sur la figure 4.6 sont conformes au métamétamodèle `EMOF` mais leurs `Union` n'est pas conforme à ce dernier.

Une implémentation d'un opérateur `Union` au niveau métamodèle est proposée et une sémantique formelle du `Package Merge` est exprimée à ce niveau. Une preuve d'équivalence sémantique est fournie[1] dans le cas de l'absence de con its entre les deux métamodèles. La spécification et un exemple sont accessibles dans l'annexe D.

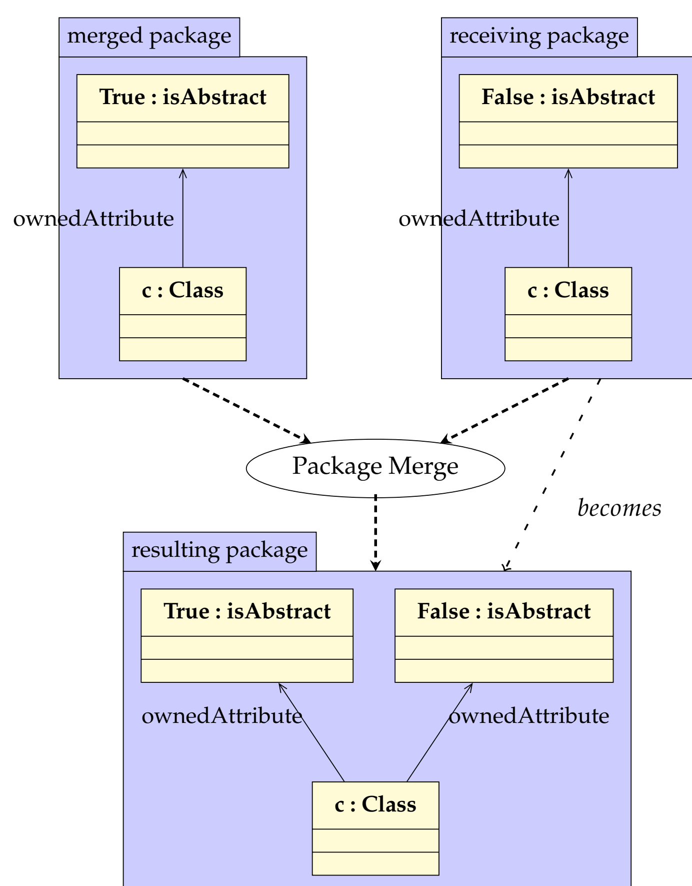


Figure 4.6 — Un exemple d'une composition non valide

[1] `http://www.irit.fr/~Mounira.Kezadri/FormalMDE/PackageMerge.html`

La résolution de ce type de conflits est possible. La règle pour résoudre un conflit sur l'attribut `isAbstract` d'une classe est :

$$isAbstract_{Resulting} = isAbstract_{Merged} \wedge isAbstract_{Receiving}$$

La liste de toutes les transformations est accessible sur la page 166 de la spécification [OMG11b].

Le résultat de la fusion des deux modèles `merged` et `receiving` dans la figure 4.6 est le modèle `receiving` selon la spécification. Dans ce qui suit, nous présentons un exemple de l'utilisation des opérateurs élémentaires pour implémenter l'opérateur `Package Merge` avec gestion de conflits.

4.2 Un exemple pour la résolution de conflits

On présente un exemple inspiré de [Zit06]. Le paquetage source (dans ce cas le paquetage `BasicEmployee` montré sur la figure 4.7) est le paquetage `receiving`. Le paquetage `EmployeeLocation` montré sur la figure 4.8 est le paquetage `merged`; c'est le paquetage qui contient les éléments additionnels qui sont fusionnés avec le paquetage `receiving`.

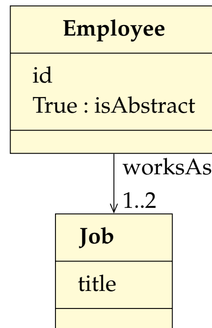


Figure 4.7 — Le métamodèle `BasicEmployee`

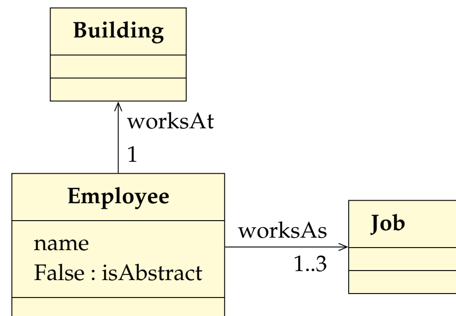


Figure 4.8 — Le métamodèle `EmployeeLocation`

La figure 4.9 montre la représentation `Coq4MDE` pour le paquetage `BasicEmployee`. Nous utilisons ici une notation indicée pour les attributs qui permet d'exprimer la dépendance entre les classes et les attributs et contourne le fait que nous ne présentons pas directement des classes attribuées dans `Coq4MDE`.

La figure 4.10 montre la représentation `Coq4MDE` pour le paquetage `EmployeeLocation`. Nous pouvons noter deux conflits entre les modèles `merged` et `receiving`. Le premier conflit concerne les attributs $upper$ de $worksAs$ (la borne maximale est égale à 2 dans le modèle `BasicEmployee` 4.9 et elle est égale à 3 dans le modèle `EmployeeLocation` 4.10). Le deuxième concerne la classe $Employee$ et les attributs $isAbstract$ dans les deux modèles (égal à True dans le modèle `BasicEmployee` 4.9 et égal à False dans le modèle `EmployeeLocation` 4.10). La règle pour résoudre le conflit à propos de l'attribut `upper` est :

$$upper_{Resulting} = max(upper_{Merged}, upper_{Receiving})$$

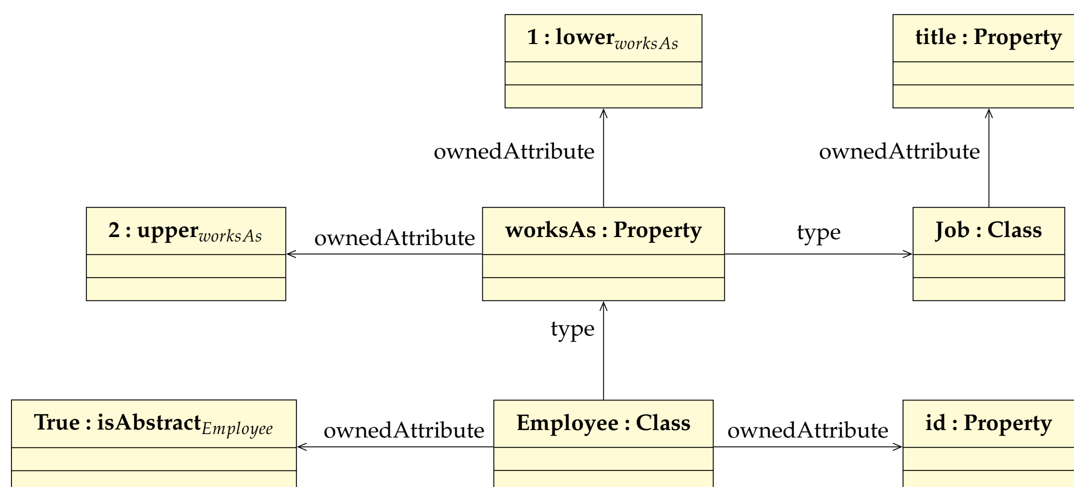


Figure 4.9 — Le paquetage `BasicEmployee` dans la notation `Coq4MDE` relativement à `EMOF`

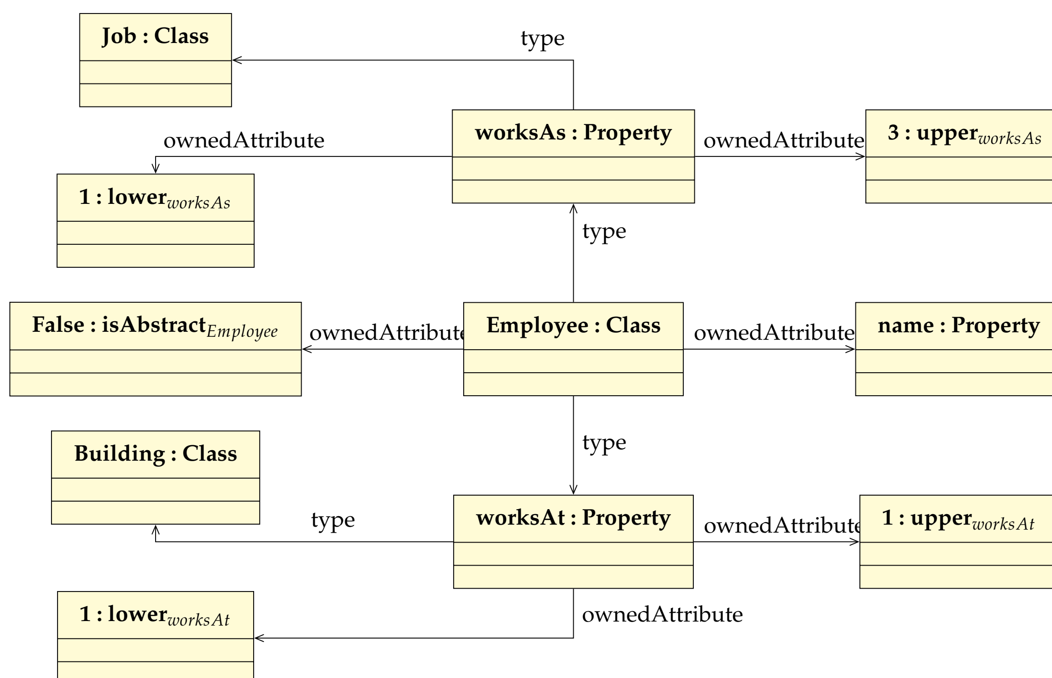


Figure 4.10 — Le paquetage `EmployeeLocation` dans la notation `Coq4MDE`

La première étape est la résolution de tous les con its. Pour ceci, l'opérateur de substitution est appliqué deux fois. La première remplace $(2 : upper_{worksAs})$ par $(3 : upper_{worksAs})$ dans le modèle `merged`. La seconde remplace $(True : isAbstract_{Employee})$ par $(False : isAbstract_{Employee})$. Le résultat de la substitution est montré sur la figure 4.11.

Une fois les con its résolus, l'étape finale est l'`Union` des modèles `merged` et `receiving` obtenus (les contraintes de l'opérateur `Union` sont satisfaites dans ce cas). Le résultat de l'union est présenté sur la figure 4.12. La syntaxe concrète correspondante au modèle résultat présenté sur la figure 4.12 est exactement la fusion des modèles `merged` et

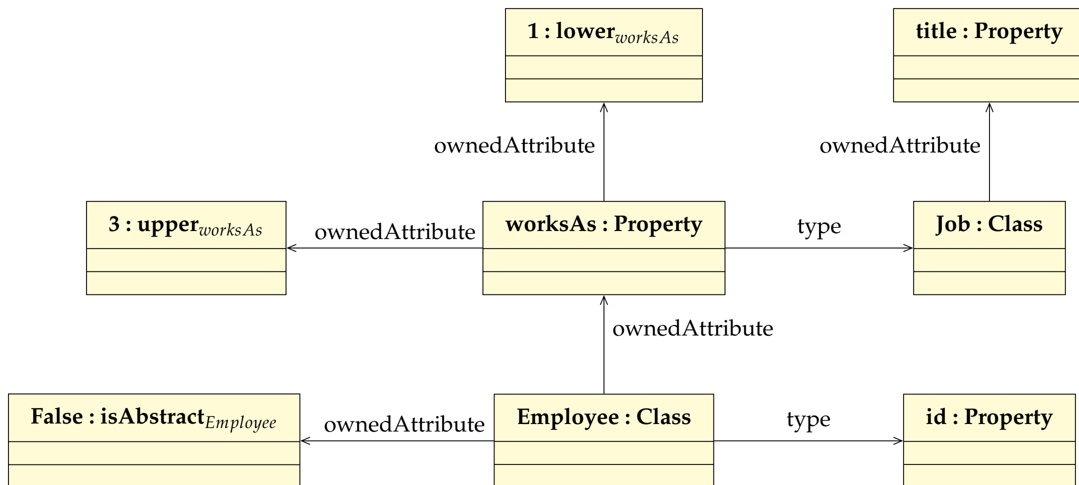


Figure 4.11 — La substitution du paquetage `merged`

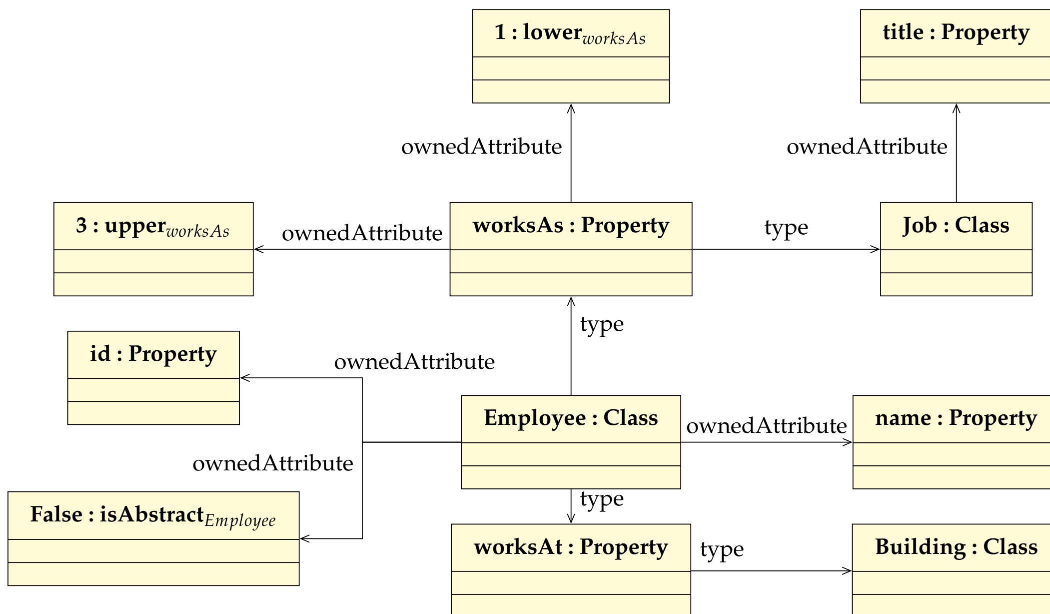


Figure 4.12 — Le paquetage résultant de l'union dans la notation `Coq4MDE`

`receiving` présentés sur la figure 4.13. Dans l'exemple précédent le `Package Merge` est exprimé en utilisant les opérateurs d'assemblage de base `Union` et `Substitution`. Définir le `Package Merge` de cette façon assure que le modèle résultat est bien typé par rapport aux paquetages `merged` et `receiving` et aussi qu'il satisfait les propriétés sémantiques du métamétamodèle.

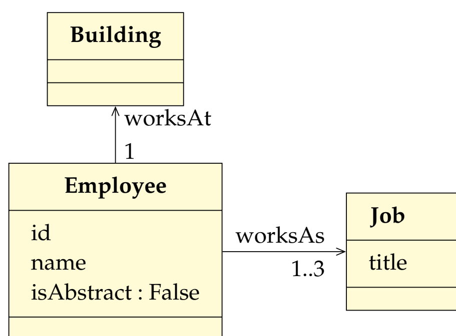


Figure 4.13 — Le métamodèle résultat

4.3 Améliorations

Dans l'exemple, nous manipulons une syntaxe concrète qui permet d'avoir des attributs spécifiques aux classes (le nom de la classe d'ou provient l'attribut est un suffixe dans le nom de l'attribut). Si nous imaginons utiliser des attributs non paramétrés, un problème se produit quand nous utilisons la `Substitution` pour la résolution des con its (par exemple si deux objets ont une même valeur pour l'attribut `lower`, vouloir remplacer cette valeur pour un objet va la remplacer obligatoirement pour les autres classes). En vue d'une formalisation complète une autre définition de l'opérateur `Substitution` est nécessaire (une version qui permet de remplacer seulement l'attribut concerné). La version de `Substitution` imaginée au lieu de remplacer un attribut par un autre, supprime le lien avec le premier (l'objet dans lequel nous voulons substituer l'attribut), ajoute le nouveau nœud s'il n'existe pas déjà et ajoute le lien avec le nouveau nœud. Nous pouvons soit montrer que cette version est équivalente à la première et donc a forcément les mêmes propriétés, soit re-vérifier les propriétés pour la nouvelle version. Un autre point important à traiter est l'ajout d'une sémantique permettant la comparaison des cardinaux qui sont pour le moment des noms d'éléments de modèles.

Notre but ici est de montrer par l'exemple en manipulant les métamodèles comme des modèles conformes à `EMOF` que nos opérateurs sont capables d'implémenter un opérateur tel que le `Package Merge`. Dans l'annexe D, nous présentons une implémentation du `Package Merge` à l'aide d'un opérateur union exprimé directement sur des métamodèles dans le cas d'absence de con its. L'implémentation complète avec gestion de con its et la preuve de l'équivalence avec la sémantique du `Package Merge` sont en cours et restent une perspective pour ce travail.

4.4 Travaux connexes de formalisation pour Package Merge

Zito dans [Zit06] présente une implémentation du `Package Merge` dans le langage de modélisation `Alloy`[2] [Jac12] et utilise l'analyseur `Alloy` pour effectuer des vérifications automatiques. `Alloy` est un langage formel de modélisation qui permet d'effectuer des analyses

[2]http ://alloy.mit.edu/alloy/

sur ses modèles. L'analyseur `Alloy` peut effectuer deux types d'analyse qui sont : la vérification de la conformité des instances par rapport au métamodèle et la vérification des assertions. Une assertion est une propriété sur le modèle qui est évaluée à vrai ou faux. Quand l'analyseur vérifie une propriété, soit il trouve que la propriété est satisfaite, soit il retourne un contre-exemple qui montre que l'assertion est fausse. L'analyseur vérifie l'assertion en énumérant d'une manière exhaustive toutes les instances possibles du modèle et en s'arrêtant quand un contre exemple est trouvé. Si l'ensemble des instances possibles est très grand (ou infini), l'analyse devient incomplète et ne permet donc pas de prouver la correction. Ce problème est résolu en demandant à l'utilisateur de préciser un intervalle pour l'analyse. `Alloy` a une performance faible quand nous analysons des modèles avec beaucoup de signatures, 20 signatures ou plus, l'analyse est limitée à un intervalle entre 5 et 10. Le métamodèle `UML` contient plus de 30 métaclasses (chacune est modélisée avec une signature). Les auteurs appliquent plusieurs stratégies pour réduire la taille du modèle `Alloy` : réduire la profondeur de l'héritage, fusionner les classes similaires, ne pas modéliser les classes qui ne comportent pas d'information et ne changent pas en appliquant la fusion (e.g. PackageImport, PackageMerge, ElementImport, Comment et Constraint), fusionner l'héritage multiple, éliminer la récursion et enfin ne pas modéliser les attributs et les associations dérivées.

L'intérêt de faire un développement prouvé en utilisant l'assistant de preuve COQ est de spécifier les propriétés d'une manière générique et de les vérifier pour toutes les instances a priori et d'avoir un opérateur correct par construction. Une des propriétés que nous pouvons vérifier est le fait qu'une instance est valide par rapport à un modèle particulier. Les assertions peuvent être aussi exprimées et vérifiées.

4.5 Conclusion

Le `Package Merge` est décrit d'une manière semi-formelle dans la spécification `UML`. C'est un opérateur très important dans le monde de la métamodélisation pour la construction de métamodèles à partir de métamodèles existants. Il est donc nécessaire qu'il soit complètement formalisé.

Nous avons montré que l'opérateur `Union` est équivalent au `Package Merge` dans le cas d'absence de con its entre les métamodèles. L'opérateur `Union` préserve les propriétés par rapport au métamétamodèle `EMOF` comme prouvé dans le chapitre 3. Nous avons proposé par la suite une façon pour résoudre les con its qui peuvent exister entre deux métamodèles en utilisant l'opérateur `Substitution` ce qui permet par la suite d'appliquer l'`Union` sur des modèles sans con its. La formalisation complète de cette deuxième version en COQ reste une perspective mais les blocs de base pour sa réalisation sont disponibles.

L'implémentation complète nécessite beaucoup de travail mais peut se faire d'une manière naturelle en suivant les étapes suivantes :

- écrire toutes les règles de résolution de con it.
- ajouter une sémantique qui permet la comparaison des cardinaux (si un objet est un nombre qui représente un cardinal).

- ajouter une sémantique booléenne pour les objets ayant une propriété booléenne (par exemple : `isAbstract` et `isOpposite`).
- écrire une procédure de détection de con its.
- résoudre les con its en utilisant l'opérateur substitution en appliquant les règles définies de résolution de con its identifiées de la spécification du `Package Merge` [OMG11b]. Par contre, pour cette spécification, différentes règles peuvent être utilisées pour résoudre les con its en cas de variation sémantique. Zito dans [Zit06] relève quelques inconsistances : par exemple, la règle générale lorsque l'élément dans le modèle `merged` n'a pas de correspondant dans le le modèle `receiving` est de le directement copier dans le modèle `resulting`, le problème se pose quand l'élément est une structure qui contient des multiplicités, la copie peut causer des problèmes et la règle pour la gestion des multiplicités peut être aussi appliquée.
- appliquer l'union sur les modèles obtenus.

Le `Package Merge` est un opérateur spécifique à un langage de modélisation particulier. Pour valider notre proposition, nous avons choisi de l'appliquer à une technologie d'introduction de composants dans n'importe quel langage de modélisation défini à partir d'un métamodèle : l'Invasive Software Composition (ISC).

5 Formalisation d'une composition de modèles (ISC)

Table des matières

5.1 ISC et la formalisation de ses opérateurs . . . 88
5.1.1 L'extension du métamodèle . . . 89
5.1.2 L'extraction et l'élimination de l'interface . . . 90
5.1.3 L'assemblage de composants . . . 92
5.1.3.1 L'opérateur bind . . . 92
5.1.3.2 L'opérateur extend . . . 93
5.2 Un exemple détaillé . . . 95
5.3 La vérification des propriétés . . . 97
5.3.1 L'opérateur bind . . . 98
5.3.1.1 Une vue sur le langage de tactiques . . . 100
5.3.2 L'opérateur bind de deux modèles avec plusieurs points de variation 102
5.3.3 L'opérateur extend . . . 105
5.3.3.1 L'opérateur extend basé sur l'union disjointe . . . 105
5.3.3.2 L'opérateur extend basé sur l'union classique . . . 107
5.3.4 L'extraction de l'interface des fragments . . . 110
5.4 Conclusion . . . 112

L'utilisation des composants permet de favoriser la réutilisation dans le développement des systèmes, d'améliorer la productivité et de réduire les coûts en particulier pour les systèmes critiques qui demandent de nombreuses activités de V&V. Un langage de modélisation dédié (DSML) n'intègre pas forcément des composants, soit car ce n'était pas une exigence initiale de la définition du langage soit pour éviter une définition trop complexe. La composition logicielle peut être faite sur plusieurs niveaux (de la fusion des modèles à

la composition du code) et de plusieurs manières. Selon le modèle du composant, le mécanisme et le langage de la composition, plusieurs types de systèmes composés peuvent être générés. Nous nous intéressons dans ce chapitre à la méthode générique Invasive Software Composition (ISC) qui permet à la fois d'enrichir un métamodèle pour permettre la définition de composants de modèles avec leurs interfaces, d'opération d'extraction de composants depuis un modèle enrichi et enfin d'opérations d'assemblage des composants ainsi obtenus.

5.1 ISC et la formalisation de ses opérateurs

La méthode de composition ISC (Invasive Software Composition) a été proposée par Aßman [Aßm03] dans le but d'ajouter une structure de composant générique à n'importe quel DSML. Cette approche est à l'origine de l'outil REUSEWARE[1] disponible sous Eclipse. Celui-ci permet d'adapter et d'étendre un langage existant en ajoutant une interface de composition sous forme de points de variation (hooks). Cette extension applique une transformation au niveau du métamodèle basée sur la spécification des ports de composition dans le langage. Les points de variation sont les éléments du modèle dont la valeur peut changer et qui permettent de construire des composants. L'avantage principal de la méthode ISC est sa généricité : elle peut être appliquée à n'importe quel langage défini par un métamodèle. La méthode conjecture dans sa version originale que, si les deux modèles sont disjoints, le résultat de la composition des fragments (modèles avec une interface de composition) conformes à un certain métamodèle est aussi conforme au même métamodèle. Elle permet dans la version REUSEWARE développée en parallèle avec nos travaux de vérifier la compatibilité des modèles à travers une notion de compatibilité de points de variation et de référence mais celle-ci ne comporte ni une formalisation ni des preuves de propriétés comme nous les présentons. Cette conformité est une des propriétés définies et vérifiées en exploitant le cadre formel d'IDM présenté dans le chapitre précédent. Nos travaux ont donc permis de prouver la conjecture proposée par Aßman.

Cette approche a été introduite pour la première fois dans [Aßm03]. Son intérêt a été illustré sur une variété de problèmes de composition [Hen09]. ISC était initialement une des techniques qui permet la composition d'artefacts logiciels en se basant sur la grammaire de la syntaxe concrète du DSML. Les composants logiciels sont souvent exprimés à l'aide de langages formels, la composition de phrases de ces langages en se basant sur la grammaire semble naturelle. La méthode propose une technique de composition basée sur la réécriture du code source. Les composants dans un système ISC sont des fragments de code source qui peuvent contenir des points de variation. Les points de variation sont les places dans lesquelles d'autres fragments peuvent être insérés durant la composition. Les points de variations ISC sont typés en se basant sur la grammaire du langage utilisé pour écrire les fragments. Récemment, ces travaux ont été étendus pour gérer les langages définis par des métamodèles [Joh11]. Un métamodèle est essentiellement un graphe typé qui décrit la structure d'un langage. Un exemple est le métamodèle UML 2, un diagramme de classe UML est un exemple d'une expression dans le langage UML 2. Reuseware peut maintenant être ins-

[1] http://www.reuseware.org

tancié à des langages de modélisation arbitraires et utilisé en combinaison avec une variété d'outils dans la démarche MDSD (Model Driven Software Development), quand la génération du langage par métamodélisation est une partie du processus de développement.

Deux types de nœuds doivent être considérés durant la composition : les points de variation `hooks` et les points de référence `prototypes`. Les points de référence sont des nœuds racines de fragments de modèle et les points de variation sont les nœuds qui peuvent être remplacés durant la composition par des points de référence.

Dans REUSEWARE, les points de variation et de référence sont groupés sous forme de ports qui représentent des interfaces de composition. Concrètement, les compositions sont définies par des programmes de composition et les ports sont liés à travers des liens de composition. Les ports peuvent être connectés seulement si les points de variation et de référence contenus dans les ports correspondent (ceci est vrai quand il y a un nombre suffisant de points de référence pour remplacer les points de variation et quand leurs types sont conformes). Dans le cadre de ce travail, nous intéressons aux connexions élémentaires entre les points de variation et de référence.

5.1.1 L'extension du métamodèle

Une interface pour un composant se compose de points de connexion qui révèlent comment le composant peut être interconnecté avec son contexte de réutilisation, les éléments qui sont communiqués en provenance et à destination de l'environnement, et les relations avec le monde extérieur qui doivent être établies. Nous devons être en mesure d'étendre tout métamodèle pour supporter la définition d'une interface de composition. Cette extension ajoute la définition de l'interface d'un fragment constituée d'un ensemble de points de variation et de référence associés à des éléments du modèle. On note $\mathtt{MM}^{\mathtt{Ext}}$ le métamodèle étendu d'un certain métamodèle `MM`. On note `ROV` la classe abstraite représentant des points de variation appelés `Hooks` et des points de référence appelés `Prototypes` (qui sont des sous-classes de `ROV`).

Dans $\mathtt{MM}^{\mathtt{Ext}}$, chaque nœud dans le graphe représentant `MM` peut être référencé par un point d'interface. Pour cette raison, une classe abstraite nommé `AbstractClasses` est ajoutée comme super-classe pour toutes les classes de `MM`. Cette classe est liée par la référence `bind` avec `ROV`. Les trois classes `ROV`, `Hook` et `Prototype` sont aussi automatiquement importées dans le `métamodèle` avec les relations d'héritage appropriées entre elles. L'extension du métamodèle présentée dans [Joh11] est définie dans le troisième niveau de modélisation. L'extension définie ici utilise seulement le deuxième niveau de modélisation (le niveau métamodèle) qui est suffisant pour nos besoins.

Dé nition 4. *Soit* $\mathtt{MM} = \langle\langle \mathtt{MMV}, \mathtt{MME}\rangle, conformsTo\rangle$ *un métamodèle.*
Soit $\mathtt{ROV}, \mathtt{Hook}, \mathtt{Prototype}, \mathtt{AbstractClasses} \in Classes, \mathtt{bind} \in References.$

$\texttt{MM}^{\texttt{Ext}}$ *est dé ni comme* $\langle\langle \texttt{MMV}^{\texttt{Ext}}, \texttt{MME}^{\texttt{Ext}}\rangle, conformsTo^{Ext}\rangle$ *tel que :*

$$\begin{array}{l}
\texttt{MMV}^{\texttt{Ext}} = \texttt{MMV} \cup \{\texttt{ROV}, \texttt{Hook}, \texttt{Prototype}, \texttt{AbstractClasses}\} \\
\texttt{MME}^{\texttt{Ext}} = \texttt{MME} \cup \{\langle \texttt{ROV}, \texttt{bind}, \texttt{AbstractClasses}\rangle\} \cup \{\forall\ \texttt{c} \in \texttt{MMV}, \langle \texttt{c}, \texttt{inh}, \texttt{AbstractClasses}\rangle\} \\
conformsTo^{Ext}(\langle \texttt{MV}, \texttt{ME}\rangle) \triangleq conformsTo(\langle \texttt{MV}, \texttt{ME}\rangle) \\
\wedge\ isAbstract(\texttt{ROV}) \\
\wedge\ subClass(\texttt{Hook}, \texttt{ROV}) \\
\wedge\ subClass(\texttt{Prototype}, \texttt{ROV}) \\
\wedge\ isAbstract(\texttt{AbstractClasses}) \\
\wedge\ \forall\ \texttt{c} \in \texttt{MMV}, subClass(\texttt{c}, \texttt{AbstractClasses})
\end{array}$$

La figure 5.1 montre un exemple d'extension du métamodèle MM. La définition de

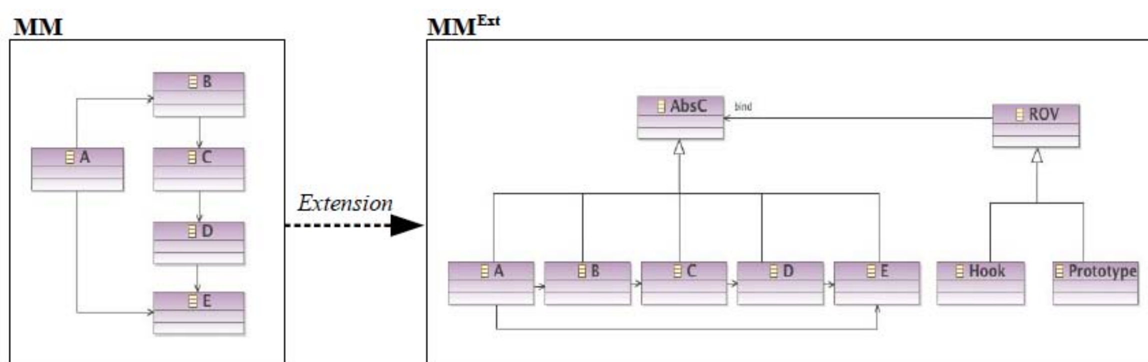


Figure 5.1 — L'extension du métamodèle

$\texttt{MM}^{\texttt{Ext}}$[2] ajoute une relation d'héritage entre chaque classe dans le métamodèle et la classe AbstractClasses[3]. Cela repose sur le parcours de tous les éléments du métamodèle. Une fonction $fold$ [4] définie au niveau métamodèle est utilisée pour ajouter une relation d'héritage entre toutes les classes du métamodèle et la classe AbstractClasses.

5.1.2 L'extraction et l'élimination de l'interface

Le but de la fonction FragmentExtraction est de construire un fragment à partir d'un modèle en définissant son interface de composition. Cette fonction prend comme paramètre : un modèle, l'objet référencé dans ce modèle et le type du point dans l'interface associé à cet objet. $FragmentExtraction\ :\ Model \times \texttt{Objects} \times \texttt{Classes} \rightarrow Model$ est défi-

[2] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/MM_Extension.html#MMExt
[3] Notée sur la figure 5.1 comme AbsC
[4] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/MM_Extension.html#foldClass

nie comme :

$$\begin{aligned}
&FragmentExtraction(\langle \mathtt{MV}, \mathtt{ME}\rangle, \mathtt{o}, \mathtt{HP}) = \langle \mathtt{MV}^{\mathtt{Ext}}, \mathtt{ME}^{\mathtt{Ext}}\rangle \\
&\textit{tel que } \mathtt{HP} \in \{\mathtt{Hook}, \mathtt{Prototype}\} \textit{ et } \exists \mathtt{c} \in \mathtt{Classes}, \langle \mathtt{o}, \mathtt{c}\rangle \in \mathtt{MV} \\
&\textit{tel que} : \\
&\mathtt{MV}^{\mathtt{Ext}} = \mathtt{MV} \cup \{\langle \mathtt{h}, \mathtt{HP}\rangle, \langle \mathtt{h}, \mathtt{ROV}\rangle, \langle \mathtt{o}, \mathtt{AbstractClasses}\rangle\} \\
&\mathtt{ME}^{\mathtt{Ext}} = \mathtt{ME} \cup \{\langle\langle \mathtt{o}, \mathtt{c}\rangle, \mathtt{inh}, \langle \mathtt{o}, \mathtt{AbstractClasses}\rangle\rangle, \\
&\langle\langle \mathtt{h}, \mathtt{ROV}\rangle, \mathtt{bind}, \langle \mathtt{o}, \mathtt{AbstractClasses}\rangle\rangle, \\
&\langle\langle \mathtt{h}, \mathtt{HP}\rangle, \mathtt{inh}, \langle \mathtt{h}, \mathtt{ROV}\rangle\rangle\}
\end{aligned}$$

Une autre version de *FragmentExtraction* peut être implémentée en spécifiant un ensemble de tuples $(\mathtt{o}, \mathtt{HP})$ pour ajouter plusieurs points simultanément.

`ElimInterface` élimine l'interface du fragment (tous les points de variation et de référence), c'est la fonction inverse de `FragmentExtraction` dans le cas où l'interface du fragment est constituée d'un seul point. Celle-ci est implémenté dans [Joh11] en utilisant l'opérateur *remove* qui est automatiquement appliqué après la composition pour rendre le modèle manipulable par des outils où les points de variation ne sont pas définis.
$ElimInterface : Model \rightarrow Model$, tel que :

$$\begin{aligned}
&ElimInterface\ \langle \mathtt{MV}^{\mathtt{Ext}}, \mathtt{ME}^{\mathtt{Ext}}\rangle = \langle \mathtt{MV}, \mathtt{ME}\rangle \\
&\textit{tel que} : \\
&\mathtt{MV} = \{\langle \mathtt{o}, \mathtt{c}\rangle \in \mathtt{MV}^{\mathtt{Ext}} | \mathtt{c} \notin \{\mathtt{Hook}, \mathtt{Prototype}, \mathtt{ROV}, \mathtt{AbstractClasses}\}\} \\
&\mathtt{ME} = \{\langle\langle \mathtt{o}, \mathtt{c}\rangle, \mathtt{r}, \langle \mathtt{o}', \mathtt{c}'\rangle\rangle \in \mathtt{ME}^{\mathtt{Ext}} | \mathtt{c}, \mathtt{c}' \notin \{\mathtt{Hook}, \mathtt{Prototype}, \mathtt{ROV}, \mathtt{AbstractClasses}\}\}
\end{aligned}$$

La figure 5.2 montre l'exemple de l'extraction de l'interface du modèle `M` en spécifiant l'élément `b` de type `B` comme un `hook` et l'opération inverse qui consiste à supprimer cette interface. Pour l'implémentation de *ElimInterface*, une fonction qui construit le graphe d'inter-

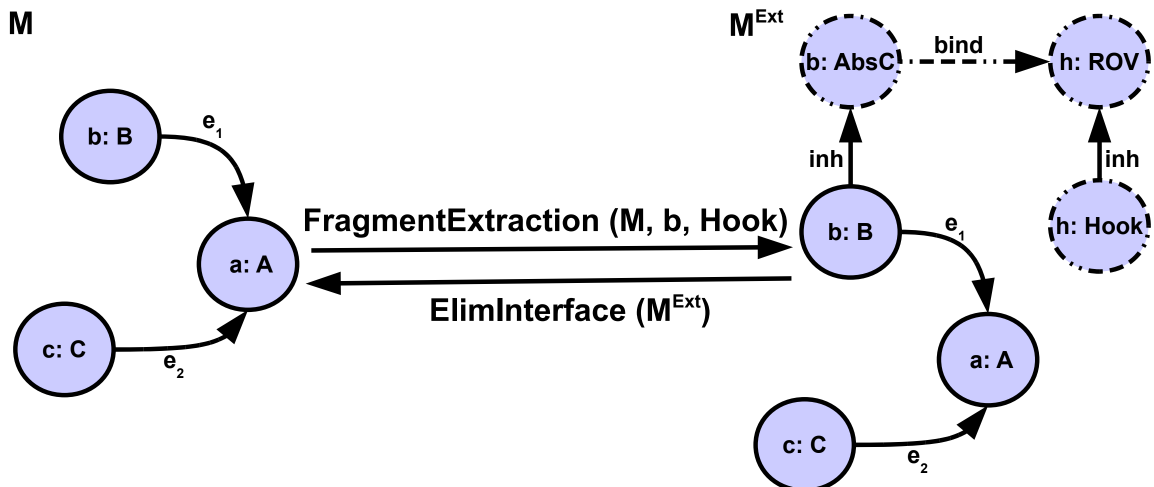


Figure 5.2 — L'extraction et l'élimination de l'interface de composition

section est utilisée. Le principe est d'éliminer les éléments par l'intersection avec un graphe contenant tous les éléments possibles sauf : `Hook`, `Prototype`, `ROV` et `AbstractClasses` et leurs liens avec les éléments de modèle dans le graphe initial. Le résultat est le graphe sans

interface [5].

COQ4MDE peut supporter la définition de composants avec interfaces de composition dans n'importe quel DSML. La section suivante décrit la formalisation des opérateurs de composition de base de ISC dans COQ4MDE. Ces opérateurs sont des variantes de l'`Union` et la `Substitution` définis dans le chapitre 3.

5.1.3 L'assemblage de composants

Cette section présente l'implémentation des opérateurs de base de ISC (*bind* et *extend*) définis dans [Aßm03] [Joh11]. La différence entre ces deux opérateurs est que le *bind* appliqué au point de variation remplace le point de variation (c-à-d., il supprime le point de variation de son fragment) alors que le *extend* appliqué sur un point de variation ne modifie pas le point de variation lui même mais l'utilise comme une position pour l'extension (c-à-d., l'élément référencé par un *hook* reste dans son fragment).

5.1.3.1 L'opérateur `bind`

L'opérateur `bind` remplace un objet `b` référencé par un point de variation du premier modèle par un objet $\mathtt{b}'$ référencé par un point de référence dans le deuxième modèle. Le modèle composé est obtenu en substituant `b` par $\mathtt{b}'$ dans les deux ensembles de nœuds et d'arcs [6].
$bind : Model \times Model \times (Objects \times Classes)$
$\times (Objects \times Classes) \rightarrow Model$ est défini comme suit :

$$
\begin{array}{l}
bind(\langle \mathtt{MV_1}, \mathtt{ME_1}\rangle, \langle \mathtt{MV_2}, \mathtt{ME_2}\rangle, \langle \mathtt{b}, \mathtt{B}\rangle, \langle \mathtt{b}', \mathtt{B}'\rangle) = \langle \mathtt{MV_3}, \mathtt{ME_3}\rangle \\
tel\ que\ \langle \mathtt{b}, \mathtt{B}\rangle \in \mathtt{MV_1}\ et\ \langle \mathtt{b}', \mathtt{B}'\rangle \in \mathtt{MV_2},\ nous\ avons : \\
\exists\, \mathtt{h}, \mathtt{p} \in Objects, \langle\langle \mathtt{h}, \mathtt{Hook}\rangle, \mathtt{inh}, \langle \mathtt{h}, \mathtt{ROV}\rangle\rangle \in \mathtt{ME_1} \\
\wedge \langle\langle \mathtt{h}, \mathtt{ROV}\rangle, \mathtt{bind}, \langle \mathtt{b}, \mathtt{AbstractClasses}\rangle\rangle \in \mathtt{ME_1} \\
\wedge \langle\langle \mathtt{b}, \mathtt{B}\rangle, \mathtt{inh}, \langle \mathtt{b}, \mathtt{AbstractClasses}\rangle\rangle \in \mathtt{ME_1} \\
\wedge \langle\langle \mathtt{p}, \mathtt{Prototype}\rangle, \mathtt{inh}, \langle \mathtt{p}, \mathtt{ROV}\rangle\rangle \in \mathtt{ME_2} \\
\wedge \langle\langle \mathtt{p}, \mathtt{ROV}\rangle, \mathtt{bind}, \langle \mathtt{b}', \mathtt{AbstractClasses}\rangle\rangle \in \mathtt{ME_2} \\
\wedge \langle\langle \mathtt{b}', \mathtt{B}'\rangle, \mathtt{inh}, \langle \mathtt{b}', \mathtt{AbstractClasses}\rangle\rangle \in \mathtt{ME_2} \\
et\ finalement : \\
\mathtt{MV_3} = substV(\langle \mathtt{b}, \mathtt{B}\rangle, \langle \mathtt{b}', \mathtt{B}'\rangle, \mathtt{MV_1}) \\
\mathtt{ME_3} = substE(\langle \mathtt{b}, \mathtt{B}\rangle, \langle \mathtt{b}', \mathtt{B}'\rangle, \mathtt{ME_1})
\end{array}
$$

tel que $substV(\langle \mathtt{b}, \mathtt{B}\rangle, \langle \mathtt{b}', \mathtt{B}'\rangle, \mathtt{MV})$ (resp. $substE(\langle \mathtt{b}, \mathtt{B}\rangle, \langle \mathtt{b}', \mathtt{B}'\rangle, \mathtt{ME})$) est la fonction qui remplace $\langle \mathtt{b}, \mathtt{B}\rangle$ par $\langle \mathtt{b}', \mathtt{B}'\rangle$ dans chaque élément de `MV` (resp. relation de `ME`). L'implémentation est basée sur la fonction de `Substitution` présentée dans le chapitre 3 tel que l'objet substitué correspond au point de variation du premier modèle et l'objet substituant correspond au point de référence du deuxième modèle. Cette fonction peut être étendue en une fonction récursive qui admet une liste de correspondance de points (Variation/Référence) et permet ainsi de remplacer plusieurs objets en même temps comme cela est indiqué sur la figure 5.3.

[5] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Interface_Elim.html#elimInt
[6] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Bind2M_Verif.html#Bind2M

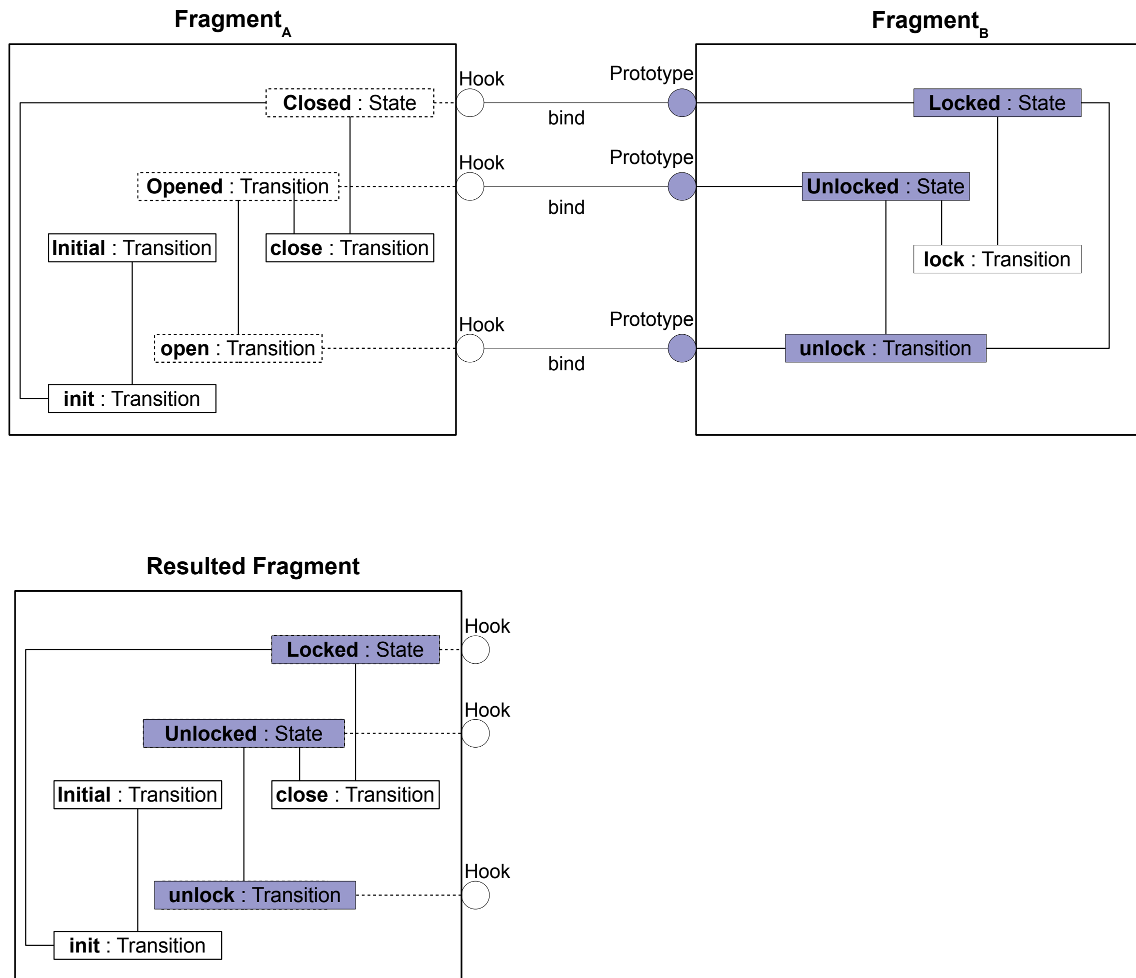


Figure 5.3 — L'opérateur `bind`

5.1.3.2 L'opérateur `extend`

Cet opérateur permet d'étendre un modèle $\langle MV_1, ME_1\rangle$ tel que le point d'extension est un objet b référencé comme un point de variation dans le modèle par un modèle $\langle MV_2, ME_2\rangle$ contenant un objet b' référencé comme un point de référence. La figure 5.4 présente un exemple d'application de l'opérateur `extend`. Cette fonction est paramétrée par un métamodèle pour assurer le typage et un nom pour le lien ajouté entre b et b'. Le modèle composé est un multi-graphe construit à partir de l'union de tous les objets de $\langle MV_1, ME_1\rangle$ et $\langle MV_2, ME_2\rangle$, tous les liens des deux modèles en plus d'un lien entre les deux objets b et b' [7].
$extend : Model \times Model \times (Objects \times Classes) \times (Objects \times Classes)$
$\times MetaModel \times References \rightarrow Model$ est défini comme suit :

[7] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Extend_Verif.html#compExt

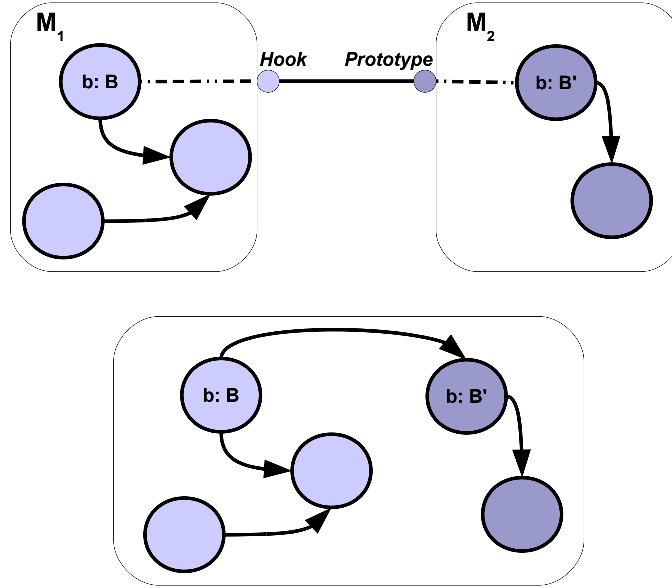


Figure 5.4 — L'opérateur `extend`

$$
\begin{aligned}
&extend(\langle \mathtt{MV}_1, \mathtt{ME}_1\rangle, \langle \mathtt{MV}_2, \mathtt{ME}_2\rangle, \langle \mathtt{b}, \mathtt{B}\rangle, \langle \mathtt{b}', \mathtt{B}'\rangle,\\
&(\langle \mathtt{MMV}, \mathtt{MME}\rangle, conformsTo), \mathtt{LinkName}) = \langle \mathtt{MV}_3, \mathtt{ME}_3\rangle\\
&\textit{tel que } \exists\ \langle \mathtt{b}, \mathtt{B}\rangle \in \mathtt{MV}_1 \textit{ et } \langle \mathtt{b}', \mathtt{B}'\rangle \in \mathtt{MV}_2,\ \textit{nous avons} :\\
&extensible(\langle \mathtt{MV}_1, \mathtt{ME}_1\rangle, \langle \mathtt{MV}_2, \mathtt{ME}_2\rangle, \langle \mathtt{b}, \mathtt{B}\rangle, \langle \mathtt{b}', \mathtt{B}'\rangle,\\
&(\langle \mathtt{MMV}, \mathtt{MME}\rangle, conformsTo), \mathtt{LinkName}) \textit{ tel que} :\\
&\mathtt{MV}_3 = \mathtt{MV}_1 \cup \mathtt{MV}_2\\
&\mathtt{ME}_3 = \mathtt{ME}_1 \cup \mathtt{ME}_2 \cup \{\langle \langle \mathtt{b}, \mathtt{B}\rangle, \mathtt{LinkName}, \langle \mathtt{b}', \mathtt{B}'\rangle\rangle\}
\end{aligned}
$$

Cette fonction peut être simplement défini à l'aide de l'`Union` de modèles comme :

$$Union\ (Union\ \langle \mathtt{MV}_1, \mathtt{ME}_1\rangle\ \langle \mathtt{MV}_2, \mathtt{ME}_2\rangle)\ (\langle \{\langle \mathtt{b}, \mathtt{B}\rangle, \langle \mathtt{b}', \mathtt{B}'\rangle\}, \{\langle \mathtt{b}, \mathtt{B}\rangle, \mathtt{LinkName}, \langle \mathtt{b}', \mathtt{B}'\rangle\rangle\}\rangle).$$

Le prédicat *extensible* vérifie qu'un modèle $\langle \mathtt{MV}_1, \mathtt{ME}_1\rangle$ dont l'interface est $\langle \mathtt{b}, \mathtt{B}\rangle$ par rapport à un certain métamodèle peut être étendu par un autre modèle $\langle \mathtt{MV}_2, \mathtt{ME}_2\rangle$ dont l'interface est $\langle \mathtt{b}', \mathtt{B}'\rangle$.

$$
\begin{aligned}
&extensible(\langle \mathtt{MV}_1, \mathtt{ME}_1\rangle, \langle \mathtt{MV}_2, \mathtt{ME}_2\rangle, \langle \mathtt{b}, \mathtt{B}\rangle, \langle \mathtt{b}', \mathtt{B}'\rangle,\\
&(\langle \mathtt{MMV}, \mathtt{MME}\rangle, conformsTo), \mathtt{LinkName}) \triangleq\\
&isExtendedH(\langle \mathtt{MV}_1, \mathtt{ME}_1\rangle, \langle \mathtt{b}, \mathtt{B}\rangle)\\
&\wedge isExtendedP(\langle \mathtt{MV}_2, \mathtt{ME}_2\rangle), \langle \mathtt{b}', \mathtt{B}'\rangle)\\
&\wedge(\mathtt{B}, \mathtt{LinkName}, \mathtt{B}')\ \in\ \mathtt{MME}
\end{aligned}
$$

Le prédicat *isExtendedH* vérifie que $\langle \mathtt{b}, \mathtt{B}\rangle$ est un point de variation dans $\langle \mathtt{MV}_1, \mathtt{ME}_1\rangle$.

$$
\begin{aligned}
&isExtendedH\langle \mathtt{MV}_1, \mathtt{ME}_1\rangle\langle \mathtt{b}, \mathtt{B}\rangle \triangleq\\
&\exists \mathtt{h} \in Objects, \langle\langle \mathtt{h}, \mathtt{Hook}\rangle, \mathtt{inh}, \langle \mathtt{h}, \mathtt{ROV}\rangle\rangle \in \mathtt{ME}_1\\
&\wedge\langle\langle \mathtt{h}, \mathtt{ROV}\rangle, \mathtt{bind}, \langle \mathtt{b}, \mathtt{AbstractClasses}\rangle\rangle \in \mathtt{ME}_1\\
&\wedge\langle\langle \mathtt{b}, \mathtt{B}\rangle, \mathtt{inh}, \langle \mathtt{b}, \mathtt{AbstractClasses}\rangle\rangle \in \mathtt{ME}_1
\end{aligned}
$$

Le prédicat *isExtendedP* vérifie que $\langle \mathtt{b}, \mathtt{B} \rangle$ est un point de référence dans le modèle $\langle \mathtt{MV}_2, \mathtt{ME}_2 \rangle$.

$$
\begin{aligned}
& isExtendedP \langle \mathtt{MV}_2, \mathtt{ME}_2 \rangle \langle \mathtt{b}, \mathtt{B} \rangle \triangleq \\
& \exists \mathtt{p}, \langle \langle \mathtt{p}, \mathtt{Prototype} \rangle, \mathtt{inh}, \langle \mathtt{p}, \mathtt{ROV} \rangle \rangle \in \mathtt{ME}_2 \\
& \wedge \langle \langle \mathtt{p}, \mathtt{ROV} \rangle, \mathtt{bind}, \langle \mathtt{b}, \mathtt{AbstractClasses} \rangle \rangle \in \mathtt{ME}_2 \\
& \wedge \langle \langle \mathtt{b}, \mathtt{B} \rangle, \mathtt{inh}, \langle \mathtt{b}, \mathtt{AbstractClasses} \rangle \rangle \in \mathtt{ME}_2
\end{aligned}
$$

Ici nous avons présenté un seul type de correspondance entre points de variation et de référence (`hook/prototype`), la méthode décrite dans [Joh11] considère aussi un autre type de correspondance (`slot/anchor`). Ce deuxième type de correspondance nécessite des conditions supplémentaires pour considérer la propriété de composition d'un arc et le traiter d'une manière particulière s'il est un arc de composition. La différence telle qu'elle est expliquée dans [Joh11] est que contrairement au `hook` et `prototype`, le point de variation `slot` et le point de référence `anchor` gardent leur contenu en cas de composition, ce qui peut s'implémenter aussi comme une variante de l'opérateur `Union`. Le premier type traité de correspondance suffit pour spécifier des compositions complexe comme celle présentée dans l'exemple de la section suivante.

5.2 Un exemple détaillé

Nous décrivons dans cette section l'utilisation des opérateurs ISC pour élaborer une composition de deux modèles. $\mathtt{M}_1$ est une machine à état qui modélise une porte avec une serrure. La porte permet les opérations : `open`, `close`, `pass`, `lock` et `unlock`. Nous voulons ajouter la possibilité de fermer la porte avec `simple` ou `double` tour, ces deux états sont décrits dans le modèle $\mathtt{M}_2$. $\mathtt{M}_1$ et $\mathtt{M}_2$ sont présentés sur la figure 5.5.

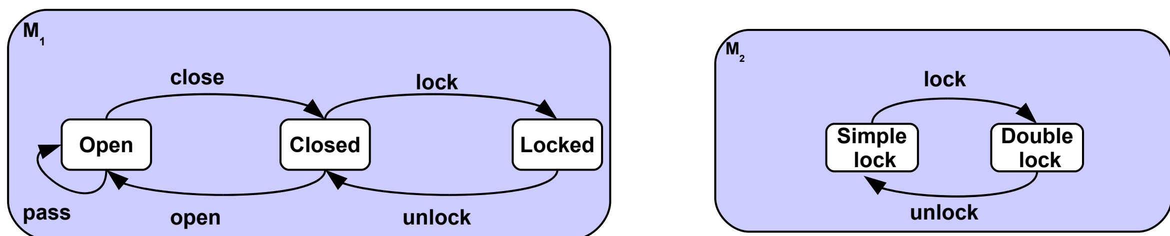


Figure 5.5 — Les modèles $\mathtt{M}_1$ et $\mathtt{M}_2$

La première étape est la définition de l'interface de chaque modèle. Ceci est fait avec la fonction *FragmentExtraction* appliquée au modèle $\mathtt{M}_1$ définit *Locked* comme un point de variation et appliquée au modèle $\mathtt{M}_2$, elle définit *Simple lock* comme un point de référence comme décrit sur la figure 5.6.

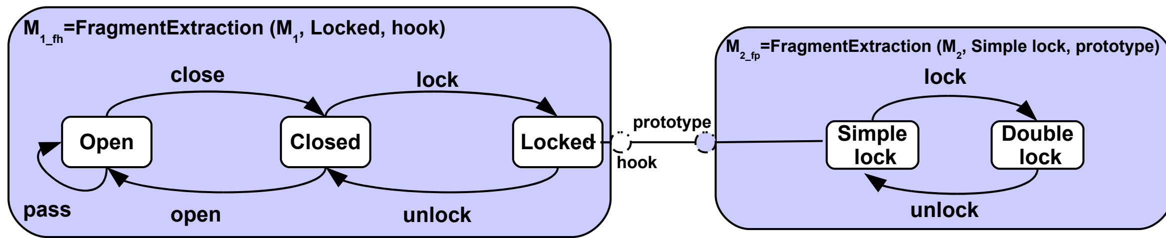


Figure 5.6 — Les points de variation et de référence pour les modèles $\mathtt{M_1}$ et $\mathtt{M_2}$

L'application de la fonction `bind` sur les deux fragments comme il est montré sur la figure 5.6 suivie de l'élimination de l'interface produit le modèle $\mathtt{M_{bind}}$ montré sur la figure 5.7.

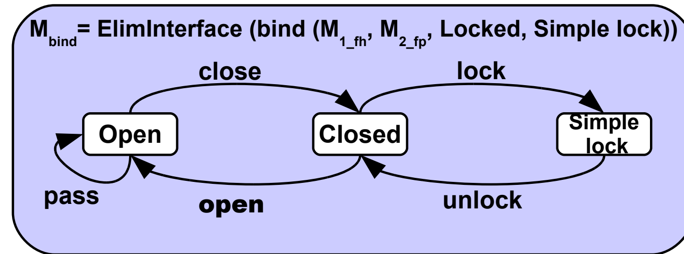


Figure 5.7 — Le modèle après l'application de la fonction *bind*

Enfin, *Simple lock* est défini dans $\mathtt{M_{bind}}$ comme un point de référence et *Double lock* est défini dans $M_{2_fp_elim}$ comme un point de variation comme montré sur la figure 5.8.

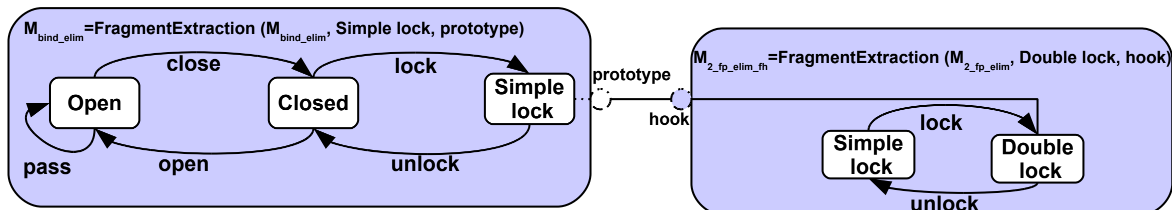


Figure 5.8 — L'extraction des fragments

L'application de la fonction *extend* sur les deux modèles présentés sur la figure 5.8 après l'élimination de l'interface génère le modèle présenté sur la figure 5.9. Le modèle est la machine à états pour une porte avec une option `simple` ou `double lock`.

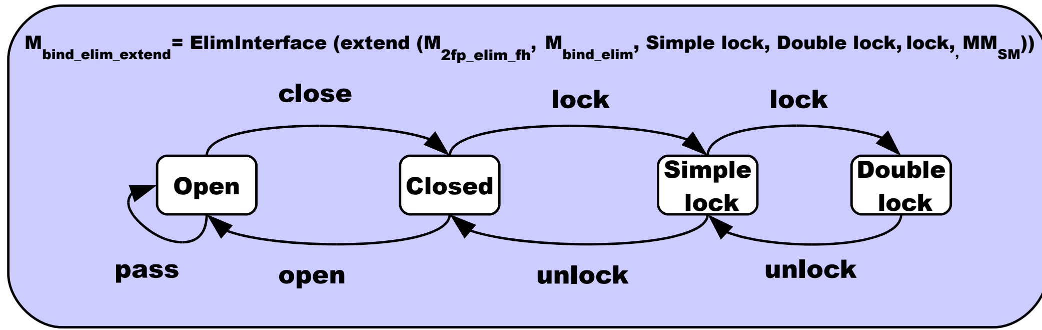


Figure 5.9 — Le modèle après l'exécution des fonction *extend* et *ElimInterface*

La contribution originale de cette partie n'est pas la définition des opérateurs de composition qui est inspirée de ISC mais leur implémentation en utilisant l'assistant de preuve Coq, leur intégration dans l'atelier Coq4MDE et la preuve que les propriétés liées à la conformité sont compositionnelles sous certaines hypothèses/préconditions pour ces opérateurs.

5.3 La véri cation des propriétés

Nous utilisons le prédicat *ValidCompositionFunction* pour vérifier le typage pour l'opérateur de composition `bind` présenté dans la section 5.1.3. Ceci correspond au théorème 15 [8].

Théorème 15. (ValidBind)
$\forall\, \mathtt{MM} \in \mathit{MetaModel}, \mathit{ValidCompositionFunction}(\mathtt{MM}, \mathtt{bind}).$

Le prédicat *ValidCompositionFunction* est défini comme suit :

$$\begin{aligned}&\mathit{ValidCompositionFunction}(MM \in \mathit{MetaModel}, f) \triangleq\\&\forall\, M1\ M2 \in \mathit{Model},\\&\mathit{InstanceOf}\ (M1, MM)\ \wedge \mathit{InstanceOf}\ (M2, MM)\\&\rightarrow \mathit{InstanceOf}\ ((f\ M1\ M2), MM)\end{aligned}$$

La preuve Coq pour ce théorème utilise des lemmes intermédiaires qui prouvent la préservation du typage par les opérations élémentaires utilisées dans la composition. Parmi ces lemmes, `conformsAddO` assure que le résultat de l'ajout d'un objet instance d'une classe dans le métamodèle à un composant instance de ce métamodèle est un composant instance du même métamodèle.

Lemme 22. (conformsAddO)
$\forall\, \langle \mathtt{MV}, \mathtt{ME}\rangle \in \mathit{Model}, \langle(\mathtt{MMV}, \mathtt{MME}), \mathit{conformsTo}\rangle \in \mathit{MetaModel},\ \mathtt{o} \in \mathtt{Objects},\ \mathtt{c} \in \mathtt{Classes},$
$\mathit{InstanceOf}(\langle \mathtt{MV}, \mathtt{ME}\rangle, \langle(\mathtt{MMV}, \mathtt{MME}), \mathit{conformsTo}\rangle) \wedge \mathtt{c} \in \mathtt{MMV}$
$\rightarrow \mathit{InstanceOf}(\langle \mathtt{MV} \cup \{\langle \mathtt{o}, \mathtt{c}\rangle\}, \mathtt{ME}\rangle, \langle(\mathtt{MMV}, \mathtt{MME}), \mathit{conformsTo}\rangle).$

[8] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Bind2M_Verif.html#ValidBind

Une autre preuve COQ a été faite pour démontrer la préservation du typage par l'opérateur de composition *extend* présenté dans la section 5.1.3. Ceci est codé dans le théorème `ValidExtend`[9].

Théorème 16. (ValidExtend)
$\forall\, \mathtt{MM} \in \mathit{MetaModel}, \mathit{ValidCompositionFunction}(\mathtt{MM}, \mathtt{extend})$

Enfin, à partir du cadre formel COQ4MDE et de la méthode de composition ISC, nous avons défini un atelier pour la composition de modèles. Les définitions de modèle et métamodèle ont été étendues pour permettre la définition de l'interface de composition de modèles. Le fragment construit est un modèle conforme au métamodèle étendu. Les opérateurs élémentaires de composition sont formalisés comme tous les éléments présentés dans ce chapitre en utilisant l'assistant de preuve COQ. La formalisation en COQ assure la terminaison[10] des opérateurs de composition, élabore une propriété de vérification compositionnelle qui permet d'exprimer et de vérifier les propriétés présentées dans le chapitre 3.

Dans ce qui suit, nous présentons pour chaque propriété élémentaire, le théorème qui prouve la préservation de la propriété pour l'opérateur `bind` défini dans la section 5.1.3.1 et le lien avec la preuve complète en COQ.

5.3.1 L'opérateur `bind`

Les vérifications de la préservation des propriétés sémantiques sont faites en premier lieu pour la fonction de substitution dans 3. L'opérateur élémentaire `bind` substitue un élément de modèle à un autre, les deux éléments du modèle doivent être du même type.

On montre dans ce qui suit que les propriétés : `subClass`, `isAbstract`, `lower` & `upper`, `isOpposite` et `areComposite` sont préservées par cet opérateur. Les preuves de ces propriétés nécessitent des conditions similaires de ceux présentées pour la fonction de substitution dans le chapitre 3.

subClass Le théorème 17 (`BindSubClassPreserved`) montre[11] que la propriété `subClass` est préservée par l'opérateur élémentaire `bind`. Donc, pour toutes classes $\mathtt{c}_1\ \mathtt{c}_2$ et pour tous éléments de modèle $\mathtt{o}_1\ \mathtt{o}_2$, si $\mathtt{c}_1$ est une *subClass* de $\mathtt{c}_2$ dans les deux modèles $\mathtt{M}_1$ et $\mathtt{M}_2$, alors $\mathtt{c}_1$ est aussi une *subClass* de $\mathtt{c}_2$ dans le modèle résultat du (*bind* $\mathtt{o}_1\ \mathtt{o}_2\ \mathtt{M}_1\ \mathtt{M}_2$).

Théorème 17. (BindSubClassPreserved)
$\forall\, \mathtt{M}_1\ \mathtt{M}_2 \in \mathtt{Model},\ \mathtt{c}_1\ \mathtt{c}_2 \in \mathtt{Classes},\ \mathtt{o}_1\ \mathtt{o}_2 \in \mathtt{Objects},$
$(\mathit{subClass}\ \mathtt{c}_1\ \mathtt{c}_2\ \mathtt{M}_1) \wedge (\mathit{subClass}\ \mathtt{c}_1\ \mathtt{c}_2\ \mathtt{M}_2) \rightarrow \mathit{subClass}\ \mathtt{c}_1\ \mathtt{c}_2\ (\mathit{bind}\ \mathtt{o}_1\ \mathtt{o}_2\ \mathtt{M}_1\ \mathtt{M}_2)$

Il n'y a donc aucune précondition sur les paramètres de l'opérateurs de composition `bind` pour que la vérification de la relation `subClass` soit compositionnelle.

[9] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Extend_Verif.html#ValidExt

[10] Nous ne pouvons pas écrire une fonction en COQ si la preuve de terminaison n'est pas donné ou déduite par COQ

[11] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Bind2M_Verif.html#Bind2MSCP

isAbstract La préservation de cette propriété par l'opérateur `bind` est prouvée[12] en utilisant le théorème 18 (`BindIsAbstractPreserved`). Ce théorème montre que toutes les classes abstraites dans deux modèles M_1 et M_2 sont aussi abstraites dans le modèle résultant de l'application du *bind* sur ces deux modèles.

Théorème 18. (BindIsAbstractPreserved)
$\forall M_1\ M_2 \in \mathit{Model},\ c \in \mathit{Classes},\ o_1\ o_2 \in \mathit{Objects},$
$(\mathit{isAbstract}\ c\ M_1) \wedge (\mathit{isAbstract}\ c\ M_2) \rightarrow \mathit{isAbstract}\ c\ (\mathit{bind}\ o_1\ o_2\ M_1\ M_2)$

Il n'y a donc aucune précondition sur les paramètres de l'opérateur de composition `bind` pour que la vérification de la relation `isAbstract` soit compositionnelle.

lower & upper Le théorème 19 (`BindLowerPreserved`) montre[13] que la propriété `lower` est préservée par l'opérateur `bind`. La vérification exige que la substitution soit injective et préserve la différence entre les éléments dans le modèle résultant. Ceci est assuré si l'élément du modèle o_2 n'est pas dans le premier modèle, ce qui garantit que l'opérateur `bind` n'ajoute pas un élément qui existe déjà dans le modèle. Enfin, la préservation de propriété `lower` est prouvée.

Théorème 19. (BindLowerPreserved)
$\forall\ \langle MV_1, ME_1\rangle\ \langle MV_2, ME_2\rangle \in \mathit{Model},\ c \in \mathit{Classes},\ r \in\ \mathit{References},$
$n \in \mathit{Natural}^{\top},\ (o_1, c_1)\ (o_2, c_2) \in \mathit{Objects},$
$c_1 = c_2 \wedge (\nexists (o, c) \in MV_1 \wedge o = o_2) \wedge\ \mathit{Injectif\ bind}$
$\wedge\ (\mathit{lower}\ c\ r\ n\ \langle MV_1, ME_1\rangle) \wedge (\mathit{lower}\ c\ r\ n\ \langle MV_2, ME_2\rangle)$
$\rightarrow (\mathit{lower}\ c\ r\ n\ (\mathit{bind}\ (o_1, c_1)\ (o_2, c_2)\ \langle MV_1, ME_1\rangle\ \langle MV_2, ME_2\rangle)).$

La préservation de la propriété `upper` est décrite[14] par le théorème `BindUpperPreserved` qui est similaire au théorème précédent pour la propriété `lower`. La condition est que le modèle ne doit pas contenir un élément dont le nom est o_2.

Nous constatons qu'il est nécessaire d'introduire des hypothèses sur les éléments des modèles pour assurer que la composition préserve les propriétés. Il s'agit donc de préconditions sur les opérateurs de composition pour assurer la préservation des propriétés sémantiques.

isOpposite Le théorème 20 (`BindIsOppositePreserved`) montre[15] que chaque paire de références opposites dans les deux modèles M_1 et M_2 restent aussi opposites en appliquant le *bind* sur les deux modèles. Enfin, la propriété `isOpposite` est préservée.

Théorème 20. (BindIsOppositePreserved)
$\forall\ M_1\ M_2 \in \mathit{Model},\ r_1\ r_2 \in \mathit{References},\ o_1\ o_2 \in \mathit{Objects},$
$(\mathit{isOpposite}\ r_1\ r_2\ M_1) \wedge (\mathit{isOpposite}\ r_1\ r_2\ M_2)$
$\rightarrow (\mathit{isOpposite}\ r_1\ r_2\ (\mathit{bind}\ o_1\ o_2\ M_1\ M_2)).$

[12] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Bind2M_Verif.html#Bind2MIAP
[13] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Bind2M_Verif.html#Bind2MLP
[14] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Bind2M_Verif.html#Bind2MUP
[15] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Bind2M_Verif.html#Bind2MIOP

Il n'y a donc aucune précondition sur les paramètres de l'opérateur de composition `bind` pour que la vérification de la relation `isOpposite` soit compositionnelle.

areComposite Le théorème 21 (`BindAreCompositeSubsPreserved`) montre[16] que l'ensemble de références composites dans deux modèles $\mathtt{M}_1$ et $\mathtt{M}_2$ sont aussi composites dans le modèle résultat de l'application de l'opérateur `bind` sur les deux modèles. Ce théorème suppose aussi que la substitution est injective et exige que le modèle substitué ne contienne pas un élément dont le nom est $\mathtt{o}_2$.

Théorème 21. (BindAreCompositeSubsPreserved)

$$\begin{array}{l}\forall\ \langle \mathtt{MV}_1,\mathtt{ME}_1\rangle\ \langle \mathtt{MV}_2,\mathtt{ME}_2\rangle \in \mathit{Model},\ \mathtt{c} \in \mathit{Classes},\\ \mathtt{R} \subset \mathit{References},\ \mathtt{o}_1\mathtt{o}_2 \in \mathit{Objects},\ \mathtt{c}_2\ \mathtt{c}_2 \in \mathit{Classes},\ \mathtt{c}_1 = \mathtt{c}_2\\ \wedge(\nexists(\mathtt{o},\mathtt{c}) \in \mathtt{MV} \wedge \mathtt{o} = \mathtt{o}_2) \wedge \mathit{Injectif\ bind}\\ \wedge\ (\mathit{areComposite}\ \mathtt{c}\ \mathtt{R}\ \langle \mathtt{MV}_1,\mathtt{ME}_1\rangle) \wedge\ (\mathit{areComposite}\ \mathtt{c}\ \mathtt{R}\ \langle \mathtt{MV}_2,\mathtt{ME}_2\rangle)\\ \rightarrow (\mathit{areComposite}\ \mathtt{c}\ \mathtt{R}\ (\mathit{bind}\ (\mathtt{o}_1,\mathtt{c}_1)\ (\mathtt{o}_2,\mathtt{c}_2)\ \langle \mathtt{MV}_1,\mathtt{ME}_1\rangle\ \langle \mathtt{MV}_2,\mathtt{ME}_2\rangle))\end{array}$$

Les preuves pour cette version de l'opérateur `bind` (deux modèles) utilisent les théorèmes de vérification pour l'opérateur élémentaire de `Substitution`, en plus d'un schéma standard pour trouver le modèle cible et les conditions d'application. Le langage de tactiques pour le système COQ [Del00] est utilisé pour définir des tactiques qui améliorent considérablement les preuves. Nous donnons ici une introduction rapide du langage de tactique de COQ.

5.3.1.1 Une vue sur le langage de tactiques

Dans un assistant de preuve, nous pouvons différencier généralement deux types de langages utilisés : le langage de preuve (permettant à l'utilisateur d'écrire les spécifications des propriétés et des preuves de propriétés) et le langage des tactiques (permettant à l'utilisateur d'écrire ses propres schémas de preuves qui seront ensuite utilisés pour construire les preuves). COQ permet de structurer une preuve comme une pile de buts, le traitement d'un but dépile celui-ci et empile les sous-buts résultant de son traitement comme expliqué dans le chapitre 2.

Un langage de tactiques permet d'améliorer l'automatisation des preuves en fournissant les outils permettant d'écrire des tactiques de preuves d'une très grande expressivité. Il s'agit donc en général d'un méta langage car il permet la manipulation du générateur de preuve lui même et pas directement des preuves.

Actuellement, la définition[17] de la syntaxe du langage des tactiques COQ (Ltac) est accessible dans le manuel de référence de COQ [Tea12]. Nous utilisons le langage Ltac pour la définition de tactiques spécifiques à notre contexte. Ltac contient des `tacticals` (permettant de combiner des tactiques) et d'autres structures permettant de faire du filtrage sur les termes et les contextes des preuves. Voici dans la table 5.1 les `tacticals` de COQ comme présentées dans [Del00].

[16] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Bind2M_Verif.html#Bind2MACP
[17] http://coq.inria.fr/refman/Reference-Manual012.html

tac_1 ; tac_2	Applique tac_1 au but puis tac_2 à tous les sous-buts
tac ; [tac_1 | ... | tac_i | ... | tac_n]	Applique *tac* au but puis tac_i au i^{me} sous-but
tac_1 Orelse tac_2	Applique tac_1 au but ou tac_2 si tac_1 échoue
Do *n* *tac*	Applique *tac* *n* fois au but et aux sous buts engendrés
Repeat *tac*	Applique *tac* jusqu'à ce qu'il échoue au but et aux sous buts engendrés
Try *tac*	Applique *tac* et n'échoue pas si l'application au but échoue
First [tac_1 | ... | tac_i | ... | tac_n]	Applique la première tac_i qui n'échoue pas au but
Solve [tac_1 | ... | tac_i | ... | tac_n]	Applique la première tac_i qui résout le but
Idtac	Laisse le but inchangé
Fail	échoue toujours

Tableau 5.1 — Les tacticals de COQ

Le langage produit des expressions qui sont des tactiques, ces expressions doivent être évaluées dans un environnement qui est un but. Nous présentons un exemple d'une première tactique utilisée pour démontrer toutes les propriétés pour l'opérateur `bind` (appliqué sur deux modèles), cette tactique est basée sur les `tacticals` présentées dans la table 5.1. La tactique 5.1, après l'introduction de la définition de la fonction `Bind2M`, essaie de distinguer les cas où le point de variation/point de référence est dans le premier ou le deuxième modèle et aussi considère le cas où les deux objets sont du même type puis tente de répéter l'application des théorèmes `SubstIOP`, `SubstSCP`, `SubstIAP`, `SubstIOP`, `SubstLP` et `SubstUP` jusqu'à résolution du but.

```
Ltac Bind2MPreserve :=
  repeat
 (intros; unfold Bind2M
  ; case HookInM; case HookInM
  ; case PrototypeInM; case PrototypeInM; trivial
    ; do 2 (case beqClass; trivial;
      try repeat (apply SubstACP    || apply SubstIOP
               || apply SubstSCP    || apply SubstIAP
               || apply SubstIOP    || apply SubstLP
               || apply SubstUP)
      ; auto)).
```

Tactique 5.1 — La tactique `Bind2MPreserve`

Une autre tactique est définie pour prouver les mêmes théorèmes pour une version de l'opérateur `bind` appliquée sur une liste de points de variation et de référence. Cette tactique utilise en plus des tacticals une opération de filtrage du but dont la syntaxe est présentée en 5.10 en s'inspirant du manuel de référence [Tea12].

```
match goal with
         | hyp1,1 ,... ,hyp1,m1 |-cpattern1=> expr1
         | hyp2,1 ,... ,hyp2,m2 |-cpattern2=> expr2
...
         | hypn,1 ,... ,hypn,mn |-cpatternn=> exprn
         |_      => exprn+1
end
```

Figure 5.10 — Une partie de la syntaxe du langage Ltac

Pour la sémantique de l'opération de filtrage, si chacune des hypothèses $hyp_{1,i}$ avec $i = 1, ..., m_1$ correspond à une hypothèse dans le but et si $cpattern_1$ correspond à la conclusion du but, alors $expr_1$ est évaluée en v_1 en substituant les hypothèses réelles aux hypothèses $hyp_{1,1}, ..., hyp_{1,m1}$. Si v_1 est une tactique, alors elle est appliquée au but. Si cette application échoue, alors une autre combinaison d'hypothèses est testée avec le même contexte de preuve. S'il n'y a pas d'autres combinaisons d'hypothèses alors le filtre du contexte suivant est testé et ainsi de suite. Si tous les filtres du contexte échouent, alors $expr_{n+1}$ est évaluée et appliquée.

```
Ltac Bind2MSHPreserve:=
repeat
  (unfold Bind2MSH; intro l
   ; induction l; trivial
     ; match goal with
         | a:_, l:_, IHl:_ |- _ =>  intros
           ; apply IHl; trivial
             ; try (apply Bind2MACP || apply Bind2MSCP
                 || apply Bind2MIAP || apply Bind2MIOP
                 || apply Bind2MLP  || apply Bind2MUP)
             ; assumption
         end).
```

Tactique 5.2 — La tactique `Bind2MSHPreserve`

La tactique `Bind2MSHPreserve` 5.2 est utilisée pour prouver automatiquement une partie des preuves présentée dans la section suivante pour l'opérateur `bind` avec une liste de points de variation. Elle introduit la définition de la version de l'opérateur `bind` avec une liste de points de variation et fait par la suite une preuve par induction sur la structure de la liste. Elle recherche dans les hypothèses, une hypothèse IHl telle que, en l'appliquant sur le but, la preuve de celui-ci se fait d'une manière triviale ou en répétant l'application des théorèmes `Bind2MACP`, `Bind2MSCP`, `Bind2MIAP`, `Bind2MIOP`, `Bind2MLP` et `Bind2MUP` qui sont déjà prouvés en utilisant la tactique précédente.

Les structures de contrôle dans Ltac permettent une programmation rigoureuse des algorithmes de preuve. Le langage permet également d'écrire des fonctions dont la terminaison n'est pas assurée car il ne s'agit pas d'un terme du λ calcul typé interne à COQ mais une fonction qui doit générer un tel terme. Sa non terminaison ne pose donc aucun problème théorique. Nous l'utilisons dans notre contexte pour simplifier des preuves auparavant réalisées manuellement, ce qui nous permet de contourner ce dernier problème.

5.3.2 L'opérateur `bind` de deux modèles avec plusieurs points de variation

Cette version est une généralisation de la version précédente de l'opérateur `bind`. Elle est caractérisée par une liste de points de variation et de référence.

`Bind2MSH` : $Model \times Model \times list\ (Objects \times Classes)$ est défini comme suit :

$$\texttt{Bind2MSH}\ \texttt{M}_1\ \texttt{M}_2\ l = \forall (o, o') \in l, bind\ \texttt{M}_1\ \texttt{M}_2\ o\ o'\ l$$

Les preuves des propriétés nécessitent les hypothèses suivantes : la compatibilité de type entre les deux éléments de modèles pour chaque paire d'éléments de modèles dans la liste,

l'injectivité de la substitution et une condition supplémentaire : le même *Prototype* n'est pas donné plus d'une fois pour assurer la préservation des multiplicités. Les mêmes hypothèses/préconditions sont nécessaires pour prouver la vérification compositionnelle des différentes propriétés considérées.

La préservation du typage par l'opérateur de composition `bind` de deux modèles avec plusieurs points de variation est démontrée à l'aide du théorème 22 (`ValidBindSH`).

Théorème 22. (ValidBindSH)
$\forall\, \mathtt{M}_1\, \mathtt{M}_2 \in \mathit{Model},\ \mathtt{l} \in \mathit{list}\ (\mathit{Objects} * \mathit{Classes}),\ \mathtt{MM} \in \mathit{MetaModel},$
$\mathit{InstanceOf}\ (\mathtt{M}_1, \mathtt{MM})\ \wedge\ \mathit{InstanceOf}\ (\mathtt{M}_2, \mathtt{MM})$
$\rightarrow \mathit{InstanceOf}\ ((\mathtt{Bind2MSH}\ \mathtt{M}_1\ \mathtt{M}_2\ l), \mathtt{MM}).$

subClass Le théorème 23 (`Bind2MSHSubClassPreserved`) montre[18] que la propriété `subClass` est préservée par l'opérateur `Bind2MSH`. Donc, pour toutes classes $\mathtt{c}_1$ $\mathtt{c}_2$ et pour toute liste *l* de correspondance entre points (variation/référence), si $\mathtt{c}_1$ est une *subClass* de $\mathtt{c}_2$ dans les deux modèles $\mathtt{M}_1$ et $\mathtt{M}_2$, alors $\mathtt{c}_1$ est aussi une *subClass* de $\mathtt{c}_2$ dans ($\mathtt{Bind2MSH}\ \mathtt{M}_1\ \mathtt{M}_2\ l$).

Théorème 23. (Bind2MSHSubClassPreserved)
$\forall\, \mathtt{M}_1\, \mathtt{M}_2 \in \mathit{Model},\ l \in \mathit{list}\ (\mathit{Objects} * \mathit{Classes}),\ c_1\ c_2 \in \mathit{Classes},$
$\mathit{subClass}\ c_1\ c_2\ \mathtt{M}_1\ \wedge \mathit{subClass}\ c_1\ c_2\ \mathtt{M}_2 \rightarrow \mathit{subClass}\ c_1\ c_2\ (\mathtt{Bind2MSH}\ \mathtt{M}_1\ \mathtt{M}_2\ l).$

isAbstract La préservation de cette propriété par l'opérateur `Bind2MSH` est prouvée en utilisant le théorème 24 (`BindIsAbstractPreserved`).

Théorème 24. (Bind2MSHIsAbstractPreserved)
$\forall\, \mathtt{M}_1\, \mathtt{M}_2 \in \mathit{Model},\ l \in \mathit{list}\ (\mathit{Objects} * \mathit{Classes}),\ c \in \mathit{Classes},$
$\mathit{isAbstract}\ c\ \mathtt{M}_1 \wedge \mathit{isAbstract}\ c\ \mathtt{M}_2 \rightarrow \mathit{isAbstract}\ c\ (\mathtt{Bind2MSH}\ \mathtt{M}_1\ \mathtt{M}_2\ l).$

Ce théorème montre[19] que toutes les classes abstraites dans deux modèles $\mathtt{M}_1$ et $\mathtt{M}_2$ sont aussi abstraites dans le modèle résultant de l'application du `Bind2MSH` sur ces deux modèles avec une liste *l* de correspondance entre points de variation et de référence.

lower & upper Le théorème `Bind2MSHLowerPreserved` montre[20] que la propriété `lower` est préservée par l'opérateur `Bind2MSH`. La vérification exige que la substitution soit injective et préserve la différence entre les éléments dans le modèle résultant. Les trois hypothèses nécessaires sont : la compatibilité de type entre les deux éléments de modèle pour chaque paire d'éléments de modèles dans la liste, l'injectivité de la substitution (utilisée dans la définition de la fonction *bind*) et une condition supplémentaire : le même *Prototype* n'est pas donné plus d'une fois dans la liste de correspondance. Notons qu'il s'agit donc de nouveau de préconditions sur l'opérateur de composition nécessaires pour prouver la préservation des propriétés sémantiques lors de la composition.

[18] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Bind2M_Verif.html#Bind2MSHSCP
[19] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Bind2M_Verif.html#Bind2MSHIAP
[20] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Bind2M_Verif.html#Bind2MSHILP

Théorème 25. (Bind2MSHLowerPreserved)
$\forall \langle \mathtt{MV_1,ME_1} \rangle \langle \mathtt{MV_2,ME_2} \rangle \in \mathtt{Model}$, $c \in \mathtt{Classes}$, $l \in list\ (\mathtt{Objects} * \mathtt{Classes})$,
$r \in \mathtt{References}$, $n \in \mathtt{Natural}$,
$\forall((\mathtt{o}_1,c_1),(\mathtt{o}_2,c_2)) \in l,\ c_1 = c_2 \wedge (\nexists(o,c) \in (\mathtt{MV_1} \cup \mathtt{MV_2}) \wedge o = \mathtt{o}_2) \wedge Injectif\ Subst$
$\wedge\forall(obj_1,obj_2) \in l, (\nexists(o,o') \in (l-(obj_1,obj_2)) \wedge o' = obj_2)$
$\wedge(lower\ c\ r\ n\ \langle \mathtt{MV_1,ME_1} \rangle) \wedge (lower\ c\ r\ n\ \langle \mathtt{MV_2,ME_2} \rangle)$
$\rightarrow (lower\ c\ r\ n\ (\mathtt{Bind2MSH}\ \langle \mathtt{MV_1,ME_1} \rangle \langle \mathtt{MV_2,ME_2} \rangle\ l))$.

La préservation de la propriété `upper` est décrite[21] en utilisant le théorème 26 (`Bind2MSHUpperPreserved`) qui est similaire au théorème précédent pour la propriété `lower`. Les trois hypothèses nécessaires sont : la compatibilité de type pour chaque paire d'éléments de modèles dans la liste et l'injectivité de la substitution.

Théorème 26. (Bind2MSHUpperPreserved)
$\forall \langle \mathtt{MV_1,ME_1} \rangle \langle \mathtt{MV_2,ME_2} \rangle \in \mathtt{Model}$, $c \in \mathtt{Classes}$, l, $r \in \mathtt{References}$, $n \in \mathtt{Natural}$,
$\forall((\mathtt{o}_1,c_1),(\mathtt{o}_2,c_2)) \in l,\ c_1 = c_2 \wedge (\nexists(o,c) \in (\mathtt{MV_1} \cup \mathtt{MV_2}) \wedge o = \mathtt{o}_2) \wedge Injectif\ Subst$
$\wedge\forall(obj_1,obj_2) \in l, (\nexists(o,o') \in (l-(obj_1,obj_2)) \wedge o' = obj_2)$
$\wedge(upper\ c\ r\ n\ \langle \mathtt{MV_1,ME_1} \rangle) \wedge (upper\ c\ r\ n\ \langle \mathtt{MV_2,ME_2} \rangle)$
$\rightarrow (upper\ c\ r\ n(Bind2MSH\ \langle \mathtt{MV_1,ME_1} \rangle \langle \mathtt{MV_2,ME_2} \rangle\ l))$.

isOpposite Le théorème `Bind2MSHIsOppositePreserved` 27 montre[22] que chaque paire de références opposites dans les deux modèles $\mathtt{M}_1$ et $\mathtt{M}_2$ sont aussi opposites en appliquant le `Bind2MSH` sur les deux modèles avec une liste de points de correspondance l. Enfin, la propriété `isOpposite` est préservée.

Théorème 27. (Bind2MSHIsOppositePreserved)
$\forall\, \mathtt{M}_1\ \mathtt{M}_2 \in \mathtt{Model}$, l, $r_1\ r_2 \in \mathtt{References}$,
$(isOpposite\ r_1\ r_2\ \mathtt{M}_1) \wedge (isOpposite\ r_1\ r_2\ \mathtt{M}_2)$
$\rightarrow (isOpposite\ r_1\ r_2\ (\mathtt{Bind2MSH}\ \mathtt{M}_1\ \mathtt{M}_2\ l))$.

areComposite Le théorème 28 (`Bind2MAreCompositePreserved`) montre[23] qu'un ensemble de références qui sont composites dans les deux modèles $\mathtt{M}_1$ et $\mathtt{M}_2$ sont aussi composites dans le modèle résultat de l'application du `Bind2MSH` sur les deux modèles avec une liste l de correspondance. Les trois hypothèses nécessaires sont : la compatibilité de type pour chaque paire d'éléments dans la liste, l'injectivité de la substitution et une condition supplémentaire : le même Prototype n'est pas donné plus d'une fois.

Théorème 28. (Bind2MAreCompositePreserved)
$\forall \langle \mathtt{MV_1,ME_1} \rangle \langle \mathtt{MV_2,ME_2} \rangle \in \mathtt{Model}$, l, $c \in \mathtt{Classes}$, $R \subset \mathtt{References}$,
$c_1 = c_2 \wedge (\nexists(o,c) \in (\mathtt{MV_1} \cup \mathtt{MV_2}) \wedge o = \mathtt{o}_2) \wedge Injectif\ Subst$
$\wedge\forall(obj_1,obj_2) \in l, (\nexists(o,o') \in (l-(obj_1,obj_2)) \wedge o' = obj_2)$
$\wedge(areComposite\ c\ R\ \langle \mathtt{MV_1,ME_1} \rangle) \wedge (areComposite\ c\ R\ \langle \mathtt{MV_2,ME_2} \rangle)$
$\rightarrow (areComposite\ c\ R\ (\mathtt{Bind2MSH}\ \langle \mathtt{MV_1,ME_1} \rangle \langle \mathtt{MV_2,ME_2} \rangle\ l))$.

[21] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Bind2M_Verif.html#Bind2MSHIUP
[22] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Bind2M_Verif.html#Bind2MSHIOP
[23] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Bind2M_Verif.html#Bind2MSHACP

5.3.3 L'opérateur `extend`

Deux variantes de l'opérateur `extend` sont implémentées et vérifiées séparément. La première version (`compositionExtend`) suppose en plus de la définition de l'opérateur `extend` présentée dans 5.1.3.2 que les deux modèles sont disjoints (ne contiennent pas un même élément) pour définir le prédicat *extensibleC*.

$$\begin{aligned}&extensibleC(\langle \mathtt{MV_1,ME_1}\rangle, \langle \mathtt{MV_2,ME_2}\rangle, \langle \mathtt{b,B}\rangle, \langle \mathtt{b',B'}\rangle,\\&(\langle \mathtt{MMV,MME}\rangle, conformsTo), \mathtt{LinkName}) \triangleq\\&isExtendedH(\langle \mathtt{MV_1,ME_1}\rangle, \langle \mathtt{b,B}\rangle)\\&\wedge isExtendedP(\langle \mathtt{MV_2,ME_2}\rangle), \langle \mathtt{b',B'}\rangle)\\&\wedge(\mathtt{B,LinkName,B'}) \in \mathtt{MME}\\&\wedge \forall v \in \mathtt{MV_1}, v \notin \mathtt{MV_2}\end{aligned}$$

La seconde version (`compositionExtendC`) ne fait pas de suppositions sur l'intersection de modèles et correspond exactement à l'opérateur `extend`. Dans cette dernière version, les modèles peuvent contenir des parties communes car ils peuvent résulter de l'extraction de composants à partir d'un même modèle.

5.3.3.1 L'opérateur `extend` basé sur l'union disjointe

Nous noterons que cet opérateur n'impose aucune précondition sur ses paramètres pour assurer que la vérification de ses propriétés est compositionnelle.

La préservation du typage par l'opérateur de composition `extend` de deux modèles avec plusieurs points de variations est prouvée à l'aide du théorème `Validextend` [24].

Théorème 29. (Validextend)
$\forall\ \mathtt{M_1\ M_2} \in \mathtt{Model}$, $\mathtt{o_1\ o_2} \in (\mathtt{Objects * Classes})$, $\mathtt{Label} \in \mathtt{References}$, $\mathtt{MM} \in \mathtt{MetaModel}$,
$(C : (extensibleC\ \mathtt{M_1\ M_2\ o_1\ o_2\ Label\ MM}))$,
$InstanceOf\ (\mathtt{M_1,MM})\ \wedge InstanceOf\ (\mathtt{M_2,MM})$
$\rightarrow InstanceOf\ ((compositionExtend\ \mathtt{M_1\ M_2\ o_1\ o_2\ Label\ MM}\ C), \mathtt{MM})$.

Le théorème 29 annonce que si les deux modèles en association avec les objets liés et l'étiquette pour le lien ajouté ainsi que le métamodèle choisi satisfont la précondition de l'opération `CompositionExtend` (le prédicat *extensibleC*), alors le typage est préservé dans le résultat de l'application de l'opérateur `CompositionExtend` sur ces éléments. Autrement dit, si les deux modèles sont instances du même métamodèle alors le résultat est aussi instance du même métamodèle.

subClass Le théorème `subClassExtendPreserved` montre que la propriété `subClass` est préservée par l'opérateur `compositionExtend`.

Théorème 30. (subClassExtendPreserved)

[24] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/Extend_Verif.html#Validextend

$\forall\, \mathtt{M_1}\, \mathtt{M_2} \in \mathit{Model},\, c_1\, c_2 \in \mathit{Classes},\, \mathtt{o_1}\, \mathtt{o_2} \in (\mathit{Objects} * \mathit{Classes}),\, \mathtt{MM} \in \mathit{MetaModel},\, \mathtt{Label},$
$(C : (\mathit{extensibleC}\, \mathtt{M_1}\, \mathtt{M_2}\, \mathtt{o_1}\, \mathtt{o_2}\, \mathtt{Label}\, \mathtt{MM})),$
$\mathit{subClass}\, c_1\, c_2\, \mathtt{M_1} \wedge \mathit{subClass}\, c_1\, c_2\, \mathtt{M_2}$
$\rightarrow \mathit{subClass}\, c_1\, c_2\, (\mathit{compositionExtend}\, \mathtt{M_1}\, \mathtt{M_2}\, \mathtt{o_1}\, \mathtt{o_2}\, \mathtt{Label}\, \mathtt{MM}\, C).$

Le théorème 30 annonce[25] que pour toutes classes c_1 c_2 si c_1 est une `subClass` de c_2 dans les deux modèles M_1 et M_2, alors c_1 est aussi une `subClass` de c_2 dans le modèle résultat de l'application de l'opérateur `compositionExtend`.

isAbstract La préservation de cette propriété par l'opérateur `compositionExtend` est prouvée[26] en utilisant le théorème 31. Ce théorème montre que toutes les classes abstraites dans deux modèles M_1 et M_2 sont aussi abstraites dans le modèle résultant de l'application du `compositionExtend` sur ces deux modèles.

Théorème 31. (isAbstractExtendPreserved)
$\forall\, \mathtt{M_1}\, \mathtt{M_2} \in \mathit{Model},\, c \in \mathit{Classes},\, \mathtt{o_1}\, \mathtt{o_2} \in (\mathit{Objects} * \mathit{Classes}),\, \mathtt{MM} \in \mathit{MetaModel},\, \mathtt{Label},$
$(C : (\mathit{extensibleC}\, \mathtt{M_1}\, \mathtt{M_2}\, \mathtt{o_1}\, \mathtt{o_2}\, \mathtt{Label}\, \mathtt{MM})),$
$\mathit{isAbstract}\, c\, \mathtt{M_1} \wedge \mathit{isAbstract}\, c\, \mathtt{M_2}$
$\rightarrow \mathit{isAbstract}\, c\, (\mathit{compositionExtend}\, \mathtt{M_1}\, \mathtt{M_2}\, \mathtt{o_1}\, \mathtt{o_2}\, \mathtt{Label}\, \mathtt{MM}\, C).$

lower & upper Le théorème 32 montre[27] que la propriété `lower` est préservée par l'opérateur `compositionExtend`.

Théorème 32. (lowerExtendPreserved)
$\forall\, \langle \mathtt{MV_1}, \mathtt{ME_1} \rangle\, \langle \mathtt{MV_2}, \mathtt{ME_2} \rangle \in \mathit{Model},\, c \in \mathit{Classes},\, r \in \mathit{References},$
$n \in \mathit{Natural},\, \mathtt{o_1}\, \mathtt{o_2} \in (\mathit{Objects} * \mathit{Classes}),\, \mathtt{MM} \in \mathit{MetaModel},\, \mathtt{Label},$
$(C : (\mathit{extensibleC}\, \mathtt{M_1}\, \mathtt{M_2}\, \mathtt{o_1}\, \mathtt{o_2}\, \mathtt{Label}\, \mathtt{MM})),$
$r \neq \mathtt{Label} \wedge (\mathit{lower}\, c\, r\, n\, \langle \mathtt{MV_1}, \mathtt{ME_1} \rangle) \wedge (\mathit{lower}\, c\, r\, n\, \langle \mathtt{MV_2}, \mathtt{ME_2} \rangle)$
$\rightarrow (\mathit{lower}\, c\, r\, n (\mathit{compositionExtend}\, \mathtt{M_1}\, \mathtt{M_2}\, \mathtt{o_1}\, \mathtt{o_2}\, \mathtt{Label}\, \mathtt{MM}\, C)).$

La préservation de la propriété `upper` est décrite[28] en utilisant le théorème 33 qui est similaire au théorème précédent pour la propriété `lower`.

Théorème 33. (upperExtendPreserved)
$\forall\, \langle \mathtt{MV_1}, \mathtt{ME_1} \rangle\, \langle \mathtt{MV_2}, \mathtt{ME_2} \rangle \in \mathit{Model},\, c \in \mathit{Classes},\, r \in \mathit{References},$
$n \in \mathit{Natural},\, \mathtt{o_1}\, \mathtt{o_2} \in (\mathit{Objects} * \mathit{Classes}),\, \mathtt{MM} \in \mathit{MetaModel},\, \mathtt{Label},$
$(C : (\mathit{extensibleC}\, \mathtt{M_1}\, \mathtt{M_2}\, \mathtt{o_1}\, \mathtt{o_2}\, \mathtt{Label}\, \mathtt{MM})),$
$r \neq \mathtt{Label} \wedge (\mathit{upper}\, c\, r\, n\, \langle \mathtt{MV_1}, \mathtt{ME_1} \rangle) \wedge (\mathit{upper}\, c\, r\, n\, \langle \mathtt{MV_2}, \mathtt{ME_2} \rangle)$
$\rightarrow (\mathit{upper}\, c\, r\, n (\mathit{compositionExtend}\, \langle \mathtt{MV_1}, \mathtt{ME_1} \rangle\, \langle \mathtt{MV_2}, \mathtt{ME_2} \rangle\, \mathtt{o_1}\, \mathtt{o_2}\, \mathtt{Label}\, \mathtt{MM}\, C)).$

isOpposite Le théorème 34 montre que pour chaque paire de références opposites dans les deux modèles M_1 et M_2 reste aussi opposites en appliquant l'opérateur

[25] http ://www.irit.fr/ Mounira.Kezadri/FormalMDE/Extend_Verif.html#SCEP
[26] http ://www.irit.fr/ Mounira.Kezadri/FormalMDE/Extend_Verif.html#IAEP
[27] http ://www.irit.fr/ Mounira.Kezadri/FormalMDE/Extend_Verif.html#LEP
[28] http ://www.irit.fr/ Mounira.Kezadri/FormalMDE/Extend_Verif.html#UEP

`compositionExtend` sur les deux modèles. Enfin, la propriété `isOpposite` est préservée[29].

Théorème 34. (isOppositeExtendPreserved)
$\forall\, \mathtt{M_1}\, \mathtt{M_2} \in \mathtt{Model},\ r_1\, r_2 \in \mathtt{References},\ \mathtt{o_1}\, \mathtt{o_2} \in (\mathtt{Objects} * \mathtt{Classes}),$
$(r_1 \neq \mathtt{Label}) \wedge (r_2 \neq \mathtt{Label})$
$(C : (\mathit{extensibleC}\ \mathtt{M_1}\ \mathtt{M_2}\ \mathtt{o_1}\ \mathtt{o_2}\ \mathtt{Label}\ \mathtt{MM})),$
$\wedge\ (\mathit{isOpposite}\ r_1\ r_2\ \mathtt{M_1}) \wedge (\mathit{isOpposite}\ r_1\ r_2\ \mathtt{M_2})$
$\rightarrow (\mathit{compositionExtend}\ \mathtt{M_1}\ \mathtt{M_2}\ \mathtt{o_1}\ \mathtt{o_2}\ \mathtt{Label}\ \mathtt{MM}\ C)).$

areComposite Le théorème 35 montre[30] qu'un ensemble de références qui sont composites dans les deux modèles $\mathtt{M_1}$ et $\mathtt{M_2}$ sont aussi composites dans le modèle résultat de l'application de l'opérateur `compositionExtend` sur les deux modèles.

Théorème 35. (areCompositeExtendPreserved)
$\forall\, \mathtt{M_1}\, \mathtt{M_2} \in \mathtt{Model},\ c \in \mathtt{Classes},\ R \subset \mathtt{References},\ \mathtt{o_1}\, \mathtt{o_2} \in (\mathtt{Objects} * \mathtt{Classes}),$
$(C : (\mathit{extensibleC}\ \mathtt{M_1}\ \mathtt{M_2}\ \mathtt{o_1}\ \mathtt{o_2}\ \mathtt{Label}\ \mathtt{MM})),$
$\mathtt{Label} \notin R \wedge (\mathit{areComposite}\ c\ R\ \mathtt{M_1}) \wedge (\mathit{areComposite}\ c\ r\ \mathtt{M_2})$
$\rightarrow (\mathit{areComposite}\ c\ R\ (\mathit{compositionExtend}\ \mathtt{M_1}\ \mathtt{M_2}\ \mathtt{o_1}\ \mathtt{o_2}\ \mathtt{Label}\ \mathtt{MM}\ C)).$

5.3.3.2 L'opérateur `extend` basé sur l'union classique

Nous noterons que les mêmes contraintes présentées dans 3.4 pour l'opérateur *Union* sont exigées.

La préservation du typage par l'opérateur de composition `compositionExtendC` est démontrée à l'aide du théorème 36.

Théorème 36. (ValidExtendC)
$\forall\, \mathtt{M_1}\, \mathtt{M_2} \in \mathtt{Model},\ \mathtt{o_1}\, \mathtt{o_2} \in (\mathtt{Objects} * \mathtt{Classes}),\ \mathtt{Label} \in \mathtt{References},\ \mathtt{MM} \in \mathtt{MetaModel},$
$(C : (\mathit{Extensible}\ \mathtt{M_1}\ \mathtt{M_2}\ \mathtt{o_1}\ \mathtt{o_2}\ \mathtt{Label}\ \mathtt{MM})),$
$\mathit{InstanceOf}\ (\mathtt{M_1}, \mathtt{MM})\ \wedge \mathit{InstanceOf}\ (\mathtt{M_2}, \mathtt{MM})$
$\rightarrow \mathit{InstanceOf}\ ((\mathit{compositionExtendC}\ \mathtt{M_1}\ \mathtt{M_2}\ \mathtt{o_1}\ \mathtt{o_2}\ \mathtt{Label}\ \mathtt{MM}\ C), \mathtt{MM}).$

subClass Le théorème 37 montre[31] que la propriété `subClass` est préservée par l'opérateur `compositionExtendC`. Donc, pour toutes classes $\mathtt{c_1}$ et $\mathtt{c_2}$, si $\mathtt{c_1}$ est une *subClass* de $\mathtt{c_2}$ dans les deux modèles $\mathtt{M_1}$ et $\mathtt{M_2}$, alors $\mathtt{c_1}$ est aussi une *subClass* de $\mathtt{c_2}$ dans le modèle résultant de l'application de l'opérateur `compositionExtend` sur les deux modèles.

Théorème 37. (subClassExtendCPreserved)
$\forall\, \mathtt{M_1}\, \mathtt{M_2} \in \mathtt{Model},\ c_1\, c_2 \in \mathtt{Classes},\ \mathtt{o_1}\, \mathtt{o_2} \in (\mathtt{Objects} * \mathtt{Classes}),\ \mathtt{MM} \in \mathtt{MetaModel},\ \mathtt{Label},$
$(C : (\mathit{Extensible}\ \mathtt{M_1}\ \mathtt{M_2}\ \mathtt{o_1}\ \mathtt{o_2}\ \mathtt{Label}\ \mathtt{MM})),$
$\mathit{subClass}\ c_1\ c_2\ \mathtt{M_1}\ \wedge \mathit{subClass}\ c_1\ c_2\ \mathtt{M_2}$
$\rightarrow \mathit{subClass}\ c_1\ c_2\ (\mathit{compositionExtendC}\ \mathtt{M_1}\ \mathtt{M_2}\ \mathtt{o_1}\ \mathtt{o_2}\ \mathtt{Label}\ \mathtt{MM}\ C).$

[29] http ://www.irit.fr/ Mounira.Kezadri/FormalMDE/Extend_Verif.html#IOEP
[30] http ://www.irit.fr/ Mounira.Kezadri/FormalMDE/Extend_Verif.html#ACEP
[31] http ://www.irit.fr/ Mounira.Kezadri/FormalMDE/ExtendCU_Verif.html#SCEP

Il n'y a donc aucune précondition sur cet opérateur de composition pour que la vérification de la relation `subClass` soit compositionnelle.

isAbstract La préservation de cette propriété par l'opérateur `compositionExtendC` est prouvée en utilisant le théorème 38. Ce théorème montre[32] que toutes les classes abstraites dans deux modèles $\mathtt{M}_1$ et $\mathtt{M}_2$ sont aussi abstraites dans le modèle résultant de l'application de `compositionExtendC` sur ces deux modèles.

Théorème 38. (isAbstractExtendCPreserved)
$\forall\, \mathtt{M}_1\, \mathtt{M}_2 \in \mathtt{Model},\, c \in \mathtt{Classes},\, \mathtt{o}_1\, \mathtt{o}_2 \in\, (\mathtt{Objects} * \mathtt{Classes}),\, \mathtt{MM} \in \mathtt{MetaModel},\, \mathtt{Label},$
$(C : (\mathit{Extensible}\, \mathtt{M}_1\, \mathtt{M}_2\, \mathtt{o}_1\, \mathtt{o}_2\, \mathtt{Label}\, \mathtt{MM})),$
$\mathit{isAbstract}\ c\ \mathtt{M}_1 \wedge \mathit{isAbstract}\ c\ \mathtt{M}_2$
$\rightarrow \mathit{isAbstract}\ c\ (\mathit{compositionExtendC}\, \mathtt{M}_1\, \mathtt{M}_2\, \mathtt{o}_1\, \mathtt{o}_2\, \mathtt{Label}\, \mathtt{MM}\, C).$

Il n'y a donc aucune précondition sur cet opérateur de composition pour que la vérification de la relation `isAbstract` soit compositionnelle.

lower & upper Le théorème 39 montre[33] que la propriété `lower` est préservée par l'opérateur `compositionExtendC`. Des hypothèses sont nécessaires contrairement à l'opérateur précédent car l'opérateur peut s'appliquer à deux modèles ayant des éléments communs et le résultat peut être un modèle non conforme au métamodèle.

Théorème 39. (lowerExtendCPreserved)
$\forall\, \langle \mathtt{MV}_1, \mathtt{ME}_1 \rangle\, \langle \mathtt{MV}_2, \mathtt{ME}_2 \rangle \in \mathtt{Model},\, c \in \mathtt{Classes},\, r \in \mathtt{References},$
$n \in \mathtt{Natural},\, \mathtt{o}_1\, \mathtt{o}_2 \in (\mathtt{Objects} * \mathtt{Classes}),\, \mathtt{MM} \in \mathtt{MetaModel},\, \mathtt{Label},$
$(C : (\mathit{Extensible}\, \mathtt{M}_1\, \mathtt{M}_2\, \mathtt{o}_1\, \mathtt{o}_2\, \mathtt{Label}\, \mathtt{MM})),$
$(\forall (o, c_1) \in (\mathtt{MV}_1 \cup \mathtt{MV}_2), r \neq \mathtt{Label} \wedge (\mathit{lowerCond}\ c\ r\ n\ \langle \mathtt{MV}_1, \mathtt{ME}_1 \rangle\, \langle \mathtt{MV}_2, \mathtt{ME}_2 \rangle)$
$\wedge (\mathit{lower}\ c\ r\ n\ \langle \mathtt{MV}_1, \mathtt{ME}_1 \rangle) \wedge (\mathit{lower}\ c\ r\ n\ \langle \mathtt{MV}_2, \mathtt{ME}_2 \rangle)$
$\rightarrow (\mathit{lower}\ c\ r\ n\ (\mathit{compositionExtendC}\, \mathtt{M}_1\, \mathtt{M}_2\, \mathtt{o}_1\, \mathtt{o}_2\, \mathtt{Label}\, \mathtt{MM}\, C)).$

La preuve nécessite une hypothèse particulière qui est que le cardinal n doit être supérieur ou égal au cardinal dans l'intersection des deux modèles. La condition est exprimée pour les deux modèle $\langle \mathtt{MV}_1, \mathtt{ME}_1 \rangle\, \langle \mathtt{MV}_2, \mathtt{ME}_2 \rangle$ comme suit :

$$\mathit{lowerCond}(\mathtt{c} \in \mathit{Classes},\, \mathtt{r} \in \mathit{References},\, \mathtt{n} \in \mathit{Natural}^{\top}) \triangleq$$
$$\mathtt{n} \geq |\{\mathtt{o}_2 \in (\mathtt{MV}_1 \cap \mathtt{MV}_2) \mid \langle \langle \mathtt{o}, \mathtt{c} \rangle, \mathtt{r}, \mathtt{o}_2 \rangle \in (\mathtt{ME}_1 \cap \mathtt{ME}_2)\}|$$

La préservation de la propriété `upper` est décrite[34] en utilisant le théorème 40 qui est similaire au théorème précédent pour la propriété `lower`.

Théorème 40. (upperExtendCPreserved)

[32] http ://www.irit.fr/ Mounira.Kezadri/FormalMDE/ExtendCU.html#IAEP
[33] http ://www.irit.fr/ Mounira.Kezadri/FormalMDE/ExtendCU.html#LEP
[34] http ://www.irit.fr/ Mounira.Kezadri/FormalMDE/ExtendCU.html#UEP

$\forall \langle \mathtt{MV_1,ME_1} \rangle \ \langle \mathtt{MV_2,ME_2} \rangle \in \mathit{Model},\ c \in \mathit{Classes},\ r \in \mathit{References},$
$n \in \mathit{Natural},\ \mathtt{o_1}\ (\mathtt{o_2} \in (\mathit{Objects} * \mathit{Classes}),\ \mathtt{MM} \in \mathit{MetaModel},\ \mathtt{Label},$
$(\forall (o, c_1) \in (\mathtt{MV_1} \cup \mathtt{MV_2}),$
$c = c_1 \to (\mathit{upperCond}\ c\ r\ n\ \langle \mathtt{MV_1,ME_1} \rangle\ \langle \mathtt{MV_2,ME_2} \rangle)$
$\wedge (\mathit{upper}\ c\ r\ n\ \langle \mathtt{MV_1,ME_1} \rangle) \wedge (\mathit{upper}\ c\ r\ n\ \langle \mathtt{MV_2,ME_2} \rangle)$
$\to (\mathit{upper}\ c\ r\ n\ (\mathit{compositionExtendC}\ \langle \mathtt{MV_1,ME_1} \rangle\ \langle \mathtt{MV_2,ME_2} \rangle\ \mathtt{o_1}\ \mathtt{o_2}\ \mathtt{Label}\ \mathtt{MM})).$

L'hypothèse assurant la préservation des propriétés sémantiques est que le cardinal n associé à la relation r en relation avec la classe c doit être strictement supérieur à la somme de ses cardinaux dans les deux modèles moins son cardinal dans le modèle d'intersection. Donc, le cardinal n doit satisfaire la condition `upperCond` pour les deux modèles $\langle \mathtt{MV_1,ME_1} \rangle\ \langle \mathtt{MV_2,ME_2} \rangle$:

$$\begin{aligned}&\mathit{upperCond}(\mathtt{c} \in \mathit{Classes},\ r \in \mathit{References},\ \mathtt{n} \in \mathit{Natural}^{\top}) \triangleq\\ &\mathtt{n} > |\{\mathtt{o_2} \in \mathtt{MV_1} \mid \langle\langle \mathtt{o,c}\rangle, \mathtt{r}, \mathtt{o_2}\rangle \in \mathtt{ME_1}\}| + |\{\mathtt{o_2} \in \mathtt{MV_2} \mid \langle\langle \mathtt{o,c}\rangle, \mathtt{r}, \mathtt{o_2}\rangle \in \mathtt{ME_2}\}|\\ &\quad -|\{\mathtt{o_2} \in (\mathtt{MV_1} \cap \mathtt{MV_2}) \mid \langle\langle \mathtt{o,c}\rangle, \mathtt{r}, \mathtt{o_2}\rangle \in (\mathtt{ME_1} \cap \mathtt{ME_2})\}|\end{aligned}$$

Cette précondition pour la composition est naturelle pour préserver le cardinal de la relation. Il est donc nécessaire d'introduire des préconditions sur cet opérateur pour que la vérification des propriétés `lower` et `upper` soit compositionnelle.

isOpposite Le théorème 41 montre que chaque paire de références opposites dans les deux modèles $\mathtt{M_1}$ et $\mathtt{M_2}$ restent aussi opposites en appliquant la fonction `compositionExtendC` sur les deux modèles. Enfin, la propriété `isOpposite` est préservée[35].

Théorème 41. (isOppositeExtendCPreserved)
$\forall\ \mathtt{M_1}\ \mathtt{M_2} \in \mathit{Model},\ r_1\ r_2 \in \mathit{References},\ \mathtt{o_1}\ \mathtt{o_2} \in (\mathit{Objects} * \mathit{Classes}),$
$(C : (\mathit{extensibleC}\ \mathtt{M_1}\ \mathtt{M_2}\ \mathtt{o_1}\ \mathtt{o_2}\ \mathtt{Label}\ \mathtt{MM})),$
$(r_1 \neq \mathtt{Label}) \wedge (r_2 \neq \mathtt{Label})$
$\wedge\ (\mathit{isOpposite}\ r_1\ r_2\ \mathtt{M_1}) \wedge (\mathit{isOpposite}\ r_1\ r_2\ \mathtt{M_2})$
$\to (\mathit{isOpposite}\ r_1\ r_2\ (\mathit{compositionExtendC}\ \mathtt{M_1}\ \mathtt{M_2}\ \mathtt{o_1}\ \mathtt{o_2}\ \mathtt{Label}\ \mathtt{MM}\ C)).$

Il n'est donc pas nécessaire d'ajouter des préconditions sur cet opérateur de composition pour que la vérification de la relation `isOpposite` soit compositionnelle.

areComposite Le théorème 42 montre[36] qu'un ensemble de références qui sont composites dans deux modèles $\mathtt{M_1}$ et $\mathtt{M_2}$ sont aussi composites dans le modèle résultat de l'application de `compositionExtendC` sur les deux modèles.

Théorème 42. (areCompositeExtendCPreserved)
Theorem *areCompositeExtendCPreserved* :
$\forall\ \mathtt{M_1}\ \mathtt{M_2} \in \mathit{Model},\ c \in \mathit{Classes},\ R \subset \mathit{References},\ \mathtt{o_1}\ (\mathtt{o_2}(\mathit{Objects} * \mathit{Classes}),$
$(C : (\mathit{extensibleC}\ \mathtt{M_1}\ \mathtt{M_2}\ \mathtt{o_1}\ \mathtt{o_2}\ \mathtt{Label}\ \mathtt{MM})),$
$\mathtt{Label} \notin R\ \wedge\ \mathit{areCompositeCond}((\mathtt{o_1}, c_1)\ (\mathtt{o_2}, c_2),\ R)$
$\wedge\ (\mathit{areComposite}\ c\ R\ \mathtt{M_1}) \wedge (\mathit{areComposite}\ c\ R\ \mathtt{M_2})$
$\to (\mathit{areComposite}\ c\ R\ (\mathit{compositionExtendC}\ \mathtt{M_1}\ \mathtt{M_2}\ \mathtt{o_1}\ \mathtt{o_2}\ \mathtt{Label}\ \mathtt{MM}\ C)).$

[35] http ://www.irit.fr/ Mounira.Kezadri/FormalMDE/ExtendCU.html#IOEP
[36] http ://www.irit.fr/ Mounira.Kezadri/FormalMDE/ExtendCU.html#ACEP

La condition AreCompositeCond pour les deux modèles $\langle \mathtt{MV_1,ME_1} \rangle$ $\langle \mathtt{MV_2,ME_2} \rangle$:

$$areCompositeCond((o_1, c_1)\ (o_2, c_2) \in (\mathtt{Objects} * \mathtt{Classes}),\ R \in References) \triangleq$$
$$|\{\mathtt{M_2} \in \mathtt{MV_1} \mid \langle\langle \mathtt{o}, \mathtt{c_1}\rangle, \mathtt{R}, \mathtt{M_2}\rangle \in \mathtt{ME_1}\}| + |\{\mathtt{M_2} \in \mathtt{MV_2} \mid \langle\langle \mathtt{o}, \mathtt{c_1}\rangle, \mathtt{R}, \mathtt{M_2}\rangle \in \mathtt{ME_2}\}| < 1$$

Il est donc nécessaire d'ajouter des préconditions sur cet opérateur de composition pour que la vérification de la relation areComposite soit compositionnelle.

5.3.4 L'extraction de l'interface des fragments

Les propriétés définies précédemment sont aussi préservées en appliquant l'extraction de l'interface des fragments. Nous annonçons les théorèmes correspondants dans ce qui suit et les preuves pour les théorèmes des 6 propriétés sont accessibles sur la page web Formal Assembly[37]. Nous noterons qu'aucune précondition supplémentaire n'est nécessaire pour assurer que la vérification est compositionnelle pour les 6 propriétés suivantes.

subClass Le théorème 43 montre[38] que la propriété subClass est préservée par l'opérateur fragmentExtractionP. Donc, pour toutes classes $\mathtt{c_1}$, $\mathtt{c_2}$ et pour toute liste l de correspondance de points (variation/référence), si $\mathtt{c_1}$ est une *subClass* de $\mathtt{c_2}$ dans un modèle M, alors $\mathtt{c_1}$ est aussi une *subClass* de $\mathtt{c_2}$ dans ($fragmentExtractionP$ M o).

Théorème 43. (ExtractPresSub)
$\forall\, \mathtt{M} \in \mathtt{Model},\ c_1\ c_2 \in \mathtt{Classes},\ o \in \mathtt{Objects},$
$c_1 \notin \{Hook,\ Prototype,\ AbsAllClasses,\ ROV\}$
$\wedge(NotExtendedVPM\ \mathtt{M}\ o) \wedge (NotExtendedEPM\ \mathtt{M}\ o)$
$\wedge\ subClass\ c_1\ c_2\ M \rightarrow subClass\ c_1\ c_2\ (fragmentExtractionP\ \mathtt{M}\ o).$

NotExtendedVPM M o et *NotExtendedEPM* M o vérifient que l'objet o n'est pas déjà déclaré comme point de variation dans l'ensemble des nœuds et d'arcs du modèle M.

isAbstract La préservation de cette propriété par l'opérateur fragmentExtractionP est prouvée[39] en utilisant le théorème 44. Ce théorème montre que toutes les classes abstraites dans un modèle M sont aussi abstraites dans le modèle résultant de l'application du fragmentExtractionP sur ce modèle.

Théorème 44. (ExtractPresIsAbstract)
$\forall \mathtt{M} \in \mathtt{Model},\ c \in \mathtt{Classes},\ o \in \mathtt{Objects},$
$c_1 \notin \{Hook,\ Prototype,\ AbsAllClasses,\ ROV\}$
$\wedge\ (NotExtendedVPM\ \mathtt{M}\ o) \wedge (NotExtendedEPM\ \mathtt{M}\ o)$
$\wedge\ isAbstract\ c\ \mathtt{M} \rightarrow isAbstract\ c\ (fragmentExtractionP\ \mathtt{M}\ o).$

[37] http://www.irit.fr/~Mounira.Kezadri/FormalAssembly.html
[38] http ://www.irit.fr/ Mounira.Kezadri/FormalMDE/Frag_Ext_Verif.html#ExtPS
[39] http ://www.irit.fr/ Mounira.Kezadri/FormalMDE/Frag_Ext_Verif.html#ExtPIA

lower & upper Le théorème 45 montre[40] que la propriété `lower` est préservée par l'opérateur `fragmentExtractionP`.

Théorème 45. (lowerExtPreserved)
$\forall \langle \mathtt{MV}, \mathtt{ME} \rangle \in \mathtt{Model},\ c \in \mathtt{Classes},\ r \in\ \mathtt{References},\ n \in \mathtt{Natural},\ o \in \mathtt{Objects},$
$c \notin \{Hook,\ Prototype,\ AbsAllClasses,\ ROV\} \wedge r \notin \{bind, inh\}$
$\wedge\ (NotExtendedVPM\ \mathtt{M}\ o) \wedge (NotExtendedEPM\ \mathtt{M}\ o)$
$\wedge\ (lower\ c\ r\ n\ \langle \mathtt{MV}, \mathtt{ME} \rangle) \rightarrow (lower\ c\ r\ n(fragmentExtractionP\ \mathtt{M}\ o)).$

La préservation de la propriété `upper` est décrite[41] en utilisant le théorème 46 qui est similaire au théorème précédent pour la propriété `lower`.

Théorème 46. (upperExtPreserved)
$\forall \langle \mathtt{MV}, \mathtt{ME} \rangle \in \mathtt{Model},\ c \in \mathtt{Classes},\ r \in\ \mathtt{References},\ n \in \mathtt{Natural},\ o \in \mathtt{Objects},$
$c \notin \{Hook,\ Prototype,\ AbsAllClasses,\ ROV\} \wedge r \notin \{bind, inh\}$
$\wedge\ (NotExtendedVPM\ \mathtt{M}\ o) \wedge (NotExtendedEPM\ \mathtt{M}\ o)$
$\wedge\ (upper\ c\ r\ n\ \langle \mathtt{MV}, \mathtt{ME} \rangle) \rightarrow (upper\ c\ r\ n(fragmentExtractionP\ \mathtt{M}\ o)).$

isOpposite Le théorème 47 montre[42] que chaque paire de références opposites dans les deux modèles $\mathtt{M}_1$ et $\mathtt{M}_2$ restent aussi opposites en appliquant la fonction `fragmentExtractionP` sur ce modèle. Enfin, la propriété `isOpposite` est préservée.

Théorème 47. (isOppositePreserved)
$\forall \mathtt{M} \in \mathtt{Model},\ r_1\ r_2 \in \mathtt{References},\ o \in \mathtt{Objects},$
$\wedge\ (NotExtendedVPM\ \mathtt{M}\ o) \wedge (NotExtendedEPM\ \mathtt{M}\ o)\ \wedge\ r_1, r_2 \notin \{Mbind, Minh\}$
$\wedge\ (isOpposite\ r_1\ r_2\ M) \rightarrow (isOpposite\ r_1\ r_2(fragmentExtractionP\ \mathtt{M}\ o)).$

areComposite Le théorème `areCompositeExtPreserved` montre[43] que les références composites dans un modèle modèle `M` restent composites dans le modèle résultat de l'application du `fragmentExtractionP` sur ce modèle.

Théorème 48. (areCompositeExtPreserved)
$\forall \mathtt{M} \in \mathtt{Model},\ c \in \mathtt{Classes},\ r \subset \mathtt{References},\ o \in \mathtt{Objects},$
$\wedge\ (NotExtendedVPM\ \mathtt{M}\ o) \wedge (NotExtendedEPM\ \mathtt{M}\ o)$
$c \notin \{Hook,\ Prototype,\ AbsAllClasses,\ ROV\} \wedge (inh, bind \notin r)$
$\wedge\ (areComposite\ c\ r\ M) \rightarrow (areComposite\ c\ r\ (fragmentExtractionP\ \mathtt{M}\ o)).$

Dans la première version de ISC [Aßm03], la notion de conformité ou la consistance est restreinte à la propriété *instanceOf* définie dans 3. Un opérateur de composition est sûr s'il permet de préserver la consistance (Theorème 5.1 de [Aßm03]). Un opérateur d'extension est sûr s'il permet juste d'ajouter des fragments qui sont indépendant du modèles (Théorème 5.2 de [Aßm03]), un exemple d'opérateur qui satisfait cette condition est l'opérateur Union disjointe (Section 5.3.3.1).

[40] http ://www.irit.fr/ Mounira.Kezadri/FormalMDE/Frag_Ext_Verif.html#LExtP
[41] http ://www.irit.fr/ Mounira.Kezadri/FormalMDE/Frag_Ext_Verif.html#UExtP
[42] http ://www.irit.fr/ Mounira.Kezadri/FormalMDE/Frag_Ext_Verif.html#IOExtP
[43] http ://www.irit.fr/ Mounira.Kezadri/FormalMDE/Frag_Ext_Verif.html#ACExtP

Dans la dernière version de ISC [Joh11] implémentée dans l'atelier REUSEWARE comme plugin Eclipse et développée en parallèle avec nos travaux, la notion du bon typage est vérifiée pour les métamodèles par rapport à EMOF. REUSEWARE manipule des métamodèles avec une syntaxe concrète et dont la syntaxe abstraite est EMOF. Cette version peut vérifier que le modèle résultat est toujours conforme à EMOF. Elle introduit le concept de compatibilité entre les points de variation (compatibilité des types), cette version d'ISC ne vérifie que les contraintes de typage et ne considère donc pas les propriétés sémantiques.

Notre démarche est originale par rapport aux travaux d'Aßman, nous fournissons a priori les préconditions qui assurent que le résultat de l'application d'un opérateur est valide (typage et contraintes sémantiques). Nous n'avons pas besoin de vérifier pour chaque application que le résultat est valide mais nous connaissons les préconditions qu'il faut respecter et si nos conditions sont satisfaites, nous pouvons assurer que le résultat de la composition est consistant.

5.4 Conclusion

à partir du cadre formel des concepts de l'IDM COQ4MDE, nous avons abordé le problème de la composition. En s'inspirant de la méthode générique de composition ISC et de l'atelier de composition REUSEWARE, nous avons d'abord proposé une extension des métamodèles permettant d'exprimer des modèles avec des interfaces de composition, nous avons défini par la suite des opérateurs pour l'extraction et la composition de fragments de modèles extraits. Ceci a donné lieu à une formalisation des opérateurs élémentaires permettant l'extraction et la composition de modèles. Toutes ces notions ont été implémentées dans l'assistant de preuve COQ suivant la même ligne directrice des travaux précédents. Nous avons commencé nos travaux avec cette formalisation des opérateurs de la méthode ISC et les preuves de préservation de leurs propriétés, nous avons défini par la suite dans le chapitre 3 des opérateurs élémentaires dont nous avons montrer l'utilisation pour définir les opérateurs d'ISC dans ce chapitre et le Package Merge dans le chapitre 4.

Cette intégration permet d'extraire des fonctions exécutables correctes par construction pour les différentes opérations relatives à la composition de modèles. La terminaison des opérations d'extraction et de composition est assurée par la définition en COQ. Les propriétés de typage ainsi qu'un ensemble de propriétés relatives au métamétamodèle EMOF sont prouvées préservées par les opérateurs de composition en introduisant pour certaines propriétés des préconditions sur les paramètres des opérateurs de composition. L'application n'est pas limitée à un langage spécifique mais peut s'étendre à tous les modèles et langages de modélisation définis par des métamodèles exprimés en MOF.

L'ensemble des opérateurs élémentaires de la méthode ISC formalisés dans ce chapitre sont : `bind` et `extend`. Deux autres opérateurs sont présentés comme élémentaires dans [Aßm03] (`rename` et `copy`). L'opérateur `rename` peut renommer un point de variation, dans notre cas il peut être implémenté à l'aide de l'opérateur substitution appliqué sur un modèle contenant une interface de composition. L'opérateur `copy` permet de copier un clone avec un autre nom et peut être implémenté à l'aide de l'opérateur union. Ces deux opérateurs peuvent être formalisés et vérifiés suivant le même schéma que les opérateurs `bind` et

extend.

à partir des opérateurs élémentaires de la méthode ISC (bind et extend) définis dans ce chapitre, des opérateurs complexes ont été construits. Les opérateurs composés permettent de faire des transformations plus complexes comme par exemple lier plusieurs points de variation en même temps par l'opérateur bind avec plusieurs points de variation.

Dans la méthode ISC [Aßm03] définie sur des fragments de code Java, l'opérateur d'extension garantit par définition qu'il ne va pas modifier le code du fragment, mais il peut changer sa sémantique. La sémantique est préservée si le code ajouté au point de variation est indépendant du code du fragment (Théorème 5.2 dans [Aßm03]). Nous avons prouvé que la sémantique est préservée dans le cas où les modèles sont disjoints mais nous avons prolongé ces travaux en proposant les préconditions les plus faibles possibles qui préservent la sémantique dans le cas non disjoint en passant par la preuve formelle de tous les théorèmes dans l'assistant de preuve COQ.

Cette proposition est une étape nécessaire dans la formalisation des techniques de composition. La prochaine étape de notre travail est de formaliser la notion de vérification du modèle en s'appuyant sur plusieurs cas d'utilisation, dans une première étape, des contraintes statiques telles que les contraintes OCL et par la suite des propriétés plus dynamiques telles que l'absence d'inter-blocage proposé dans le cadre de l'atelier BIP. Le résultat attendu de notre travail est de définir un cadre pour déterminer les préconditions minimales nécessaires pour les opérateurs de composition pour assurer que la vérification d'une propriété est compositionnelle en s'appuyant sur la construction de la preuve qu'elle l'est effectivement, c'est-à-dire que les préconditions permettent d'assurer comme postcondition la satisfaction de la propriété souhaitée.

Le codage des modèles exploité par COQ4MDE nous a permis de mener à bout nos expériences. Les preuves réalisées sont toutefois assez lourdes sans être complexes. L'état de l'art de l'exploitation des formalismes catégoriques en informatique a montré que ceux-ci permettent généralement de simplifier les preuves en réutilisant les propriétés des opérateurs catégoriques. Nous nous sommes donc intéressés à un autre codage des modèles issu de la théorie des catégories, la cadre catégorique offre une description mathématique élégante et intègre naturellement des opérateurs riches.

6 Perspectives : Vers une catégorie de modèles

Table des matières

6.1 Une catégorie pour les modèles . . . 116
6.1.1 Une première approche intuitive . . . 118
6.1.2 Une seconde approche . . . 119
6.2 Proposition d'interprétation des opérateurs algébriques . . . 122
6.3 Conclusion . . . 127
6.4 Bilan des travaux présentés . . . 129
6.4.1 La formalisation syntaxique . . . 129
6.4.2 La formalisation sémantique . . . 130
6.5 Perspectives . . . 130
6.5.1 Compatibilité de composants à base d'ontologies . . . 130
6.5.2 Compatibilité comportementale lors de l'assemblage . . . 131
6.5.3 Modélisation d'autres opérateurs . . . 132
6.5.4 Opérateurs sur les ontologies . . . 132
A.1 L'union . . . 137
A.2 Les ensembles disjoints . . . 137
A.3 L'intersection . . . 138
A.4 La différence . . . 138
A.5 La substitution . . . 139

Les travaux réalisés dans les chapitres précédents reposent sur des preuves complexes sans être mathématiquement difficiles. Ceci est lié à la représentation des modèles comme des types dépendants dont chaque manipulation impose la preuve de la construc-

tion d'un type modèle. Le raisonnement sur ces types est assez délicat notamment concernant l'égalité. D'autres approches ont été envisagées pour représenter les modèles comme [PM11] et [GSMP11] mais elles n'éliminent pas complètement ces difficultés et en introduisent d'autres. Un autre aspect est que le cadre défini n'intègre pas naturellement des opérateurs riches (nous devons définir et formaliser tout nos opérateurs avec la possibilité de construction d'opérateurs de haut niveau à partir d'opérateurs formalisés). Nous avons donc envisagé d'expérimenter l'utilisation d'un cadre fondé sur la théorie des catégories [HS73] [Pie91] [Gol06] avec sa gamme d'opérateurs algébriques.

Ce chapitre décrit les résultats préliminaires à l'implémentation d'une catégorie de modèles et l'étude de l'interprétation des opérateurs algébriques classiques dans notre contexte. Ce cadre catégorique permet d'améliorer l'expression et la compréhension des opérateurs élémentaires, d'avoir un cadre commun permettant d'exprimer et de formaliser des opérateurs plus riches d'une manière plus naturelle.

6.1 Une catégorie pour les modèles

Nous présentons une définition générique de la notion de catégorie empruntée à Awodey [Awo10].

Dé nition 5. *Une catégorie est composée des éléments suivants :*

- *Des objets : A, B, C, ...*
- *Des èches : f, g, h, ...*
- *Chaque èche f a deux composantes nommées successivement le* `domaine` *et le* `codomaine` *de f : $dom(f)$ et $codom(f)$.*

  Nous écrivons : $f : A \rightarrow B$ pour indiquer que $A = dom(f)$ et $B = codom(f)$. Ceci est représenté graphiquement par :

$$A \bullet\!\xrightarrow{f} B$$

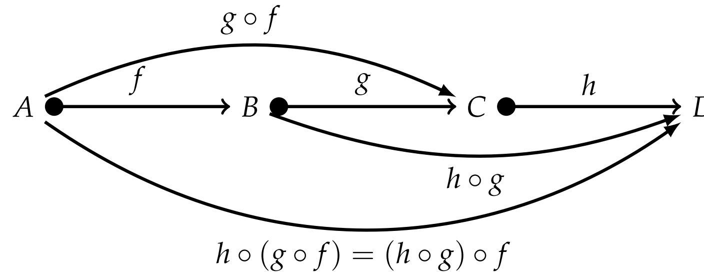
- *Étant donnés deux èches : $f : A \rightarrow B$ et $g : B \rightarrow C$, avec $codom(f) = dom(g)$, il existe une èche : $g \circ f : A \rightarrow C$ nommée la composition de f et g ($\circ$ est l opération de composition).*

$$A \bullet\!\xrightarrow{f} B \bullet\!\xrightarrow{g} C \qquad (g \circ f : A \rightarrow C)$$

- *Pour chaque objet A, il existe une èche identité : $1_A : A \rightarrow A$.*

  Ces données doivent satisfaire les règles :

– *L associativité à gauche et à droite de la composition :*
Pour tous morphismes (èches) $f: A \to B$, $g: B \to C$, $h: C \to D$, *on a :* $h \circ (g \circ f) = (h \circ g) \circ f$.



– *La composition avec des morphismes identité :*
Pour chaque $f: A \to B$: $f \circ 1_A = f = 1_B \circ f$.

Toute structure qui satisfait ces conditions est une catégorie.

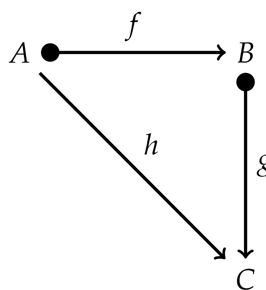
Dé nition 6. *Un foncteur* $F : C \to D$ *entre une catégorie C et une catégorie D est une fonction des objets de C vers les objets de D et des èches de C vers les èches de D tel que :*

▷ $F(f : A \to B) = F(f) : F(A) \to F(B)$

▷ $F(g \circ f) = F(g) \circ F(f)$

▷ $F(1_A) = 1_{F(A)}$

Remarque 6.1. *Les diagrammes sont utilisés pour montrer que la composition des morphismes commute. Par exemple le schéma ci-dessous commute si pour tous les morphismes* f *et* g, *il existe un morphisme* h *tel que :* $h = g \circ f$.



Une catégorie offre de nombreuses constructions qui sont des instances spécifiques de notions plus génériques qui sont les limites et les colimites de diagrammes (produit/coproduit, égalisateur/coégalisateur, pullback/pushout) et des théorèmes associés qui sont exploitables dans toutes les catégories. Le bon choix des objets et des èches permet ensuite de réutiliser ces constructions et propriétés.

La définition d'une catégorie de modèles requiert la définition des objets, des èches et la preuve des propriétés associées. Selon nos ré exions, deux choix sont possibles que nous détaillons dans ce qui suit avec pour chaque solution ses avantages et ses inconvénients.

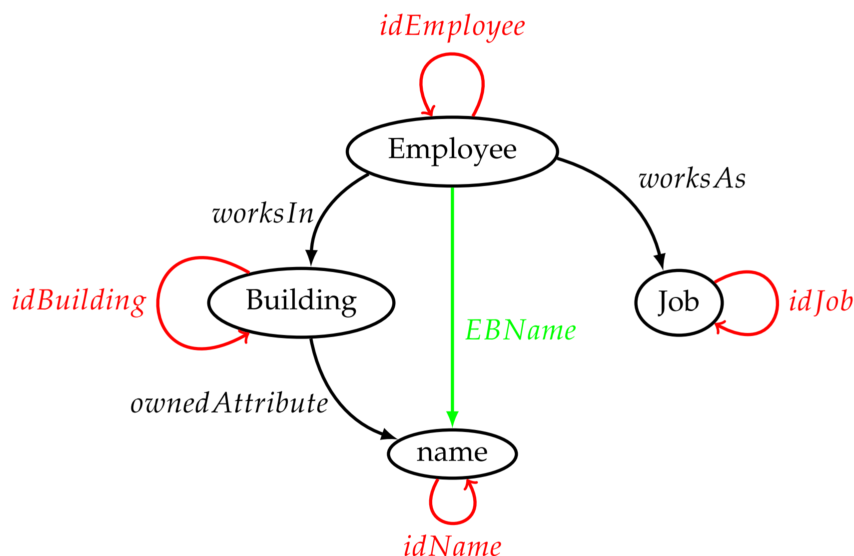
Remarque 6.2. *Les modèles sont dé nis comme des multigraphes, donc, nous parlons d un modèle ou d un graphe pour faire référence au graphe qui représente ce même modèle.*

6.1.1 Une première approche intuitive

Pour un graphe donné, nous pouvons considérer les nœuds comme les objets de la catégorie et les arcs comme les morphismes associés. La manipulation du graphe est plus simple car elle évite dans un premier temps les complications liées à l'utilisation des types dépendants. Mais cette solution impose l'ajout de morphismes supplémentaires pour obtenir une catégorie. En effet, un graphe quelconque ne constitue pas une catégorie car deux propriétés ne sont satisfaites : l'identité pour chaque nœud et la composition entre morphismes. Une solution possible consiste à construire la fermeture transitive ré exive des morphismes et calculer la plus petite catégorie à partir du graphe.

Voici un exemple d'application de cette démarche pour construire une catégorie pour le modèle présenté dans 4.13. La catégorie correspondante peut être construite comme suit :

1. représenter `Employee`, `Building`, `Job` et `name` comme des objets ;
2. considérer `worksIn`, `worksAs` et `ownedAttribute` comme étant des morphismes ;
3. ajouter les èches d'identité pour chaque objet (en rouge ci-dessous) ;
4. pour chaque paire de èches composables, ajouter une èche pour la fonction construite par leur composition si celle-ci n'existe pas déjà (en vert ci-dessous).



Les liens de composition et d'identité selon cette proposition ne sont pas des liens fictifs mais sont considérés comme des requêtes sur la structure du graphe qui indiquent pour un nœud donné, l'ensemble des nœuds accessibles transitivement et ré exivement. Un autre inconvénient pour cette solution est qu'il est difficile de garder les propriétés du modèle complet comme les objets manipulés sont les nœuds, notamment cette structure ne comporte

pas en elle-même les propriétés de correction d'un modèle par exemple le fait qu'un lien ne doit exister que si ses objets source et cible sont dans le modèle. Nous présentons alors une deuxième approche en s'appuyant sur des constructions catégoriques.

6.1.2 Une seconde approche

Il s'agit de considérer les modèles comme les objets de la catégorie et les morphismes de modèles comme les èches. La notion de morphisme est très générale, dans notre contexte nous considérons des applications entre les modèles. C'est l'approche adoptée dans [LCR+97] [EHK+96] [PP93]. Cette solution présente également des avantages et des inconvénients. Parmi ses avantages, nous pouvons utiliser la structure de modèle définie dans Coq4MDE et les opérateurs définis au niveau modèle (les caractéristiques sémantiques du modèle sont conservées par la structure). Par contre, comme les objets sont des modèles, l'utilisation des types dépendant n'est pas évitée. Concernant la représentation des structures algébriques en Coq, plusieurs solutions existent comme : les enregistrement (Record) [GGMR09] et les classes typées (Class) [SO08]. Nous avons choisi de définir la catégorie de modèles en se basant sur une formalisation de la métathéorie des catégories en Coq[1]. Cette formalisation utilise le type Class et elle est utilisée pour la définition d'un foncteur (Définition 6.1) pour une transformation décrite dans [Meg12].

Les objets de la catégorie : Dans cette version, les objets de la catégorie sont des paires constituées de l'ensemble des nœuds et des liens tels que cette paire constitue un graphe. L'objet graphe est défini comme $tvses = (vs, es)$ tel que vs est l'ensemble des nœuds et es celui des liens. Pour chaque lien $e \in es$, $e = ((e_1, e_2), a)$ est tel que $e_1 \in vs$ et $e_2 \in vs$. Les objets sont définis comme suit :

```
Definition OModele:= {tvses: (uniset GE.Vertex * uniset GE.edge) |
  forall (e: GE.edge),
    let (vs, es) := tvses in
      let (edg, a) := e in let (e1, e2) := edg in
        In es e
        -> In vs e1
        /\ In vs e2}%type.
```

Les homomorphismes de la catégorie : Un morphisme de modèle est défini par un triplet de fonctions $mapve = (mapv, mape, mapa)$ avec une contrainte qui permet de vérifier que l'application des fonctions $mapv$, $mape$ et $mapa$ préserve la structure du modèle. C'est une fonction totale d'un graphe simple vers un autre graphe simple définie comme suit :

```
Definition MorModele :
  OModele -> OModele -> Type :=
  fun g g' => {map: (((GE.Vertex) -> (GE.Vertex)) * ((GE.edge) -> (GE.edge)) * ((GE.Arc) ->
        (GE.Arc)))|
    forall (e: GE.edge),
      let (mapve, mapa):= map in let (mapv, mape) := mapve in
        let (edg, a) := e in let (e1, e2) := edg in
          match g, g' with
            exist (vs, es) graph, exist (vs', es') graph' =>
```

[1] http://www.cs.berkeley.edu/~megacz/coq-categories/

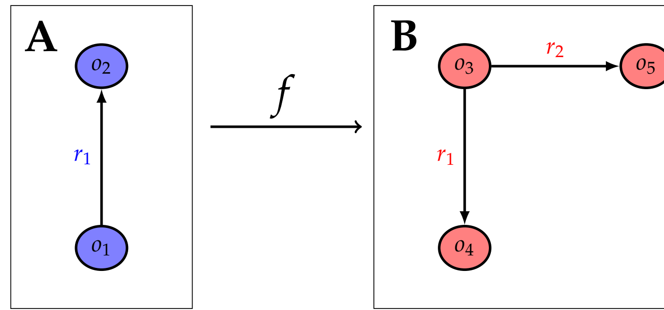


Figure 6.1 — Deux modèles A et B et un morphisme f entre les deux

```
        In es e ->
        (fst (fst (mape e)) = mapv e1)
        /\ (snd (fst (mape e)) = mapv e2)
        /\ In es' (mape e)
      end}.
```

Exemple 6.1. *Voici un exemple de deux objets A et B de la catégorie des modèles : Un morphisme mapve possible entre les deux objets A et B peut être dé ni avec le triplet de fonctions $(mapv, mape, mapa)$ suivantes :*

$$f = \begin{cases} mapv = \{o_1 \rightarrow o_3, o_2 \rightarrow o_4\} \\ mape = \{((o_1, o_2), r_1) \rightarrow ((o_3, o_4), r_1)\} \\ mapa = \{r_1 \rightarrow r_1\} \end{cases}$$

Un autre morphisme possible est :

$$f = \begin{cases} mapv = \{o_1 \rightarrow o_3, o_2 \rightarrow o_5\} \\ mape = \{((o_1, o_2), r_1) \rightarrow ((o_3, o_5), r_2)\} \\ mapa = \{r_1 \rightarrow r_2\} \end{cases}$$

Le premier morphisme satisfait bien la dé nition de $MorModele$ car : pour l arc $((o_1, o_2), r_1)$ du modèle A, $mape((o_1, o_2), r_1) = ((mapv\ o_1, mapv\ o_2), mapa\ r_1)$ et $mape((o_1, o_2), r_1)$ appartient aux arcs de B. Le deuxième morphisme satisfait aussi la dé nition de $MorModele$.

Remarque 6.3. *Plusieurs homomorphismes de graphes peuvent être dé nis, notamment toutes les fonctions de type (OModele → OModele → Type) peuvent être considérées. Parmi les propriétés des homomorphismes :*

- *l homomorphisme préserve la structure car les nœuds sont liés aux nœuds et les arcs sont liés aux arcs.*
- *l homomorphisme n est pas forcément injectif car les nœuds destinations peuvent être associés à plusieurs nœuds sources.*
- *l homomorphisme n est pas forcément surjectif car les nœuds destinations peuvent n être associés à aucun nœud source comme présenté dans les deux exemples précédents.*

Le morphisme identité : `MorId` est le morphisme qui pour chaque nœud associe le même nœud, pour chaque arc associe le même arc et pour chaque étiquette d'arc associe la même étiquette. Il est typé en COQ comme suit :

```
Definition MorId: forall g: OModele, MorModele g g.
```

Il est défini à l'aide du triplet de fonctions $(((fun\ x => x), (fun\ x => x), (fun\ x => x)))$ avec la preuve que ces fonctions constitue un morphisme de g vers g.

La composition des morphismes : `MorComp` est la fonction de composition des morphismes. Elle est définie en combinant l'application des deux fonctions. Soit $homg = (mapg, condh) : g \rightarrow g'$ un morphisme qui vérifie qu'un modèle g' est le résultat de l'application de la fonction $mapve$ sur le modèle g, soit $homg' = (mapg', condh') : g' \rightarrow g''$ alors la fonction map dans le morphisme composition est un triplet constitué de la composition des trois composantes respectives des deux fonctions. Elle est typée en COQ comme suit :

```
Definition MorComp:
  forall (g: OModele) g' g'',
    (MorModele g g') -> (MorModele g' g'') -> (MorModele g g'').
```

Soit : mapg= (mapv, mape, mapa) et mapg'= (mapv', mape', mapa'), alors la définition de la composition utilise le triplet de fonctions : $((fun\ x => mapv'\ (mapv\ x)), (fun\ x => mape'\ (mape\ x)), (fun\ x => mapa'\ (mapa\ x)))$ en association avec la preuve de la condition sur le modèle composite.

L'équivalence des morphismes : *MorEqv* défini l'équivalence de morphisme comme suit : deux morphismes hom et hom' sont dits équivalents si leur résultat pour chaque nœud x sont les mêmes.

```
Definition MorEqv g g' (hom: MorModele g g') (hom': MorModele g g') :=
  forall x, (fst (fst (proj1_sig hom))) x = (fst (fst (proj1_sig hom'))) x.
```

La catégorie des modèles : La catégorie des modèles définie avec ces notions d'objets (*OModele*), morphisme (*MorModele*), morphisme identité (*MorId*), composition de morphismes (*MorComp*) et équivalence de morphismes (*MorEqv*) comme une instance du type classe défini dans [2] est la suivante :

```
Instance ModeleCat (Ob:= OModele):
  Category (Ob) ((MorModele)) :=
  {
    id          :=                  MorId
    ; comp      := fun a b c        => MorComp a b c
    ; eqv       := fun a b H H'     => MorEqv a b H H'
  }.
```

Nous avons choisi d'utiliser une bibliothèque COQ qui définit la notion de catégorie. Elle permet de considérer des morphismes qui respectent eux même la structure de catégorie et donc demande de prouver des propriétés supplémentaires :

[2] http://www.cs.berkeley.edu/~megacz/coq-categories/

1. L'équivalence de morphismes (*eqv_equivalence*) est une relation d'équivalence (réflexive, symétrique et transitive). Elle est définie comme suit :

```
Lemma eqv_equivalence: forall a b, RelationClasses.Equivalence (MorEqv a b).
```

2. La composition de morphismes (*MorComp*) respecte la propriété *left_identity* (la composition avec le morphisme identité à gauche) :

```
Lemma left_identity:   forall a b (f : MorModele a b),
   MorEqv a b (MorComp a a b (MorId a) f) f.
```

3. La composition de morphismes (*MorComp*) respecte la propriété *right_identity* (la composition avec le morphisme identité à droite) :

```
Lemma right_identity:   forall a b (f : MorModele a b),
   MorEqv a b (MorComp a b b f (MorId b)) f.
```

4. La composition de morphismes (*MorComp*) est une relation associative :

```
Lemma associativity:   forall a b (f : MorModele a b) c (g : MorModele b c)
     d (h : MorModele c d),
   MorEqv a d (MorComp a c d (MorComp a b c f g) h)
              (MorComp a b d f (MorComp b c d g h)).
```

Nous avons choisi pour l'implémentation de la catégorie de modèles cette deuxième approche qui considère les modèles comme les objets. Nous proposons dans ce qui suit la signification de quelques opérateurs algébriques pour cette catégorie.

Selon le principe de dualité, chaque définition concernant une catégorie peut être transformée en sa définition duale en remplaçant les mots domaine par codomaine et toutes les compositions de type $f \circ g$ par $g \circ f$. Si une proposition est vraie dans une catégorie, alors la proposition duale est vraie dans la catégorie duale. Ce principe de dualité permet de construire la preuve du théorème dual à chaque fois qu'un théorème est prouvé. Ce principe permet donc de réduire de moitié les preuves à faire.

Le principe de dualité permet à la théorie des catégories de construire les objets initiaux et terminaux, les produits, les coproduits, les égalisateurs, les coégalisateurs, les pullbacks et les pushouts qui sont des exemples de constructions universelles et co-universelles que nous présentons brièvement dans ce qui suit ainsi que leurs applications dans notre contexte.

6.2 Proposition d'interprétation des opérateurs algébriques

Nous étudions dans cette section la sémantique des opérateurs catégoriques dans la catégorie de modèles. Le but est de définir les briques de base pour l'exploitation de ces notions dans cette catégorie particulière. Nous nous intéressons dans ce cadre à l'extraction à partir d'une catégorie définie à l'aide d'objets de type modèle et d'un morphisme entre ces objets des opérateurs catégoriques que nous pourrons appliquer et leurs significations.

La catégorie duale :

Dé nition 7. *La catégorie duale pour une catégorie C est notée C^{op}. Les objets de la catégorie duale sont les mêmes que les objets de C ; les èches sont les èches opposées des èches dans C.*

La catégorie duale pour la catégorie de modèles est une catégorie dont les objets sont du type *OModele* et dont le morphisme est le morphisme opposé à *MorModele* (obtenu en inversant ses deux paramètres). Les èches de composition et d'identité sont donc définies d'une façon triviale.

Les objets initiaux et terminaux :

Dé nition 8. *Un objet* 0 *est appelé un objet initial pour une catégorie si pour chaque objet A, il existe exactement une èche de* 0 *vers A. D une façon duale, un objet* 1 *est appelé objet nal ou terminal si pour chaque objet A il y a exactement une èche de A vers* 1*.*

Remarque 6.4. *Les objets initiaux et terminal d une catégorie peuvent ne pas exister.*

Concernant la catégorie des modèles (de manière semblable à celle des ensembles), l'objet initial est le modèle vide car pour tout modèle, il existe une fonction qui permet de le construire à partir d'un modèle vide. L'objet final peut être n'importe quel modèle A contenant un seul élément avec un lien ré exif, car il existe un morphisme de n'importe quel modèle B vers ce modèle, ce dernier morphisme permet de lier tous les nœuds de A au nœud unique dans B ainsi que tous les liens de A à l'arc unique de B.

Remarque 6.5. *Les opérateurs tel qu ils sont présentés concernent les objets dans la catégorie de modèles. Ces opérateurs peuvent concerner la catégorie elle même.*

Produit :

Dé nition 9. *Un produit de deux objets A et B est un objet $A \times B$ en association avec deux projections $\pi_1 : A \times B \to A$ et $\pi_2 : A \times B \to B$, tel que pour chaque objet C et paires de èches $f : C \to A$ et $g : C \to B$, il existe un morphisme unique $\langle f, g \rangle : C \to A \times B$ tel que : $\pi_1 \circ \langle f, g \rangle = f$ et $\pi_2 \circ \langle f, g \rangle = g$.*

Dans la catégorie des ensembles, le produit de deux objets est leur produit cartésien. Le produit dans la catégorie de modèles est le produit catégorique des deux graphes représentant les deux modèles. Le produit catégorique peut être obtenue comme suit :

- ▷ L'ensemble des nœuds est le produit cartésien des ensembles de nœuds de A et B.
- ▷ Un arc r, r' existe entre (o_1, o_2) et (o'_1, o'_2) s'il existe un lien r entre o_1 et o_2 dans A et un lien r' entre o'_1 et o'_2 dans B.

Exemple 6.2. *Nous présentons sur la gure 6.2 l exemple du produit des deux objets A et B.*

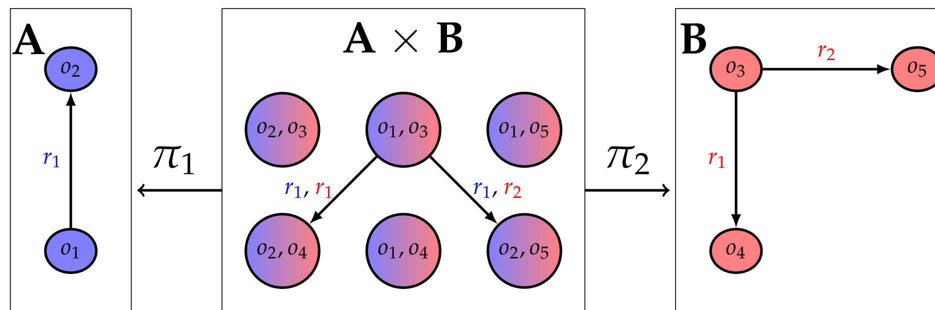


Figure 6.2 — Le produit des deux modèles A et B

Remarque 6.6. *π_1 et π_2 sont les fonctions de projection pour les objets A et B. Par exemple π_1 est dé ni ainsi pour tout objet (o_1, o_2), arc $(((o_1, o_2), (o_3, o_4)), (r_1, r_2))$ et étiquette d arc (r_1, r_2) dans le modèle comme suit :*

$$\pi_1 = \begin{cases} mapv(o_1, o_2) = o_1 \\ mape(((o_1, o_2), (o_3, o_4)), (r_1, r_2)) = ((o_1, o_3), r_1) \\ mapa(r_1, r_2) = r_1 \end{cases}$$

Remarque 6.7. *Pour l instant, le produit ne semble pas avoir une utilité et une interprétation usuelle dans le contexte de modèles.*

Coproduit :

Dé nition 10. *Un coproduit de deux objets A et B est un objet $A + B$ en association avec deux projections ι_1 et ι_2 tel que pour tout objet C avec $f : A \to C$ et $g : B \to C$, il existe un morphisme unique $\langle f, g \rangle : A + B \to C$.*

Pour la catégorie de modèles, le coproduit utilise aussi le concept de l'union disjointe qui ressemble à l'union classique sauf que si les deux modèles ont des éléments en commun, ces éléments sont présentés deux fois dans le résultat.

Exemple 6.3. *Voici sur la gure 6.3 l exemple du coproduit pour les objets A et B.*

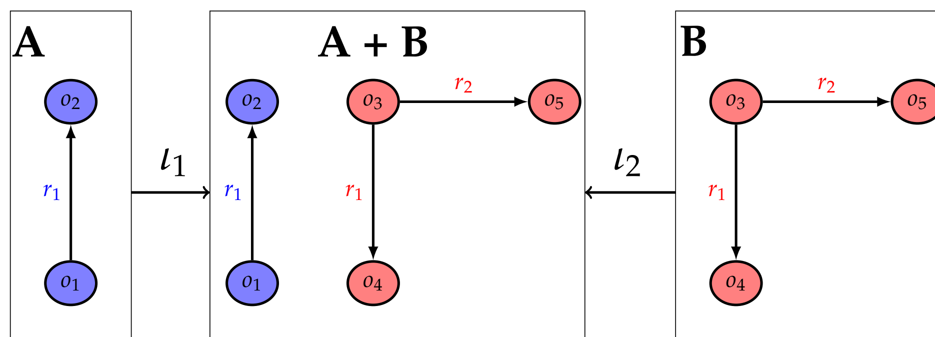


Figure 6.3 — Le coproduit des deux modèles A et B

Remarque 6.8. *ι_1 et ι_2 sont dé nies comme les fonctions d appartenance de A et B à $A + B$.*

Remarque 6.9. *Cette construction dé nit une version spéciale de la fusion de modèles. En relation avec les propriétés EMOF, le coproduit de modèles est un opérateur qui conserve la propriété du bon typage des deux modèles de base, les classes abstraites et l héritage mais peut ne pas préserver les multiplicités.*

Égalisateur :

Dé nition 11. *Dans une catégorie C, étant donné deux èches parallèles f et $g : A \rightarrow B$, un égalisateur de f et g est composé de E un objet et e un morphisme tel que $e : E \rightarrow A$, avec : $f \circ e = g \circ e$. Étant donné $z : Z \rightarrow A$ avec $f \circ z = g \circ z$, il existe alors un $u : Z \rightarrow E$ unique tel que : $e \circ u = z$.*

Pour les ensembles, étant donné deux fonctions $f, g : A \rightrightarrows B$, soit E le sous-ensemble de A où les deux morphismes coïncident : $\{x \in A \mid f(x) = g(x)\}$. La fonction d'inclusion $e : E \rightarrow A$, qui relie chaque élément $x \in X$ au même élément x considéré comme un élément de A, est un égalisateur de f et g.

Pour la catégorie de modèles, si f et g sont des morphismes de modèles, alors leur égalisateur est un morphisme $e : E \rightarrow A$ donné au niveau des nœuds comme l'ensemble E de nœuds égaux par les fonctions f et g et au niveau des arcs par : $E(x, y) \Leftrightarrow A(e(x), e(y))$. Autrement dit, si x et y appartiennent à E et s'il existe un arc entre eux dans A alors l'arc appartient aussi à E. Simplement dit, ceci construit le sous-modèle de A où $\forall x \in A, f(x) = g(x)$ avec les liens associés à ses éléments.

Exemple 6.4. *Pour les deux morphismes f et g suivants, l exemple 6.4 montre le résultat du calcul de l égalisateur.*

$$f = \begin{cases} mapv = \{o_1 \rightarrow o_3, o_2 \rightarrow o_4\} \\ mape = \{((o_1, o_2), r_1) \rightarrow ((o_3, o_4), r_1)\} \\ mapa = \{r_1 \rightarrow r_1\} \end{cases}$$

$$g = \begin{cases} mapv' = \{o_1 \rightarrow o_3, o_2 \rightarrow o_5\} \\ mape' = \{((o_1, o_2), r_1) \rightarrow ((o_3, o_5), r_2)\} \\ mapa' = \{r_1 \rightarrow r_2\} \end{cases}$$

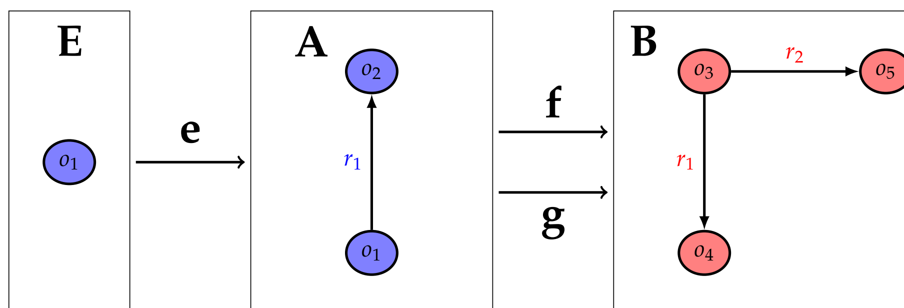


Figure 6.4 — Un exemple d'égalisateur

Coégalisateur :

Dé nition 12. *Pour deux èches parallèles $f, g : A \rightrightarrows B$ dans une catégorie C, un coégalisateur est composé de Q un objet et $q : B \to Q$ un morphisme, avec la propriété $q \circ f = q \circ g$. Tel que, étant donné Z et $z : B \to Z$, si $z \circ f = z \circ g$, alors il existe un $u : Q \to Z$ unique tel que : $u \circ q = z$.*

Un coégalisateur est une généralisation d'un quotient en utilisant une relation d'équivalence. Pour la catégorie des ensembles pour une paire de fonctions parallèles $f, g : A \rightrightarrows B$, l'égalisateur peut être construit en calculant le quotient de B par la relation d'équivalence générée par l'équation $f(x) = g(x)$ pour tout $x \in A$.

Concernant la catégorie de modèles, pour une relation d'équivalence *map* entre $x \in A$ et $y \in B$ (les éléments équivalents à x par *map*, aussi appelé classe d'équivalence de x), tel que le coégalisateur est le quotient de B par cette relation d'équivalence. Le résultat est un modèle ou les éléments équivalents sont regroupés dans un seul élément.

Exemple 6.5. *Voici un exemple de deux modèles A et B avec les mêmes morphismes f et g présentés dans l exemple précédent, l coégalisateur est le modèle Q présenté sur la gure 6.5 en association avec le morphisme q.*

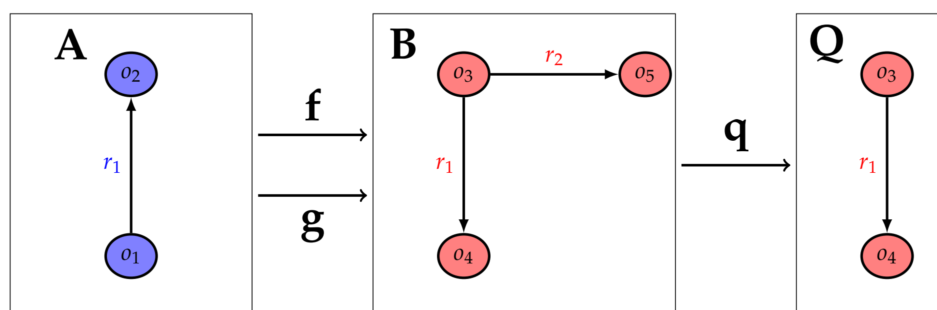


Figure 6.5 — Un exemple de coégalisateur

Un opérateur très intéressant dans le contexte des modèles est le `Package Merge` présenté dans le chapitre 4. Cet opérateur peut être implémenté comme une combinaison de l'union disjointe et du quotient [RB88] et donc peut être défini en utilisant le coproduit en combinaison avec le coégalisateur [Mar11]. Supposons que nous ayons deux modèles A et B et une fonction map (qui relie les parties égales des deux modèles), la fusion des deux modèles peut être obtenue comme suit :

1. un coproduit est calculé pour les deux modèles A et B.
2. la fonction map est utilisée pour calculer les classes d'équivalence.
3. Le quotient du modèle B par rapport aux classes d'équivalences est exactement la fusion des deux modèles.

Une autre variante peut être imaginée ou les classes d'équivalence sont définies d'une manière syntaxique (la relation d'équivalence est une égalité syntaxique) ceci permet de synthétiser les classes d'équivalence entre les nœuds, les arcs et les étiquettes pour l'étape 2, les étapes 1 et 3 restent les mêmes.

Nous pouvons aussi récupérer dans ce cadre catégorique l'opérateur substitution en utilisant la notion de foncteur (Définition 6.1). Dans ce cas, la substitution est utilisée pour construire le foncteur de la catégorie des graphes à la catégorie des graphes substitués.

6.3 Conclusion

Nous avons présenté deux alternatives classiques pour la description d'une catégorie de modèles ainsi que les détails d'implémentation de la solution choisie. Nous avons par la suite décrit nos ré exions pour compléter cette catégorie avec les notions habituelles dans les cadres catégoriques.

Ces opérateurs algébriques peuvent ne pas préserver le type de propriétés que nous avons vérifié dans le chapitre 3 pour des opérateurs élémentaires. À ce niveau, nous avons mené des ré exions informelles sur les propriétés préservées, les preuves en COQ ouvriront les portes pour des travaux futurs. Premièrement, l'opérateur produit génère un modèle qui est fondamentalement et structurellement différent des deux modèles de base (ce qui introduit un changement des nœuds et des arcs), donc aucune des propriétés ne peut être préservée. Pour le coproduit et le coégalisateur, le bon typage, l'héritage, les classes abstraites sont préservées par contre les cardinaux peuvent ne pas être préservés (le premier peut poser une non conformité par rapport à la borne maximale et le deuxième par rapport à la borne minimale). Pour le coégalisateur, le modèle résultat peut préserver toutes le propriétés. Enfin, la combinaison d'un coproduit avec un coégalisateur préserve toutes les propriétés. Cette combinaison peut être considérée comme une manière de restaurer la préservation des propriétés une fois elles ont été altérées par une application d'un opérateur coproduit. Une autre piste donc à étudier est l'utilisation du coégalisateur pour la restauration de la consistance dans le cadre de transformations bidirectionnelles.

Nous avons indiqué dans ce chapitre comment la théorie des catégories peut être utile dans notre contexte de modèles. Selon l'état de l'art en modélisation par la théorie des catégories, l'implémentation concrète de ces constructions universelles et couniverselles pour ce contexte particulier devrait être très avantageuse non seulement pour l'implémentation des opérateurs sur les modèles d'une manière élégante comme l'utilisation de la combinaison du coégalisateur et le coproduit pour l'implémentation du `Package Merge` mais aussi pour des preuves plus simples pour des propriétés complexes.

Conclusion générale et perspectives

Ce chapitre présente le bilan des travaux réalisés dans cette thèse et quelques perspectives. L'objectif initial était l'étude de la réutilisation dans un contexte de certification et qualification. Ce sujet est très vaste et nous nous sommes focalisée sur deux aspects : la réutilisation d'outils de vérification et la formalisation de la composition dans un langage de modélisation supportant des activités de vérification. Nous avons exploité les deux paradigmes de modélisation majoritairement utilisés qui sont au centre de l'ingénierie logicielle et des problèmatiques de réutilisation : l'Ingénierie Dirigée par les Modèles (IDM – UML, EMOF, OO) et le Web sémantique (RDF, OWL, ontologies). Le Web sémantique offre une plus grande expressivité et est censé être plus accessible aux les profanes. Dans ce cadre, les ontologies permettent de modéliser collaborativement un monde ouvert en perpétuelle évolution et de raisonner sur les modèles produits. Les modèles correspondent généralement à des spécifications d'un existant validées par des êtres humains sans sémantique explicite qui permettrait de les vérifier en les confrontant à d'autres modèles. Les outils disponibles sont souvent des prototypes académiques. L'IDM est plus mature, dispose d'un plus grand nombre d'outils et est mieux maîtrisée par le monde industriel. Elle est plutôt exploitée par une communauté d'utilisateurs plus restreinte. Il est généralement possible d'associer une sémantique explicite aux modèles ce qui permet d'exploiter des techniques de vérification formelles qui s'appuient sur la sémantique des langages de modélisation. Ces deux approches ont été appliquées indépendamment dans les deux parties de cette thèse.

6.4 Bilan des travaux présentés

Cette thèse est présentée essentiellement en deux parties. La première partie concerne la construction d'une ontologie du domaine de la $V\&V$ que nous appelons *formalisation syntaxique*. La seconde partie présente une formalisation de la construction et de la vérification d'un assemblage de composants pré-vérifiés en s'appuyant sur une formalisation des concepts de l'IDM et que nous appelons *formalisation sémantique*. Dans cette dernière partie, les composants que nous considérons sont des morceaux de modèles conformes au même métamodèle et la vérification sémantique considérée est la conformité.

6.4.1 La formalisation syntaxique

Nous avons principalement décrit dans cette partie l'ontologie de domaine de $V\&V$. La VVO est une ontologie qui contient une conceptualisation générale du domaine avec une population riche de méthodes et d'outils liés. Cette ontologie est utilisée initialement comme nous l'avons montré dans l'étude de cas comme une base de connaissances pour répondre à

des requêtes. Elle est librement accessible pour être étendue et validée par la communauté.

6.4.2 La formalisation sémantique

Nous avons présenté une formalisation avec l'assistant à la preuve COQ de quelques opérateurs élémentaires d'assemblage de composants. Ces opérateurs sont exprimés dans le cadre d'IDM formelle COQ4MDE. Nous avons prouvé la préservation des propriétés en relation avec le métamétamodèle EMOF ainsi que la propriété du bon typage pour ces opérateurs élémentaires. La propriété de la terminaison des opérateurs est toujours assurée dans un développement en COQ. Les formalisations des opérateurs de plus haut niveau peuvent être obtenues en combinant celles des opérateurs élémentaires. Le processus de formalisation de l'opérateur Package Merge de EMOF est aussi expliqué à base d'opérateurs élémentaires. D'autres opérateurs de composition de la méthode ISC sont aussi formalisés et les preuves de propriétés sont aussi présentées.

Nous avons présenté nos travaux préliminaires pour une catégorie de modèles et nos ré exions sur la signification des opérateurs catégoriques. Le cadre algébrique catégorique permet d'exprimer des opérateurs très riches d'une manière assez intuitive. L'implémentation concrète de ces opérateurs en COQ et les propriétés que nous pouvons vérifier et conserver par application des opérateurs algébriques restent des perspectives.

6.5 Perspectives

Nous présentons dans ce qui suit quelques pistes pour des travaux futurs. Premièrement, l'ontologie VVO est conçue actuellement dans un but de partage de connaissances, elle peut avoir beaucoup d'autres utilisations, nous présentons dans un premier point des exemples d'utilisation (points 6.5.1 et 6.5.2). Deuxièmement, la formalisation sémantique proposée en COQ offre un cadre pour manipuler des modèles et spécifier des techniques de vérification. Cela permet ensuite de prouver la correction de modèles et d'étudier la conservation de propriétés lors de l'assemblage de composants. Ce cadre peut être étendu pour supporter d'autres opérateurs et vérifier d'autres propriétés (point 6.5.3). Il peut aussi être utilisé pour décrire et vérifier des opérateurs sur des ontologies (point 6.5.4).

6.5.1 Compatibilité de composants à base d'ontologies

Suivant le principe du SOA (Service-oriented architectures) [Erl08] pour l'interopérabilité des services web et ce qui se fait dans le domaine du SWS (Semantic Web Service) [KT06] pour l'élaboration d'applications à base d'ontologies, l'ontologie peut être utilisée pour automatiser la composition par la définition des connexions sémantiques entre les composants en exprimant d'une manière explicite les composants compatibles. Nous pouvons soit exprimer la compatibilité comme une intention et dans ce cas spécifier les composants qui devraient être composables, soit l'exprimer comme une certitude et dans ce cas donner la preuve de composabilité.

6.5.2 Compatibilité comportementale lors de l'assemblage

Les vérifications en COQ assurent l'égalité de types (contrainte explicite dans la fonction de substitution), remplacer un type par un autre qui soit compatible nécessite une manière d'exprimer une hiérarchie pour les types et une notion de compatibilité plus exible. Le type exprime la contrainte minimale pour autoriser la composition en préservant le typage, l'ajout de préconditions sur le même paramètre permet de rendre compositionnelles d'autres propriétés. Pour l'instant, l'équivalence est déclarée explicitement sans s'appuyer sur le contenu structurel et comportemental des types. Plusieurs aspects doivent être étudiés dans ce but. D'une part, les langages de métamodélisation dans l'IDM se focalisent sur les aspects structurels du langage. La sémantique n'apparaît qu'à travers le choix des noms des métaclasses et des relations. La sémantique statique peut être exprimée avec des formules de la logique du premier ordre FOL (First Order Logic) manipulant des instances des métaclasses et des relations (OCL est utilisé par l'OMG pour exprimer ces contraintes pour EMOF). Les propriétés considérées dans cette thèse en terme de vérification compositionnelle sont de cette nature (structurelle). Pour traiter des propriétés comportementales, comme les travaux réalisés autour du langage BIP [BBS06] en terme de vérification compositionnelle de l'absence d'interblocage, il est nécessaire de pouvoir donner une sémantique comportementale pour les langages spécifiés par un métamodèle. Plusieurs travaux proposent d'introduire un méta langage d'actions pour exprimer cette sémantique à travers des méthodes associées aux métaclasses comme Kermeta [MFJ05] et XCore[3]. Cette forme opérationnelle permet d'exprimer des contraintes sous une forme axiomatique à base de pré/post conditions sur les méthodes et d'invariants sur les métaclasses. Elle n'est pas forcément très adaptée à la définition de propriétés sur le comportement du langage qui exigent de raisonner sur l'évolution de l'état du système de façon semblable à une sémantique de trace. Des travaux sont en cours au sein du projet ANR GEMOC[4] pour offrir les extensions adéquates au niveau des langages de métamétamodélisation en s'appuyant sur une combinaison de l'approche opérationnelle pour les détails internes de manipulations des métaclasses en exploitant Kermeta et sur une sémantique de traces en s'appuyant sur CCSL [And09]. Les résultats de cette thèse et les travaux de formalisation en COQ du langage Fiacre par Garnacho et al. [GBF13] à base de Timed Transition System de Henziger [HMP92] pourront servir de base à la formalisation sémantique de ces extensions de langages de métamétamodélisation. L'objectif de GEMOC est de construire un atelier de spécification de langages permettant de donner différentes sémantiques plus ou moins détaillées selon les activités de V&V considérées et les phases du cycle de vie dans lesquelles les modèles sont réalisés. D'autre part, il faut aussi considérer d'autres formes de propriétés que nous souhaitons préserver par composition, ou vérifier de manière compositionnelle. Autrement dit, nous pouvons considérer d'autres notions de conformité pour lesquelles nous étudierons la compositionalité. Dans le cadre de ce manuscrit, nous avons d'abord considéré le typage, que nous avons ensuite enrichi avec d'autres contraintes comme la multiplicité. Il serait pertinent d'étudier les relations sémantiques entre les différentes formes de conformité pour en déduire la préservation de la compositionalité à partir de la relation entre les conformités. Nous montrons la préservation pour la conformité au sens EMOF qui sur-contraint le typage. Autrement dit, le respect de la

[3] http://wiki.eclipse.org/Xcore
[4] http://gemoc.org/

conformité au sens EMOF, assure le typage. Nous pouvons donc en déduire la préservation du typage sans refaire les preuves.

6.5.3 Modélisation d'autres opérateurs

Nous pouvons formaliser d'autres opérateurs de composition élémentaires ou de plus haut niveau pour lesquels nous pouvons suivre la même démarche de vérification. Les opérateurs élémentaires déjà formalisés offre un cadre riche pour implémenter une grande famille d'opérateurs d'assemblage de haut niveau. Dans ce travail, nous avons exprimé d'autres opérateurs qui nous ont servi pour nos vérifications d'opérateurs élémentaires comme l'opérateur d'intersection de modèles utilisé pour la vérification de l'opérateur Union. L'opérateur intersection peut être utilisé pour formaliser d'autres opérateurs sur les modèles comme $EMFDiff$, l'opérateur *Slicing* et l'opérateur *Pruning* sur les métamodèles dont le but est de construire des vues spécifiques des métamodèles. Ces opérateurs manipulent la structure du modèle. La programmation par aspect en programmation orientée objet et les technologies monadiques en programmation fonctionnelle permettent d'assembler des comportements. Une perspective à long terme pour la poursuite de ces travaux consiste à formaliser ce type d'opérateurs de composition comportementale. Ces travaux seront partiellement abordés dans le cadre du projet GeMoc cité précédemment.

Nous pouvons étudier la correction des opérateurs composites d'une manière générique comme illustré sur la figure 6.6. Pour exprimer le fait que si une propriété particulière (par exemple $conformsTo$) est préservée par l'application d'un opérateur de composition f_1 et par l'application d'un opérateur de composition f_2, alors cette propriété est préservée par l'application de l'opérateur $(f_1||f_2)$.

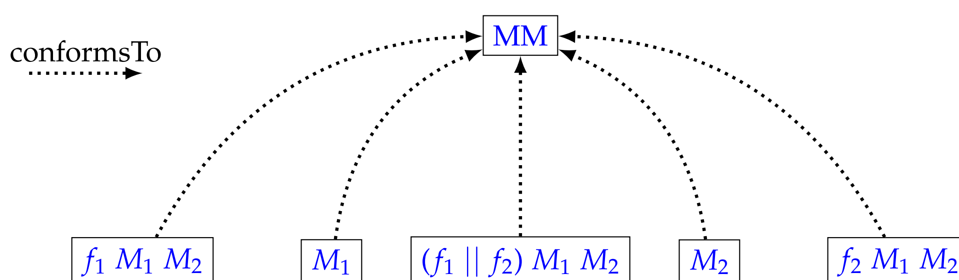


Figure 6.6 — La vérification du conformsTo pour la composition de fonctions

Une autre approche est de spécifier les conditions sur les opérateurs d'une manière générique pour qu'ils préservent par définition une certaine propriété. Par exemple n'importe quel opérateur qui ne modifie pas le type des éléments de modèles préserve par définition la propriété $instanceOf$.

6.5.4 Opérateurs sur les ontologies

Une ontologie peut être vue elle même comme un modèle conforme à un métamodèle. Nous pouvons appliquer tous les opérateurs d'assemblage déjà étudiés dans cette thèse sur des

modèles ontologiques, ceci dans un but de formaliser l'assemblage certifié pour les ontologies. L'exemple le plus simple d'application est la fusion formelle d'ontologies. En effet, une ontologie comporte des formules de logique du premier ordre exprimées avec OWL. De nombreuses ontologies sont très volumineuses et composées de sous-ontologies faiblement liées qui forment naturellement des composants c'est-à-dire des parties pouvant être assemblées. Leur vérification est souvent très coûteuse et bénéficierait d'un traitement compositionnel. Les expériences conduites dans cette thèse concernent des propriétés locales. Les préconditions restent donc relativement simples pour les opérateurs de composition. Il est nécessaire de poursuivre les travaux en considérant des propriétés plus globales pour mesurer la complexité des préconditions nécessaires pour réaliser la vérification compositionnelle. Ces travaux seraient proches de la prise en compte d'OCL évoquées dans la section précédente.

Troisième partie

Annexes

A Une extension pour la bibliothèque Uniset de COQ

Cette annexe présente des propriétés supplémentaires prouvées sur la structure Uniset (une représentation des ensembles à l'aide de fonctions caractéristiques) de la bibliothèque standard de COQ que nous exploitons dans les autres preuves.

A.1 L'union

Ces propriétés concernent l'union telle qu'elle est définie dans la bibliothèque Uniset[1] et utilisée dans le chapitre 3.

Le lemme *unionAddLeftSym* montre que l'ajout d'un élément à l'union de deux ensembles est égal à l'union du résultat de l'ajout de l'élément au premier ensemble avec le deuxième ensemble. Le théorème *unionAddRightSym* est similaire en commutant les deux côtés de l'égalité.

Lemme 23. (unionAddLeftSym) $\forall\ s\ s'\ v,\ add(union\ s\ s')\ v = union(add\ s\ v)\ s'.$

Lemme 24. (UnionIntroLeft) $\forall\ v\ s\ s',\ v \in s \rightarrow\ v \in (union\ s\ s').$

Lemme 25. (UnionIntroRight) $\forall\ v\ s\ s',\ v \in s' \rightarrow\ v \in (union\ s\ s').$

Lemme 26. (UnionElim) $\forall\ v\ s\ s',\ v \in (union\ s\ s') \rightarrow\ v \in s\ \vee\ v \in s'.$

A.2 Les ensembles disjoints

Deux ensembles sont dits disjoints s'il n'existe pas d'éléments appartenant en même temps aux deux ensembles. Voici un ensemble de théorèmes concernant les ensembles disjoints :

Lemme 27. (DisjointSym) $\forall\ s\ s',\ Disjoint\ s\ s' \rightarrow Disjoint\ s'\ s.$

Lemme 28. (DisjointAddLeftIntro) $\forall\ v\ s\ s',\ v \notin s' \rightarrow Disjoint\ s\ s' \rightarrow Disjoint\ (add\ s\ v)\ s'.$

Lemme 29. (DisjointAddRightIntro) $\forall\ v\ s\ s',\ v \notin s \rightarrow Disjoint\ s\ s' \rightarrow Disjoint\ s\ (add\ s'\ v).$

[1] http://coq.inria.fr/stdlib/Coq.Sets.Uniset.html

Lemme 30. (DisjointAddLeftElim1) $\forall\ v\ s\ s',\ Disjoint\ (add\ s\ v)\ s' \rightarrow\ v \notin s'$.

Lemme 31. (DisjointAddRightElim1) $\forall\ v\ s\ s',\ Disjoint\ s\ (add\ s\ v) \rightarrow\ v \notin s$.

Lemme 32. (DisjointAddLeftElim2) $\forall\ v\ s\ s',\ Disjoint\ (add\ s\ v)\ s' \rightarrow\ Disjoint\ s\ s'$.

Lemme 33. (DisjointAddRightElim2) $\forall\ v\ s\ s',\ Disjoint\ s\ (add\ s'\ v) \rightarrow\ Disjoint\ s\ s'$.

A.3 L'intersection

L'intersection (*inter*) est implémentée en utilisant les booléens tel que, x est un membre de l'ensemble $A_1 \cap A_2$ si la valeur de $\chi_{A_1}(x) \wedge \chi_{A_2}(x)$ est satisfaite, x n'appartient pas à l'intersection dans le cas contraire.

$$\chi_{A_1 \cap A_2}(x) = \chi_{A_1}(x) \wedge \chi_{A_2}(x)$$

Lemme 34. (interEmptyLeft) $\forall\ s,\ inter\ Emptyset\ s = Emptyset$.

Lemme 35. (interEmptyRight) $\forall\ s,\ inter\ s\ Emptyset = Emptyset$.

Lemme 36. (interAddLeft1) $\forall\ v\ s\ s',\ v \in s' \rightarrow inter\ (add\ s\ v)\ s' = add\ (inter\ s\ s')\ v$.

Lemme 37. (interAddLeft2) $\forall\ v\ s\ s',\ v \notin s' \rightarrow\ inter\ (add\ s\ v)\ s' = inter\ s\ s'$.

Lemme 38. (interAddRight1) $\forall\ v\ s\ s', v \in s \rightarrow\ inter\ s\ (add\ s'\ v) = add\ (inter\ s\ s')\ v$.

Lemme 39. (interAddRight2) $\forall\ v\ s\ s',\ v \notin s \rightarrow\ inter\ s\ (add\ s'\ v) = inter\ s\ s'$.

Lemme 40. (interIntro) $\forall\ v\ s\ s',\ v \in s \rightarrow\ v \in s'\ \rightarrow\ v \in (inter\ s\ s')$.

Lemme 41. (interElimLeft) $\forall\ v\ s\ s',\ v \in (inter\ s\ s') \rightarrow\ v \in s$.

Lemme 42. (interElimRight) $\forall\ v\ s\ s', v \in (inter\ s\ s') \rightarrow\ v \in s'$.

Lemme 43. (DisjointInter) $\forall\ s\ s',\ Disjoint\ s\ s' \rightarrow\ inter\ s\ s' = Emptyset$.

Lemme 44. (interDisjoint) $\forall\ s\ s',\ inter\ s\ s' = Emptyset \rightarrow\ Disjoint\ s\ s'$.

A.4 La différence

La différence ($diff$) entre deux ensembles est implémentée comme suit : x est un membre de l'ensemble $A_1 - A_2$ si la valeur de $\chi_{A_1}(x) \wedge \neg\chi_{A_2}(x)$ est satisfaite, x n'appartient pas à l'union dans le cas contraire.

$$\chi_{A_1 - A_2}(x) = \chi_{A_1}(x) \wedge \neg\chi_{A_2}(x)$$

Lemme 45. (diffIntro) $\forall\ v\ s\ s',\ v \in s \rightarrow\ v \notin s'\ \rightarrow\ v \in (diff\ s\ s')$.

Lemme 46. (diffElimLeft) $\forall\ v\ s\ s',\ v \in (diff\ s\ s') \rightarrow\ v \in s$.

Lemme 47. (diffElimRight) $\forall\ v\ s\ s',\ v \in (diff\ s\ s') \rightarrow\ v \notin s'$.

A.5 La substitution

Cette section présente des théorèmes concernant l'opération map sur les ensembles qui est une version générique de la substitution. Elle permet d'appliquer une certaine fonction f sur tous les éléments d'un ensemble. L'application de la fonction f sur l'ensemble défini par χ_A est définie pour chaque élément x comme suit :

$$mapu\ f\ \chi_A(x) = f(\chi_A(x))$$

Lemme 48. (mapuElim) $\forall\ f\ s\ v,\ v \in (mapu\ f\ s) \rightarrow\ (f\ v) \in s.$

Lemme 49. (mapuIntro) $\forall\ f\ s\ w\ v,\ (f\ v) \in s \rightarrow f\ = f\ w \rightarrow w \in (mapu\ f\ s).$

Lemme 50. (mapuEmpty) $\forall\ f,\ mapu\ f\ Emptyset = Emptyset.$

Lemme 51. (mapuAddSym) $\forall\ f\ v'\ s,\ mapu\ f\ (add\ s\ (f\ v)) = (equiv\ f\ v) \cup (mapu\ f\ s).$

Les définitions et lemmes présentés dans ce chapitre sont utilisés pour l'implémentation des opérateurs et les preuves correspondants au niveau des graphes et des modèles.

B Calcul des cardinaux

Table des matières

B.1 Les itérateurs de graphes 142
B.1.1 Les itérateurs des nœuds et des liens 142
B.1.2 Le module Forall 143
B.1.3 Le module `Fold` 144
B.1.4 Le module Exists 144
B.2 Les itérateurs sur les modèles 145
B.3 Exemples de descriptions de propriétés 147
B.3.1 La définition de la propriété *lower* 147
B.3.2 La définition de la propriété *subClass* 147

La formalisation des propriétés sémantiques tel que `lower` dans la section 3.4.5 demande le calcul d'un cardinal sur la structure du modèle. Ceci fait intervenir des itérateurs sur les composantes du modèle. Les ensembles des nœuds et des liens sont représentés comme des fonctions caractéristiques qui sont utilisées pour former les graphes en deuxième niveau qui constituent les modèles d'ou la difficulté d'itérer sur les modèles.

D'une façon générale, un objet composé tel qu'un ensemble, doit fournir une manière de parcourir et d'accéder à ses éléments d'une façon séquentielle. Un itérateur doit connaître l'élément courant et les éléments qui sont déjà parcourus. Un itérateur peut être vu comme un type abstrait qui a deux opérations élémentaires : référencer un élément particulier dans l'ensemble et modifier son état pour désigner l'élément suivant sans jamais désigner deux fois le même élément. Le but d'un itérateur est de permettre de traiter chaque élément de l'ensemble indépendamment de la structure interne de l'ensemble.

Remarque B.1. *Toutes les fonctions utilisées pour exprimer les propriétés sont écrites dans une syntaxe* COQ *qui permet de générer du code calculable et non pas juste des propositions.*

B.1 Les itérateurs de graphes

Nous présentons des itérateurs avec des définitions génériques qui permettent de parcourir les graphes en appliquant une certaine fonction ou en vérifiant une certaine condition sur chaque élément dans les ensembles des nœuds et des graphes. Ces itérateurs utilisent la structure du graphe pour parcourir d'une façon séquentielle les éléments des deux ensembles.

B.1.1 Les itérateurs des nœuds et des liens

La fonction `iter` dans le module `V` définit un type pour les itérateurs sur les nœuds d'un graphe `g`. `R` est le type de retour de la fonction appliquée aux nœuds du graphe. `tvses` est la paire des ensembles des nœuds et des arcs utilisés pour construire le graphe `g`.

```
Definition iter R (g : graph tvses) := In g → R.
```

`In` caractérise tous les nœuds qui satisfont la condition d'appartenir à l'ensemble des nœuds du graphe `g`, elle est utilisée dans la définition de `iter` pour choisir un élément de l'ensemble des nœuds de g. `In` est définie dans le module `V` comme suit :

```
Definition In (g : graph tvses) := { v : typ | Uniset.In (fst tvses) v }.
```

Exemple B.1. *(*`M.V.iter bool g`*) est le type des fonctions d itération qui pour les nœuds du graphe* `g` *retournent un booléen.*

Remarque B.2. *Les dé nitions du module des nœuds (*`V`*) sont pré xées par* `M.V` *et celles du module des liens (*`E`*) par* `M.E`*.*

Le lemme `Add_iter` montre que pour un graphe `g` et un nœud `v`, si nous avons un itérateur de nœuds pour le graphe (`AddV v g`) alors nous pouvons déduire un itérateur de nœuds pour le graphe `g`. Une définition similaire adaptée pour l'itérateur des liens notée `M.E.Add_iter` est définie dans le module `E`.

```
Lemma Add_iter :
forall (R : Set) tvses' v (n : ~ Uniset.In (fst tvses') v) (g' : graph tvses'),
(iter R (AddV tvses' v n g')) -> (iter R g').
```

Ceci est prouvé grâce au lemme `Add_in_S`. Un autre lemme dans le module `E` est défini en remplaçant `AddV` par `AddA`.

```
Lemma Add_in_S :
forall tvses' v (n : ~ Uniset.In (fst tvses') v) (g' : graph tvses'),
In g' -> In (AddV tvses' v n g').
```

Nous présentons dans ce qui suit 3 modules de COQ4MDE [TCCG07] qui définissent des itérateurs sur les graphes ainsi que leurs propriétés. Ces itérateurs sont utilisés par la suite pour définir des itérateurs plus spécifiques pour les modèles.

B.1.2 Le module Forall

Le module `Forall` définit un opérateur qui permet de vérifier des conditions sur tous les nœuds et/ou tous les arcs d'un graphe.

La fonction `elements` définie dans le module `Forall` permet d'appliquer en même temps sur un graphe un itérateur sur les nœuds et un itérateur sur les arcs. L'itérateur sur les nœuds permet d'accéder à tous les nœuds de l'ensemble pour vérifier une certaine condition sur chaque nœud et retourner une valeur booléenne. D'une manière similaire, l'itérateur sur les arcs permet d'accéder à tous les arcs de l'ensemble pour vérifier une certaine condition sur chaque arc et retourner ainsi une valeur booléenne. La fonction retourne la valeur fausse si un des itérateurs trouve un nœud/arc qui ne satisfait pas sa condition.

```
Fixpoint elements tvses (g : graph tvses) { struct g } : (V.iter bool g) -> (E.iter bool g)
    -> bool :=
 match g in pgraph tvses return (V.iter bool g) -> (E.iter bool g) -> bool with
 | Nil                          => fun _ _ => true
 | AddA tvses' s d a Hd Hs Ha g' => fun fv fe => let fe' := E.Add_iter s d a Hd Hs Ha g' fe
                                                 in if fe (E.Add_in_O s d a Hd Hs Ha g')
                                                    then elements tvses' g' fv  fe'
                                                    else false
 | AddV tvses' v n             g' => fun fv fe => let fv' := V.Add_iter v n g' fv
                                                 in if fv (V.Add_in_O v n g')
                                                    then elements tvses' g' fv' fe
                                                    else false
 end.
```

Cette définition permet d'utiliser la structure du graphe pour itérer les ensembles des nœuds et des arcs de ce graphe en vérifiant les conditions définies par les itérateurs. Dans le cas d'un graphe vide le résultat de la vérification est toujours vraie. Pour les constructeurs `AddV` et `AddA` (les constructeurs pour l'ajout d'un nœud `v` ou un arc `e` à un graphe `g`), les itérateurs sont synthétisés pour le graphe `g` à l'aide de `V.Add_iter` (Lemma B.1.1) et `E.Add_iter`. Après vérification des conditions définies par les itérateurs sur `v` ou `e`, si les conditions sont satisfaites la fonction est appelée récursivement, sinon le résultat de la vérification est faux.

La propriété `Add_in_O` utilisée dans la définition de `elements` permet de choisir l'élément à itérer (nœud/arc) en donnant la preuve qu'il appartient à l'ensemble des nœuds/arcs. Pour sa définition dans le module `E` présentée dans le Lemme suivant, nous choisissons l'arc (src, a, dst) en prouvant qu'il satisfait la condition d'appartenance à l'ensemble des arcs.

```
Lemma Add_in_O : forall tvses' src dst a (Hd : Uniset.In (fst tvses') dst) (Hs : Uniset.In
    (fst tvses') src) (Ha :~ Uniset.In (snd tvses') (src, dst, a)) (g' : graph tvses'),
 In (AddA tvses' src dst a Hd Hs Ha g').
```

Les propriétés `elements_correct` et `elements_complete` permettent de passer de la définition de la fonction `elements` (l'itérateur `Forall` des nœuds/arcs/nœuds et arcs) à une quantification universelle classique sur ces derniers et d'une quantification universelle sur les nœuds et les arcs à la définition de `elements`. Nous définissons ainsi une équivalence entre ces deux représentations.

```
Lemma elements_correct :
 forall tvses (g : graph tvses) (pv : V.iter bool g) (pe : E.iter bool g),
  elements g pv pe = true -> (forall v, pv v = true) /\ (forall e, pe e = true).
Lemma elements_complete :
 forall tvses (g : graph tvses) (pv : V.iter bool g) (pe : E.iter bool g),
   (forall v, pv v = true) -> (forall e, pe e = true) -> elements g pv pe = true.
```

B.1.3 Le module `Fold`

Ce module permet de définir des fonctions qui peuvent itérer les nœuds et les arcs des graphes en générant certains résultats à l'aide de la fonction `fold`. Cette fonction est paramétrée par son type de retour R, deux itérateurs et une valeur initiale qui est également de type R. Elle parcourt le graphe en utilisant ses itérateurs (pour les nœuds et pour les arcs). L'itérateur sur les nœuds est du type $(V.iter\ (R \rightarrow R))$ (c'est n'importe quelle fonction qui permet pour un nœud du graphe et pour une valeur initiale de type R de retourner une certaine valeur de type R). La fonction qui permet d'itérer les arcs est typée par $(E.iter\ (R \rightarrow R))$. La fonction `fold` applique récursivement les deux fonctions d'itérations et se termine en exploitant la structure du graphe.

```
Fixpoint fold (R : Set) tvses (g : graph tvses) { struct g } :
 (V.iter (R -> R) g) -> (E.iter (R -> R) g) -> R -> R :=
 match g in pgraph tvses return (V.iter (R -> R) g) -> (E.iter (R -> R) g) -> R -> R with
 | Nil                          => fun _ _ t => t
 | AddA tvses' s d a Hd Hs Ha g' => fun fv fe t => let fe' := E.Add_iter s d a Hd Hs Ha g'
     fe
           in fe (E.Add_in_O s d a Hd Hs Ha g') (fold R tvses' g' fv  fe' t)
 | AddV tvses' v n              g' => fun fv fe t => let fv' := V.Add_iter v n g' fv
           in fv (V.Add_in_O v n g')              (fold R tvses' g' fv' fe  t)
 end.
```

B.1.4 Le module Exists

Nous nous intéressons maintenant à un opérateur qui itère tous les éléments du graphe en cherchant des nœuds/arcs qui satisfont les conditions spécifiées par les itérateurs. Ce type d'opérateur est défini dans le module `Exists` à l'aide de la fonctions `elements`. La définition de cette fonction ressemble à celle de `elements` dans le module `Forall` mais avec un traitement particulier qui permet de retourner la valeur `true` dès qu'un nœud ou un arc qui satisfait la condition de l'opérateur trouvé.

```
Fixpoint elements tvses (g : graph tvses) { struct g } : (V.iter bool g) -> (E.iter bool g)
     -> bool :=
 match g in pgraph tvses return (V.iter bool g) -> (E.iter bool g) -> bool with
 | Nil                          => fun _ _ => false
 | AddA tvses' s d a Hd Hs Ha g' => fun fv fe => let fe' := E.Add_iter s d a Hd Hs Ha g' fe
                                              in if fe (E.Add_in_O s d a Hd Hs Ha g')
                                                 then true
                                                 else elements tvses' g' fv  fe'
 | AddV tvses' v n              g' => fun fv fe => let fv' := V.Add_iter v n g' fv
                                              in if fv (V.Add_in_O v n g')
                                                 then true
                                                 else elements tvses' g' fv' fe
 end.
```

Propriétés sur l'itérateur Exists

Les théorèmes `elements_correct` et `elements_complete` permettent de passer de la fonction `elements` définie dans le module `Exists` à la quantification existentielle classique en COQ.

```
Lemma elements_correct :
 forall tvses (g : graph tvses) (pv : V.iter bool g) (pe : E.iter bool g),
  elements g pv pe = true -> { v | pv v = true } + { e | pe e = true}.

Lemma elements_complete :
 forall tvses (g : graph tvses) (pv : V.iter bool g) (pe : E.iter bool g),
   (exists v, pv v = true) \/ (exists e, pe e = true) -> elements g pv pe = true.
```

B.2 Les itérateurs sur les modèles

Dans le cadre de nos définitions de propriétés sur les modèles, nous devons itérer les objets et les liens dans les modèles pour vérifier certaines propriétés ou calculer certaines fonctions.

Les modules définis pour les itérateurs sur les graphes sont utilisés pour la définition des itérateurs permettant d'exprimer les propriétés sur les modèles.

La fonction `forallObject` permet d'itérer les objets d'un modèle pour vérifier une certaine condition, elle est définie en utilisant la fonction `elements` du module `Forall`. La fonction d'itération des nœuds est un paramètre de la fonction `forallObject` alors que l'itérateur des arcs est défini comme une fonction dont la valeur booléenne de retour est toujours vraie.

```
Definition forallObject (m : Model) : (forall v, VertexIn m v -> bool) -> bool :=

 match m return (forall v, VertexIn m v -> bool) -> bool with
 | model tvses g => fun it => let it' := fun v_in_g : M.V.In g => match v_in_g with
                                                                  | exist v in_g => it v in_g
                                                                  end
 in
  M.Forall.elements g it' (fun _ => true)
 end.
```

La fonction `forallLink` permet d'itérer tous les objets d'un modèle pour vérifier une certaine condition précisée dans son itérateur de liens. Elle est définie aussi en utilisant la fonction `elements` du module `Forall` tel que son paramètre est l'itérateur des liens, l'itérateur des nœuds est défini comme une fonction dont la valeur booléenne de retour est toujours vraie.

```
Definition forallLink (m : Model) : (forall e, EdgeIn m e -> bool) -> bool :=
  match m return (forall e, EdgeIn m e -> bool) -> bool with
  | model tvses g => fun it => let it' := fun e_in_g : M.E.In g => match e_in_g with
                                                                  | exist e in_g => it e in_g
                                                                  end
 in
  M.Forall.elements g (fun _ => true) it'
 end.
```

La fonction `existsObject` permet d'itérer l'ensemble des objets de modèle pour vérifier l'existence d'un objet qui satisfait une condition spécifiée dans son paramètre. Elle est

définie en utilisant la fonction `elements` du module `Exists`, elle prend comme paramètre un itérateur sur les objets du modèle qui vérifie qu'il existe bien un objet satisfaisant sa condition, l'itérateur des liens est défini comme une fonction dont la valeur booléenne de retour est toujours fausse.

```
Definition existsObject (m : Model) : (forall v, VertexIn m v -> bool) -> bool :=
 match m return  (forall v, VertexIn m v -> bool) -> bool with
 | model tvses g => fun it => let it' := fun v_in_g : M.V.In g => match v_in_g with
                                                          | exist v in_g => it v in_g
                                                          end
 in
  M.Exists.elements g it' (fun _ => false)
 end.
```

La fonction `existsLink` est définie en utilisant la fonction `elements` du module `Exists`, elle prend comme paramètre un itérateur sur les liens du modèle qui vérifie qu'il existe bien un lien dans le modèle vérifiant sa condition, l'itérateur des objets est défini comme une fonction dont la valeur booléenne de retour est toujours fausse.

```
Definition existsLink  (m : Model) : (forall e, EdgeIn m e -> bool) -> bool :=
 match m return  (forall e, EdgeIn m e -> bool) -> bool with
 | model tvses g => fun it => let it' := fun e_in_g : M.E.In g => match e_in_g with
                                                          | exist e in_g => it e in_g
                                                          end
 in
  M.Exists.elements g (fun _ => false) it'
 end.
```

La fonction `foldObject` est définie en utilisant la fonction `elements` du module `Fold`, elle prend comme paramètre un itérateur sur les objets du modèle et calcule le résultat obtenu par l'application de la fonction d'itération. La fonction d'itération des objets est n'importe quelle fonction qui pour un objet dans le modèle et une valeur initial du type R retourne une valeur de type R. L'itérateur des liens du modèle est défini comme une fonction identité.

```
Definition foldObject (R : Set) (m : Model) : (forall v, VertexIn m v -> R -> R) -> R -> R
    :=
 match m return (forall v, VertexIn m v -> R -> R) -> R -> R with
 | model tvses g => fun it => let it' := fun v_in_g : M.V.In g => match v_in_g with
                                                          | exist v in_g => it v in_g
                                                          end
 in
  M.fold g it' (fun _ => fun x => x)
 end.
```

La fonction `foldLink` est définie en utilisant la fonction `elements` du module `Fold`, elle prend comme paramètre un itérateur sur les liens du modèle et calcule le résultat obtenu par l'application de la fonction d'itération avec sa valeur initiale. L'itérateur des objets du modèle est défini comme une fonction identité.

```
Definition foldLink (R : Set) (m : Model) :  (forall v, EdgeIn m v -> R -> R) -> R -> R :=
 match m return (forall e, EdgeIn m e -> R -> R) -> R -> R with
 | model tvses g => fun it => let it' := fun e_in_g : M.E.In g =>
                                        match e_in_g with   | exist e in_g => it e in_g     end
 in M.fold g (fun _ => fun x => x) it'
 end.
```

B.3 Exemples de descriptions de propriétés

Pour exprimer des propriétés comme `subClass`, `isAbstract`, `lower`, `upper`, `isOpposite` et `areComposite` présentées dans le chapitre 3, nous sommes amenée à itérer les objets et les liens des modèles pour vérifier l'existence de certains liens et parfois calculer leurs cardinaux. Nous présentons deux exemples de propriétés qui utilisent les itérateurs de modèles déjà définis.

B.3.1 La dé nition de la propriété *lower*

La propriété *lower* définie dans la section 3.4.5 a comme paramètre une classe c, une référence r et un entier n. Elle permet pour tout modèle $\langle \mathtt{MV}, \mathtt{ME} \rangle$ de vérifier que :

$$\forall\, o \in \mathtt{Objects}, \langle \mathtt{o}, \mathtt{c} \rangle \in \mathtt{MV} \Rightarrow |\{\mathtt{m}_2 \in \mathtt{MV} \mid \langle\langle \mathtt{o}, \mathtt{c}\rangle, \mathtt{r}, \mathtt{m}_2\rangle \in \mathtt{ME}\}| \geq \mathtt{n}$$

.

```
Definition lower (c: Abstract.Vertex) (r: Reference) (n: nat) : PropertyS:=
  fun m =>
    forallObject m
    (fun o => fun o_in_m : VertexIn m o =>
      if (beqClass (classOf o) c) then
        (le_gt_dec_bool n (foldLink nat m
          (fun l => fun l_in_m : EdgeIn m l => fun card =>
            if ((andb (andb ((beqClass (classOf (fst (fst l))) c))
              (beqObject (objectOf (fst (fst l))) (objectOf o)))
            (beqReference (refOf (snd l)) r))) then S card
              else card) 0))
        else true).
```

Cette définition utilise `forallobject` pour itérer tout les objets du modèle. Pour tout objet `o` vérifiant la condition d'avoir comme type c, elle permet de vérifier la contrainte `lower` en calculant le nombre de liens de la forme recherchée à l'aide de la fonction `foldLink` avec comme valeur initiale 0.

B.3.2 La dé nition de la propriété *subClass*

La propriété `subClass` définie dans la section 3.1 exprime que c' est une sous-classe directe de c et donc tous les objets instances de c et c' doivent être liés par un lien d'héritage. Cette propriété est exprimée à l'aide de `forallObject` et `existsLink` comme suit :

```
Definition subClass c c' : PropertyS:=
  fun m =>
    forallObject m
    (fun o => fun o_in_m : VertexIn m o => if (beqClass (classOf o) c) then
      (existsLink m (fun l => fun l_in_m : EdgeIn m l =>
        (andb (beqObject (objectOf (fst (fst l))) (objectOf o))
          (andb (beqObject (objectOf (snd (fst l))) (objectOf o))
            (andb (beqClass (classOf (fst (fst l))) c')
              (andb (beqClass (classOf (snd (fst l))) c)
                ((beqReference (refOf (snd l)) inh))))))))
      else true).
```

C La preuve InstanceOf de l'opérateur Substitution

Cette annexe présente une preuve mathématique pour le premier théorème présenté dans le chapitre 3. Le théorème *ValidSubstitution* prouve la préservation de la propriété *instanceOf* par l'opérateur `Substitution`. La méthode de Lamport [Lam95] est utilisée pour rédiger la preuve. Nous avons choisi de ne pas présenter le reste des preuves dans le même format mais de référencer pour chaque théorème le lien vers le code de la preuve dans le format de l'assistant de preuve Coq.

Théorème 49. (ValidSubstitution)
$\forall\, \mathtt{M} \in \mathit{Model},\ \mathtt{MM} \in \mathit{MetaModel}$
$\mathit{InstanceOf}\ (\mathtt{M}, \mathtt{MM}) \rightarrow \mathit{InstanceOf}\ ((\mathit{Substitution}\ \mathtt{o}_1\ \mathtt{o}_2\ \mathtt{M}), \mathtt{MM})$

SUPPOSONS: M le $\mathtt{Model}\langle \mathtt{MV}, \mathtt{ME}\rangle$
MM le $\mathtt{MetaModel}\langle \mathtt{MMV}, \mathtt{MME}, \mathit{conformsTo}\rangle$
$H : \mathit{InstanceOf}\ (\langle \mathtt{MV}, \mathtt{ME}\rangle, \langle \mathtt{MMV}, \mathtt{MME}, \mathit{conformsTo}\rangle)$.

PROUVONS: $\mathit{InstanceOf}\ ((\langle \mathit{SubstV}\ \mathtt{o}_1\ \mathtt{o}_2\ \mathtt{MV}, \mathit{SubstE}\ \mathtt{o}_1\ \mathtt{o}_2\ \mathtt{ME}\rangle, \langle \mathtt{MMV}, \mathtt{MME}, \mathit{conformsTo}\rangle)$.

ESQUISSE DE LA PREUVE: Nous supposons que le modèle M est une instance du métamodèle MM et nous voulons vérifier que le modèle après la substitution d'un objet o_1 par un objet o_2 est aussi une instance de MM. Nous vérifions que la fonction de *Substitution* ne change pas les types des nœuds et des liens et de ce fait nous pouvons conclure que tous les nœuds et les liens restent conformes aux mêmes nœuds et liens dans le métamodèle initial.

PREUVE:

⟨1⟩1. Après l'introduction de la définition du *instanceOf* de la section 3.4.1, de *SubstV* et *SubstE*, l'hypothèse H est transformé ainsi :
$H : (\forall \langle \mathtt{o}, \mathtt{c}\rangle, \langle \mathtt{o}, \mathtt{c}\rangle \in \mathtt{MV} \rightarrow \mathtt{c} \in \mathtt{MMV})$
$\wedge\ (\forall \langle \langle \mathtt{o}, \mathtt{c}\rangle, \mathtt{r}, \langle \mathtt{o}', \mathtt{c}'\rangle\rangle, \langle \langle \mathtt{o}, \mathtt{c}\rangle, \mathtt{r}, \langle \mathtt{o}', \mathtt{c}'\rangle\rangle \in \mathtt{ME} \rightarrow \langle \mathtt{c}, \mathtt{r}, \mathtt{c}'\rangle \in \mathtt{MME})$.
Le but courant est transformé ainsi :
$(\forall \langle \mathtt{o}, \mathtt{c}\rangle, \langle \mathtt{o}, \mathtt{c}\rangle \in (V.\mathit{image}\ \mathit{mapv}\ (\langle \mathtt{MV}, \mathtt{ME}\rangle)\ g) \rightarrow \mathtt{c} \in \mathtt{MMV})$
$\wedge\ (\forall \langle \langle \mathtt{o}, \mathtt{c}\rangle, \mathtt{r}, \langle \mathtt{o}', \mathtt{c}'\rangle\rangle, \langle \langle \mathtt{o}, \mathtt{c}\rangle, \mathtt{r}, \langle \mathtt{o}', \mathtt{c}'\rangle\rangle \in (E.\mathit{image}\ \mathit{mapv}\ \mathit{mapa}\ (\langle \mathtt{MV}, \mathtt{ME}\rangle)\ g)$
$\rightarrow \langle \mathtt{c}, \mathtt{r}, \mathtt{c}'\rangle \in \mathtt{MME})$.

⟨1⟩2. L'hypothèse H est découpée en deux hypothèses :
$H_0 :\ \forall \langle \mathtt{o}, \mathtt{c}\rangle, \langle \mathtt{o}, \mathtt{c}\rangle \in \mathtt{MV} \rightarrow \mathtt{c} \in \mathtt{MMV}$.
$H_1 :\ \forall \langle \langle \mathtt{o}, \mathtt{c}\rangle, \mathtt{r}, \langle \mathtt{o}', \mathtt{c}'\rangle\rangle, \langle \langle \mathtt{o}, \mathtt{c}\rangle, \mathtt{r}, \langle \mathtt{o}', \mathtt{c}'\rangle\rangle \in \mathtt{ME} \rightarrow \langle \mathtt{c}, \mathtt{r}, \mathtt{c}'\rangle \in \mathtt{MME}$.
Le but courant est découpé en deux sous-buts :

1. $\langle \texttt{o}, \texttt{c} \rangle \in (V.image\ mapv\ (\langle \texttt{MV}, \texttt{ME} \rangle)\ g) \rightarrow \texttt{c} \in \texttt{MMV}$
2. $\forall \langle \langle \texttt{o}, \texttt{c} \rangle, \texttt{r}, \langle \texttt{o}', \texttt{c}' \rangle \rangle, \langle \langle \texttt{o}, \texttt{c} \rangle, \texttt{r}, \langle \texttt{o}', \texttt{c}' \rangle \rangle \in (E.image\ mapv\ mapa\ (\langle \texttt{MV}, \texttt{ME} \rangle)\ g)$
$\rightarrow \langle \texttt{c}, \texttt{r}, \texttt{c}' \rangle \in \texttt{MME}$

$\langle 2 \rangle$1. Nous commençons par prouver le premier sous-but qui correspond à la partie gauche de la conjonction :
SUPPOSONS: $H_2 : \ \langle \texttt{o}, \texttt{c} \rangle \in (V.image\ mapv\ (\langle \texttt{MV}, \texttt{ME} \rangle)\ g)$.
PROUVONS: $(\texttt{c} \in \texttt{MMV})$
PREUVE:
$\langle 3 \rangle$1. En généralisant le lemme 52 en utilisant H_2, nous obtenons une nouvelle hypothèse : $H_4 : \exists (o', c') \in MV \mid mapv\ (o', c') = (o, c)$.
$\langle 3 \rangle$2. Nous introduisons la définition de *mapv*, nous pouvons conclure que $c' = c$ alors nous aurons comme hypothèse : $H_5 : (o', c) \in MV$.
$\langle 3 \rangle$3. En appliquant H_0 avec comme paramètre (o', c) et H_5.
$\langle 3 \rangle$4. **Q.E.D.**

$\langle 2 \rangle$2. Nous prouvons maintenant la deuxième partie du but :
$\forall \langle \langle \texttt{o}, \texttt{c} \rangle, \texttt{r}, \langle \texttt{o}', \texttt{c}' \rangle \rangle, \langle \langle \texttt{o}, \texttt{c} \rangle, \texttt{r}, \langle \texttt{o}', \texttt{c}' \rangle \rangle \in (E.image\ mapv\ mapa\ (\langle \texttt{MV}, \texttt{ME} \rangle)\ g)$
$\rightarrow \langle \texttt{c}, \texttt{r}, \texttt{c}' \rangle \in \texttt{MME}$
SUPPOSONS: En ayant comme hypothèse supplémentaire à H_0 et H_1, l'hypothèse $H_2 : \forall \langle \langle \texttt{o}, \texttt{c} \rangle, \texttt{r}, \langle \texttt{o}', \texttt{c}' \rangle \rangle$,
$\langle \langle \texttt{o}, \texttt{c} \rangle, \texttt{r}, \langle \texttt{o}', \texttt{c}' \rangle \rangle \in (E.image\ mapv\ mapa\ (\langle \texttt{MV}, \texttt{ME} \rangle)\ g)$
PROUVONS: Ce sous but revient à prouver : $\langle \texttt{c}, \texttt{r}, \texttt{c}' \rangle \in \texttt{MME}$
PREUVE:
$\langle 3 \rangle$1. Ici, nous généralisons le lemme 53 en utilisant l'hypothèse H_2, nous obtenons une nouvelle hypothèse : $H_4 : \exists \langle \langle \texttt{o}_1, \texttt{c}_1 \rangle, \texttt{r}_1, \langle \texttt{o}'_1, \texttt{c}'_1 \rangle \rangle$,
$\langle \langle \texttt{o}_1, \texttt{c}_1 \rangle, \texttt{r}_1, \langle \texttt{o}'_1, \texttt{c}'_1 \rangle \rangle \in \texttt{ME} \mid mape\ \langle \langle \texttt{o}_1, \texttt{c}_1 \rangle, \texttt{r}_1, \langle \texttt{o}'_1, \texttt{c}'_1 \rangle \rangle = \langle \langle \texttt{o}, \texttt{c} \rangle, \texttt{r}, \langle \texttt{o}', \texttt{c}' \rangle \rangle$.
$\langle 3 \rangle$2. Nous introduisant la définition de *mape*, nous pouvons conclure que :
$c_1 = c, c'_1 = c'$ et $r_1 = r$,
alors nous aurons comme hypothèse : $H_5 : \langle \langle \texttt{o}_1, \texttt{c} \rangle, \texttt{r}, \langle \texttt{o}'_1, \texttt{c}' \rangle \rangle \in \texttt{ME}$.
$\langle 3 \rangle$3. En appliquant H_0 avec comme paramètre $\langle \langle \texttt{o}_1, \texttt{c} \rangle, \texttt{r}, \langle \texttt{o}'_1, \texttt{c}' \rangle \rangle$ et H_5.
$\langle 3 \rangle$4. **Q.E.D.**

□

Les lemmes utilisés dans cette preuve sont :

Lemme 52. (V.imageElim)
$\forall\ mapv,\ mapa,\ \langle \texttt{MV}, \texttt{ME} \rangle \in Model, v \in (image\ \langle \texttt{MV}, \texttt{ME} \rangle) \rightarrow \exists w \in MV \wedge mapv\ w = v.$

Lemme 53. (E.imageElim)
$\forall\ mapv,\ mapa,\ \langle \texttt{MV}, \texttt{ME} \rangle \in Model, e \in (image\ \langle \texttt{MV}, \texttt{ME} \rangle) \rightarrow \exists w \in \texttt{ME} \wedge mape\ w = e.$

Les preuves de ces deux lemmes sont faites par induction sur la structure du graphe et font intervenir d'autres théorèmes que nous ne présentons pas ici mais qui sont accessibles avec notre code COQ.

D Package Merge

Table des matières

D.1 Préliminaires 151
D.2 L'Union 154
D.3 Les preuves d'équivalence sémantique 154
D.4 Exemple 154

D.1 Préliminaires

`Package Merge` est l'opérateur défini dans le chapitre 4. Nous nous intéressons dans cette annexe à sa sémantique que nous décrivons en utilisant le prédicat `mergePackage`. Ce prédicat assure que le paquetage M_{result} est le résultat de la fusion des deux paquetages $M_{receiving}$ et M_{merged}.

Soit $M_{result} \in$ `MetaModel`, le résultat de la fusion de deux métamodèles : M_{merged} et $M_{receiving}$.

- ▷ M_{result} est le métamodèle $\langle\langle MV_{result}, ME_{result}\rangle, conformsTo_{result}\rangle$
- ▷ M_{merged} est le métamodèle $\langle MV_{merged}, ME_{merged}\rangle, conformsTo_{merged}\rangle$
- ▷ $M_{receiving}$ est le métamodèle $\langle MV_{receiving}, ME_{receiving}, conformsTo_{receiving}\rangle$

En se basant sur [Zit06], nous écrivons la sémantique du `Package Merge` sous forme de règles. Premièrement, nous définissons plusieurs constructions :

- ▷ $ownedMembers\langle MV, ME\rangle \triangleq MV \cup ME$
- ▷ $ownedAssociations\langle MV, ME\rangle \triangleq ME$
- ▷ $ownedClasses\langle MV, ME\rangle \triangleq MV$
- ▷ $ownedReferences\langle MV, ME\rangle \triangleq \{r \mid \exists \langle o, r, o'\rangle \in ME\}$

Les conditions sont détaillées ici :

1. Pour chaque élément du modèle M_{merged}, s'il n'y a pas un élément correspondant[1] dans $M_{receiving}$, cet élément doit appartenir au modèle M_{result}.

$$\begin{array}{l} ownedMembers_{merged} \triangleq \\ \forall\, e_1\, e_2, e_1 \in (ownedMembers\ M_{merged}) \\ \wedge (e_2 \notin (ownedMembers\ M_{receiving}) \wedge\ matches(e_1, e_2)) \\ \rightarrow e_1 \in (ownedMembers\ M_{result}) \end{array}$$

2. Pour chaque élément du modèle $M_{receiving}$, s'il n'y a pas un élément correspondant dans M_{merged}, cet élément doit appartenir au modèle M_{result}.

$$\begin{array}{l} ownedMembers_{receiving} \triangleq \\ \forall\, e_1\, e_2, e_1 \in (ownedMembers\ M_{receiving}) \\ \wedge (e_2 \notin (ownedMembers\ M_{merged}) \wedge\ matches(e_1, e_2)) \\ \rightarrow e_1 \in (ownedMembers\ M_{result}) \end{array}$$

3. Si un élément du modèle M_{merged} est égal à un élément de modèle $M_{receiving}$, alors cet élément doit appartenir au modèle M_{result}.

$$\begin{array}{l} ownedMembers_{receivingANDmerged} \triangleq \\ \forall\, e_1\, e_2, e_1 \in (ownedMembers\ M_{merged}) \\ \wedge (e_2 \in (ownedMembers\ M_{receiving}) \wedge\ matches(e_1, e_2)) \\ \rightarrow e_1 \in (ownedMembers\ M_{result}) \end{array}$$

4. Pour toutes les associations (respectivement les classes), contenues dans M_{merged}, et les associations (respectivement classes) contenues dans $M_{receiving}$, il existe une association, respectivement classe dans M_{result} qui est le résultat de la fonction merge des deux éléments de modèle.

$$\begin{array}{l} ownedClasses_{receivingANDmerged} \triangleq \\ \forall\, c_1\, c_2, c_1 \in (ownedClasses\ M_{merged}) \\ \wedge (c_2 \in (ownedClasses\ M_{receiving}) \wedge\ matches(c_1, c_2)) \\ \rightarrow \exists\, c_3 \in (ownedClasses\ M_{result}) \wedge\ mergeClasse(c_1, c_2, c_3) \end{array}$$

$$\begin{array}{l} ownedAssociations_{receivingANDmerged} \triangleq \\ \forall\, a_1 a_2, a_1 \in (ownedAssociations\ M_{merged}) \\ \wedge (a_2 \in (ownedAssociations\ M_{receiving}) \wedge\ matches(a_1, a_2)) \\ \rightarrow \exists\, a_3 \in (ownedAssociations\ M_{result}) \wedge\ mergeAssociation(a_1, a_2, a_3) \end{array}$$

   Les opérations `mergeAssociation` et `mergeClass` sont définies comme étant l'égalité de leurs trois paramètres.

5. M_{result} doit contenir uniquement les classes et les associations qui sont le résultat de la fusion de M_{merged} et $M_{receiving}$. Sans cette condition, rien ne peut empêcher M_{result} d'avoir des éléments additionnels provenant d'autres paquetages.

[1] *matches* est définie dans notre cas par l'égalité.

$$
\begin{aligned}
& ownedMembers_{receivingORmerged} \triangleq \\
& \forall\, e_1, e_1 \in (ownedMembers\ M_{result}) \\
& \rightarrow \exists\, e_2, (e_2 \in (ownedMembers\ M_{receiving}) \\
& \vee e_2 \in (ownedMembers\ M_{merged})) \wedge\ matches(e_1, e_2)
\end{aligned}
$$

6. Les deux métamodèles doivent contenir les mêmes contraintes sur les classes et les associations.

$$
\begin{aligned}
& mergeIsValid(M_{merged}, M_{receiving}) \triangleq \\
& \forall\, e_1\ e_2, e_1 \in (ownedClasses\ M_{merged}) \rightarrow e_2 \in (ownedClasses\ M_{receiving}) \\
& \rightarrow matches(e_1, e_2) \rightarrow (classConstraints(e_1, e_2)) \\
& \wedge \forall\, e_1\ e_2,\ e_1 \in (ownedProperties\ M_{merged}), e_2 \in (ownedProperties\ M_{receiving}) \\
& \rightarrow matches(e_1, e_2) \rightarrow (PropertyConstraints(e_1, e_2))
\end{aligned}
$$

Le prédicat `classConstraints` vérifie que la première classe a la même contrainte *areComposite* que la seconde (si elle existe).

$$
\begin{aligned}
& classConstraints(c_1, c_2) \triangleq \\
& \forall\, x_1\ x_2, \\
& (((c_1, Class), ownedAttribute, (r, Property)) \in ME_{receiving} \\
& \wedge((r, Property), ownedAttribute, (areComposite, Property)) \in ME_{receiving} \\
& \wedge((areComposite, Property), type, (x_1, Boolean)) \in ME_{receiving}) \\
& \wedge(((c_2, Class), ownedAttribute, (r, Property)) \in ME_{receiving} \\
& \wedge((r, Property), ownedAttribute, (areComposite, Property)) \in ME_{receiving} \\
& \wedge((areComposite, Property), type, (x_2, Boolean)) \in ME_{receiving}) \\
& \rightarrow x_1 = x_2
\end{aligned}
$$

Pour chaque référence, le prédicat *propertyConstraint* doit être satisfait.

$$
\begin{aligned}
& PropertyConstraints(r_1, r_2) \triangleq \\
& forall\ M, (conformsTO_{merged} \rightarrow (isOpposite\ r_1\ r_{1'}\ M) \\
& \rightarrow (conformsTO_{received} \rightarrow (isOpposite\ r_2\ r_{2'}\ M))
\end{aligned}
$$

Alors, le prédicat `mergePackage` vérifie qu'un modèle M_{result} est le résultat du merge de deux modèles $M_{receiving}$ et M_{merged}. Le prédicat est décrit comme suit :

$$
\begin{aligned}
& mergePackage(M_{receiving}, M_{merged}, M_{result}) \triangleq \\
& mergeIsValid(M_{receiving}, M_{merged}) \wedge\ ownedMembers_{receiving} \\
& \wedge\ ownedMembers_{merged} \wedge\ ownedMembers_{receivingANDmerged} \\
& \wedge\ ownedClasses_{receivingANDmerged} \wedge\ ownedAssociations_{receivingANDmerged} \\
& ownedMembers_{receivingORmerged}
\end{aligned}
$$

Les règles de correspondance basées sur l'égalité des noms : Si les noms de deux éléments dans les modèles `merged` et `receiving` correspondent, ils sont considérés comme le même élément s'ils sont du même type, ils sont alors fusionnés pour former un élément dans le paquetage résultat. La représentation des modèles dans COQ4MDE comme des graphes aplatis en utilisant les ensembles représentés comme des fonctions caractéristiques permet de faire ceci en utilisant l'opérateur `Union`.

D.2 L'Union

La version de l'`Union` présentée dans le chapitre 3 est décrite sur des modèles qui peuvent être des métamodèles exprimés comme des modèles conformes à `MOF`. Cette version de l'`Union` est décrite directement sur la structure de métamodèle.

$$\begin{aligned}&metaUnion\ \langle MV_1, ME_1\rangle, conformsTo_1\rangle\ \langle MV_2, ME_2, conformsTo_2\rangle \triangleq\\ &UnionIsValid\ \langle MV_1, ME_1\rangle, conformsTo_1\rangle\ \langle MV_2, ME_2, conformsTo_2\rangle\\ &\rightarrow \langle\langle MV_1 \cup MV_2, ME_1 \cup ME_2\rangle, conformsTo_1\ and\ conformsTo_2\rangle\end{aligned}$$

Le prédicat `UnionIsValid` vérifie que les classes et les références contenues dans les deux métamodèles ont les mêmes contraintes.

D.3 Les preuves d'équivalence sémantique

Le but est de prouver que l'union est sémantiquement équivalente à l'opérateur du fusion. Ceci peut être fait en prouvant le théorème `SemanticsEqMergeUnion`[2]. Le théorème montre que le résultat de l'application de l'opérateur `metaUnion` est aussi le résultat du merge des deux modèles : M_{merged} et $M_{receiving}$. La preuve nécessite la validation de chaque règle présentée dans 4.1.

$$\texttt{Theorem}\ SemanticsEqMergeUnion : \forall\ M_{merged},\ M_{receiving} \in MetaModel,\\ mergePackage\ M_{merged}\ M_{receiving}\ (metaUnion\ M_{merged}\ M_{receiving})$$

D.4 Exemple

Cet exemple est inspiré de [ZD06]. La figure D.1 est le modèle du paquetage `BasicEmployee` dont le contenu doit être étendu avec l'opération de fusion, on l'appelle le paquetage `receiving`. La figure D.2 est le modèle `receiving` dans la notation COQ4MDE.

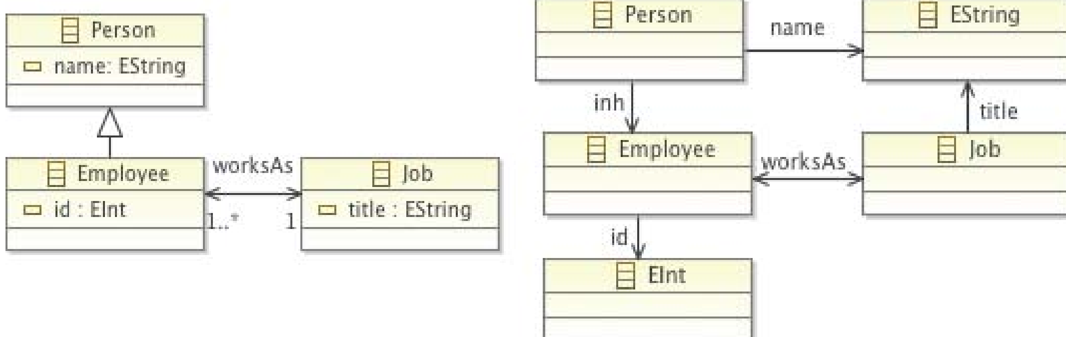


Figure D.1 — Le métamodèle `receiving` *Figure D.2* — Le métamodèle `receiving` (notation COQ4MDE)

[2] `http://www.irit.fr/~Mounira.Kezadri/FormalMDE/PackageMerge.html`

Plus formellement, le métamodèle $M_{receiving}$ est décrit dans COQ4MDE avec ses trois composantes : $MV_{receiving}$, $ME_{receiving}$ et $conformsTo_{receiving}$ comme suit :

$$MV_{receiving} = \{Person, Employee, Job, EString, EInt\}$$

$$ME_{receiving} = \{(Person, inh, Employee), (Person, name, EString),$$
$$((Employee, worksAs, Job), (Job, title, EString), (Employee, id, EInt), (Job, worksAs, Employee)\}$$

$$conformsTo_{receiving} = InstanceOf\ M_{receiving} \wedge\ subClass\ Person\ Employee$$
$$\wedge\ lower\ Employee\ worksAs\ 1 \wedge\ upper\ Employee\ worksAs\ 1 \wedge\ lower\ Job\ worksAs\ 1$$

La figure D.3 représente le paquetage `EmployeeLocation`, il contient les éléments qui doivent être intégrés dans le paquetage `receiving`, on l'appelle le paquetage `merged`. La figure D.4 est le modèle `merged` dans la notation COQ4MDE.

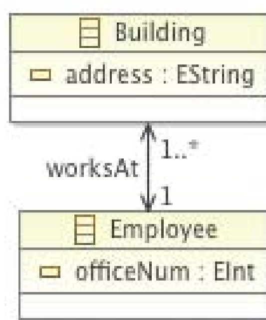


Figure D.3 — Le métamodèle `merged`

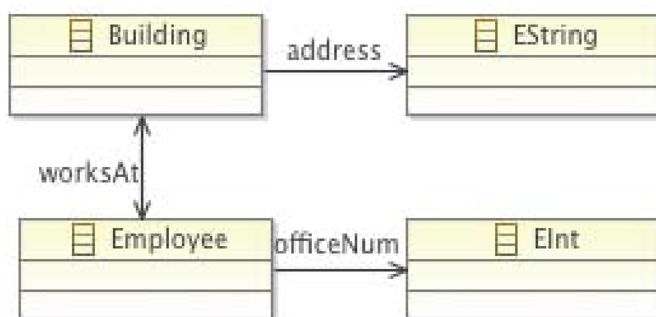


Figure D.4 — Le métamodèle `merged` (notation COQ4MDE)

Plus formellement, le métamodèle M_{merged} est décrit dans COQ4MDE avec ses trois composantes : MV_{merged}, ME_{merged} et $conformsTo_{merged}$ comme suit.

$$MV_{merged} = \{Building, Employee, EString, EInt\}$$

$$ME_{merged} = \{(Building, address, EString), ((Employee, worksAt, Building),$$
$$(Building, worksAt, Employee), (Employee, officeNum, EInt)\}$$

$$conformsTo_{merged} = InstanceOf\ M_{merged} \wedge\ lower\ Employee\ worksAt\ 1$$
$$\wedge\ lower\ Building\ worksAt\ 1 \wedge\ upper\ Employee\ worksAt\ 1$$

La figure D.5 est le résultat attendu de la fusion des deux paquetages. L'opérateur `metaUnion` permet d'obtenir la fusion des deux métamodèles. Cet opérateur vérifie le prédicat `unionIsValid` avant l'application de l'opérateur. L'application de l'opérateur `metaUnion` sur les métamodèles $M_{receiving}$ et M_{merged} génère le métamodèle M_{result} décrit avec MV_{result}, ME_{result} et $conformsTo_{result}$ et qui correspond exactement au résultat attendu de la fusion des deux métamodèles.

$$MV_{result} = \{Building, Employee, EString, EInt, Person, Job\}$$

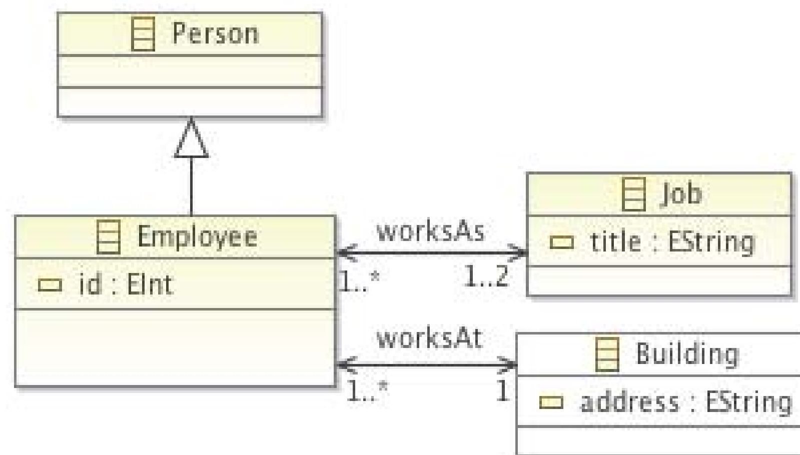


Figure D.5 — Le metamodèle `resulting`

$ME_{result} = \{(Building, address, EString), (Employee, worksAt, Building),$
$(Employee, officeNum, EInt), (Person, inh, Employee),$
$(Person, name, EString), (Employee, worksAs, Job), (Job, title, EString),$
$(Employee, id, EInt), (Building, worksAt, Employee), (Job, worksAs, Employee)\}$

$conformsTo_{result} =$
$InstanceOf\ M_{merged} \wedge\ lower\ Employee\ worksAt\ 1 \wedge\ lower\ Building\ worksAt\ 1$
$\wedge\ upper\ Employee\ worksAt\ 1 \wedge\ InstanceOf\ M_{receiving} \wedge\ subClass\ Person\ Employee$
$\wedge\ lower\ Employee\ worksAs\ 1 \wedge\ upper\ Employee\ worksAs\ 1 \wedge\ lower\ Job\ worksAs\ 1$

La figure D.6 est le métamodèle `resulting` décrit avec la notation COQ4MDE.

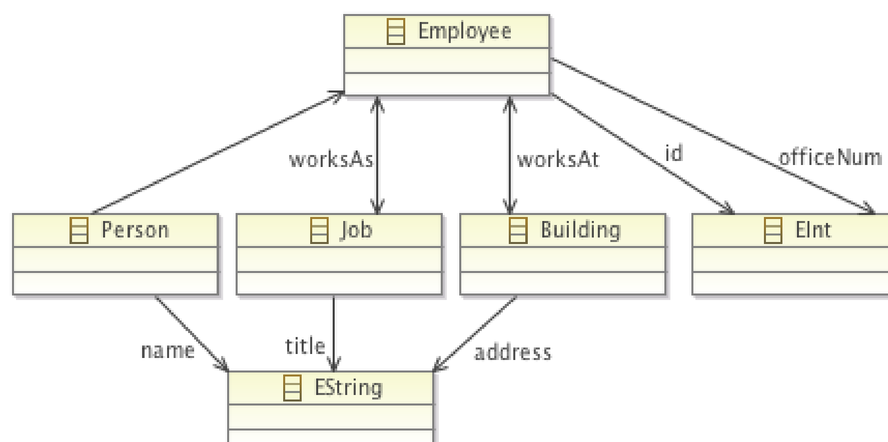


Figure D.6 — Le métamodèle `resulting` dans la notation COQ4MDE

E Les fichiers Coq

Table des matières

E.1 Les définitions et les propriétés 157
E.2 Les preuves 158

Nos développements Coq sont accessibles sur la page web[1] générée automatiquement avec Coqdoc ou sur la page web FormalAssembly [2] avec une interface plus conviviale. Voici la liste des fichiers Coq :

E.1 Les dé nitions et les propriétés

- Util.v (615 lignes de code) : Des propriétés pour les fonctions caractéristiques et diverses propriétés.
- Graph.v (2047 lignes de code) : La définition de la structure du graphe et ses propriétés.
- meta.v (329 lignes de code) : La définition des modèles et leurs opérateurs.
- Def.v (164 lignes de code) : Des définitions pour la manipulation de modèles.
- Properties.v (1482 lignes de code) : La définition des propriétés `EMOF`.
- MM_Conforms.v (175 lignes de code) : La définition de la notion $instanceOf$.
- Frag_Interface_Extraction.v (373 lignes de code) : La définition de l'extraction de l'interface de fragments.
- PackageMerge.v (290 lignes de code) : La définition de la fusion de paquetages.

[1] http://www.irit.fr/~Mounira.Kezadri/FormalMDE/
[2] http://www.irit.fr/~Mounira.Kezadri/FormalAssembly.html

E.2 Les preuves

- ▷ Frag_Ext_Verif.v (716 lignes de code) : La vérification des propriétés pour l'extraction de l'interface des fragments.
- ▷ Interface_Elim.v (274 lignes de code) : La vérification des propriétés pour l'élimination de l'interface des fragments.
- ▷ Subst_Verif.v (2656 lignes de code) : La vérification des propriétés de l'opérateurs `Substitution`.
- ▷ Bind2M_Verif.v (787 lignes de code) : La vérification des propriétés pour l'opérateur `bind` de deux modèles.
- ▷ Union_Verif.v (733 lignes de code) : La vérification de propriétés pour l'opérateur `Union`.
- ▷ Extend_Verif.v (746 lignes de code) : La vérification des propriétés pour l'opérateur `extend` basé sur l'union disjointe.
- ▷ ExtendCU_Verif.v (2425 lignes de code) : La vérification des propriétés de l'opérateur `extend` basé sur l'Union classique.

Bibliographie

[ACHH93] R. Alur, C. Courcoubetis, T. Henzinger, and P. Ho. Hybrid automata : An algorithmic approach to the specification and verification of hybrid systems. *Hybrid systems*, pages 209–229, 1993. 18

[ALN+91] J.R. Abrial, M.K.O. Lee, D.S. Neilson, P.N. Scharbach, and I.H. Sørensen. The b-method. In *VDM 91 Formal Software Development Methods*, pages 398–405. Springer, 1991. 50

[Ana12] K. Anastasakis. Uml2alloy-reference manual, 2012. 50

[And09] C. André. Syntax and semantics of the clock constraint specification language (ccsl). *INRIA Research Report*, 2009. `http://hal.inria.fr/inria-00384077/PDF/RR-6925.pdf`. 131

[Ash08] P.J. Ashenden. *The designer s guide to VHDL*, volume 3. Morgan Kaufmann, 2008. 24

[Asp00] D. Aspinall. Proof general : A generic tool for proof development. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 38–43. Springer, 2000. 40

[Aßm03] U. Aßmann. *Invasive software composition*. Springer, 2003. 88, 92, 111, 112, 113

[Awo10] S. Awodey. Category theory, volume 52 of oxford logic guides, 2010. 116

[B6̈0] Richard J. Büchi. Weak Second-Order Arithmetic and Finite Automata. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 6(1-6) :66–92, 1960. 18

[BAG03] D. Bourigault and N. Aussenac-Gilles. Construction d'ontologies à partir de textes. In *Actes de la 10ème conférence annuelle sur le Traitement Automatique des Langues*, pages 27–50, 2003. 13

[Bar97] F.J. Barros. Modeling formalisms for dynamic structure systems. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 7(4) :501–515, 1997. 20

[BB05] O. Bodenreider and A. Burgun. Biomedical ontologies. *Medical Informatics*, pages 211–236, 2005. 27

[BBNS10] S. Bensalem, M. Bozga, T.H. Nguyen, and J. Sifakis. Compositional verification for component-based systems and application. *Software, IET*, 4(3) :181–193, 2010. 71

[BBS06] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *Software Engineering and Formal Methods, 2006. SEFM 2006. Fourth IEEE International Conference on*, pages 3–12. IEEE, 2006. 71, 131

[BC11] Y. Bertot and P. Castéran. Le coq'art (v8). 2011. `http://www.labri.fr/perso/casteran/CoqArt/coqartF.pdf`. 41

[BCC06] G. Brusa, M.L. Caliusco, and O. Chiotti. A process for building a domain ontology : an experience in developing a government budgetary ontology. In *Proceedings of the second Australasian workshop on Advances in ontologies-Volume 72*, pages 7–15. Australian Computer Society, Inc., 2006. 12

[BCCLG13] Franck Barbier, Pierre Castéran, Eric Cariou, and Olivier Le Goaer. Adaptive Software based on Correct-by-Construction Metamodels. In B. Cristina Pelayo Gracia-Bustelo O. Sanjuan Martinez V. Garcia Diaz, J.M. Cueva Lovelle, editor, *Progressions and Innovations in Model-Driven Software Engineering*, Advances in Systems Analysis, Software Engineering, and High Performance Computing (ASASEHPC), pages 308–325. IGI Global, July 2013. 49

[Béz05] J. Bézivin. On the unification power of models. *Software and Systems Modeling*, 4(2) :171–188, 2005. 36

[BGG+92] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with embedding hardware description languages in hol. In *Proceedings of the IFIP TC10/WG*, volume 10, pages 129–156. Citeseer, 1992. 48

[BHP00] P.A. Bernstein, A.Y. Halevy, and R.A. Pottinger. A vision for management of complex models. *ACM Sigmod Record*, 29(4) :55–63, 2000. 39

[BHS05] F. Baader, I. Horrocks, and U. Sattler. Description logics as ontology languages for the semantic web. *Mechanizing Mathematical Reasoning*, pages 228–248, 2005. 11

[BK+08] C. Baier, J.P. Katoen, et al. *Principles of model checking*, volume 26202649. MIT press, 2008. 11, 22, 23, 24, 25

[Bla08] C. Blais. Toward a verification, validation, and accreditation (VV&A) ontology. Technical report, DTIC Document, 2008. 27

[BM10] A. Boronat and J. Meseguer. An algebraic semantics for mof. *Formal Asp. Comput.*, 22(3-4) :269–296, 2010. 48

[Boh13] N. Bohr. I. on the constitution of atoms and molecules. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 26(151) :1–25, 1913. 36

[BRV04] B. Berthomieu*, P.O. Ribet, and F. Vernadat. The tool tina–construction of abstract state spaces for petri nets and time petri nets. *International Journal of Production Research*, 42(14) :2741–2756, 2004. 25

[BS11] X. Blanc and O. Salvatori. *MDA en action : Ingénierie logicielle guidée par les modèles*. Eyrolles, 2011. 37, 38

[BW03] D. Bjørner and M. Wirsing. *An Ontology for a TripTych Formal Software Development*. Springer-Verlag, 2003. 28

[BW06] A.D. Brucker and B. Wolff. The hol-ocl book. 2006. 49

[BY03] J. Bengtsson and W. Yi. Timed automata : Semantics, algorithms and tools. *Lectures on Concurrency and Petri Nets*, pages 87–124, 2003. 18

[CCO+04] S. Chaki, E.M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/event-based software model checking. In *Integrated Formal Methods*, pages 128–147. Springer, 2004. 24

[CCS98] ACM CCS. Acm computing classification system [1998 version], 1998. 28

[CDE+02] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F. Quesada. Maude : specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2) :187–243, 2002. 48

[CDGL+05] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Dl-lite : Tractable description logics for ontologies. In *Proceedings of the National Conference on Arti cial Intelligence*, volume 20, page 602. Menlo Park, CA ; Cambridge, MA ; London ; AAAI Press ; MIT Press ; 1999, 2005. 11

[CE08] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *25 Years of Model Checking*, pages 196–215, 2008. 24

[CFG10] J. Cervelle, E. Formenti, and P. Guillon. Ultimate Traces of Cellular Automata. *Arxiv preprint arXiv :1001.0251*, 2010. 18

[CFL+04] P. Castells, B. Foncillas, R. Lara, M. Rico, and J. Alonso. Semantic web technologies for economic and financial information management. *The Semantic Web : Research and Applications*, pages 473–487, 2004. 27

[CH+86] T. Coquand, G. Huet, et al. The calculus of constructions. *INRIA Research Report*, 1986. 40

[Chl11] A. Chlipala. Certified programming with dependent types, 2011. 47

[Chr03] J. Chrzaszcz. Implementing modules in the coq system. *Theorem Proving in Higher Order Logics*, pages 270–286, 2003. 40, 57, 58

[Cla02] S. Clarke. Extending standard UML with model composition semantics. *Science of Computer Programming*, 44(1) :71–100, 2002. 39

[CLST11] D. Calegari, C. Luna, N. Szasz, and Á. Tasistro. A type-theoretic framework for certified model transformations. In *Formal Methods : Foundations and Applications*, pages 112–127. Springer, 2011. 49

[Coq01] C. Coquand. The interactive theorem prover agda, 2001. 39

[Cru86] D.A. Cruse. *Lexical semantics*. Cambridge University Press, 1986. 13

[CTB12] B. Combemale, X. Thirioux, and B. Baudry. Formally defining and iterating infinite models. In *Model Driven Engineering Languages and Systems*, pages 119–133. Springer, 2012. 56

[CTG+11] H Chale, O. Taofifenua, T. Gaudré, A. Topa, N. Lévy, and J.L. Boulanger. Reducing the gap between formal and informal worlds in automotive safety-critical systems. In *INCOSE Symposium*, 2011. 28

[CW96] E.M. Clarke and J.M. Wing. Formal methods : State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4) :626–643, 1996. 28

[DDFV09] M. Didonet Del Fabro and P. Valduriez. Towards the efficient development of model transformations using model weaving and matching transformations. *Software and System Modeling*, 8(3) :305–324, 2009. 39

[Del00] D. Delahaye. A tactic language for the system coq. In *Logic for Programming and Automated Reasoning*, pages 377–440. Springer, 2000. 40, 100

[DGD06] D. Djuric, D. Gasevic, and V. Devedzic. The tao of modeling spaces. *Journal of Object Technology*, 5(8) :125–147, 2006. 15

[DK05] P.R. D'Argenio and J.P. Katoen. A theory of stochastic systems part I : Stochastic automata. *Information and computation*, 203(1) :1–38, 2005. 18

[DS+08] S. Després, S. Szulman, et al. Réseau terminologique versus ontologie. *Toht 2008*, pages 17–34, 2008. 13

[EC80] E.A. Emerson and E.M. Clarke. *Characterizing correctness properties of parallel programs using xpoints*. Springer, 1980. 25

[EH86] E.A. Emerson and J.Y. Halpern. "sometimes" and "not never" revisited : on branching versus linear time temporal logic. *Journal of the ACM (JACM)*, 33(1) :151–178, 1986. 24

[EHK+96] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. *Algebraic Approaches to Graph Transformation : Part II : Single Pushout Approach and Comparison with Double Pushout Approach*. Citeseer, 1996. 119

[Erl08] T. Erl. *Soa : principles of service design*, volume 1. Prentice Hall Upper Saddle River, 2008. 130

[FDH11] J.V. Fonou-Dombeu and M. Huisman. Combining ontology development methodologies and semantic web platforms for e-government domain ontology development. *International Journal of Web & Semantic Technology (IJWesT)*, 2(2) :12–25, 2011. 12

[FH04] C. Ferdinand and R. Heckmann. ait : Worst-case execution time prediction by static program analysis. In *Building the Information Society*, pages 377–383. Springer, 2004. 11

[FL99] M. Fernández-López. Overview of methodologies for building ontologies. 1999. 12

[FT13] M. Fernández and J. Terrell. Assembling the proofs of ordered model transformations. *arXiv preprint arXiv :1302.5174*, 2013. 49

[G+93] T.R. Gruber et al. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2) :199–220, 1993. 9

[G+95] T.R. Gruber et al. Toward principles for the design of ontologies used for knowledge sharing. *International journal of human computer studies*, 43(5) :907–928, 1995. 13

[GAA+13] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S.O. Biha, et al. A machine-checked proof of the odd order theorem. 2013. 39

[GBF13] M. Garnacho, J.P. Bodeveix, and M. Filali. Mechanized Semantics of Real-Time Concurrent Systems. *IRIT Research Report*, 2013. `http://www.irit.fr/~Manuel.Garnacho/Publications/MechRT.pdf`. 131

[GC05] E. Giménez and P. Castéran. A tutorial on [co-] inductive types in coq, 2005. `http://coq.inria.fr/distrib/v8.2/files/RecTutorial.pdf`. 40

[GDD09] D. Gaševic, D. Djuric, and V. Deved ic. *Model driven engineering and ontology development*. Springer, 2009. 36

[GGMR09] F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In *Theorem Proving in Higher Order Logics*, pages 327–342. Springer, 2009. 119

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL : a theorem proving environment for higher order logic*. Cambridge University Press, 1993. 39

[Gol06] R. Goldblatt. *Topoi : the categorial analysis of logic*, volume 98. Dover Publications, 2006. 116

[Gon] G. Gonthier. A computer-checked proof of the four colour theorem, 2005. *URL http ://research. microsoft. com/ gonthier/4colproof. pdf*. 39

[GPFLC91] A. Gomez-Perez, M. Fernández-López, and O. Corcho. Ontological engineering. *AI Magazine*, 36 :56, 1991. 12

[GRL07] O. Gendreau, P.N. Robillard, and P. Labreche. Peer review as a v v standard compliance technique : An aviation industry case study. In *Electrical and Computer Engineering, 2007. CCECE 2007. Canadian Conference on*, pages 679 –681, april 2007. 23

[GSMP11] M. Giorgino, M. Strecker, R. Matthes, and M. Pantel. Verification of the schorr-waite algorithm–from trees to graphs. *Logic-Based Program Synthesis and Transformation*, pages 67–83, 2011. 49, 116

[H+02] I. Horrocks et al. Daml+oil : A description logic for the semantic web. *IEEE Data Engineering Bulletin*, 25(1) :4–9, 2002. 11

[Hen09] J. Henriksson. *A Lightweight Framework for Universal Fragment Composition : With an Application in the Semantic Web*. PhD thesis, 2009. 88

[Héo10] M. Héon. *OntoCASE : méthodologie et assistant logiciel pour une ingénierie ontologique fondée sur la transformation d un modèle semi-formel*. PhD thesis, Université de Québec à Montréal, 2010. 30

[Hil00] R. Hilliard. Ieee-std-1471-2000 recommended practice for architectural description of software-intensive systems. *IEEE, http ://standards. ieee. org*, 2000. 12, 16, 20

[HKR+04] M. Horridge, H. Knublauch, A. Rector, R. Stevens, and C. Wroe. A practical guide to building owl ontologies using the protégé-owl plugin and co-ode tools edition 1.0. *University of Manchester*, 2004. 16

[HM01] V. Haarslev and R. Müller. Racer system description. *Automated Reasoning*, pages 701–705, 2001. 14

[HMP92] T. A Henzinger, Z. Manna, and A. Pnueli. Timed transition systems. In *Real-Time : Theory in Practice*, pages 226–251. Springer, 1992. 131

[HMU79] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to automata theory, languages, and computation*, volume 2. Addison-wesley Reading, MA, 1979. 18

[HPWD09] T. Hardin, F. Pessaux, P. Weis, and D. Doligez. Focalize-reference manual. *LIP6-CEDRIC, dec*, 2009. 50

[HS73] H. Herrlich and G.E. Strecker. *Category theory*. Allyn and Bacon Boston, 1973. 116

[HZG03] Q. Huo, H. Zhu, and S. Greenwood. A multi-agent software environment for testing web-based applications. *COMPSAC-NEW YORK-*, pages 210–215, 2003. 27

[ISO12] ISO. Road vehicles – functional safety – part 10 : Guideline on iso 26262. *Available on : http ://www.iso.org/*, 2012. 29

[Jac12] D. Jackson. Software abstractions-logic, language, and analysis, revised edition, 2012. 50, 84

[JB06] F. Jouault and J. Bézivin. Km3 : a dsl for metamodel specification. In *Formal Methods for Open Object-Based Distributed Systems*, pages 171–185. Springer, 2006. 52

[JBCV98] D. Jones, T. Bench-Capon, and P. Visser. Methodologies for ontology development. In *Proc. IT&KNOWS Conference of the 15th IFIP World Computer Congress*, pages 20–35. Citeseer, 1998. 12

[Jea08] C. Jeanneret. An analysis of model composition approaches. Master's thesis, Ecole Polytechnique Fédérale de Lausanne, 2007-2008. 39

[Joh11] Jendrik Johannes. *Component-Based Model-Driven Software Development*. PhD thesis, vorgelegt an der Technischen Universität Dresden Fakultät Informatik, 2011. 88, 89, 91, 92, 95, 112

[KCP+11] M. Kezadri, B. Combemale, M. Pantel, X. Thirioux, et al. A proof assistant based formalization of components in mde. In *8th International Symposium on Formal Aspects of Component Software (FACS 2011)*, 2011. 59

[KL00] W.D. Kelton and A.M. Law. *Simulation modeling and analysis*. McGraw Hill Boston, MA, 2000. 11

[KLM+97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, June 1997. 39

[KP10a] M. Kezadri and M. Pantel. First steps toward a Verification and validation ontology (student paper). In *International Conference on Knowledge Engineering and Ontology Development (KEOD), Valencia, Spain, 25/10/2010-28/10/2010*, page (electronic medium), http ://www.insticc.net, octobre 2010. INSTICC - Institute for Systems and Technologies of Information, Control and Communication. 7

[KP10b] Mounira Kezadri and Marc Pantel. Premières expériences pour l'édition correcte par construction de modèles (short paper). In Yamine Aït-Ameur, editor, *Approches Formelles dans l Assistance au Développement de Logiciels (AFADL), LISI / ENSMA, Poitiers, France, 06/06/2010-11/06/2010*, pages 217–221, http ://www.lisi.ensma.fr, juin 2010. LISI-ENSMA. 64

[KP12] M. Kezadri and M. Pantel. First steps toward a Verification and validation ontology (regular paper). In *International Conference on Embedded Real Time Software and Systems (ERTS2), Toulouse, France, 01/02/2012-03/02/2012*, page (electronic medium). SIA/3AF/SEE, février 2012. 7

[KRPP10] D. Kolovos, L. Rose, R. Paige, and F.A.C. Polack. The epsilon book. *Structure*, 178, 2010. 39

[KT06] M. Korotkiy and J. Top. Onto-soa : from ontology-enabled soa to service-enabled ontologies. In *Telecommunications, 2006. AICT-ICIW 06. International Conference*

on Internet and Web Applications and Services/Advanced International Conference on, pages 124–124. IEEE, 2006. 130

[Küh06] Thomas Kühne. Matters of (meta-) modeling. *Software & Systems Modeling*, 5(4) :369–385, 2006. 49

[Lam95] L. Lamport. How to write a proof. *The American mathematical monthly*, 102(7) :600–608, 1995. 149

[Law05] MV Lawson. Finite automata. *Handbook of networked and embedded control systems*, pages 117–143, 2005. 18

[LCR+97] M. Loewe, U. Corradini, Montanari, F. Rossi, H. Ehrig, H. Ehrig, R. Heckel, and Owe. *Algebraic approaches to graph transformation part I : Basic concepts and double pushout approach*, volume 97, pages 163–245. World Scientific Publishing Co., Inc., 1997. 119

[Let05] K. Letkeman. Comparing and merging uml models in ibm rational software architect. *IBM Rational, July*, 2005. 39

[Let08] P. Letouzey. Extraction in coq : An overview. *Logic and Theory of Algorithms*, pages 359–369, 2008. 40

[LJ99] E.A. Lee and II. John. *Overview of the ptolemy project*. Electronics Research Laboratory, College of Engineering, University of California, 1999. 71

[LP92] Z. Luo and R. Pollack. *LEGO proof development system : User s manual*. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science, 1992. 39

[LSV96] E. A Lee and A. Sangiovanni-Vincentelli. The tagged signal model-a preliminary version of a denotational framework for comparing models of computation. *Memorandum UCB/ERL M*, 96, 1996. 71

[LYX09] Q. Liu, Z. Ynag, and J. Xie. Description and verification of pattern-based composition in coq. In *Advances in Computational Science and Engineering*, pages 231–245. Springer, 2009. 71

[Mar11] J. Marchand. Formal and tool-supported operator for multi-formalism modelling. 2011. 126

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4) :184–195, 1960. 46

[MCF03] S.J. Mellor, T. Clark, and T. Futagami. Model-driven development : guest editors' introduction. *IEEE software*, 20(5) :14–18, 2003. 37

[Mci68] D. Mcilroy. Mass-Produced Software Components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 88–98, 1968. 2

[MD10] F. Madiot and G. Dupé. Emf facet : A non-intrusive tooling to extend metemodels, 2010. 39

[Meg12] A. Megacz. Hardware design with generalized arrows. In Andy Gill and Jurriaan Hage, editors, *Implementation and Application of Functional Languages*, Lecture Notes in Computer Science, pages 164–180. Springer Berlin / Heidelberg, 2012. 119

[MFJ05] P.A. Muller, F. Fleurey, and J.M. Jézéquel. Weaving executability into object-oriented meta-languages. In *Model Driven Engineering Languages and Systems*, pages 264–278. Springer, 2005. 131

[MH88] M.L. Minsky and J. Henry. *La société de l esprit*. InterEditions, 1988. 36

[ML75] P. Martin-Löf. An intuitionistic theory of types : Predicative part. *Studies in Logic and the Foundations of Mathematics*, 80 :73–118, 1975. 40

[MLS97] E. Mikk, Y. Lakhnechi, and M. Siegel. Hierarchical automata as model for statecharts. *Advances in Computing Science—ASIAN 97*, pages 181–196, 1997. 18

[MPSP+09] B. Motik, P.F. Patel-Schneider, B. Parsia, C. Bock, A. Fokoue, P. Haase, R. Hoekstra, I. Horrocks, A. Ruttenberg, U. Sattler, et al. Owl 2 web ontology language : Structural specification and functional-style syntax. *W3C Recommendation*, 27, 2009. 16

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The de nition of Standard ML*. MIT press, 1997. 46

[MVH+04] D.L. McGuinness, F. Van Harmelen, et al. OWL web ontology language overview. *W3C recommendation*, 10 :2004–03, 2004. 15

[NPW02] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL : a proof assistant for higher-order logic*, volume 2283. Springer, 2002. 39, 49

[OJ04] J Souquières D Okalas Ossami and JP Jacquot. Construction de spécifications multi-vues uml et b. *Journal en ligne : Informations, Savoirs, Décisions & Médiations (ISDM)*, 2004. 50

[OMG07] OMG. Unified modeling language (omg uml)-infrastructure(v2. 1.2). *Available on : http ://www. omg. org/spec/UML/2.1.2*, 2, 2007. 76

[OMG09] OMG. Ontology definition metamodel(v1.0). *Available on : http ://www.omg.org/spec/ODM/1.0*, 2009. 16

[OMG11a] OMG. Omg meta object facility (mof) core specification (version 2.4.1). *Available on : http ://www.omg.org/spec/MOF/2.4.1*, 2.4.1, 2011. 53

[OMG11b] OMG. Unified modeling language (omg uml)-infrastructure(v2.4.1). *Available on : http ://www.omg.org/spec/UML/2.4.1*, 2.4.1, 2011. 75, 76, 81, 86

[OMG12] OMG. Object constraint language, version 2.3.1. *http ://www.omg.org/spec/OCL/2.3.1/PDF/*, 2012. 37

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. Pvs : A prototype verification system. In *Automated Deduction—CADE-11*, pages 748–752. Springer, 1992. 50

[Per04] D. Perrin. *In nite words : automata, semigroups, logic and games*. Academic Press, 2004. 18

[Pie91] B.C. Pierce. *Basic category theory for computer scientists*. MIT press, 1991. 116

[PM11] C. Picard and R. Matthes. Coinductive graph representation : the problem of embedded lists. *Electronic Communications of the EASST, Special issue Graph Computation Models, GCM 10*, 2011. 49, 116

[PM12] C. Paulin-Mohring. *Course notes LASER summerschool 2011*, chapter Introduction to the Coq proof-assistant for practical software verification. Lecture Notes in Computer Science. Springer-Verlag, 2012. to appear. 41

[PMV03] T. Pender, E. McSheffrey, and L. Varveris. *UML bible*. Wiley Chichester, 2003. 37

[Pnu77] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977. 24

[Poe06] I. Poernomo. The meta-object facility typed. In *SAC*, pages 1845–1849, 2006. 48, 49

[Poe08] I. Poernomo. Proofs-as-model-transformations. In *ICMT*, pages 214–228, 2008. 48

[PP93] F. Parisi-Presicce. Single vs. double pushout derivations of graphs. In *Graph-Theoretic Concepts in Computer Science*, pages 248–262. Springer, 1993. 119

[Pri03] Arthur N. Prior. *Time and modality*. Oxford University Press, 2003. 24

[PT10] I. Poernomo and J. Terrell. Correct-by-construction model transformations from partially ordered specifications in coq. In *ICFEM*, pages 56–73, 2010. 48, 49

[QS82] J. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on Programming*, pages 337–351. Springer, 1982. 25

[RB88] D.E. Rydeheard and R.M. Burstall. *Computational category theory*, volume 152. Prentice Hall Englewood Cliffs, 1988. 126

[RG05] I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In *Computer Aided Veri cation*, pages 82–97. Springer, 2005. 11

[RKL+05] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web service modeling ontology. *Applied Ontology*, 1(1) :77–106, 2005. 13

[RRRDV07] J. Raúl Romero, J.E. Rivera, F. Durán, and A. Vallecillo. Formal and tool support for model driven engineering with maude. *Journal of Object Technology*, 6(9) :187–207, 2007. 48

[RTC12a] RTCA / EUROCAE. "DO-178/ED-12 : Software considerations in airborne systems and equipment certification", 2012. 1, 29

[RTC12b] RTCA / EUROCAE. "DO-330/ED-215 : Software Tool Qualification Considerations" - clarifying software tools and avionics tool qualification, 2012. 1, 29

[RW11] N. Rozanski and E. Woods. *Software systems architecture : working with stakeholders using viewpoints and perspectives*. Addison-Wesley Professional, 2011. 20

[Saï97] A. Saïbi. Typing algorithm in type theory with inheritance. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 292–301. ACM, 1997. 40

[SB06] C. Snook and M. Butler. Uml-b : Formal modeling and design aided by uml. *ACM Trans. Softw. Eng. Methodol.*, 15(1) :92–122, January 2006. 50

[Sel03] B. Selic. The pragmatics of model-driven development. *Software, IEEE*, 20(5) :19–25, 2003. 37

[SMB97] B. Steffen, T. Margaria, and V. Braun. The electronic tool integration platform : concepts and design. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1) :9–30, 1997. 28

[SMH08] R. Shearer, B. Motik, and I. Horrocks. Hermit : A highly-efficient owl reasoner. In *Proceedings of the 5th International Workshop on OWL : Experiences and Directions (OWLED 2008)*, pages 26–27, 2008. 14

[SO08] M. Sozeau and N. Oury. First-class type classes. In *Theorem Proving in Higher Order Logics*, pages 278–293. Springer, 2008. 119

[SPG+07] E. Sirin, B. Parsia, B.C. Grau, A. Kalyanpur, and Y. Katz. Pellet : A practical owl-dl reasoner. *Web Semantics : science, services and agents on the World Wide Web*, 5(2) :51–53, 2007. 14

[Sta81] R.M. Stallman. *EMACS the extensible, customizable self-documenting display editor*, volume 2. ACM, 1981. 40

[TCCG07] X. Thirioux, B. Combemale, X. Crégut, and P.L. Garoche. A Framework to Formalise the MDE Foundations. In Richard Paige and Jean Bézivin, editors, *International Workshop on Towers of Models (TOWERS)*, pages 14–30, Zurich, June 2007. 47, 52, 142

[Tea12] The Coq Development Team. The coq proof assistant reference manual : Version 8.4. 2012. 40, 100, 101

[Tiw08] A. Tiwari. Abstractions for hybrid systems. *Formal Methods in System Design*, 32(1) :57–83, 2008. 17

[TM02] D.E. Thomas and P.R. Moorby. *The Verilog® Hardware Description Language*, volume 2. Springer, 2002. 24

[Tsa] D. Tsarkov. Owl : fact++. 14

[TV10] J. Troya and A. Vallecillo. Towards a rewriting logic semantics for atl. In *ICMT*, pages 230–244, 2010. 48

[Usc96] M. Uschold. Building ontologies : Towards a unified methodology. *Technical report-University of Edinburgh arti cial intelligence applications institute AIAI TR*, 1996. 12

[VBC98] P.R.S. Visser and T.J.M. Bench-Capon. A comparison of four ontologies for the design of legal knowledge systems. *Arti cial Intelligence and Law*, 6(1) :27–57, 1998. 27

[VD86] D. Van Dalen. Intuitionistic logic. *Handbook of philosophical logic*, 3 :225–339, 1986. 40

[WC53] J.D. Watson and F.H.C. Crick. A structure for deoxyribose nucleic acid. *Nature*, 421(6921) :397–3988, 1953. 36

[XB03] F. Xie and J.C. Browne. Verified systems by composition from verified components. *ACM SIGSOFT Software Engineering Notes*, 28(5) :277–286, 2003. 71

[ZD06] A. Zito and J. Dingel. Modeling uml2 package merge with alloy. In *First Alloy Workshop*, 2006. 50, 154

[Zit06] A.P. Zito. *UML s Package Extension Mechanism : Taking a Closer Look at Package Merge*. Queen's University, 2006. 76, 78, 81, 84, 86, 151

[Zwe94] P. Zweigenbaum. Menelas : an access system for medical records using natural language. *Computer methods and programs in Biomedicine*, 45(1) :117–120, 1994. 27

Liste des figures

1.1 La classe SymbolicSimulation et sa relation avec la classe Simulation dans le fichier OWL . . . 11
1.2 La relation entre la classe CBMC et ModelChecking présentée dans protégé . 12
1.3 Un réseau terminologique partiel associé au terme bruit . . . 13
1.4 L'architecture globale du VVO . . . 17
1.5 Une partie du premier niveau de la hiérarchie des formalismes . . . 18
1.6 Une partie de la hiérarchie du formalisme Automata . . . 19
1.7 La définition du concept FiniteAutomata . . . 19
1.8 Une partie de la hiérarchie du FormalismElement . . . 19
1.9 Une partie de la hiérarchie des points de vue . . . 21
1.10 La définition du concept concurrency . . . 21
1.11 Une partie de la hiérarchie des techniques de V&V . . . 22
1.12 La définition du PeerReview . . . 23
1.13 La définition du Testing . . . 23
1.14 La définition de Emulation . . . 24
1.15 La définition de Simulation . . . 24
1.16 la hiérarchie de TemporalLogic . . . 25
1.17 la hiérarchie des PetriNets . . . 26
1.18 Un exemple de requête . . . 26
1.19 Les concepts de base pour l'ontologie des méthodes formelles . . . 28

2.1 Les notions de base de la technologie objet . . . 36
2.2 Les notions de base de l'IDM . . . 36
2.3 L'architecture à quatre niveaux du MDA . . . 38

3.1 Les définitions de Model et MetaModel par un diagramme de classe . . . 52
3.2 Un exemple de modèle en notation diagramme d'objets . . . 54

3.3 Le modèle M_1 60
3.4 Le modèle M_2 60
3.5 L'union des deux modèles M_1 et M_2 61
3.6 Le modèle résultat de la substitution 63
3.7 Les concepts de base de EMOF 65
3.8 La vérification du $conformsTo$ 66
3.9 Un extrait du métamodèle des ports 69

4.1 Une vue conceptuelle de la sémantique du Package Merge 76
4.2 Le paquetage `merged` 77
4.3 Le paquetage `receiving` 77
4.4 Le paquetage `resulting` 77
4.5 Une partie du métamétamodèle EMOF (notation COQ4MDE) 79
4.6 Un exemple d'une composition non valide 80
4.7 Le métamodèle `BasicEmployee` 81
4.8 Le métamodèle `EmployeeLocation` 81
4.9 Le paquetage `BasicEmployee` dans la notation `Coq4MDE` relativement à `EMOF` 82
4.10 Le paquetage `EmployeeLocation` dans la notation `Coq4MDE` 82
4.11 La substitution du paquetage `merged` 83
4.12 Le paquetage résultant de l'union dans la notation `Coq4MDE` 83
4.13 Le métamodèle résultat 84

5.1 L'extension du métamodèle 90
5.2 L'extraction et l'élimination de l'interface de composition 91
5.3 L'opérateur `bind` 93
5.4 L'opérateur `extend` 94
5.5 Les modèles $\mathtt{M}_1$ et $\mathtt{M}_2$ 95
5.6 Les points de variation et de référence pour les modèles $\mathtt{M}_1$ et $\mathtt{M}_2$ 96
5.7 Le modèle après l'application de la fonction *bind* 96
5.8 L'extraction des fragments 96
5.9 Le modèle après l'exécution des fonction $extend$ et $ElimInterface$ 97
5.10 Une partie de la syntaxe du langage Ltac 101

6.1 Deux modèles A et B et un morphisme f entre les deux 120
6.2 Le produit des deux modèles A et B 124
6.3 Le coproduit des deux modèles A et B 124

6.4 Un exemple d'égalisateur 125
6.5 Un exemple de coégalisateur 126
6.6 La vérification du conformsTo pour la composition de fonctions 132

D.1 Le métamodèle `receiving` 154
D.2 Le métamodèle `receiving` (notation COQ4MDE) 154
D.3 Le métamodèle `merged` 155
D.4 Le métamodèle `merged` (notation COQ4MDE) 155
D.5 Le metamodèle `resulting` 156
D.6 Le métamodèle `resulting` dans la notation COQ4MDE 156

Index

A
areComposite 65, 70, 71, 98, 110
assistant à la preuve 130
assistant à la preuve 35, 39, 46–50
assistant de preuve . . . 1–3, 63, 72, 85, 97, 98, 100, 112, 113

B
base de connaissance 8

C
catégorie 116–127
catégorie duale 123
certification 1
CESAR 1
CESAR 8, 12
classe abstraite 54
Classes 53
classification 8, 18, 20, 21, 28
coégalisateur 117, 125–127
concept 10–22, 25, 27, 30
conceptualisation 7–9, 12, 14, 30
conformité 55
conformsTo 55
coproduit 124–127

E
Eclipse 52
égalisateur 117, 125, 126
EMOF . . . 2, 64, 65, 71, 72, 79, 80, 85, 112, 130
explicite 1

F
formalisme 8, 15, 17, 18, 20, 22, 25, 26, 30

G
graphe 58, 59

H
héritage 53, 54

I
IDM 51, 52
implicite 1
instance 10, 11, 15, 16, 20, 21, 25, 28
InstanceOf 11, 63, 64
instanceOf 55
isAbstract 54, 65–67, 98, 99, 108
ISC 4, 86, 88, 92, 95, 97, 98, 111–113, 130

isOpposite . 65, 69, 70, 98–100, 104, 107, 109, 111

L
langage de modélisation 8, 17
logique constructive 52
lower . 64, 65, 67–69, 72, 98, 99, 103, 104, 106, 108, 109, 111

M
macroscopique 1
MDA 1
MDD 1
MDE 1
MetaModel 52, 59, 64, 65, 69
microscopique 1
Model 52
MOF 53, 60, 64, 72, 75, 112, 129–131
multigraphe 52

O
Objects 53
objet final 123
objet initial 123
ontologie 1, 7–20, 25, 27, 28, 30

P
Package Merge . . 4, 75–77, 79–81, 83–86, 151
polymorphisme d'héritage 53
population 8, 19
produit 123
Ptolemy 8, 12

Q
qualification 1

R
References 53
relation sémantique 8
réseaux de Petri 25, 26

S
sémantique 1
spécification formelle 8, 9
subClass 54, 65, 66, 98, 103, 105–108, 110
Substitution 63, 65–68, 75, 79, 83–85
syntaxique 1
système 8, 9, 16–18, 20–27, 30

U
Union 60, 61, 63–65, 67, 68, 70, 75, 78–80, 82, 83, 85, 92, 94, 95
Uniset 57, 60
upper 64, 65, 67–69, 72, 98, 99, 104, 106, 108, 109, 111

V
validation 7, 9, 22, 25
vérification 9, 22, 23, 25, 26
vue 8, 17, 18, 20, 21
V&V . 7–10, 12–14, 16, 17, 20–22, 25–27, 30, 52
V&V V&V sont 31
VVO 7–18, 20–23, 25–27, 30, 53

Glossaire

CESAR Cost-Efficient methods and processes for SAfety Relevant embedded systems. 1, 8, 12

CIC Calculus of Inductive Constructions. 40, 46

CMOF Complete Meta-Object Facility. 64

DAL Design Assurance Level. 29

DSML domain specific modeling languages. 87, 88, 91

EMOF Essential Meta-Object Facility. 2, 64, 65, 71, 130

EMOF Meta-Object Facility. 53, 60, 64, 72, 73, 129–132

IDM Ingénierie Dirigée par les Modèles. 2, 3, 31, 35, 36, 38, 47, 48, 50, 52, 129–131

ISC Invasive Software Composition. 86, 88, 91, 92, 95–97, 111, 112, 130

MDA Model Driven Architecture. 1, 16, 38

MDD Model Driven Development. 1

MDE Model Driven Engineering. 1

MMISS MultiMedia Instruction in Safe Systems. 27

OCL Object Constraint Language. 37, 49, 112, 131

OMG Object Management Group. 37, 38, 73, 131

OO Orienté Objet. 129

OWL Web Ontology Language. 14–16, 25, 27, 31, 129

RDF Resource Description Framework. 15, 16, 129

RDFS Resource Description Framework Schema. 15

SOA Service-oriented architectures. 130

TQL Tool Qualification Level. 29

UML Unified Modeling Language. 28, 37, 39, 49, 60, 64, 73, 88, 129

URI Universal Resource Identifier. 14

V&V Verification and Validation. 1–3, 7–10, 12–14, 16, 17, 20–22, 25–27, 30, 87, 131, 175

VVO Verification and Validation Ontology. 3, 7–18, 20–23, 25–27, 29, 30, 53, 129

W3C Web Ontology Working Group. 15, 16

XML Extensible Markup Language. 15

TITRE : ASSISTANCE À LA VALIDATION ET VÉRIFICATION DE SYSTÈMES CRITIQUES : ONTOLOGIES ET INTÉGRATION DE COMPOSANTS.
Encadrants : Marc Pantel & Xavier Thirioux

Résumé

Les activités de validation et vérification de modèles sont devenues essentielles dans le développement de systèmes complexes. Les efforts de formalisation de ces activités se sont multipliés récemment étant donné leur importance pour les systèmes embarqués critiques. Notre travail s'inscrit principalement dans cette voie. Nous abordons deux visions complémentaires pour traiter cette problématique. La première est une description syntaxique implicite macroscopique basée sur une ontologie pour aider les concepteurs dans le choix des outils selon leurs exigences. La seconde est une description sémantique explicite microscopique pour faciliter la construction de techniques de vérification compositionnelles.

Mots clés : formalisation, Vérification et Validation, ontologie, Ingénierie Dirigée par les Modèles, assemblage, assistant à la preuve

---

TITLE : SUPPORT FOR THE VALIDATION AND VERIFICATION OF CRITICAL SYSTEMS: ONTOLOGIES AND INTEGRATION OF COMPONENTS.
Supervisors : Marc Pantel & Xavier Thirioux

Abstract

The validation and verification of models have become essential in the development of complex systems. The efforts for the formalization of these activities have increased recently being given their importance for critical embedded systems. Our work lies mainly in this direction. We discuss two complementary visions for addressing this issue. The first is a syntactic implicit macroscopic description based on an ontology to help designers in the choice of tools depending on their requirements. The second is a microscopic explicit semantic description aiming to facilitate the construction of compositional verification techniques.

Keywords : formalisation, Verification and Validation, ontology, Model Driven Engineering, composition, proof assistant

---

Mounira KEZADRI
Institut de Recherche en Informatique de Toulouse - UMR 5505
INPT, Toulouse