HAL
open science

Acquisition et analyse des exigences pour le développement logiciel : une approche dirigée par les modèles.

Erwan Brottier

To cite this version:

Erwan Brottier. Acquisition et analyse des exigences pour le développement logiciel : une approche dirigée par les modèles.. Génie logiciel [cs.SE]. Université Rennes 1, 2009. Français. NNT : . tel-00512174

HAL Id: tel-00512174

https://theses.hal.science/tel-00512174v1

Submitted on 27 Aug 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N° d'ordre : 4078 ANNÉE 2009

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale Matisse

présentée par

Erwan Brottier

préparée à l'unité de recherche IRISA
IRISA/TRISKELL
IFSIC

Acquisition et analyse des exigences pour le développement logiciel : une approche dirigée par les modèles

Thèse soutenue à Rennes le 14 décembre

devant le jury composé de :

Françoise ANDRÉ
Professeur, IRISA / *président*

Daniel DEVEAUX
Maître de conférences, Valoria / *rapporteur*

Pierre KELSEN
Professeur, Université du Luxembourg / *rapporteur*

Yves LE TRAON
Professeur, Université du Luxembourg / *directeur de thèse*

Benoit BAUDRY
Chargé de recherche IRISA / *examinateur*

Bertrand NICOLAS
Architecte logiciel, France Télécom R&D Lannion / *examinateur*

Remerciements

Je tiens à remercier tout d'abord Yves Le Traon, mon directeur de Thèse. Yves m'a d'abord apporté humainement, ce qui est de loin le plus important. Il m'a aussi ouvert les portes du doctorat. Je me rappelle nos premières discussions sur les routes bretonnes durant mon stage à France Télécom. Quatre heures à tuer entre Rennes et Lannion chaque semaine, mais tellement de sujets abordés. Entre théologie, histoire et philosophie, le chemin parcouru fut des plus enrichissants. J'espère qu'au Luxembourg, tu pourras enfin souffler un peu et profiter de ta grande famille. Je remercie aussi Bertrand Nicolas, mon chef d'équipe à France Télécom. Bertrand m'a beaucoup appris sur le fonctionnement d'une multi-nationale et sur les institutions politiques plus généralement.

Je remercie les personnes que j'ai cotoyées à France Télécom et à l'Irisa pour leur aide précieuse et les bons moments passés. Il est difficile d'en faire une liste exhaustive. Je pense à Maryvonne, ma collègue de bureau à Lannion, Jacques et son humour détonnant, Marianno pour ses conseils, Grégoire et Sébastien pour leur aide. Je pense aussi à Sakku, Chi dung et Yves-Marie. Merci à toute l'équipe Triskell à Rennes et bien sûr à Jean-Marc pour m'avoir accueilli. Merci à Didier, François et Cyril pour leur aide ; Noël, Franck et Robert pour les bœufs jazz ; Reda, Julien, Jean-Marie et Gilles pour les fous rires (merci Gilles pour le soutien psychologique). Je pense aussi à Olivier, Freddy, Romain et Brice. Je remercie particulièrement Benoît, mon encadrant, qui m'a formé sur le long terme à la rédaction d'articles scientifiques. Merci aussi à Philippe de l'Ensieta pour ses explications concernant les méthodes formelles.

Merci à ma mère, Annie, mon père, Christian et mes beaux parents Gérard et Pascale. Sans leur éducation, il m'aurait été difficile d'écrire ce manuscrit. En qualité de professeur de français, Christian a entièrement relu et corrigé ce manuscrit ; s'il reste une faute d'orthographe, il va sans dire que j'en suis responsable (quoique ! ;-). Merci à toute ma famille pour leur compréhension lorsqu'il m'était impossible d'être des leurs. Une thèse est une activité chrono-phage, pavée d'imprévus, où l'organisation est d'autant plus importante qu'elle n'est pas ma première qualité (c'est peu dire …). Un grand tour de France s'impose donc, et ma valise est prête. Merci à mes ami(e)s de longue date pour leur soutien festif. Il n'est pas aisé de comprendre les angoisses d'un doctorant. Merci donc pour votre légèreté communicative. Je remercie particulièrement Marion pour sa patience. Désolé pour tout ce que nous n'avons pas pu partager par manque de temps.

Le domaine de l'ingénierie des exigences est en soit particulièrement formateur. Il m'a fallu du temps pour comprendre les fondements de cette partie des sciences de l'information. J'ai d'abord été surpris par les préoccupations du domaine (objectifs, stratégies, interactions sociales …), bien éloignées de celles strictement liées à l'implémentation d'un logiciel (conception, test …). J'ai compris par la suite qu'un logiciel n'est qu'une des nombreuses formes que peut prendre un système. C'est aussi de l'organisation d'un groupe de personnes dont il s'agit et plus généralement de l'ingénierie sociale. Cette connaissance m'est désormais utile dans bien des domaines.

Enfin, je suis conscient de la chance qui m'a été donné de faire ce travail. La production d'une thèse est une épreuve exigeante. Etalé sur quatre ans, ce travail de longue haleine forge la patience et la détermination. Il apprend à chercher, classer, et comparer l'information, ce qui de nos jours est un atout de taille. Il force à lire, écrire et penser simultanément dans deux langues, ce qui est une expérience étonnante. La découverte de ce monde m'a beaucoup appris sur moi-même et m'a permis d'affiner mes projets futurs.

Table des matières

INTRODUCTION **11**

PREMIERE PARTIE : ETAT DE L'ART ET MOTIVATION **16**

CHAPITRE I FIABILISATION DES EXIGENCES 18
1 Processus d'ingénierie des exigences *19*
2 Capture des exigences *20*
2.1 Décomposition et classement de l'information 21
2.2 Langages de modélisation des exigences 22
2.3 Approche de modélisation par patrons de spécifications et langage naturel contraint 24
3 Détection d'incohérences *28*
3.1 Notion d'incohérence et classifications 28
3.2 Principales approches de détection d'incohérences 29
4 Synthèse *32*
CHAPITRE II UN CADRE CONCEPTUEL POUR LA METAMODELISATION : L'INGENIERIE DIRIGEE PAR LES MODELES. 34
1 Théorie des langages *35*
2 Métamodélisation *36*
2.1 Modèles et métamodèles 37
2.2 Relations entre modèles, métamodèles et langages 39
2.3 Niveaux de modélisation : l'architecture MDA 40
3 Transformation de modèles *41*
3.1 Définition et langages actuels 41
3.2 Spécification par transformation de modèles de la sémantique d'un langage 42
4 Composition de modèles *45*
4.1 Principes généraux 45
4.2 Approches de composition de modèles 47
5 Synthèse *49*
CHAPITRE III PROBLEMATIQUES, PLATE-FORME DE RECHERCHES ET SPECIFICATIONS ETUDIEES 50
1 Problématique *51*
2 Approche proposée et réalisation concrète *53*
2.1 Description générale de l'approche 53
2.2 La plate-forme R2A : une plate-forme de recherche dirigée par les modèles 55
3 Spécifications d'exigences étudiées *57*
3.1 Etudes de cas réalisées 57
3.2 Exemple d'une spécification d'exigences : le système SRV 57

DEUXIEME PARTIE : CONTRIBUTIONS **61**

CHAPITRE IV LE FORMALISME RM ET LE LANGAGE RDL 63
1 Le formalisme RM *64*
1.1 Vues, niveaux de description et niveaux de modélisation 64
1.2 Sémantique statique 66
1.3 Sémantique dynamique 71
2 Le langage RDL : un langage naturel contraint pour exprimer le domaine d'un système *74*
2.1 Intérêt et Principes et du RDL 74
2.2 Patrons de spécification RDL et expressivité du langage 76
2.3 Description du langage RDL : le métamodèle RDM 80
CHAPITRE V CARACTERISATION D'UN PROCESSUS DE COMPOSITION DE SPECIFICATIONS PARTIELLES HETEROGENES 83
1 Exemple d'une spécification d'exigences opérationnelles *84*
2 Analyse de la problématique *87*
2.1 Description générale d'un processus de composition 87
2.2 Interprétation 91
2.3 Traitement des superpositions sémantiques 96
3 Résumé des caractéristiques d'un processus de composition d'exigences *99*
CHAPITRE VI OBTENTION D'UNE SPECIFICATION OPERATIONNELLE GLOBALE : PROCESSUS DE COMPOSITION SIMPLE 100
1 Un processus itératif en trois étapes *101*
1.1 Vue générale du processus 101
1.2 Description des trois étapes du processus 103
1.3 Itération et symétrie du processus 104

1.4 Relâchement du métamodèle cœur 105
2 Interprétation 106
2.1 Vue générale 106
2.2 Notion de règle d'interprétation 107
2.3 Nature des modèles intermédiaires produits 115
3 Fusion 117
3.1 Vue générale 117
3.2 Exemples de fragments à fusionner et caractérisation de l'étape de fusion 119
3.3 Approche adoptée et implémentation avec le langage FRL 120
3.4 Illustration 125
4 Analyse 129
4.1 Vue générale 130
4.2 Support de la traçabilité 131
4.3 Analyse statique du modèle global 134
4.4 Illustration 137
CHAPITRE VII CREATION DE PLATES-FORMES SPECIALISEES : PROCESSUS DE COMPOSITION DOUBLE 139
1 Motivation 140
2 Cadre conceptuel pour l'adaptabilité de la plate-forme R2A 141
3 Processus de composition double 145
3.1 Vue générale du processus 145
3.2 Exemple de sémantique de composition définie sur le MOF 146
3.3 Directives de composition 147
3.4 Cohérence du métamodèle cœur et de sa sémantique de fusion 149

CONCLUSION ET PERSPECTIVES 150

TROISIEME PARTIE : ANNEXES 154

ANNEXE A INDEX. 155
ANNEXE B PROCESSUS D'INGENIERIE DES EXIGENCES. 158
ANNEXE C CADRE CONCEPTUEL EN INGENIERIE DES EXIGENCES. 162
ANNEXE D ARCHITECTURE DE LA PLATE-FORME R2A. 166
ANNEXE E DESCRIPTION DES VUES DU FORMALISME RM. 168
ANNEXE F LANGAGE RDL. 171
ANNEXE G GRAMMAIRE EBNF DU LANGAGE IRL. 175
ANNEXE H GRAMMAIRE EBNF DU LANGAGE FRL. 179
ANNEXE I SPECIFICATION IRL DE LA FONCTION SEMANTIQUE DU LANGAGE RDL DEFINIE SUR RM (F : RDL → RM). 181
ANNEXE J SPECIFICATION IDL DE REGLES DE DETECTION D'INCOHERENCES POUR LE METAMODELE RM. 186
ANNEXE K SPECIFICATION FRL DE LA SEMANTIQUE COMPOSITIONNELLE DU METAMODELE RM (Φ_{RM})... 187
ANNEXE L MODELES INTERMEDIAIRES PRODUITS DURANT LA COMPOSITION DES MODELES DE LA FIGURE 43. 191
ANNEXE M SPECIFICATION FRL DE LA SEMANTIQUE COMPOSITIONNELLE DU META-METAMODELE MOF (Φ_{MOF}) 197

PUBLICATIONS 199

BIBLIOGRAPHIE 200

Introduction

L'informatisation croissante de notre société a profondément modifié la nature des logiciels et la manière de les construire. L'explosion de la puissance de calcul des ordinateurs a permis l'automatisation de tâches toujours plus complexes, augmentant fortement la taille et la complexité des spécifications logicielles produites. L'exacerbation de la concurrence a contraint l'industrie logicielle à mettre en place des méthodes de développement plus productives[1]. Enfin, le remplacement de l'humain par la machine pour des tâches mettant en jeu la vie humaine a forcé l'industrie à mieux contrôler (sinon prouver) la qualité des logiciels produits. Ces nombreux défis sont à l'origine des processus de développement appliqués par l'industrie logicielle pour maîtriser la complexité des problèmes adressés, améliorer la productivité et quantifier la qualité des logiciels livrés.

La privatisation progressive des grands groupes télécoms et leur mise en concurrence au niveau international a profondément bouleversé les conditions de développement logiciel au sein du groupe *France Télécom* (FT). Les activités d'implémentations sont externalisées chez des prestataires pour ne garder in situ que les étapes d'analyses amont et les étapes de validation. Cette situation vise à limiter les coûts de développement mais restreint la pleine maîtrise de la qualité des logiciels produits ; seules les *spécifications d'exigences* restent sous la responsabilité des équipes de FT. Pour pallier ce problème, le groupe s'intéresse à la définition d'une *plate-forme de fiabilisation des exigences*. Cette plate-forme vise à intégrer dans un même environnement de développement des techniques de *vérification et validation* (V&V) afin de mesurer la qualité des exigences produites puis de vérifier si ces dernières sont correctement satisfaites une fois le logiciel livré.

Le développement d'une telle plate-forme est une tâche techniquement ardue. Une spécification d'exigences est dans ce contexte un modèle formel, productif tout au long du processus de développement. C'est aussi une tâche coûteuse. Pour être rentable, cette plate-forme nécessite un environnement de développement favorisant sa réutilisation dans des contextes industriels variés. L'*ingénierie dirigée par les modèles* (IDM) [1-3] vise à systématiser le développement logiciel. C'est une approche prometteuse pour intégrer au sein d'un même environnement des techniques V&V modernes et une grande variété de langages de modélisation [2]. Elle offre au génie logiciel un *cadre conceptuel homogène* pour la formalisation des langages et de leurs relations, ainsi que des techniques de réutilisation pour la conception logicielle. Nous adoptons dans cette thèse le point de vue technologique de l'IDM pour penser les concepts et les préoccupations de l'ingénierie des exigences.

Problématiques abordées

L'*ingénierie des exigences* [4-6] (IE) regroupe les méthodes et techniques visant à produire une spécification d'exigences exhaustive, cohérente et reflétant effectivement les besoins à l'origine d'un projet logiciel. De nombreuses méthodes outillées ont été proposées dans ce domaine pour capturer et analyser formellement les exigences, mais ces dernières restent inexploitées à l'échelle industrielle [7-10]. La suite présente les problématiques liées à ce *manque de transfert industriel*, puis décrit les contributions. Ces problématiques sont résumées ci-dessous :

(i) La non-spécification des *liens sémantiques* entre artefacts de développement durant les premières phases du développement : entre exigences (*superpositions sémantiques*) et entre exigences et artefacts de développement aval (*liens de traçabilité*).

(ii) La difficulté pour les parties prenantes de produire une spécification formelle des exigences.

(iii) Le manque de flexibilité et d'interopérabilité des outils V&V actuels.

[1] A titre d'exemple, le temps de réalisation d'un service télécom varie entre trois et six mois, contre plusieurs années avant la privatisation progressives des groupes télécoms.

L'application de techniques V&V nécessite une spécification formelle des exigences [11], appelée *spécification opérationnelle*. La production d'une telle spécification est une activité particulièrement ardue, sinon la plus difficile d'un processus de développement logiciel [12]. Une spécification d'exigences est une collection de *spécifications partielles, hétérogènes et potentiellement incohérentes*. Ces spécifications sont partielles car résultant d'un travail d'équipe impliquant des *parties prenantes* hétéroclites de part leurs compétences, intérêts et rôles au sein du projet logiciel [4]. Elles sont hétérogènes car décrites à l'aide de langages adaptés à la description de préoccupations variées [1] (DSL [13-15] pour *Domain-Specific Language*). Elles sont potentiellement incohérentes (conflits quant aux intérêts défendus, divergences conceptuelles …) puisque décrites par des parties prenantes (i) ayant chacune un point de vue singulier sur ce que doit être le futur logiciel [16] et (ii) n'ayant généralement qu'une vague idée de leurs besoins [17].

La description des exigences à l'aide de spécifications partielles et hétérogènes contribue à la mise en oeuvre des principes de *séparation des préoccupations* [18] et de *décomposition des problèmes* [19], encouragés par les méthodes de développement modernes comme RUP d'IBM [11]. Ces principes favorisent la compréhension, la communication et l'identification de solutions réutilisables (*patrons de spécifications* [20], *aspects* [21], *composants* [22], *lignes de produits* [23]). L'application de ces principes à différents *niveaux d'abstraction* (exigence, architecture, conception, code) favorise les activités de vérification (conception par contrats [24], model-checking [25])[2]. Cette tendance est renforcée par les approches de développement génératives issues de l'IDM. Ces approches prônent une description des logiciels au niveau d'abstraction le plus élevé possible[3], les spécifications de plus bas niveau étant automatiquement générées. Là où les spécifications de haut niveau étaient utilisées comme des artefacts informels de documentation et supports de discussions (utilisation « contemplative »), l'idée est d'en faire des *modèles productifs*.

Une approche dirigée par les modèles améliore l'automatisation du développement logiciel mais nécessite de décrire précisemment les *liens sémantiques* entre spécifications partielles[4], ce qui est particulièrement complexe pour une spécification d'exigences vu sa nature (collection de spécifications partielles hétérogènes potentiellement incohérentes). Les *liens de traçabilité* [26-29] entre exigences et artefacts de développement aval sont rarement spécifiés car leurs descriptions et mises à jour sont manuelles et très coûteuses [28]. Cette situation provoque une désynchronisation entre deux niveaux d'abstraction [2], ce qui n'est plus tenable car les exigences évoluent désormais alors que les logiciels sont déjà en cours d'implémentation [28, 30]. Les liens de traçabilité sont aussi utiles pour les activités V&V. Sans eux, il est difficile pour les parties prenantes d'interpréter au niveau des exigences une erreur détectée à un niveau d'abstraction moindre [31] (là où les techniques de vérification opèrent).

Les *superpositions sémantiques* [32] sont aussi rarement spécifiées car trop complexes. Ces liens sémantiques identifient quels éléments des spécifications partielles représentent la même information. Ils doivent être identifiés pour mesurer la *cohérence d'une spécification d'exigences*. Les approches de *comparaison structurelle* [33-38] détectent par ce biais des *incohérences statiques*[5], mais ne sont pas adaptées à des spécifications hétérogènes (les règles de cohérence doivent être spécifiées pour chaque paire possible de langages de description des exigences). De plus, ces approches sont incapables de fournir une mesure globale de la cohérence statique d'une spécification d'exigences ; il est impossible de détecter un *manque d'information* car les spécifications partielles sont confrontées deux à deux [39]. Cette situation freine l'utilisation de techniques de vérification formelle afin de détecter des *incohérences dynamiques*[6] car ces techniques acceptent uniquement une spécification des exigences statiquement cohérente.

---

[2] Les techniques de vérification confrontent des spécifications à différents niveaux d'abstraction ; les techniques de validation consistent en la confrontation entre une spécification et des parties prenantes.

[3] L'abstraction diminue la distance conceptuelle entre le domaine du problème et le domaine de la solution [2]. Elle permet de s'affranchir dans une certaine mesure de ce que Brooks appelle la complexité accidentelle [12] (liée à la construction d'une solution et non au problème adressé).

[4] Au même titre que le passage aux approches de développement orientées objet a déplacé la complexité au niveau des relations entre objets.

[5] Relatives à la sémantique statique d'une information, à sa représentation par une structure normalisée.

[6] Relatives à la sémantique dynamique d'une information.

La *composition de modèles* apparaît comme une approche prometteuse pour dépasser ces limitations [40], mais les approches actuelles [40-46] ne sont pas adaptées à la production d'une spécification opérationnelle d'exigences. Certaines sont trop informelles [44-46]. D'autres acceptent des expressions d'un seul langage de description des exigences [40-43]. De plus, elles supposent que les spécifications partielles sont cohérentes avant composition [40]. La nature des langages supportés est aussi problématique. Ces langages sont généralement des *formalismes*, inadaptés à la majorité des parties prenantes [47], limitant ainsi l'implication des utilisateurs lors de la spécification des exigences[7]. Les langages semi-formels comme UML [50] sont plus accessibles pour les parties prenantes mais comportent des variations sémantiques sources d'*ambiguïtés*. De récents travaux [8, 51-53] proposent des langages à base de *patrons de spécifications*, munis d'une *syntaxe naturelle contrainte*. Ces langages sont plus accessibles mais se bornent à la description des *exigences non-fonctionnelles*.

Enfin, une plate-forme de fiabilisation doit être adaptable : les langages et outils V&V requis lors du déploiement de cette plate-forme dépendent de la nature du projet logiciel et plus généralement du contexte industriel et de la nature du projet logiciel [30]. Les langages supportés par les outils V&V actuels ne sont pas adaptables. Ces outils ne sont pas interopérables car implémentés de manière ad-hoc. Enfin, la production d'une plate-forme intégrant un grand nombre de langages et de fonctionnalités pose le problème de sa maintenance et de son évolutivité.

Contribution

Jusqu'à présent, la communauté de l'IDM s'est intéressée pour l'essentiel à la productivité des *spécifications de conception* (premier niveau d'abstraction du code). Nous proposons dans ce manuscrit une approche de développement dirigée par les modèles centrée sur les exigences, réalisée par la *plate-forme R2A* (pour *Requirements to Analysis*). La réalisation et l'évaluation de cette approche dans sa globalité dépassent le cadre d'une seule thèse. Nous focalisons dans cette thèse sur un élément essentiel de cette approche, à savoir un processus de composition de modèles. Ce processus de composition est implémenté au sein de la plate-forme R2A. La Figure 1 schématise le processus au sein de l'approche proposé. Le travail effectué ici s'articule autour de trois contributions :

(1) La première contribution de cette thèse est un processus générique de composition de modèles appelé *processus de composition simple*. Ce processus produit un modèle global des exigences (ou simplement *modèle global*) à partir de spécifications d'exigences partielles et hétérogènes. Ce modèle global représente une spécification opérationnelle des exigences, conforme à l'ensemble des spécifications partielles fournies en entrée. Le processus est paramétrable pour un contexte industriel donné : un ensemble de *langages d'entrée* utilisés pour exprimer les exigences, un *formalisme* (dont le modèle global est une instance) et un ensemble de règles spécifiant la sémantique de ces langages. Ce processus accepte des spécifications partielles potentiellement incohérentes. Il offre une capacité de détection d'incohérences statiques supérieure aux techniques de comparaison structurelle (détection de manque d'information). Il produit aussi un *modèle de traçabilité* entre spécifications partielles et modèle global. Enfin, ce processus accepte des spécifications partielles potentiellement ambiguës. La gestion des ambiguïtés est spécifiée au niveau des règles sémantiques. Ces caractéristiques favorisent l'implication des parties prenantes et l'application des techniques de vérification formelle pour la détection d'incohérences dynamiques.

(2) La deuxième contribution de cette thèse porte sur un processus de composition à deux niveaux d'abstraction, appelé *processus de composition double*. Ce processus est une extension du processus de composition simple. Le processus de composition double consiste en l'application du processus de composition simple à deux niveaux d'abstraction. Au *niveau conception* (niveau méta), le formalisme du processus de composition simple[8] est produit par composition de *composants conception*. Au *niveau instance*, ce formalisme est instancié par composition des spécifications partielles (production du modèle global). Un composant de conception représente un *type d'informations* susceptibles d'être capturée par le processus de composition

[7] Cause importante d'échecs de projets logiciels, voir les études statistiques de référence [48-49].
[8] C'est aussi le formalisme de la plate-forme R2A.

simple (*composant ontologique*), ou une *fonctionnalité* nécessaire à l'application d'une ou plusieurs techniques V&V comme la simulation des exigences par exemple (*composant fonctionnel*). Les composants ontologiques embarquent une *sémantique compositionnelle* décrivant la manière dont leurs instances sont composées une fois instanciées. Les composants fonctionnels embarquent une *sémantique opérationnelle* (sémantique dynamique de la fonctionnalité représentée) et une *sémantique de déploiement* (contraintes structurelles devant être satisfaites avant l'exécution de la fonctionnalité). Le processus de composition double favorise la flexibilité de la plate-forme en autorisant une décomposition de sa conception suivant les préoccupations considérées par les ingénieurs.

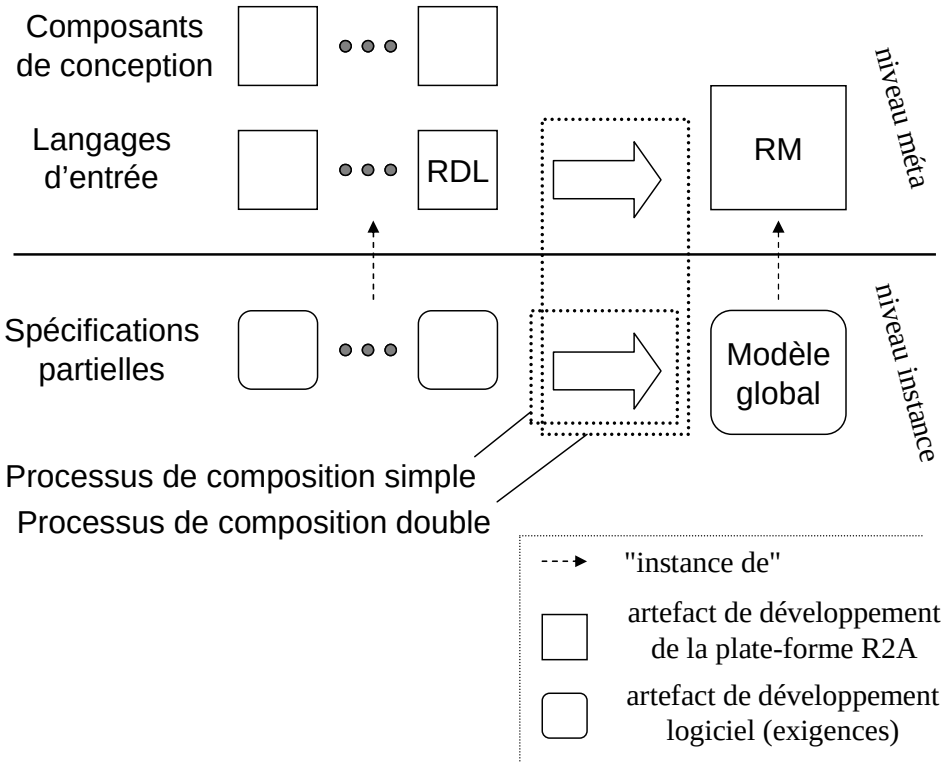


Figure 1 – Vue générale de l'approche proposée.

(3) La troisième contribution de cette thèse est le développement de la plate-forme R2A, un prototype de recherche mettant en œuvre les deux processus de composition exposés. Cette plate-forme est actuellement spécialisée pour les besoins industriels de FT. Son formalisme, appelé *RM* (pour *Requirements Metamodel*) est centré sur les actions des acteurs du système. Il symbolise les comportements possibles d'un *domaine* (exigences et environnement) à l'aide de la logique du premier ordre étendue d'une logique modale pour exprimer les exigences non-fonctionnelles *temps réel*. Les langages acceptés par la plate-forme sont les diagrammes d'activité UML, les diagrammes de classes, un langage naturel contraint, le *RDL* (pour *Requirements Description Language*) et le formalisme lui-même. L'obtention d'une vue unifiée des exigences (le modèle global) autorise l'intégration au sein de la plate-forme d'une batterie d'outils V&V et d'algorithmes de génération d'artefacts de développement aval. Les fonctionnalités principales supportées par la plate-forme R2A sont l'extraction d'une première esquisse de *spécification d'analyse*, la génération d'objectifs de test système, la simulation des exigences fonctionnelles (voir les travaux antérieurs de Clémentine Nebut [54-58]) et la vérification d'exigences non-fonctionnelles par intégration d'un outil de vérification formelle. La description de ces fonctionnalités dépasse le cadre de cette thèse. Nous présentons uniquement les langages RM et RDL.

Les processus de composition simple et double, et plus généralement la plate-forme R2A dans son ensemble ont été entièrement développés à l'aide de technologies issues de l'IDM. Ce choix technologique favorise la maîtrise de l'évolutivité de la plate-forme. La plate-forme est spécifiée à l'aide de *métamodèles* et *transformations de modèles*. Les premiers décrivent l'aspect statique des

langages et fonctionnalités supportées, les deuxièmes l'aspect dynamique. Nous adoptons une terminologie issue de l'IDM dans l'ensemble de ce manuscrit.

Organisation du manuscrit

Ce manuscrit est structuré en trois parties.

La première partie fournit un état de l'art des travaux en relation avec cette thèse et se termine par une description de l'approche proposée et de sa réalisation concrète (la plate-forme R2A). Le Chapitre I propose une vue générale de l'ingénierie des exigences, de ses problématiques et insiste sur les formalismes utilisés et les techniques de vérification de la cohérence des exigences. Le Chapitre II présente le domaine de l'ingénierie dirigée par les modèles ainsi que les travaux portant sur la composition de modèles. Le Chapitre III présente les problématiques abordées, la plate-forme R2A et introduit les contributions. Il fournit un exemple de spécification d'exigences non formalisée d'un système de réunion virtuelle et montre, exemples à l'appui, la difficulté de produire et mesurer la cohérence d'une spécification opérationnelle d'exigences.

La deuxième partie présente les contributions. Le Chapitre IV détaille le langage d'entrée RDL et le formalisme actuel de la plate-forme, le RM. Ces langages sont utilisés dans la suite pour illustrer les processus de composition. Le Chapitre V caractérise un processus de composition pour les exigences. Le Chapitre VI détaille le processus de composition simple et l'illustre à l'aide d'un ensemble de spécifications hétérogènes raffinant la spécification d'exigences du système de réunion virtuelle introduite dans le Chapitre III. Il illustre l'utilisation de la plate-forme pour la détection d'incohérences. Enfin, le Chapitre VII présente le processus de composition double et l'illustre par la production de la plate-forme R2A actuelle. Nous présentons les perspectives de recherche envisagées en conclusion.

La troisième partie regroupe les annexes.

Première partie : Etat de l'art et motivation

Préambule

L'objectif de cette thèse est la mise en place d'une plate-forme d'ingénierie des exigences à l'aide d'une approche dirigée par les modèles afin d'améliorer la qualité des logiciels produits. Nous avons identifié deux axes importants pour ce faire : (i) la systématisation de la capture et de l'analyse des exigences d'un projet logiciel et (ii) l'amélioration de l'adaptabilité de cette plate-forme. La majeure partie de cette thèse porte sur la problématique de composition de modèles : (i) composition de spécifications d'exigences partielles et hétérogènes pour l'obtention d'une spécification globale et analysable des exigences dans un premier temps, (ii) composition de modèles de conceptions pour la production de plates-formes d'ingénierie des exigences adaptées à un contexte industriel donné. Cette première partie dresse un état de l'art multidisciplinaire des méthodes et techniques modernes d'ingénierie logicielle relatives à ces problématiques. Les problématiques abordées sont les suivantes :

- Comment modéliser informations et langages ? Comment traiter l'information dans un contexte multi-langage ? En particulier, comment décrire les relations sémantiques entre différents langages ?
- Qu'est-ce qu'une exigence ? Comment modéliser les exigences d'un système ? Quelles informations doivent être capturées lors d'un processus d'ingénierie des exigences ?
- Comment mesurer la qualité d'une spécification d'exigences ? Quelles caractéristiques d'une spécification d'exigences peuvent être évaluées par les techniques de vérification et de validation modernes ? Quelles sont les difficultés induites par l'application de ces techniques ?
- Comment séparer les préoccupations au niveau des exigences et au niveau de la conception d'une plate-forme d'ingénierie des exigences ? Comment composer des modèles capturant des préoccupations diverses ? Quelles sont les limitations des approches de composition contemporaines ?

Cette thèse emprunte des notions provenant de domaines de recherches variés, partageant les problématiques de modélisation et de traitement de l'information. L'ingénierie dirigée par les modèles et l'ingénierie des exigences partagent un grand nombre de notions communes relatives à la métamodélisation. Cependant, ces notions ne sont pas toujours nommées de la même manière. Nous portons donc une attention particulière à la définition d'une terminologie uniforme.

Organisation

Le Chapitre I présente le domaine de l'ingénierie des exigences. Il se focalise sur les problématiques liées à la fiabilisation des exigences : modélisation précise des exigences et analyse des exigences dans le but de détecter des incohérences). Ce chapitre discute dans un premier temps des langages de modélisation et des formalismes utilisés en ingénierie des exigences. Les principes généraux de décomposition des problèmes et de séparation des préoccupations sont discutés. Les techniques de vérifications formelles permettant leurs analyses sont finalement étudiées. L'objectif ici n'est pas de présenter une liste exhaustive de ces techniques, mais plutôt de lister les types d'incohérences pouvant être identifiées, analyser les limites de ces techniques et souligner la grande variété des outils existants (et par conséquent le défi industriel résultant de l'intégration de ces techniques au sein d'une plate-forme industrielle d'analyse des exigences).

Le Chapitre II présente dans un premier temps les notions clés de la théorie des langages. Il détaille ensuite le cadre conceptuel de métamodélisation proposé par l'ingénierie dirigée par les modèles. Ce cadre conceptuel unifie les perspectives et les notions propres à la modélisation et au traitement de l'information en génie logiciel. Nous adoptons en priorité le vocabulaire et la perspective

propres à l'ingénierie dirigée par les modèles. Dans un deuxième temps, ce chapitre fait l'inventaire des travaux concernant la composition de modèles. Nous restreignons notre étude aux approches existant dans les domaines de l'ingénierie des exigences, de l'ingénierie dirigée par les modèles et du développement orienté aspects (AOSD). Nous nous focalisons en particulier sur les travaux produisant des modèles formels.

Le Chapitre III décrit les les limitations des recherches actuelles quiresponsables du manque de transfert industriel des travaux en ingénierie des exigences favorisant la fiabilisation des exigences. Ce chapitre présente ensuite une vue générale de l'approche dans laquelle s'inscrit les contributions de cette thèse. Nous présentons la plate-forme R2A, développée pour France Télécom et implémentant cette approche. Nous donnons enfin un exemple d'une première spécification d'exigences informelles et illustrons les difficultés rencontrées durant un processus d'ingénierie des exigences. Cette spécification est utilisée tout au long de la deuxième partie de cette thèse pour illustrer les contributions.

Chapitre I Fiabilisation des exigences

Résumé

L'ingénierie des exigences [4] (IE) considère comme principal indicateur du succès d'un projet logiciel le niveau d'adéquation entre les exigences identifiées et les besoins réels ayant motivé le projet [4]. Elle vise à l'amélioration de la qualité du logiciel produit par la définition d'un processus rigoureux d'évaluation des besoins et des exigences [59]. Comme toute ingénierie, l'IE s'appuie sur la définition d'un processus (*processus d'ingénierie des exigences*) systématisant la production d'un produit réalisé (*spécification des exigences*[9]) à partir d'un produit initial (*spécification des besoins*[10]) par modélisations et raffinements successifs. Un processus IE implique un grand nombre d'individus, *parties prenantes* du projet logiciel (clients, futurs utilisateurs, prestataires, développeurs et testeurs, analystes, commerciaux, décideurs …). Il supporte une démarche de qualité : (i) il regroupe un ensemble de méthodes, techniques et outils favorisant la maîtrise de la qualité des spécifications amont, et dans la mesure du possible, (ii) il fournit des moyens de mesure de cette qualité (indicateurs précieux pour les chefs de projets et les qualiticiens).

Ce chapitre n'a pas pour ambition de présenter une liste exhaustive des problématiques et travaux proposés dans la littérature. L'IE est une discipline foncièrement pluridisciplinaire [4], empruntant à de nombreux autres domaines de recherches des réflexions et techniques facilitant la collecte, la rédaction, la compréhension et l'évaluation de *spécifications partielles* décrites dans des langages variés [16]. Ce chapitre structure plutôt une partie des travaux portant sur la modélisation et la vérification des exigences dans un but de détection d'incohérences. L'Annexe B fournit plus de détails sur l'ingénierie des exigences, ses principales activités et son cadre conceptuel. Enfin, notons que ce chapitre fait référence à des notions propres à la théorie des langages (syntaxe, sémantique, ambigüité …) définies en Chapitre II.

La section 1 donne une vue générale d'un processus d'ingénierie et souligne les principales difficultés liées à la production d'une spécification opérationnelle des exigences (spécification suffisement précise et formelle pour être automatiquement analysée). Les deux sections suivantes se focalisent sur les travaux directement liés au sujet de cette thèse. La section 2 présente les langages de modélisation utilisés en ingénierie des exigences et discute des techniques favorsiant la séparation des préoccupations et la décomposition des problèmes. La section 3 fournit une classification des incohérences susceptibles d'être identifiées au sein d'une spécification d'exigences et présente les principales techniques d'analyse proposées pour les détecter. Cette section souligne les limitations des travaux existants. La section 4 conclut.

[9] SRS dans la littérature anglophone (*Software Requirements Specification*).

[10] Une spécification des besoins définit les objectifs du système (*goals*). Une part importante des travaux en IE porte sur la description des besoins d'un système (cf. Annexe B). Ces besoins sont informels ; ils sont raffinés en exigences formelles. La description des besoins et leur raffinement ne sont pas abordés dans cette thèse.

1 Processus d'ingénierie des exigences

Un processus d'ingénierie des exigences (processus IE) constitue la partie amont du processus de développement logiciel[11] (voir la Figure 2). Il supporte les premiers raffinements du processus de développement ; il guide et accompagne le passage de besoins informels formulés par les parties prenantes en une première spécification d'exigences très abstraites (*spécification des besoins*). Le reste du processus consiste en raffinements progressifs de cette spécification jusqu'à l'obtention d'une *spécification d'exigences*[12]. Une fois les *spécifications amonts* produites, elles sont regroupées au sein du *cahier des charges*. Ce dernier sert de point d'entrée pour les phases de développements aval (durant lesquelles les *spécifications avals* sont produites). Il reflète les consensus et décisions prises par les parties prenantes ; pour l'industriel, c'est en outre un document contractuel définissant les rôles et responsabilités des clients et prestataires.

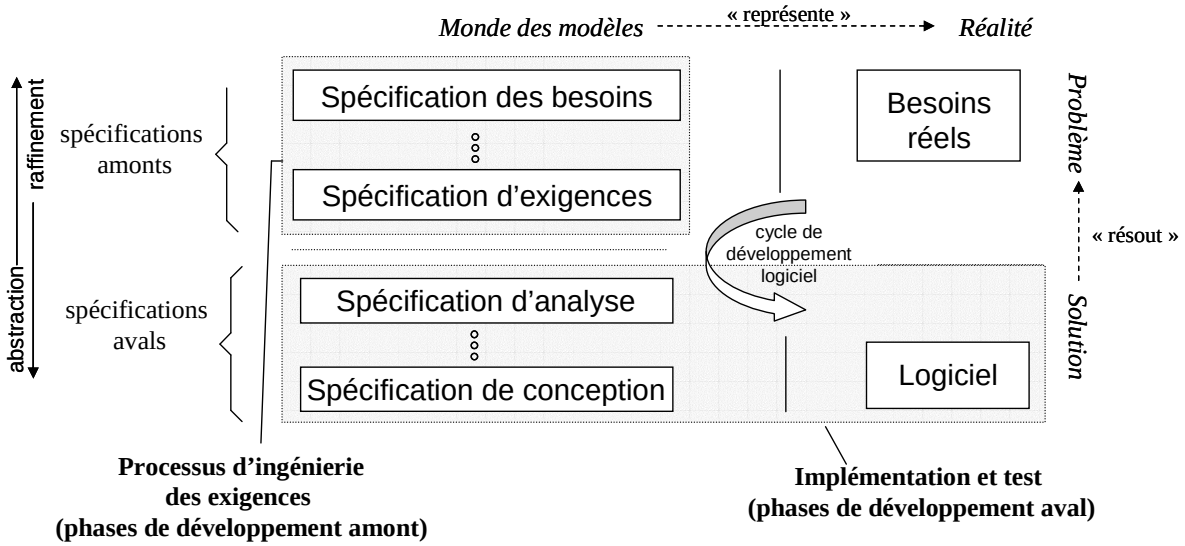


Figure 2 – Le processus d'ingénierie des exigences au sein du processus de développement logiciel.

La spécification d'exigences résultant d'un processus IE décrit un *domaine*, c'est-à-dire les exigences d'un système et l'environnement dans lequel ce dernier est déployé (voir la terminologie IE en Annexe B). Cette spécification est plus ou moins précise, suivant les besoins de l'industriel et le rôle joué par les exigences au sein du processus de développement. Dans cette thèse, nous nous intéressons à la production d'une spécification formelle des exigences, appelée *spécification opérationnelle des exigences*. Une telle spécification autorise l'application d'outils de vérification formelle (voir section 3). Elle formalise uniquement[13] les informations nécessaires à l'application des techniques d'analyse formelle désirée, ce qui implique de bien choisir le ou les langage(s) de modélisation proposé(s) aux parties prenantes[14] (voir section 2).

Les parties prenantes regroupent un ensemble hétéroclite de personnes[15] : elles n'ont pas les mêmes compétences, connaissances et expertises ; elles ne décrivent pas les mêmes préoccupations et ne défendent pas les mêmes intérêts au sein du projet. En conséquence, une spécification d'exigences n'est pas un monolithique ; c'est une collection de *spécifications partielles*. Ces spécifications partielles sont *hétérogènes* car (i) exprimées dans divers langages de modélisation adaptés aux

[11] Partie centrale du processus de développement en spirale de Boehm [60].

[12] La spécification d'exigences constitue une abstraction des spécifications avals, tout comme la spécification des besoins est une abstraction de la spécification d'exigences.

[13] La formalisation complète des exigences d'un système est justifiée pour des systèmes critiques dont les effets sur l'environnement sont potentiellement dangereux et susceptibles de mettre en péril des vies humaines.

[14] Ce choix dépend du niveau d'automatisation désiré (techniques de vérification et de validation visées), de la nature du projet, de la culture de l'entreprise et des habitudes des parties prenantes.

[15] Futurs utilisateurs du système, experts de domaines métiers, chefs de projets, analystes, architectes, ingénieurs, commerciaux …

rédacteurs [61], (ii) portant sur des aspects divers du domaine d'un système [62] et (iii) capturant des préoccupations [45] variées.

> **Définition – *spécification opérationnelle des exigences*** : Une spécification d'exigences est dite opérationnelle lorsqu'elle est formelle et simulable. L'obtention d'une spécification opérationnelle est un pré-requis pour l'application de techniques d'analyse formelle.

> **Définition – spécification partielle** : Une spécification partielle est un élément d'une spécification d'exigences. Elle capture un ensemble d'informations sur le domaine du système en construction. Une spécification partielle regroupe un ou plusieurs modèles expressions d'un ou plusieurs langages de modélisation. Les informations capturées par une spécification partielle sont incomplètes par définition.

La maîtrise d'une spécification d'exigences est une problématique importante en ingénierie des exigences. La bonne gestion d'un grand nombre de spécifications partielles hétérogènes tout au long du processus IE impacte fortement la qualité de la spécification des exigences. Cette gestion est difficile pour les raisons suivantes :

Communication de l'information. Un processus IE est par nature un travail de groupes, ce qui implique des problèmes de communication (compréhension, modération …). Deux approches complémentaires peuvent être appliquées pour faciliter la communication entre parties prenantes : (i) le classement de l'information ([19, 61, 63-65] par exemple) et (ii) l'utilisation de langages adaptés aux parties prenantes comme les langages naturels contraints [8, 51-53]. Les travaux portant sur ces deux approches sont discutés en section 2.1 et section 2.3.

Cohérence de l'information. Les exigences sont rarement cohérentes entre elles. Les incohérences reflètent des désaccords, des différences d'opinions entre parties prenantes, des incompréhensions concernant un domaine métier, des divergences d'objectifs [66]. La gestion de spécifications potentiellement incohérentes est de ce fait une problématique importante [33, 38-39, 66-67]. La problématique de détection d'incohérences est étudiée en section 3.

Evolution de l'information. Les logiciels et leur environnement évoluent et font l'objet de nouvelles versions. Les phases de développement avals et amonts sont de plus en plus réalisées en parallèle pour faire face aux contraintes de mise sur le marché dans un contexte industriel fortement concurrentiel ; les exigences évoluent alors que le logiciel est en cours d'implémentation [30]. La maîtrise de l'évolution des exigences et la mesure de l'impact sur le logiciel en cours de développement deviennent dès lors critiques [68].

Traçabilité de l'information. La description des liens de traçabilité [26-29] entre exigences et artefacts de développement logiciel est une nécessité pour l'industrie [29, 69]. On appelle *modèle de traçabilité* [28] un modèle capturant ce type d'informations. La traçabilité est utile pour identifier les artefacts avals impactés par l'évolution des exigences (suppression ou la modification d'une exigence). C'est aussi un moyen utile pour déterminer l'origine au niveau besoins des incohérences détectées au niveau exigences[16]. Un certain nombre de logiciels commerciaux offrent des solutions pour la gestion de liens de traçabilité mais ces derniers n'offrent pas plus de fonctionnalités qu'un tableur perfectionné (voir [70] pour plus de détails).

2 Capture des exigences

Cette section discute des langages et méthodologies utilisés en IE pour produire une spécification opérationnelle des exigences. La section 2.1 discute des approches existantes de *classement de l'information*. Ces dernières favorisent la *séparation des préoccupations* et la *décomposition des problèmes*, ce qui limite les problèmes de communications de l'information. La section 2.2 énumère les principaux langages de modélisation proposés en IE. Ces langages sont comparés afin de déterminer ce qu'un langage de modélisation des exigences doit capturer au minimum pour construire une spécification opérationnelle. Enfin, la section 2.3 présente les approches de modélisation par

[16] En ce sens, la détection d'incohérences participe à la validation.

patron de spécifications et *langages naturels contraints*. Ces approches offrent des langages formels offrant un bon compromis entre expressivité et accessibilité, ce qui favorise l'implication des parties prenantes dans la modélisation de spécifications opérationnelles.

2.1 Décomposition et classement de l'information

La manipulation, l'évolution et la compréhension d'une spécification d'exigences sont d'autant plus difficiles que le système étudié est complexe et sa spécification précise (ce qui est nécessaire pour automatiser son analyse). Cette difficulté a pour origine la limitation de la mémoire immédiate humaine, la lenteur des mécanismes de la mémoire à long terme et plus généralement l'incapacité de l'être humain à percevoir un problème dans sa globalité [71]. Pour atténuer cette difficulté, les ingénieurs IE utilisent des méthodes de décomposition et de classement de l'information favorisant la décomposition des problèmes [19] et la séparation des préoccupations [18, 72]. La décomposition d'un problème tend à fragmenter un problème initial en un ensemble de problèmes plus simples à résoudre car isolés [19][17]. La séparation des préoccupations facilite la compréhension d'une spécification, la factorisation éventuelle d'éléments réutilisables et la maîtrise de son évolution [18].

Les méthodes de décomposition de l'information proposées dans la littérature diffèrent suivant la nature de l'information considérée : les exigences sont fragmentées suivant les préoccupations quelles décrivent (points de vue), la conception d'un logiciel est fragmentée en composant fonctionnels, le code relatif à une notion est encapsulé dans une classe elle-même dans un paquetage. Un des points communs à la majorité de ces méthodes est l'utilisation de mécanismes d'abstraction [62] tels que la classification, l'encapsulation ou les lois de composition (arbres et-ou, états composite, composition de fonctions …). L'utilisation de ces méthodes aboutit à la description d'une spécification à l'aide d'une collection de spécifications partielles. Nous avons identifié quatre types de méthodes de classification dans la littérature. Nous les présentons succinctement ci-dessous :

Mécanismes d'abstraction. Une technique pour morceler une information consiste à tirer partie des mécanismes d'abstraction supportés par les langages de modélisation (voir [62] pour plus de détails sur les mécanismes d'abstraction) : mécanismes de classification, de composition ou encore de contextualisation. L'adoption des langages orientés objet a démocratisé ces mécanismes. Les diagrammes de classes UML offrent par exemple la classification via l'héritage et la contextualisation via les paquetages. Les machines à états offrent la composition grâce à la notion d'états composites. La décomposition logique (et-ou) des machines à états proposée par STATEMATE [73] est un autre exemple de mécanisme de composition. Les mécanismes d'abstraction offre un moyen de morceler une spécification décrite dans un langage de modélisation donné.

Langages à perspectives multiples. Ces langages permettent de décrire un système à l'aide de plusieurs perspectives (ou vues). Chaque perspective est spécialisée pour la description d'une ou plusieurs préoccupations particulières et la sémantique du langage définit les liens sémantiques entre ces perspectives[18]. Une spécification décrite avec un langage de ce type est une collection de modèles instances de différentes perspectives. UML [50] est un langage à perspectives multiples par exemple. Le diagramme de cas d'utilisation est une perspective UML utile pour représenter les principales fonctionnalités d'un système. Les diagrammes d'états, les diagrammes de séquences et les diagrammes d'activités sont utiles pour exprimer la dynamique d'un système. Le langage RM-ODP [74] est un autre exemple de langage offrant plusieurs vues[19]. Notons que cette méthode de classification est complémentaire à celle des mécanismes d'abstraction : chaque perspective peut être considéré comme un langage pouvant supporter des mécanismes d'abstraction.

[17] L'unidimensionnalisation systématique des problèmes en sciences a cependant des limites (cette problèmatique relevant de la méthodologie scientifique est cher à Edgar Morrin ; voir ses travaux sur la complexité).

[18] C'est rarement le cas dans les faits. Les correspondances entre les perspectives d'UML ou de RM-ODP par exemple sont sujettes à débats.

[19] Ces vues sont appelées « points de vue » dans ce langage (à ne pas confondre avec la notion de point de vue discutée plus bas).

Points de vue. Le classement par *points de vue* [16, 33-34, 37, 61, 65-66] est une méthode générale de gestion et de classement de spécifications partielles d'exigences décrites par un grand nombre de parties prenantes. Un point de vue qualifie l'information capturée par les spécifications partielles ; il associe aux spécifications partielles qu'il regroupe un ensemble de critères de classification (langages de modélisation utilisés, préoccupations décrites, noms des rédacteurs)[20]. Un point de vue attache aussi aux spécifications des informations utiles pour le processus IE (historique des évolutions des informations capturées par le point de vue, règles de cohérence vérifiées ou à vérifier) [37]. Les points de vue favorise le maintien des liens de traçabilité, la cohérence entre spécifications partielles [33-34] et la découverte de nouvelles exigences [39, 66]. Cette méthode de classement est complémentaire à celles précédemment décrites car elle est indépendante des langages choisis pour exprimer les exigences. Par exemple, les informations sur la fonctionnalité d'un système peuvent être classées au sein d'un cas d'utilisation, lui-même classé dans un point de vue.

Aspects. Les langages à perspectives multiples et/ou les points de vue ne supportent qu'un unique partitionnement de l'information. De ce fait, l'information relative à une préoccupation a des chances d'être dispersée dans les spécifications partielles formant la spécification d'exigences (Sommerville parle d'orthogonalité entre points de vue et préoccupations [65]). Cette problématique propre à la modélisation est appelée tyrannie de la décomposition dominante [75]. Initialement appliquées aux artefacts de développement aval[21], les approches de développement par aspect [44-46, 76-82] (AOSD pour *aspect-oriented software development*) proposent des méthodes pour séparer plus finement les préoccupations au niveau des exigences [44-46, 79-82]. Dans ce contexte, un aspect revêt de multiples définitions :

- un aspect désigne une fonctionnalité du système dont la description est fortement liée aux autres fonctionnalités (définition adoptée par Theme [44] et Early-AIM [80]). Pour un système de gestion de librairie, les actions "emprunter un livre" ou "s'abonner" sont des aspects dans ce sens.
- un aspect est un type d'informations au sens de Mylopoulos [62] (définition adoptée par [83]). Par exemple : aspect statique, aspect comportemental et aspect temps réel.
- un aspect est une préoccupation dont la description est dispersée dans plusieurs exigences (définition adoptée par [45-46, 79, 81, 84]). La sécurité, la compatibilité, le temps de réponse, la confidentialité sont des exemples d'aspects dans ce sens.

2.2 Langages de modélisation des exigences

Le langage naturel est le langage le plus utilisé pour décrire les premières spécifications d'exigences. C'est un langage utilisable et compréhensible par l'ensemble des parties prenantes, ce qui facilite la communication et la validation des informations produites. C'est en outre le langage le plus généraliste qui soit car il n'est pas spécialisé pour la description d'un type d'informations particulier (ce qui permet aux parties prenantes de décrire également tous types d'informations). Le langage naturel reste néanmoins un langage informel, ce qui rend impossible une description formelle de sa sémantique. Ce défaut est handicapant dans le cadre d'un processus IE pour deux raisons :

- ***l'ambiguïté de ses expressions***. Plusieurs sens peuvent être raisonnablement attachés à une même expression (voir les exemples d'ambiguïté relevés dans la spécification SRV). Il peut donc exister autant d'interprétations d'une spécification en langage naturel qu'il existe de lecteurs et rédacteurs, ce qui est source d'incompréhensions et d'incohérences difficilement identifiables.

[20] Ces informations sont appelées critères de classification dans la suite (La notion de point de vue est modélisée dans le RM, voir Annexe E).

[21] Au niveau implémentation (e.g. [76-77]), un aspect est une portion de code que l'on retrouve répétée et entrelacée dans le code d'un logiciel (par exemple un code de génération de traces pour la recherche de bogs). Au niveau conception (e.g. [78]), un aspect est un fragment de modèle de conception (ensemble de classes et d'associations) pouvant être réutilisée dans différents projets (par exemple un mécanisme de contrôle d'accès). Les composants de conception du Chapitre VII sont des aspects en ce sens.

- ***l'impossibilité d'analyser formellement ses expressions***. Une spécification en langage naturel doit être raffinée avant de pouvoir être analysée formellement. C'est une activité cruciale d'un processus IE visant une mesure outillée de la qualité des exigences puisque seules des spécifications opérationnelles peuvent être analysées. Or l'utilisation traditionnelle de langages de modélisation munis d'une syntaxe hermétique limite l'implication des parties prenantes. Cette situation diminue la confiance accordée aux diagnostics résultats de l'utilisation d'outils de vérification formelle (en particulier, on notera qu'un raffinement est d'autant plus source d'erreurs qu'il est effectué par un petit nombre de parties prenantes).

Cette section traite uniquement des langages dédiés à la spécification d'exigences opérationnelles. Jackson considère dans [85] que la logique des prédicats du premier ordre augmentée d'opérateurs de logiques temporelles est suffisante pour capturer une spécification opérationnelle, c'est-à-dire un domaine (ontologie statique et dynamique au sens de Mylopoulos [62]). Cependant, les langages proposés dans la littérature ont chacun leur particularité. Nous ne cherchons pas à faire un inventaire des langages proposés dans la littérature, mais plutôt de déterminer leurs points communs et les informations devant être modélisées pour pouvoir appliquer les techniques de vérification présentées en section 3. Les langages de modélisation présentés ici sont pour la plupart des formalismes (cf. Chapitre II). Les langages de modélisation munis d'une syntaxe plus adaptés à l'implication des parties prenantes font l'objet de la section suivante.

Un grand nombre de langages ont été proposés pour exprimer les exigences opérationnelles. La Figure 3 donne quelques exemples d'expressions de ces langages. Nous les classons en trois grandes catégories (il existe bien d'autres classifications possibles, voir par exemple celle de Lamsweerde [11]) :

- ***Machines à états (FSA)***. On trouve dans cette catégorie les langages représentant les propriétés fonctionnelles d'un système à l'aide d'état et de transition (*FSA*, *Finite State Automata*) dont les gardes sont des conditions sur les phénomènes[22] de l'environnement. Ces langages permettent donc de décrire qu'une partie des exigences des assertions environnementales (ces deux types de propriétés sont d'ailleurs rarement distingués). Le langage STATEMATE [73] d'Harel (voir Figure 3), le diagramme de machine à états UML [50] (une adaptation des travaux de Harel) ou encore SDL [86] en sont de bons exemples.
- ***Logiques modales***. Ces langages permettent la description des propriétés non-fonctionnelles d'un système. Leurs expressions sont donc des contraintes devant être vérifiées par les propriétés fonctionnelles (voir définition en Annexe B). Ces langages sont particulièrement utilisés pour spécifier les propriétés devant être évaluées dans une perspective de vérification à base de model-checking[23] (voir section 3). La diversité des langages de logiques modales reflète la diversité des outils de vérification formelle. Ces langages peuvent être classés dans différentes catégories : (i) langage temporel (relation de causalité) comme CTL [87], LTL [88] et QRE [89], (ii) temporel temporisé (quantification du temps en plus des relations de causalité) comme IF[24] [90] et TCTL [91], (iii) symbolique (calcul sur des intervalles de temps) comme PRTCTL [92]. Ils peuvent aussi être textuels ou graphiques. La Figure 3 propose quatre exemples d'expressions de logique modale, sémantiquement équivalentes : elles spécifient que le phénomène P ne peut être vrai que si le phénomène S a été vrai dans le passé.
- ***Langages généralistes orientés objets***. Un modèle conceptuel des connaissances des parties prenantes sur le domaine a un rôle prépondérant en IE [93]. Ce constat est à l'origine de l'utilisation de techniques de modélisation issues du domaine de la représentation des connaissances, et finalement de l'adoption de l'approche objet en IE. Les langages RML [93-94], Telos[25] [95], Albert II (sic) [96] et KAOS [97-99][26] et UML [50] (préalablement muni

[22] Evénement ou condition sur l'état du système (voir la terminologie en Annexe B).

[23] C'est-à-dire une confrontation entre un modèle fonctionnel et un modèle non-fonctionnel.

[24] IF est un langage plus expressif et généraliste que les logiques modales. C'est un FSA étendu pour la description de propriétés non-fonctionnelles. Nous avons implémenté une transformation du RM vers IF dans la plate-forme R2A pour détecter des incohérences dynamique.

[25] Une extension de RML.

d'une sémantique formelle) sont représentatifs de cette évolution. Ces langages ont un niveau d'abstraction moins élevé que ceux des deux premières catégories. Ils sont plus généralistes car ils permettent de décrire à la fois les propriétés fonctionnelles et non-fonctionnelles, même si leur expressivité concernant les deuxièmes est inférieure à celle des logiques modales (UML est particulièrement pauvre à cet égard). Ils offrent en outre des concepts de haut niveau d'abstraction comme les contraintes structurelles et des règles déductives (voir l'exemple de Telos dans la Figure 3), les notions d'action, d'agent, de pré-condition et de post-condition (voir KAOS dans la figure). Enfin, on notera que Telos (et KAOS dans une moindre mesure) a la particularité d'offrir des mécanismes de métamodélisation ce qui le rend plus flexible et extensible que les autres langages. Pour résumer on peut considérer ces langages comme un assemblage orienté objet de concepts propres aux deux premières catégories, étendu de concepts plus concrets propres au cadre conceptuel de l'IE (cf. Annexe B).

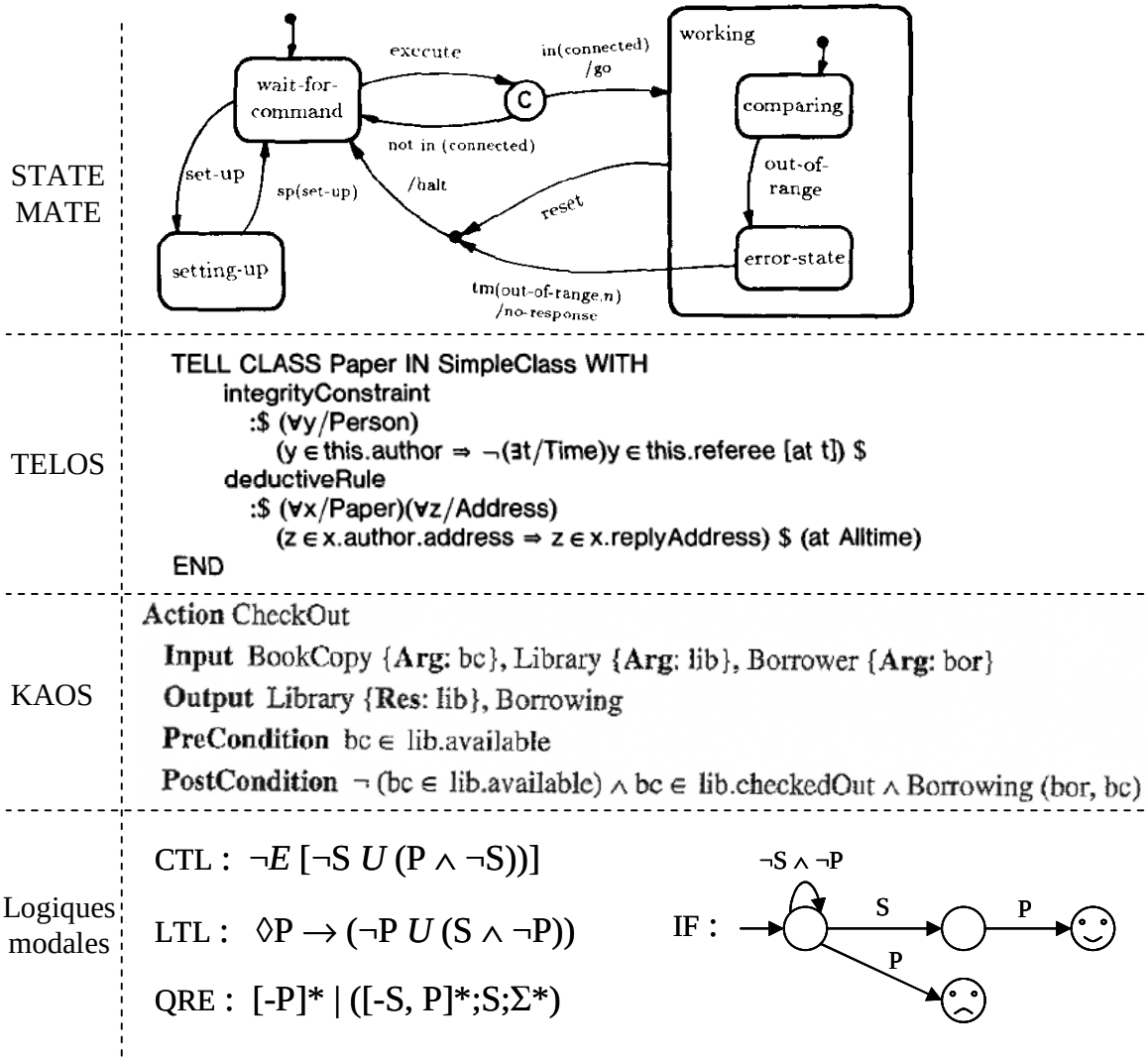


Figure 3 - Exemples de langages formels utilisés en ingénierie des exigences pour l'expression de spécifications opérationnelles (ces exemples ne sont pas représentatifs de l'expressivité de ces langages).

2.3 Approche de modélisation par patrons de spécifications et langage naturel contraint

[26] Nous considérons ici uniquement la partie du langage utile pour l'étape de production d'une spécification opérationnelle (la notion d'objectif, par exemple, est hors de propos).

Nous avons vu que la plupart des langages utilisés en IE pour décrire une spécification opérationnelle ne sont pas accessibles à l'ensemble des parties prenantes, ce qui limite leur implication et rend difficile la validation des spécifications partielles produites. En outre, les ingénieurs sont peu enclins à utiliser les logiques modales, ce qui rend les techniques de vérification formelle inutilisables dans l'industrie [9]. Nous présentons dans cette section les langages de modélisation basés sur des patrons de spécification [8, 51-53]. Nous proposons dans ce manuscrit un langage de modélisation à base de patrons de spécification (le RDL, voir le Chapitre IV).

Les langages à base de patrons de spécification sont moins expressifs que les langages présentés plus haut, mais sont plus accessibles à l'ensemble des parties prenantes. Ils participent à l'implication des parties prenantes dans la rédaction et la validation d'une spécification opérationnelle. Ils favorisent aussi le transfert technologique vers l'industrie des outils de vérification formelle (voir section 3). La section 2.3.1 présente la notion de patron de spécifications, la section 2.3.2 fait l'inventaire des langages de ce type proposés dans la littérature. Enfin, la section 2.3.3 illustre l'utilisation d'un patron de spécifications.

2.3.1 Principe

Un *patron de spécifications* est similaire à un patron de conceptions de Gamma [100]. Il capitalise un savoir-faire dans le but de résoudre un type de problème donné. Alors qu'un patron de conception de Gamma décrit une solution de conception logicielle, un patron de spécifications décrit une solution pour exprimer simplement (mais formellement) une contrainte temporelle[27] [8, 51]. Les langages à base de patrons de spécifications sont généralement munis d'une syntaxe textuelle imitant le langage naturel (langage naturel contraint). L'expressivité d'un langage à base de patrons de spécification dépend du domaine visé (le ou les logiques temporelles). Un patron comporte les éléments suivants :

- ***Un nom et une description de son intention***.
- ***Un ensemble d'exemples d'application***.
- ***Un champ d'application***. Un *champ d'application* décrit l'espace temporel durant lequel la propriété capturée doit être vraie. Cet espace temporel est spécifié en fonction d'un ou plusieurs phénomènes du système. Techniquement, le champ d'application détermine quand la propriété est évaluée durant la simulation des exigences fonctionnelles du système. La Figure 4 présente les champs d'application proposés par Dwyer [8]. Dans cette figure, Q et R sont des phénomènes. A titre d'exemple, le champ d'application « Before Q » signifie qu'une propriété munie de ce champ est évaluée tant que le phénomène Q n'a pas été observé.
- ***Un ensemble de formules logiques paramétrées***. Chaque patron est associé à une ou plusieurs expressions logiques. Chaque expression spécifie la sémantique du patron dans un formalisme particulier. Les formalismes choisis sont typiquement des langages acceptés en entrée par des outils de vérification formelle comme les model-checkers (voir section 3). Les formules associées à un patron sont paramétrées, c'est-à-dire qu'elles comportent un ensemble de variables. Une propriété décrite à l'aide d'un patron consiste en une formule où chaque variable a été instanciée par une expression représentant un phénomène. Une propriété instance d'un langage à base de patrons de spécification est donc une instance d'un de ses patrons (voir l'illustration plus loin).

2.3.2 Travaux existants

Dwyer [8, 51] propose un ensemble de patrons de spécification adaptés à la description de propriétés temporelles (causalité). La sémantique des patrons de Dwyer est définie sur cinq formalismes communément acceptés en entrée d'outils de vérification formelle (CTL [87], LTL [88], GIL [101], INCA [102] et QRE [89]). Pour valider l'approche, Dwyer a réalisé une étude visant à déterminer dans quelle proportion les spécifications d'exigences non-fonctionnelles rencontrées dans la pratique peuvent être capturées par ses patrons [8]. Cette étude portant sur plus de 500 spécifications de propriétés formelles conclut entre autres que 92% des spécifications rencontrées pouvaient être capturées par un des patrons. La Figure 5 présente une description succincte des patrons de Dwyer.

[27] C'est à dire des propriétés optatives (exigences) non-fonctionnelles.

-

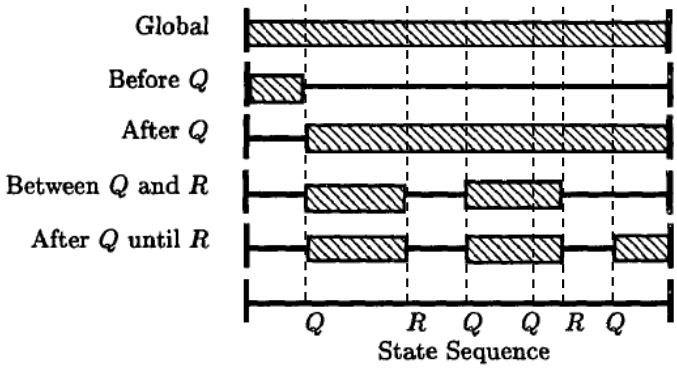


Figure 4 - Les six champs d'application proposés par Dwyer (extrait de [8]).

OCCURRENCE	Absence	A given state/event does not occur within a scope
	Existence	A given state/event must occur within a scope
	Bounded Existence	A given state/event must occur k times, at least k times or at most k times within a scope
	Universality	A given state/event occurs throughout a scope
ORDER	Precedence	A given state/event must always be preceded by a given state/event within a scope
	Response	A given state/event must always be followed by a given state/event within a scope
	Chain Precedence	A given sequence of states/events must always be preceded by a given sequence of states/events (generalization of Precedence Pattern).
	Chain Response	A given sequence of states/events must always be followed by a given sequence of states/events (generalization of Response Pattern).

Figure 5 – Résumé des patrons de spécification de Dwyer [8].

Cheng et Konrad [52] proposent une extension du système de patrons de spécification de Dwyer. Cette extension consiste en l'ajout de patrons pour l'expression de propriétés temporelles temporisées (temps réels). Les formalismes visés diffèrent puisque les logiques impliquées sont de nature différente. Ces formalismes sont MTL [103] (extension de LTL [88]), TCTL [91] (extension de CTL [87]) et RTGIL [104] (extension de GIL [101]). Comme les patrons de Dwyer, les patrons de Cheng ont été définis pour modéliser les propriétés non-fonctionnelles les plus rencontrées dans les spécifications d'exigences. Cheng propose en outre un langage naturel contraint pour aider les parties prenantes à exprimer les propriétés. Ce langage naturel contraint est décrit par une grammaire anglaise très restrictive (une propriété ne peut être représentée que par une seule phrase), ce qui limite les ambiguïtés et facilite l'apprentissage du langage. La Figure 6 présente les patrons ajoutés par Cheng.

Smith propose une approche complémentaire aux travaux de Dwyer, appelée Propel [53]. C'est une approche d'identification progressive de propriétés temporelles. Elle se base sur les patrons de Dwyer étendus avec un ensemble d'options, ce qui augmente l'expressivité des patrons. Le patron « response » de Dwyer spécifie qu'un événement *e1* est toujours suivi d'un autre événement *e2*. A titre d'exemple, le patron « response » de Smith permet en plus de spécifier via les options si *e1* peut être répété plusieurs fois avant *e2*, si des événements peuvent survenir entre *e1* et *e2* ou encore si le patron peut se répéter. Smith munit comme Cheng son langage d'une syntaxe de type langage naturel contraint. Un patron est instancié à l'aide d'un ensemble de phrases (une pour le patron et les autres pour les options. A la différence de Dwyer et Cheng, la sémantique des patrons de Smith est définie sur un seul langage (un FSA étendu pour exprimer les options).

Category	Pattern Name	Description
Duration	Minimum Duration	Describes the minimum amount of time a state formula has to hold once it becomes true. (Ex.: Engine starter system "The system has a minimum 'off' period of 120 seconds before it reenters the cranking mode.")
	Maximum Duration	Captures that a state formula always holds for less than a specified amount of time. (Ex.: Engine starter system "The system can only operate in engine cranking mode for no longer than 10 seconds at one time.")
Periodic	Bounded Recurrence	Denotes the amount of time in which a state formula has to hold at least once. (Ex.: ABS system "The ABS controller checks for wheel skidding every 10 milliseconds.")
Real-time Order	Bounded Response	Restricts the maximum amount of time that passes after a state formula holds until another state formula becomes true. (Ex.: ABS system "From direct client input, detection of and response to rapid deceleration must occur within 0.015 seconds.")
	Bounded Invariance	Specifies the minimum amount of time a state formula must hold once another state formula is satisfied. (Ex.: Engine starter system "If Error 502 is sent to the Driver Information System, then the braking system is inhibited for 10 seconds.")

Figure 6 – Résumé des patrons de spécification ajoutés par Cheng [52] pour le temps réel.

2.3.3 Illustration

Nous illustrons dans cette section l'instanciation d'un patron temps réel de Cheng. Le patron "Bounded Recurrence" permet de décrire la satisfaction périodique d'un phénomène. Une propriété de ce type spécifie que le phénomène en question est vraie au moins une fois tous les *x* unités de temps. La Figure 7 fournit la sémantique de ce patron sous la forme de formules MTL paramétrées (les sémantiques du patron dans les langages TCTL et RTGIL ne sont pas présentées). Chaque formule correspond à la sémantique du patron pour un champ d'application donné. Les variables *P* et *c* représentent le phénomène observé et la durée de la période. Les variables *Q* et *R* représentent les phénomènes observés pour décrire le champ d'application.

$$\underline{\text{MTL:}}\ \begin{cases} \text{Globally:} & \Box(\Diamond_{\leq c}\mathtt{P}) \\ \text{Before R:} & \Diamond\mathtt{R} \rightarrow ((\Diamond_{\leq c}(\mathtt{P} \vee \mathtt{R}))\ \mathcal{U}\ \mathtt{R}) \\ \text{After Q:} & \Box(\mathtt{Q} \rightarrow \Box(\Diamond_{\leq c}\mathtt{P})) \\ \text{Between Q and R:} & \Box((\mathtt{Q} \wedge \neg\mathtt{R} \wedge \Diamond\mathtt{R}) \rightarrow ((\Diamond_{\leq c}(\mathtt{P} \vee \mathtt{R}))\ \mathcal{U}\ \mathtt{R})) \\ \text{After Q until R:} & \Box((\mathtt{Q} \wedge \neg\mathtt{R}) \rightarrow ((\Diamond_{\leq c}(\mathtt{P} \vee \mathtt{R}))\ \mathcal{W}\ \mathtt{R})) \end{cases}$$

Figure 7 - Formules paramétrées MTL de la sémantique du patron "Bounded Recurrence" de Cheng [52].

(1) SafetyReq32 : « Si l'état d'urgence est décrété, un message SOS doit être envoyé au moins toute les 60 secondes. »

(2) After ***Q*** until ***R***, it is always the case that ***P*** holds at least every ***c*** time unit(s).

(3) After <u>(alarmMode ≥ 3)</u> until <u>(alarmMode < 3)</u>, it is always the case that <u>(in(SendSOS))</u> holds at least every <u>60</u> time unit(s).

(4) $\Box$ ((<u>(alarmMode ≥ 3)</u> ∧ ¬<u>(alarmMode < 3)</u>)
→ ((◊≤60 (<u>(in(SendSOS))</u> ∨ <u>(alarmMode < 3)</u>) *W* <u>(alarmMode < 3)</u>))

Figure 8 - Illustration de l'instanciation du patron "Bounded Recurrence" de Cheng.

La Figure 8 illustre la description d'une exigence de sécurité par instanciation de ce patron. L'exigence porte sur l'envoi récurrent d'un message SOS (1). L'utilisateur choisit d'abord le patron et identifie l'expression associée en langage naturel contraint (2). Il instancie ensuite les variables avec les expressions adéquates suivant la modélisation préalable des phénomènes concernés (3). Dans ce cas, on suppose d'une part que l'état d'urgence est décrété lorsque la variable *alarmMode* a une valeur supérieure ou égale à trois et d'autre part que le système envoie un message SOS lorsqu'il est dans l'état *SendSOS* (on considère que l'unité de temps est la seconde). Finalement, l'utilisateur choisit un des formalismes proposés par le langage, en l'occurrence MTL. La formule paramétrée est

sélectionnée (cinquième phrase de la Figure 7). Les variables de cette formule sont instanciées (4). Il ne reste plus qu'à évaluer la propriété MTL obtenue avec l'outil de vérification formelle visé en lui fournissant au préalable la spécification des exigences fonctionnelles. La section suivante présente les outils de vérification formelle utilisée en IE.

3 Détection d'incohérences

Les incohérences entre exigences révèlent des divergences conceptuelles, des conflits d'intérêts entre les parties prenantes ou tout simplement des erreurs de spécification. Leur détection est une activité de premier plan au sein d'un processus IE. Elle guide le processus d'identification des exigences et dans certains cas favorisent la découverte de nouvelles exigences [34]. Nous proposons dans cette section un tour d'horizon des techniques de vérification formelle proposées dans la littérature pour évaluer la cohérence d'une spécification opérationnelle d'exigences. Nous commençons par définir ce qu'est une incohérence en section 3.1 et proposons une classification des différents types d'incohérences rencontrées durant un processus IE. La section 3.2 présente ensuite les trois types d'approches rencontrées dans la littérature, toutes basées sur la confrontation de modèles.

3.1 Notion d'incohérence et classifications

La notion d'incohérence revêt différentes définitions dans la littérature suivant la nature des modèles confrontés. Bien que ces définitions soient relatives à une approche de détection particulière, toutes s'accordent sur le fait que deux spécifications sont incohérentes s'il est impossible de les satisfaire conjointement [105]. Autrement dit, considérer ces deux spécifications vraies implique alors un non-sens. Nous donnons ici une définition très générale de la notion d'incohérence, en accord avec l'ensemble des travaux dans la littérature :

Définition - *incohérence* : une incohérence est détectée si une règle de cohérence est non satisfaite lors de la confrontation des spécifications évaluées [32]. Les types d'incohérences pouvant être détectés dépend de ce que capturent les spécifications confrontées [105].

Définition - *règle de cohérence* : Une règle de cohérence est un invariant logique entre deux spécifications (ou modèles plus généralement).

Il existe un grand nombre de classifications possibles des incohérences. Ces classifications sont utiles pour comprendre plus précisément la notion d'incohérence. Elaasar et Briand en proposent cinq [106] relatives au langage de modélisation UML, mais néanmoins pertinentes pour tout langage de modélisation. Axel van Lamsweerde propose une autre classification [105] relative aux causes d'une incohérence. Nous présentons ici une synthèse de ces deux perspectives. Ces perspectives étant fortement liées, nous avons choisi la perspective d'Elaasar et Briand comme perspective dominante. Cette synthèse, décrite ci-dessous, est utilisée dans la section suivante pour caractériser les travaux proposés dans la littérature :

- ***incohérence syntaxique et sémantique*** : On dit d'une incohérence qu'elle est syntaxique si elle peut être détectée par la seule connaissance de la syntaxe des langages utilisés pour décrire les modèles analysés. Les autres sont qualifiées de sémantique.
- ***incohérence statique et dynamique*** : une incohérence est statique si elle est relative aux sémantiques statiques des langages utilisés, dynamique si elle est relative aux sémantiques dynamiques[28].
- ***incohérence intra- et inter-modèle*** : une incohérence peut être détectée entre plusieurs modèles (incohérence inter-modèle) ou au sein d'un même modèle (incohérence intra-modèle).
- ***niveau de modélisation où l'incohérence a été détectée*** : Une exigence peut être détectée à différents niveaux de modélisation (cf. Chapitre II) : niveau instance ou M0, niveau produit ou M1 et niveau processus ou M2. Par exemple, une incohérence dynamique détectée par un

[28] Une incohérence dynamique est aussi appelée déviation par Lamsweerde, si elle est de niveau instance.

model-checker est de niveau instance alors que des règles de cohérence définies au niveau processus permettent la détection d'incohérences de niveau produit [105].

- ***nature de l'erreur*** : une incohérence peut refléter différents types d'erreurs comme un manque d'information[29] (sous-spécification) ou une contradiction logique [34] (appelée aussi conflit [105]). Axel van Lamsweerde ajoute à cette liste les *collisions* [105] (*collision terminologique*, la *collision dénotationnelle* et la *collision structurelle*)[30]. La notion de collision est définie en section 4 du Chapitre II.

3.2 Principales approches de détection d'incohérences

Cette section vise à caractériser le pouvoir de détection d'incohérences des techniques contemporaines de vérification des exigences. Nous présentons tout d'abord en section 3.2.1 la conformité d'une spécification en tant qu'expression d'un langage comme une technique de vérification. Les techniques de comparaison de spécifications partielles sont ensuite présentées en section 3.2.2. Enfin, nous présentons brièvement en section 3.2.3 les techniques de vérification formelle permettant de détecter des incohérences dynamiques.

3.2.1 Evaluation de la conformité d'une spécification en tant qu'expression d'un langage

Une spécification d'exigences est une expression d'un langage définissant un ensemble de règles syntaxiques et sémantiques (voir Chapitre II). Ces dernières sont des règles de cohérences par définition puisqu'elles définissent une relation logique entre les expressions du langage (des modèles) et le langage lui-même (un autre modèle). L'évaluation de cette relation (relation de conformité) permet de détecter des incohérences statiques et syntaxiques. Les incohérences détectées par cette approche sont uniquement intra-modèles puisque les incohérences potentiellement détectables concernent une spécification seule. Elles peuvent être de niveau M1 ou M2, suivant le niveau de modélisation du modèle évaluée.

3.2.2 Comparaison structurelle de spécifications partielles

La détection d'incohérences par évaluation de la conformité d'une spécification unique n'est pas adaptée à la détection d'incohérences durant les premières phases du processus d'ingénierie des exigences. En effet, nous avons vu en section 2.1 que les exigences sont spécifiées au sein d'une collection de spécifications partielles. C'est pourquoi beaucoup d'approches [33-38] portent sur la détection d'incohérences entre des spécifications partielles traditionnellement classées par points de vue. Pour toutes ces approches, une règle de cohérence consiste en une comparaison structurelle des modèles analysés dans le but de détecter une contradiction logique. C'est une expression de la logique du premier ordre, similaire à une expression OCL [107] (cf. Chapitre II) mais comportant deux contextes au lieu d'un pour pouvoir comparer deux modèles. Elle vise à détecter une superposition sémantique (voir la définition ci-dessus) entre deux parties de modèles distincts, puis le cas échéant, de vérifier si ces deux parties satisfont à un invariant. Lorsque ce dernier est violé, une incohérence est détectée [32]. La détection d'une incohérence présuppose donc l'existence de redondances identifiables dans l'information analysée (on ne peut comparer que ce qui est comparable …).

Définition – *superposition (sémantique)* : une superposition sémantique est une situation où deux fragments de modèles appartenant chacun à un des deux modèles confrontés représentent une même information [32].

[29] En IE, les travaux considèrent rarement une sous-spécification comme une incohérence. Nous adoptons la position de Elaasar et Briand ce qui permet d'être cohérent avec la notion de conformité de l'IDM. A titre d'exemple, un modèle de catalogues de produits est incohérent si un de ses produits n'a pas de nom.

[30] Une collision terminologique est une situation où des termes différents sont utilisés pour la désignation d'un même concept, une collision dénotationnelle correspond à la situation inverse et une collision structurelle est une généralisation de la collision terminologique à la structure de l'information.

La Figure 9 illustre la notion de règle de cohérence pour les approches de comparaison. Elle présente deux extraits de points de vue (les spécifications partielles (a) et (b)) décrivant un protocole de communication de téléphone fixe, ainsi qu'un exemple de règle de cohérence (c). Cette règle est définie pour tout point de vue (*VP*) expression du diagramme de machines à états (*STD*) munie d'une sémantique définie sur le domaine D_S. Elle spécifie que si une transition entre deux états est représentée dans un point de vue (VP_S), et que ces deux états sont aussi représentés dans un autre point de vue (VP_D), alors la transition doit être aussi représentée dans VP_D. On remarque que cette règle n'est pas satisfaite par les points de vue (a) et (b), puisque la transition « callee replaces receiver » dans (a) n'est pas représentée dans (b). Il y a donc incohérence dans ce cas.

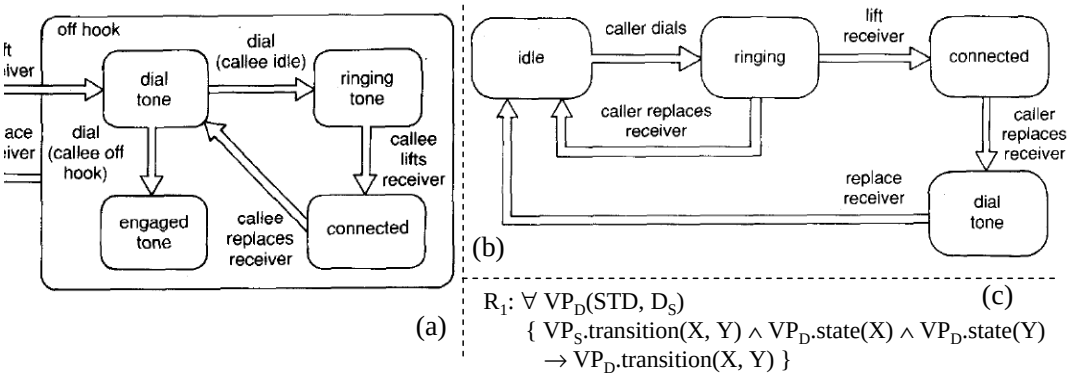


Figure 9 -Exemple d'une règle de cohérence (approche par comparaison, extrait de [34]).

Les incohérences détectées par les approches de comparaison sont syntaxiques (et sémantiques d'un point de vue statique), statiques, inter-modèles et sont appliquées au niveau de modélisation M1. Une limitation importante de ces approches est qu'elles ne permettent pas de détecter un manque d'information ; les règles de cohérence ne s'appliquent que localement puisque la comparaison ne se fait qu'entre paire de modèles [39].

3.2.3 Confrontation dynamique

Les approches précédemment citées ne peuvent détecter d'incohérences dynamiques car elles ne s'intéressent qu'à la structure des modèles (type d'informations statique). La détection d'incohérences dynamiques consiste à analyser l'évolution d'une spécification opérationnelle au cours du temps. Elles s'appuient sur les méthodes formelles via l'utilisation d'outils mathématiques dédiés (par exemple les model-checkers [90, 108-112]). Nous présentons dans cette section les principaux travaux en IE basés sur le model-checking.

Un model-checker est un outil de vérification formelle dont l'algorithme calcule le produit cartésien synchrone de structures de Kripke. Ces structures capturent soit une description d'un processus du système, soit une contrainte logique non-fonctionnelle devant être vérifiée par le système. Les descriptions des processus du système sont spécifiées à l'aide de langages FSA comme Promela (le langage du model-checker Spin [109]). Les contraintes sont décrites à l'aide de logiques modales (cf. section 2.2). Le résultat du produit cartésien des descriptions fonctionnelles représente les comportements du système. Dans cette approche, une contrainte logique non-fonctionnelle est satisfaite si le résultat du produit cartésien de sa négation avec la description fonctionnelle est vide (dans le cas contraire, le résultat décrit un ensemble de contre-exemples invalidant la contrainte). Un model-checker permet donc de confronter une description fonctionnelle considérée correcte avec un ensemble de propriétés non-fonctionnelles que l'on veut voir satisfaites par la description fonctionnelle. Pour l'appliquer à la vérification d'une spécification opérationnelle d'exigences, on procède comme suit :

- Les propriétés de l'environnement (par définition indicatives) sont considérés comme faisant partie de la description du système.
- Les exigences fonctionnelles sont aussi considérées comme faisant partie de la description du système (bien qu'optatives par définition). Elles vont donc être simulées.

- Seules les exigences non-fonctionnelles gardent leur caractère optatif (souhaité). Elles jouent alors le rôle de règles de cohérence et sont vérifiées une à une. Une incohérence est alors détectée si une exigence non-fonctionnelle est violée (le résultat du produit cartésien synchrone n'est pas vide).

Les model-checkers sont de puissants outils mathématiques. Cependant, les exigences acceptées en entrée doivent être exprimées à l'aide de langages formels difficiles à manipuler par les parties prenantes (logiques modales et FSA). De plus, un model-checker ne peut accepter en entrée une collection de spécifications partielles, ce qui n'est pas compatible avec le résultat d'une approche d'indentification des exigences par points de vue ou plus généralement par aspects (cf. section 2.1). Pour pallier cette difficulté, de nombreux outils (e.g. [47, 113-115]) ont été proposés, tous proposant des langages d'entrée plus adaptés pour les parties prenantes (en particulier les langages à base de patrons de spécification présentés en section 2.3) et acceptant une spécification d'exigences sous la forme d'une collection de spécifications partielles. Les architectures de ces outils sont très similaires ; elles partagent toutes les caractéristiques suivantes :

- Le cœur de l'architecture est un model-checker acceptant des modèles instances de FSA ou de logiques modales.
- L'achitecture comporte un mécanisme de traduction du ou des langage(s) de modélisation supporté(s) par l'outil vers le ou les langage(s) d'entrée du ou des model-checker(s).
- L'architecture comporte un mécanisme de composition de spécifications partielles car le model-checker accepte un unique modèle. Spider [47] utilise par exemple l'outil Hydra [116].

Outil			Model-checker associé		
Nom	Langages d'entrée		Nom	Langages d'entrée	
	Indicatif	Optatif		Indicatif	Optatif
TCM [113]	Diagramme d'activité UML	LTL [88]	NuSMV [110]	?	CTL [87], LTL [88]
Spider [47]	Machines à états et diagrammes de classes UML	Patrons de spécification [52]	Spin [109]	Promela	LTL [88]
Hugo [114]	Machines à états UML	diagrammes de collaboration UML	Spin [109]	Promela	FSA Büchi
OBP [115]	SDL [86]	CDL + observateurs IF [90]	IF [90]	FSA Büchi	FSA Büchi (observateur)

Tableau 1 – Exemples d'outils de vérification des exigences, langages d'entrée et model-checker associés.

Le Tableau 1 répertorie quelques outils de vérification dynamique proposés dans la littérature. Par exemple, l'outil Hugo [114] accepte un ensemble de machines à états UML et un ensemble de diagrammes de collaboration UML. Les premiers sont transformés en un modèle Promela, les deuxièmes en automates de büchi, puis chaque automate de büchi est confronté au modèle Promela grâce au model-checker Spin [109]. Ce tableau souligne la grande variabilité des model-checkers visés et des langages d'entrée supportés. Le choix d'un model-checker dépend des propriétés à vérifier et donc du but de l'analyse[31]. Les langages supportés sont souvent semi-formels[32] (e.g. diagrammes UML), ce qui facilite leur utilisation par les parties prenantes.

[31] Les model-checkers visent tous à démontrer la possibilité ou non d'atteindre un état particulier, étant donné un ensemble de contraintes sur les chemins susceptibles d'y amener. Cependant, chacun est spécialisé pour la résolution en un temps raisonnable d'un type de problèmes. A titre d'exemple, Eshuis [113] présente TCM [117] comme un outil ciblant initialement deux model-checkers (Spin [109] et NuSMV [110]) mais

4 Synthèse

Une spécification d'exigences est une collection de spécifications partielles, hétérogènes et potentiellement incohérentes. Ces spécifications peuvent être exprimées à l'aide de (i) formalisme ou de (ii) langages à base de patrons de spécification. Ces derniers sont moins expressifs mais plus accessibles que les formalismes, ce qui favorise l'implication des parties prenantes. En outre, ils ne permettent de spécifier que les exigences non-fonctionnelles. Les formalismes diffèrent quant à leur expressivité. On constate néanmoins un certain nombre de caractéristiques qu'un formalisme doit comporter pour pouvoir représenter un domaine :

- Représentation orientée objet des connaissances du domaine.
- Support d'une logique modale pour décrire les propriétés non-fonctionnelles temporelles.
- Support de mécanismes de métamodélisation pour pouvoir étendre ou modifier facilement l'expressivité du langage et décrire facilement une grande variété de domaines métiers différents).

approches / types détectables	Evaluation de la conformité d'un modèle	Comparaison structurelle de modèles	Confrontation dynamique de modèles
syntaxique	×	×	
sémantique	×	×	×
statique	×	×	
dynamique			×
intra-modèle	×		×
inter-modèle		×	
M2	×		
M1	×	×	
M0			×
manque d'information	×		
contradiction logique	×	×	×
collisions			

Tableau 2 - Comparaison des pouvoirs de détection d'incohérences des trois approches de détection d'incohérences.

Concernant la détection d'incohérences, nous avons vu trois types d'approches principales. Le Tableau 2 compare les pouvoirs de détection de ces approches, suivant leur capacité à détecter les types d'incohérences introduits en section 3.1. On peut faire les remarques suivantes :

- la comparaison structurelle est une approche puissante pour identifier les exigences car elle aide à la découverte de désaccords conceptuels entre les parties prenantes [34].
- L'évaluation de la conformité d'un modèle est plus complète que les autres concernant les types d'incohérences détectées mais elle ne s'applique qu'à un modèle à la fois.
- la comparaison structurelle travaille au niveau local puisqu'elle consiste en la confrontation de modèles deux à deux. Il n'est ainsi pas possible de détecter des manques d'informations (sous-spécification) [39], contrairement à l'évaluation de la conformité d'un modèle (un manque d'informations détecté entre deux modèles peut très bien être résolu par un troisième modèle).

remarque que le deuxième est plus approprié à la recherche de boucles infinis car implémentant un algorithme symbolique de recherche de chemins.

[32] La sémantique de ces langages comportent donc des variations sémantiques (voir par exemple la discussion dans [114] concernant les machines à états UML ou les travaux de formalisation de langages semi-formels [118-119]), ce qui ajoute à la variabilité des outils.

- la confrontation dynamique de modèles est la seule à détecter des incohérences dynamiques. Cependant elle nécessite au préalable une spécification opérationnelle globale et cohérente statiquement pour pouvoir être appliquée.
- aucune des approches ne facilite la détection de collisions.

Le chapitre suivant présente (i) l'ingénierie dirigée par les modèmes et (ii) les travaux portant sur la composition de modèles. L'ingénierie dirigée par les modèles semble un bon cadre conceptuel de métamodélisation pour produire des outils industriels intégrant dans un même environnement les techniques de vérification modernes et une grande variété de langages de modélisation [2]. La composition de modèles quant à elle est une piste encore peu étudiée pour détecter des incohérences [40]. Elle peut être une solution intéressante puisque l'obtention d'un modèle global (i) autorise l'application d'une approche basée sur l'évaluation de la conformité d'un modèle et (ii) est un prérequis pour l'application de techniques de vérification formelle.

Chapitre II Un cadre conceptuel pour la métamodélisation : l'ingénierie dirigée par les modèles.

Résumé

L'*ingénierie dirigée par les modèles* [1-3, 120-121] (IDM[33]) englobe l'ensemble des approches de génie logiciel où les informations produites durant un projet logiciel sont systématiquement modélisées et les modèles résultants successivement transformés jusqu'à obtention du logiciel [2]. Là où les spécifications de haut niveau étaient utilisées comme artefacts de documentation et supports de discussions (utilisation « contemplative »), l'idée de l'IDM est de les rendre productives tout au long du cycle de développement. Cette approche générative promeut une description systématique des logiciels à l'aide de spécifications plus abstraites que le code (exigences, architecture, conception). Elle diminue la distance conceptuelle entre le domaine du problème et le domaine de la solution [2], ce qui permet de s'affranchir dans une certaine mesure de ce que Brooks appelle la complexité accidentelle [12] (liée à la construction d'une solution et non au problème adressé).

Les idées au cœur de l'IDM ne sont pas neuves. En attestent les travaux dans le domaine du MIC [122-123] (*model-integrated computing*) dédiés au développement de systèmes embarqués complexes. Dans cette approche, les modèles sont systématiquement utilisés pour développer le logiciel mais aussi pour générer des environnements de développement spécifiques aux projets de développement. On peut aussi citer les travaux portant sur les langages spécifiques de domaine [13-15] (*DSL* pour *domain-specific language*), les usines logicielles [124], ou encore les approches de développement par aspects [44-46, 76-82] (AOSD pour *aspect-oriented software development*). Les actions du consortium industriel de normalisation OMG [125] (*object management group*) ont fortement amplifié l'intérêt porté à l'IDM. Parmi celles-ci, on peut souligner le MDA [126] (*model-driven architecture*), approche industrielle de développement prônant une séparation stricte entre spécifications indépendantes et spécifications spécifiques à une plate-forme donnée. Cette initiative est à l'origine d'une première normalisation des notions clés de l'IDM.

Ce chapitre fournit un cadre conceptuel homogène pour la formalisation des langages et la manipulation de spécifications partielles hétérogènes. Ce cadre conceptuel se base pour l'essentiel sur les normes proposées par l'OMG (en particulier le MDA) ; il incorpore quelques notions de modélisation propres à l'ingénierie des exigences et la théorie des langages formels. La section 1 définit la notion de *langage* et discute des notions d'*ambiguïté* et de *règles sémantiques*. La section 2 présente la manière dont l'IDM représente les langages et leurs expressions à l'aide de *métamodèles* et *modèles*. Elle décrit aussi la notion de *métamodélisation* et sa réalisation concrète à travers l'architecture MDA des niveaux de modélisation. La section 3 présente les langages de *transformation de modèles*, dédiés à la manipulation des modèles et la description de relations entre métamodèles (en particulier de *relation sémantique*). Enfin, la problématique de *composition de modèles* est étudiée en section 4 et la section 5 donne un résumé succinct du chapitre.

[33] MDE (*model-driven engineering*) dans la littérature anglophone.

1 Théorie des langages

Un langage de modélisation (ou simplement langage) est un ensemble d'expressions, toutes conformes à une *syntaxe.* Une syntaxe est un système formel de règles produisant l'ensemble des expressions du langage (une expression appartient à un langage si elle est syntaxiquement correcte). Un langage est muni d'une *sémantique* qui attribue à chaque expression un ou plusieurs sens. La sémantique d'un langage est spécifiée par une *relation sémantique* associant à chaque expression un ou plusieurs éléments d'un *domaine*. Le domaine d'un langage définit l'expressivité de ce dernier. Un langage est donc une syntaxe munie d'une sémantique définie sur un domaine. Les définitions proposées ici sont conformes à celles données par Harel [127], excepté le fait que nous définissons la sémantique d'un langage comme une relation et non une fonction. Cela permet d'inclure dans notre définition d'un langage les langages non formels (voir la notion d'ambiguïté plus loin).

Définition – ***langage*** : Un langage L = (N, f) est la donnée d'une syntaxe N et d'une relation sémantique f. Par commodité, on considère aussi un langage comme l'ensemble des expressions syntaxiquement correctes.

Définition – ***domaine (d'un langage)*** : Le domaine d'un langage est l'ensemble des informations pouvant être représentées par celui-ci. Le domaine d'un langage détermine son expressivité (un type d'informations). Il détermine aussi le type d'analyses pouvant être effectuées.

Définition – ***relation sémantique d'un langage*** : Soit L = (N, f) un langage définit sur un domaine D, la sémantique de L est spécfiée par la relation sémantique $f : N \rightarrow D$. Cette relation associe les expressions conformes à la syntaxe N de L et les éléments du domaine D de L. Dans ce contexte, f(e) correspond au sens de l'expression e si $e \in L$.

Une relation sémantique d'un langage L peut être définie par un ensemble de *règles sémantiques* portant sur les éléments du domaine de L. Ces règles sont traditionnellement partitionnées suivant qu'elles portent sur la structure des éléments du domaine (règles statiques) ou les évolutions possibles de ces structures dans le temps (règles dynamiques). Les règles dynamiques d'un langage sont généralement définies pour des expressions satisfaisant les règles statiques (dans le cas d'une expression représentant un programme, celui-ci ne peut pas être exécuté s'il comporte des erreurs détectées durant l'analyse statique). Le Tableau 3 fournit quatre exemples de règles statiques et dynamiques pour le langage Java et un langage classique de machine à état. Ces deux classes de règles sémantiques sont caractérisées ci-dessous :

- ***sémantique statique (structurelle)*** : Ensemble des règles sémantiques contraignant la structure des expressions possibles du langage. Tout comme les règles syntaxiques, une règle statique est un invariant satisfait par toutes les expressions du langage. La sémantique statique définit aussi la signification des expressions du langage.
- ***sémantique dynamique*** : Ensemble des règles sémantiques contraignant les évolutions possibles des expressions du langage. La sémantique dynamique d'un langage spécifie donc le comportement de ses expressions. A noter que certains langages n'ont pas de sémantique dynamique (par exemple la vue UML [50] décrivant les diagrammes de classes).

	Java	Machine à état
Sémantique statique	Une classe doit redéfinir l'ensemble des constructeurs de sa super-classe.	Il n'existe qu'un état de départ par état composite.
Sémantique dynamique	Si une classe comporte au moins deux fonctions également nommées, la fonction considérée lors d'un appel est celle dont les types des paramètres sont le plus spécialisés.	Une transition est tirée lorsque le flot de contrôle est situé dans son état source et que sa garde est satisfaite pour l'état courant du système.

Tableau 3 - Exemples de règles sémantiques.

Les langages de modélisation sont usuellement classés suivant la nature de leur relation sémantique. On distingue trois grandes catégories : les langages informels, formels et semi-formels. Un langage est informel lorsque sa sémantique n'est pas explicitement spécifiée. Toute syntaxe seule peut être considérée comme un langage informel. Le langage naturel est représentatif de cette famille. Un langage est formel lorsque toutes ses expressions sont non ambiguës, c'est-à-dire qu'elles n'ont qu'un et un seul sens. Axel van Lamsweerde propose dans [11] une classification des langages formels utilisés en ingénierie logicielle (par exemple Z [128] ou les logiques modales). Ces langages ont la particularité d'être défini sur un formalisme ; leur domaine est algèbrique (logique du premier ordre, logique modale …), ce qui offre une capacité d'analyse automatique de ses expressions (preuves, inférences de faits par déduction). Enfin, on qualifie un langage de semi-formel lorsque sa sémantique n'est pas entièrement définie (par exemple UML [50], i* [129] et KAOS [97]). Ces langages peuvent comporter intentionnellement des *variations sémantiques* [41, 114, 118-119] et certaines de leurs expressions sont ambigües.

Définition – ***ambiguïté*** : Une expression e d'un langage L = (N, f) est ambiguë ssi son image par f est un ensemble d'éléments du domaine sémantique. Plus formellement, on a :

Soit un langage L = (N, f) et e une expression telle que $e \in L$, e est ambiguë ssi card (f(e)) > 1.

Définition – ***langage formel*** : Un langage L = (N, f) est formel ssi son domaine sémantique est algébrique et l'ensemble de ses expressions sont non ambiguës. La relation sémantique d'un langage formel est donc une fonction. Dans la suite, nous utiliserons le terme de *fonction sémantique* pour désigner la relation sémantique d'un langage formel. Plus formellement, on a :

Un langage L = (N, f) est formel ssi $\forall e \in L$, card (f(e)) = 1

Notation – *fonction sémantique* : On utilisera également le terme de « fonction sémantique » pour désigner la relation sémantique d'un langage formel.

Un langage informel est accessible à tout un chacun et permet d'exprimer aisément une grande variété d'informations. Cependant un langage informel est source d'incompréhensions (problème des ambigüités) et ses expressions manquent de concision. A l'inverse, les langages formels permettent d'obtenir des spécifications directement analysables par des outils de vérification formelle (cf. section 3 du Chapitre I) mais sont rarement accessibles aux non mathématiciens (la majorité des parties prenantes à un projet logiciel) car leur syntaxe est hermétique. Dans ce manuscrit, nous nous intéressons aux langages formels ou semi-formels, le but étant de produire une spécification des exigences analysables par des outils de vérification formelle. En particulier, nous nous intéressons aux langages naturels contraints (cf. section 2.3 du Chapitre I) car ces langages offrent un bon compromis entre accessibilité et non ambigüité de leurs expressions.

2 Métamodélisation

Quelle que soit la discipline scientifique considérée, un *modèle* est une abstraction d'une information dans le but de résoudre un problème. Un modèle est une abstraction dans la mesure où il *représente* uniquement une partie de l'information considérée. Cette représentation est superficielle pour deux raisons. En premier lieu, il est rarement possible de décrire exhaustivement une information. Deuxièmement, il n'est pas souhaitable de tout modéliser ; un "bon modèle" représente uniquement les aspects de l'information nécessaires à la résolution du problème visé[34]. La description d'informations hétérogènes comme les exigences nécessitent donc l'utilisation d'une variété de langages, chacun adaptés à la description d'un type d'informations particulier.

[34] A titre d'exemple, il est inutile de modéliser les forces électromagnétiques d'un environnement pour calculer en physique classique la trajectoire d'un corpuscule (il est plus raisonnable de modéliser la force de gravité). Il en va tout autrement en physique nucléaire si le corpuscule est un électron. Cet exemple souligne l'importance du but ayant motivé l'acte de modélisation et l'impact qu'il a sur la perception de l'information modélisée.

Cette section décrit le cadre conceptuel offert par l'IDM pour manipuler de manière homogène une grande variété de langages. Ce cadre conceptuel founit un environnement de métamodèlisation adapté à la description des exigences à l'aide d'une collection de spécifications partielles et hétérogènes. Les sections 2.1 illustre comment spécifier la sémantique statique d'un langage à l'aide d'un métamodèle et de contraintes OCL. La section 2.2 présente les relations entre modèles, métamodèles et langages ; elle clarifie le le lien entre les notions de la théorie des langages présentées en section 1 et les notions propres à l'IDM. Enfin, la section 2.3 décrit l'architecture MDA des niveaux de modélisation faisant de l'IDM un cadre conceptuel de métamodélisation homogène.

2.1 Modèles et métamodèles

Un modèle est un graphe d'*objets* et de *liens* typés représentant une information. La Figure 10 fournit un exemple de modèle représentant un catalogue de produits. Chaque objet est instance d'un concept (par exemple « Catalog »), encapsule un ensemble de données (par exemple « luxury goods ») et est lié à d'autres objets par des liens (par exemple « category »). Il est utile dans certains cas de désigner une partie d'un modèle pour des raisons de réutilisation, ou de description. On parle alors de *fragment de modèle*[35]. Cette notion est par exemple utilisée dans le cadre de la génération de modèles de test [130]. Notons qu'un modèle est aussi un fragment de modèle, et inversement. Nous donnons une définition ci-dessous :

Définition - ***Fragment de modèle*** : Ensemble d'objets et de liens appartenant à un modèle. Un fragment de modèle peut comporter des liens dont la source ou la destination n'est pas définie. Les valeurs des propriétés des objets peuvent aussi ne pas être définies.

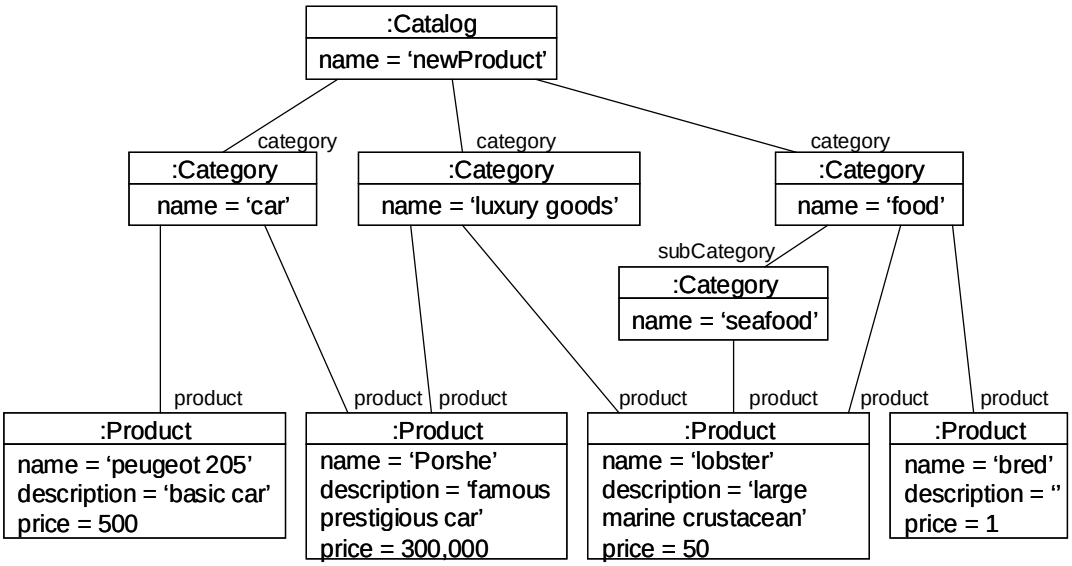


Figure 10 - Exemple de modèle de catalogue de produits conforme au métamodèle de la Figure 11.

Le traitement automatique d'un ensemble de catalogues de produits nécessite de définir précisément ce qu'est un modèle de catalogues de produits. Cette description est réalisée à l'aide d'un *métamodèle* en IDM. Un métamodèle représente un *type d'informations*. Il détermine une manière singulière de représenter les informations de ce type : concepts clés, nature des relations entre ces concepts. Un métamodèle sert le plus souvent à représenter la syntaxe abstraite d'un langage de modélisation. Il peut aussi représenter un domaine de connaissances (ontologie [131]). Dans le premier cas, un métamodèle définit une structure de données adaptée pour représenter les arbres syntaxiques d'un langage. Dans le deuxième cas, il définit une structure de données adaptée pour la représentation d'une connaissance particulière. En IDM, la notion de métamodèle est un outil normalisé de modélisation, comme les DTD dans le monde du web, les grammaires pour la communauté des langages formels, ou les modèles conceptuels pour les bases de données.

[35] On parle aussi de « constellation d'objet » dans certains travaux en IE.

Un métamodèle est un graphe regroupant un ensemble de *métaclasses* et de *relations*. La Figure 11 présente un métamodèle simplifié de catalogues de produits (le modèle conceptuel d'une base de données d'un système de gestion informatisé des stocks d'un distributeur). Ce métamodèle définit le concept de catalogue (représentée par la métaclasse CATALOG), regroupant un ensemble de produits (métaclasse PRODUCT) classés par catégories (métaclasse CATEGORY). Une métaclasse comporte un ensemble de *propriétés* comme l'*attribut* « prix » (*price*) et l'*association* « catégorie » (*category*) pour la métaclasse PRODUCT. Une relation est spécifiée par une ou deux associations liant deux métaclasses. Une association représente une direction d'une relation et définit une cardinalité. Lorsqu'une relation est spécifiée par deux associations, on dit que l'une est l'opposée de l'autre (par exemple les associations « parent » et « subCategory » sont opposées).

Comme le suggère la Figure 11, un métamodèle est aussi un modèle, puisque c'est une expression d'un langage de modélisation. En IDM, un métamodèle est conforme au langage *MOF* [132] (voir section 2.3). MOF comporte des mécanismes d'abstraction dont la composition et l'héritage. L'association *category* de la métaclasse CATALOG par exemple est une composition, représentée par un losange noir. Le métamodèle des catalogues de produits illustre aussi l'héritage. On peut considérer un métamodèle comme un ensemble de *contraintes structurelles* contraignant les informations pouvant être représentées et décrivant une partie de la sémantique statique du langage spécifié[36]. Ces contraintes structurelles sont (i) les cardinalités définies par les associations et (ii) les règles propres aux mécanismes d'abstraction du MOF. Par exemple, un objet de type « Category » peut comporter une chaîne de caractère instance de l'attribut « name » puisque la métaclasse « Category » est un super-type de la métaclasse « NamedElement ». Autre exemple, la composition interdit par définition le fait qu'un objet se contienne lui-même.

Il est important de noter qu'un métamodèle n'est qu'une manière possible de représenter la syntaxe abstraite d'un langage de modélisation, tout comme il est possible de proposer des langages différents pour capturer un même type d'informations. Cette variabilité est accentuée par l'utilisation de langages de modélisation divers mais dont les types d'informations sont sémantiquement liés (partageant par exemple une même préoccupation). Les approches de modélisation à l'aide de DSL [14] (*Domain Specific Language*) illustrent le besoin d'une séparation des préoccupations par l'usage de langages variés. Dans ce contexte, le type d'informations capturées correspond à une union des types d'informations des langages employés.

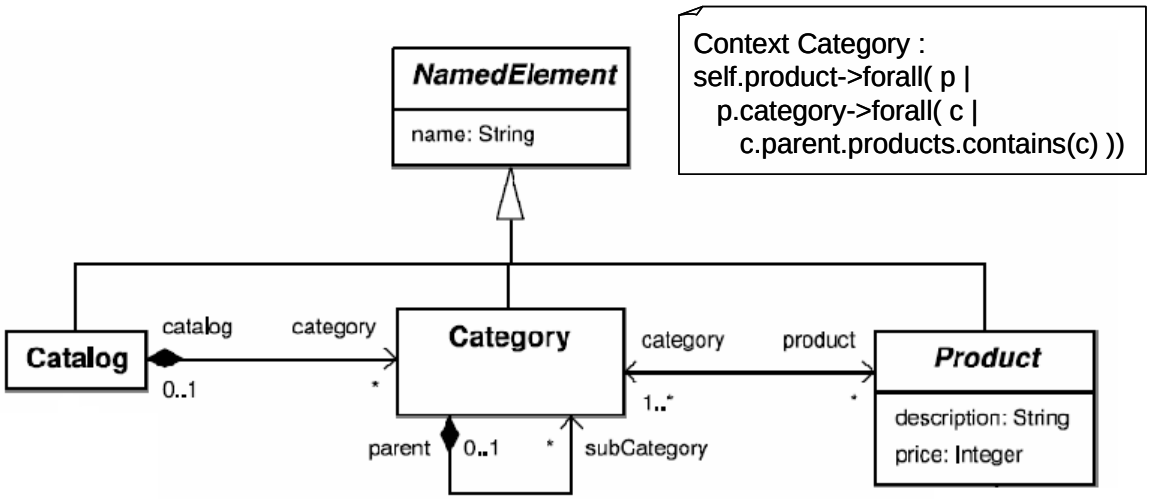


Figure 11 - Exemple de métamodèle représentant des catalogues de produits.

Le langage MOF n'est pas assez expressif pour décrire toutes les contraintes structurelles sur un ensemble de modèles. Par exemple, il est impossible de spécifier avec MOF une règle structurelle R1 statuant qu'un produit d'une catégorie est aussi un produit de toutes ses sous-catégories. En IDM, on utilise dans ce cas une expression du langage textuel OCL [107] (*Object Constraint Language*), plus expressif que le MOF. La description du métamodèle de la Figure 11 comporte une expression OCL

[36] Ce sont des règles de cohérence intra-modèle (voir la section 3.2.1 du Chapitre I).

représentant R1 (le texte dans l'annotation). MOF et OCL ne sont cependant pas suffisants pour décrire toutes les contraintes structurelles d'un métamodèle[37]. Certaines doivent être spécifiées à l'aide d'autres langages ou simplement informellement. Sans entrer dans les détails, on peut retenir qu'une spécification d'OCL (i) n'a pas d'effet de bord, (ii) ne peut spécifier que des règles sémantiques statiques pour l'essentiel et (iii) est suffisamment simple pour être utilisable par des non mathématiciens.

2.2 Relations entre modèles, métamodèles et langages

Tout comme un langage est un ensemble d'expressions syntaxiquement correctes, un métamodèle est un ensemble de modèles dit conformes à ce métamodèle, c'est-à-dire satisfaisant ses contraintes structurelles. Au préalable, les éléments de ces modèles doivent aussi être des instances des éléments du métamodèle (métaclasses ou relations). Les modèles conformes au métamodèle sont tous instances de ce métamodèle, ce qui n'est pas toujours le cas dans l'autre sens. La *relation d'instanciation* et la *relation de conformité* entre modèles et métamodèles sont définies ci-dessous. Le modèle de la Figure 10 est conforme au métamodèle de la Figure 11 (on peut donc dire que ce modèle représente une expression du langage de catalogues de produits). Cette relation de conformité est essentielle pour l'IDM et est analogue à la relation d'appartenance liant une expression textuelle à une grammaire, un programme à son langage de programmation ou encore un document XML à sa DTD (*Document Type Description*). La relation de conformité permet de donner un type aux modèles, ce qui facilite l'écriture de programme comme les transformations de modèles (cf. section 3).

Définition – ***relation d'instanciation*** : La relation d'instanciation R est un ensemble de couple (a, b) où b est un métamodèle et a est un modèle dont chaque élément (objets et liens) est une instance d'un élément de b (métaclasses et propriétés). On dit qu'un modèle a est *instance d'*un métamodèle b ssi (a, b) □ R.

Définition – ***relation de conformité*** : Un modèle a est conforme à un métamodèle b ssi a (i) est une instance de b, (ii) les contraintes structurelles définies par b et exprimées avec le MOF (cardinalité, composition et héritage) sont satisfaites par a et (iii) les contraintes OCL de b sont satisfaites par a. La relation de conformité est un couple (a, b) où a est un modèle *conforme au* métamodèle b.

La Figure 12 résume les différentes notions abordées précédemment et insiste sur leurs relations. Un langage de modélisation permet la capture d'un certain type d'informations (son domaine). Ce type d'informations est déterminé par les buts ayant motivé la construction du langage. Un type d'informations peut être considéré comme la composition de type d'informations plus élémentaire (représentant des préoccupations). Une information de ce type est capturée par une expression appartenant à ce langage. L'information capturée par une expression est incluse dans le type d'informations capturé par son langage. Un métamodèle peut représenter la syntaxe abstraite d'un langage de modélisation tout comme un type d'informations (le métamodèle de la Figure 11 par exemple). Il détermine une manière de représenter les expressions de ce dernier. Toute expression d'un langage de modélisation est représentée par un modèle conforme au métamodèle associé.

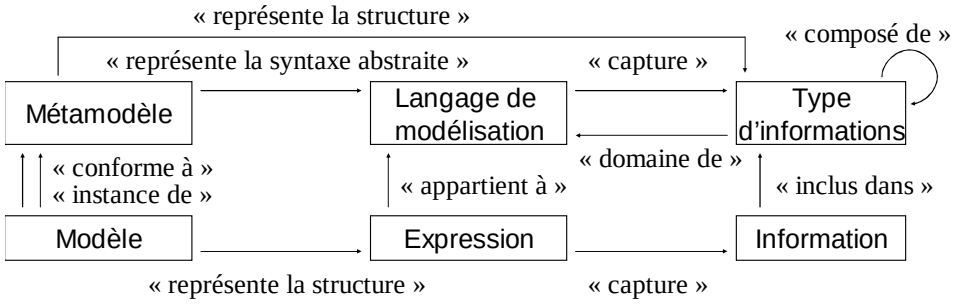


Figure 12 – Schématisation des relations entre les notions de métamodèle, langage type d'informations.

[37] Certaines règles récursives entre autres.

2.3 Niveaux de modélisation : l'architecture MDA

La Figure 12 schématise les relations entre des artefacts de modélisation appartenant uniquement à deux niveaux de modélisation (modèle, expression et information d'un côté, métamodèle, langage et type d'informations de l'autre). En IDM, un métamodèle est aussi un modèle[38], c'est-à-dire qu'il est aussi une expression d'un langage, appelé *MOF* (*Meta Object Facility*) [132][39]. Ce langage est représenté par le méta-métamodèle MOF, se situant à un niveau de modélisation au-dessus des métamodèles. Le MOF permet de représenter un grand nombre de langage de modélisation, incluant par exemple UML ou OCL.

La Figure 13 présente l'architecture MDA [126, 133], définie par l'OMG [125] pour supporter la métamodélisation. Cette architecture généralise les relations discutées dans la section précédente à un ensemble quelconque de niveaux de modélisation. L'architecture MDA comporte quatre niveaux de modélisation. Le MOF est situé au niveau le plus haut (M3). Il a la particularité d'être conforme à lui-même (il s'auto-décrit), ce qui permet de limiter à quatre le nombre de niveau de modélisation de l'architecture MDA (M0 à M3). Les niveaux M1 et M2 correspondent à ceux présentés en Figure 12[40]. Dans ce contexte, le métamodèle UML appartient au niveau M2, les modèles UML comme les diagrammes de classes appartiennent au niveau M1. Enfin, les instances au niveau M0 représentent le système instancié (monde), c'est-à-dire le système en fonctionnement (par exemple, une instance d'une classe UML définie dans un diagramme de classe du niveau M1 appartient au niveau M0).

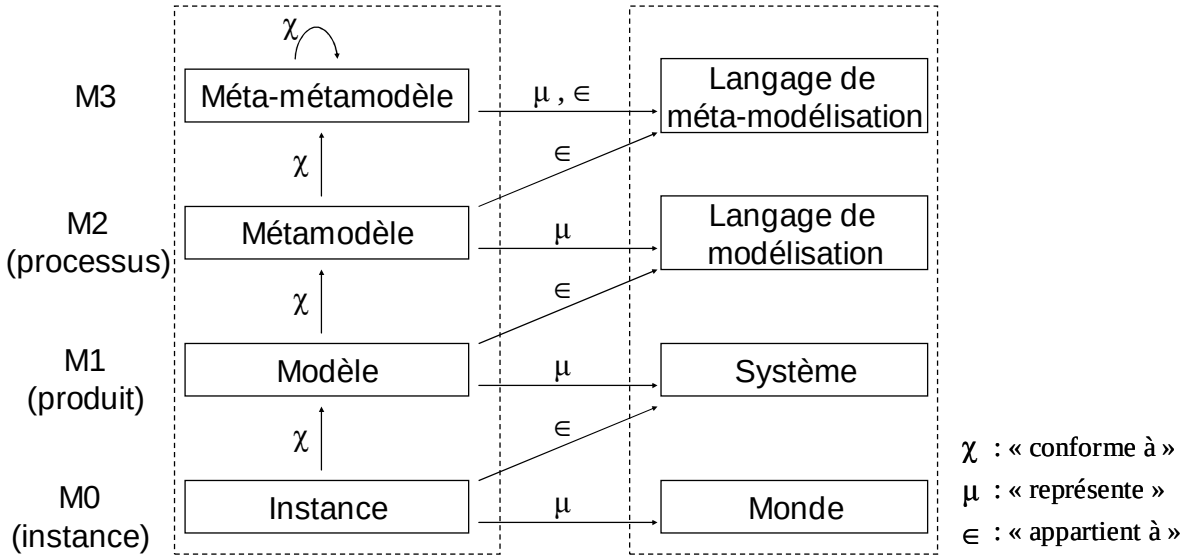


Figure 13 - L'architecture MDA à quatre niveaux de modélisation.

Des architectures similaires sont utilisées en ingénierie des exigences [95, 97, 134-135] pour représenter de manière synchrone le domaine des exigences à capturer (un métamodèle) d'une part, et la ou les spécification(s) d'exigences capturées (un modèle) d'autre part. Dans ces travaux, les niveaux de modélisation portent des noms variés. Nous prendrons ici la terminologie de Laamsweerde [98] car elle souligne bien le rôle de chacun des niveaux de modélisation dans un processus d'ingénierie des exigences : *niveau instance* pour M0, *niveau produit* pour M1 et *niveau processus* pour M2 (M1 est aussi appelé « niveau domaine » dans [97] par exemple, et M2 « niveau méta » dans [134]). Le niveau M3 n'est pas clairement défini comme dans le cas de l'architecture MDA. L'architecture MDA offre trois avantages :

- **Adaptabilité des formalismes de représentation des connaissances**. Dans les travaux en IE, le formalisme utilisé pour capturer les exigences est un métamodèle décrit au niveau M2

[38] Le terme « modèle » ne présume pas du niveau de modélisation en IDM. Néanmoins, on parle souvent de modèle pour désigner un artefact de niveau M1. Le terme est ambigu en IE car un modèle désigne tantôt un métamodèle, tantôt une spécification d'exigences (niveau produit).

[39] On peut comparer MOF à EBNF pour les grammaires ou OWL pour les ontologies.

[40] A la différence près que les notions d'expressions et de modèles sont ici confondues.

capturant les exigences d'un logiciel. Comme le souligne Mylopoulos, une architecture du type MDA rend le formalisme variable[41], ce qui est un mécanisme puissant pour l'évolution d'un outil de capture des exigences [95].

- **Manipulation uniforme des artefacts de modélisation IDM**. L'architecture MDA permet une manipulation homogène des modèles peu importe le niveau de modélisation auxquels ils appartiennent. Par exemple, un même code est applicable à la fois au MOF et à l'ensemble des métamodèles (ajout d'un patron de conception « Visiteur » par exemple).
- **Manipulation homogène de différents langages**. L'existence d'un niveau M3 est essentielle pour une description formelle de langages variés, et pour spécifier leurs relations sémantiques. L'architecture MDA joue un rôle important dans cette thèse, puisque nous nous intéressons à la composition de spécifications d'exigences dans un contexte multi-langage.

3 Transformation de modèles

MOF et OCL permettent de décrire la syntaxe abstraite des langages de modélisation ainsi qu'une partie de leur sémantique statique [136]. Le reste de la sémantique d'un langage est spécifié à l'aide de *transformation de modèles* ou simplement *transformation*. Dans cette section, nous définissons la notion de transformation en section 3.1 et présentons les principaux langages actuellement disponibles. Nous discutons ensuite en section 3.2 de la réalisation de la relation sémantique d'un langage formel dans le cadre conceptuel de l'IDM.

3.1 Définition et langages actuels

Comme illustré par la Figure 14, une transformation de modèles est un programme dont les valeurs d'entrées et de sorties sont des modèles. Une transformation accepte en entrée des modèles conformes à un ensemble de métamodèles sources et produit après exécution un ensemble de modèles conformes à un ensemble de métamodèles cibles. Les modèles sont rarement conformes à leurs métamodèles durant leur manipulation par une transformation. Ils le sont épisodiquement avant et après transformation. Pendant l'exécution, les contraintes structurelles violées sont les bornes inférieures des cardinalités ; les bornes supérieures ne sont jamais violées (cela rend inexécutable le code de la transformation). L'utilisation de transformation de modèles dans le cadre conceptuel de l'IDM est intéressante pour au moins quatre raisons :

- Les données traitées sont fortement typées (relation de conformité). Les métamodèles sources et cibles jouent le rôle de contrats (au sens de Meyer [24]), ce qui favorise la détection d'erreurs de programmation et offre une opportunité intéressante pour la validation des transformations [137][42].
- Les données traitées sont débarrassées des informations relatives à la syntaxe concrète des langages représentés par les métamodèles sources et cibles, ce qui limite la complexité du code.
- Il existe une variété de langages dédiés à la transformation de modèles. Ces langages favorisent la description de transformations complexes (limitation de la taille du code, meilleure structuration, instructions spécialisées …).
- Une transformation est elle-même un modèle conforme au métamodèle représentant un langage de transformation de modèles. Elle peut donc aussi être manipulée.

Comme tout programme, une transformation peut réaliser des traitements extrêmement variés comme le raffinement d'un modèle, la composition de modèles (voir section 4), la traduction d'un formalisme vers un autre (spécification d'une fonction sémantique, voir section 3.2). En pratique, on

[41] Mylopoulos ne cite pas le MDA car ce dernier a été défini à posteriori, mais ils parlent des mêmes principes.

[42] Les contraintes structurelles sont des abstractions du code de la transformation, ce qui permet une vérification par confrontation de niveaux d'abstractions (technique classique de vérification et de test).

choisit un langage de transformation en fonction de la nature de la transformation envisagée. Pour générer du code, on utilisera par exemple un langage de patrons. Pour réaliser une transformation structurelle simple, on privilégiera un langage déclaratif. Pour réaliser une transformation plus algorithmique, on préférera un langage généraliste impératif. Un grand nombre d'approches et de langages associés existent pour spécifier une transformation (voir les travaux de Czarnecki [138-139] pour une taxonomie complète). Les langages de transformation de modèles sont encore à l'état de recherche et les transformations sont encore peu utilisées à l'échelle industrielle. On distingue actuellement deux grandes approches : (i) les approches à base de règles (la majorité) et (ii) les approches opérationnelles.

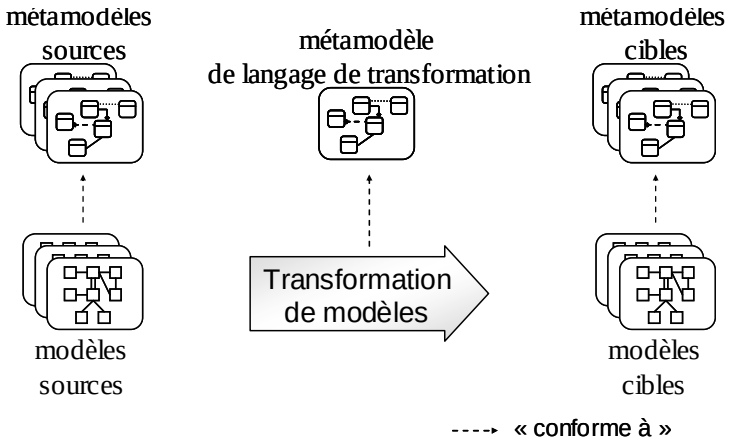


Figure 14 - Schématisation d'une transformation de modèles.

Dans une approche à base de règles (Tefkat [140], ATL [141], SmartQVT[43] [143], EML [144]), une transformation spécifie un ensemble de relations liant des fragments de modèles d'entrée à des fragments de modèles de sortie. Dans ce contexte, une règle est composée de deux parties : un *motif* et une *production*. Le motif spécifie les fragments de modèles sur lesquels la règle s'applique. La production spécifie les fragments produits lorsque la règle s'applique. Le code de la production est fonction des éléments du motif. Les approches à base de règles diffèrent suivant des critères tels que l'ordonnancement des règles (implicite, ordre total ou partiel) et la nature du code (déclaratif, impératif, hybride). L'approche opérationnelle (par exemple, Kermeta [145-146]) considère une transformation comme un ensemble d'actions définies directement au sein du métamodèle source. C'est une approche orientée objet, où le code de la transformation est structuré en fonction des éléments du métamodèle : chaque métaclasse est munie d'un ensemble d'opérations instanciées pour chacune de ses instances.

3.2 Spécification par transformation de modèles de la sémantique d'un langage

La sémantique d'un langage consiste en une relation sémantique entre une syntaxe abstraite et un domaine. Dans cette section, nous présentons les deux approches possibles pour décrire la sémantique d'un langage dans le cadre conceptuel de l'IDM. La première consiste en l'implémentation d'une transformation de la syntaxe abstraite du langage (un métamodèle) vers la syntaxe abstraite d'un langage formel où la sémantique dynamique est déjà implémentée par un outil (section 3.2.1). Dans ce cas, on réutilise un formalisme. La deuxième approche consiste en une implémentation de la sémantique dynamique au niveau de la syntaxe abstraite du langage (section 3.2.2).

3.2.1 Homéomorphisme

[43] SmartQVT est une implémentation de la norme QVT [142] (*Query View Transformation*) de l'OMG. Cette norme constitue un premier pas vers une uniformisation des approches à base de règles incluant trois langages complémentaires (comprenant des constructions déclaratives et impératives).

L'approche par *homéomorphisme* consiste en la spécification d'une transformation de modèles entre le métamodèle représentant le langage et un métamodèle représentany son domaine. Cette approche permet de réutiliser la sémantique d'un langage déjà existant (le domaine), ce qui correspond à la définition d'un langage donnée en section 1. Dans la littérature, ce type de transformation est parfois appelé projection, morphisme ou encore transformation homéomorphe [47, 116][44]. Elle est par exemple utilisé dans le cadre du projet PUML [147] visant à munir UML [50] d'une sémantique formelle définie sur le formalisme Z [128]. Il est alors possible de profiter conjointement de la syntaxe accessible de UML et des outils d'analyses formelles associés à Z. Les outils de vérification présentés en section 3.2.3 du Chapitre I implémentent aussi des homéomorphismes, réalisés de manière ad-hoc.

Les langages à base de règles sont les plus indiqués dans ce cas. Cette approche nécessite généralement la spécification d'un programme transformant une instance du métamodèle du domaine en un texte sémantiquement équivalent[45]. Ce texte est ensuite fourni à un outil implémentant la sémantique du domaine. Ce programme n'est pas nécessaire si l'outil a été implémenté en suivant une approche dirigée par les modèles. Dans ce cas, l'outil accepte directement le modèle du domaine produit par la transformation homéomorphe. La Figure 15 schématise les deux possibilités d'implémentation d'un homéomorphisme, suivant que l'outil réutilisé est implémenté avec des technologies IDM ou non.

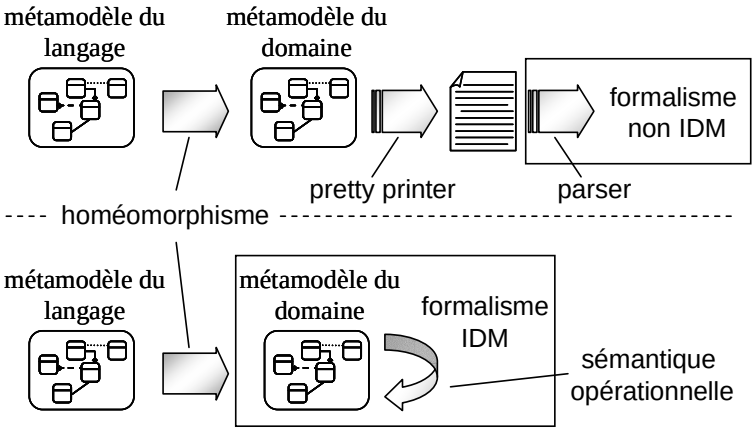


Figure 15 – Schématisation d'une spécification de la sémantique dynamique d'un langage par transformation homéomorphe.

3.2.2 Sémantique opérationnelle embarquée

La deuxième approche consiste en la spécification d'une *sémantique opérationnelle* au niveau du métamodèle du langage. Une sémantique opérationnelle est décrite par un ensemble d'opérations manipulant les concepts du métamodèle du langage. Ces opérations sont spécifiées à l'aide d'un langage d'action comme Kermeta [146]. Dans ce cas, le métamodèle du langage représente à la fois la syntaxe abstraite et le domaine sémantique du langage[46] (syntaxe et domaine se confondent). Cette approche est indiquée pour décrire la sémantique d'un formalisme (comme par exemple la sémantique du formalisme IDM de la Figure 15). La suite de cette section présente brièvement Kermeta et illustre la spécification d'une sémantique opérationnelle avec ce langage.

> **Définition** – ***sémantique opérationnelle*** : sémantique dynamique exécutable, décrite par un ensemble d'opérations associées aux métaclasses du métamodèle. Tout modèle conforme à un métamodèle muni d'une sémantique opérationnelle est exécutable.

Le langage d'action Kermeta [145] est une extension d'une version simplifiée de EMOF[47]. Kermeta permet ainsi de spécifier la sémantique opérationnelle de métamodèles conformes à EMOF.

---

[44] Mathématiquement, un homéomorphisme conserve la structure de l'information suivant les lois des langages vu comme des groupes (en particulier les lois de composition).

[45] Ce type de programme est appelée « pretty printer » en compilation (fonction inverse du « parsing »).

[46] La fonction sémantique est dans ce cas l'identité, c'est-à-dire que tout modèle est l'image de lui-même pas la fonction sémantique.

[47] EMOF (*Essential MOF*) est une version simplifier de MOF.

Le langage d'action est impératif, orienté objet et a été spécialement conçu pour intégrer un processus de développement dirigé par les modèles. Il fournit les constructions classiques d'un langage de programmation orienté objet (encapsulation, héritage …) et des constructions dédiées à la manipulation de modèles (constructions « à la OCL » comme la navigation, les opérations sur les collections telles *select* ou *collect*). Le langage d'action de Kermeta est particulièrement adapté aux activités de métamodélisation comme la spécification des langages de modélisation, la simulation et prototypage de métamodèle.

La Figure 16 fournit (a) un métamodèle possible de machine à état et (b) le même métamodèle muni d'une sémantique opérationnelle de simulation (ajout d'opérations et de relations). Ces opérations sont embarquées dans les différents concepts du métamodèle (par exemple *fire()* dans la métaclasse TRANSITION). L'association *currentState* entre les concepts FSM et STATE a été ajoutée pour maintenir en mémoire l'état courant d'une machine à état en cours de simulation. La Figure 17 fournit une implémentation en Kermeta de l'opération *step* embarquée par le concept STATE (cf. Figure 16). La spécification informelle de cette action est donnée ci-dessous :

- L'action *step* ne peut être appelée que sur l'état courant (voir la pré-condition).
- Si un évènement entrant satisfait l'événement d'entrée d'une transition de l'état courant, alors la transition est tirée et l'état courant mis à jour (soit la redirection du lien instance de *currentState* vers la cible TRANSITION.*target* de la transition tirée).
- Si plusieurs transitions sont satisfaites, la machine à état est indéterministe et une exception est levée.

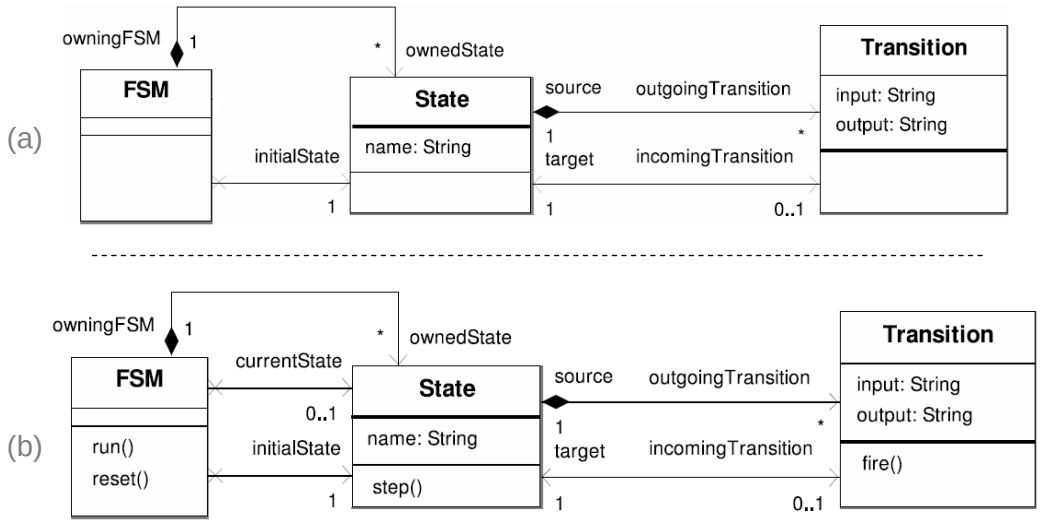


Figure 16 - Un métamodèle possible d'une machine à état (a) et le même métamodèle muni d'une sémantique opérationnelle embarquée de simulation (b).

```
operation step(c : String) : String
   pre owningFsm.currentState == self
   var validTransitions : Collection<Transition>
   validTransitions := outgoingTransition.select{t|t.input.equals(c)}
   if validTransitions.size > 1 then
      raise NonDeterminism.new
   end
   result := validTransitions.one.fire
end
```

Figure 17 - Code Kermeta pour l'action « step ».

Cette approche a plusieurs avantages. Tout modèle conforme au métamodèle de la Figure 16 (b) est exécutable par construction. Pour le modeleur (celui qui crée des machines à états dans l'exemple de la Figure 16), c'est un mécanisme intéressant pour la vérification des modèles, puisqu'un modèle ne se comportant pas correctement n'est pas valide structurellement (sauf erreur au niveau de la spécification de la sémantique opérationnelle). Pour le métamodeleur (celui qui conçoit la sémantique

opérationnelle), l'exécutabilité facilite la validation de la sémantique opérationnelle définie au niveau du métamodèle. Cependant, il est important de noter qu'une sémantique opérationnelle n'est pas toujours facile à spécifier de cette manière. Cela dépend de ce qui est représenté par le métamodèle. Lorsque le langage représenté est un formalisme, une sémantique opérationnelle est facile à définir comme une composition des concepts de sa syntaxe abstraite. Dans le cas contraire, il est plus intéressant de définir un homéomorphisme vers un métamodèle de la syntaxe abstraite du domaine (première approche), quitte à munir celui-ci dans un deuxième temps d'une sémantique opérationnelle.

4 Composition de modèles

La composition de modèles [40-46, 78] est un problème général, utile à un grand nombre de domaines informatiques. C'est une étape nécessaire si l'on désire analyser automatiquement une spécification décrite à l'aide d'une collection de spécifications partielles[48]. C'est en outre une technique supérieure aux approches de comparaison en ce qui concerne la détection d'incohérences [40]. La composition de modèles revêt plusieurs appellations, suivant le domaine en question et la nature de ce qui est composé : composition de points de vue, de vues, de spécifications partielles, intégration de vues, fusion, amalgamation ou encore tissage d'aspects. Nous préférerons dans la suite l'appellation *composition de modèles*, cette dernière étant plus générale car ne faisant aucune hypothèse sur ce qui est composé.

La section 4.1 introduit les principes généraux d'un processus de composition de spécifications partielles hétérogènes. Elle présente un processus de composition comme une transformation de modèles et définit un cadre conceptuel IDM pour la composition. La section 4.2 présente les travaux clés sur la composition de modèles dans les domaines de l'ingénierie des exigences et le domaine du développement logiciel par aspect (AOSD). Ces travaux sont décrits avec la terminologie proposée en sections 1 et 2, bien que la plupart ne soient pas construits avec des technologies IDM. Cette section n'offre pas une rétrospective complète. Néanmoins, les travaux présentés couvrent un large spectre et sont représentatifs des travaux plus anciens.

4.1 Principes généraux

La composition de modèles a pour but l'obtention d'un *modèle global*. Ce modèle global capture une partie des informations initialement dispersées dans une collection de *modèles d'entrée*. Il représente la spécification opérationnelle obtenue après composition et sert de modèle d'entrée aux outils d'analyse une fois exempt d'incohérences statiques (cf. section 3 du Chapitre I). Un modèle d'entrée est conforme à un *métamodèle d'entrée* représentant la syntaxe abstraite d'un *langage d'entrée*. Le modèle global est instance du *métamodèle cœur* représentant la syntaxe abstraite d'un *formalisme*[49] (langage cible du processus de composition). Le formalisme spécifie le domaine sémantique choisi pour la composition. Le RM présenté dans le Chapitre IV est un exemple de formalisme pour un processus de composition, tout comme KAOS [97], Telos [95], RML [93-94] ou encore Albert II [96].

Définition – *langage d'entrée (et métamodèle d'entrée)* : langage dont les expressions peuvent être acceptées en entrée d'un processus de composition. C'est un langage de modélisation, défini par un métamodèle (sa syntaxe abstraite) et une fonction sémantique définie sur le formalisme. Un langage d'entrée est représenté par un métamodèle d'entrée. Un modèle d'entrée est une instance d'un métamodèle d'entrée.

Définition – *formalisme (et métamodèle cœur)* : Langage dont le résultat d'un processus de composition est une expression. C'est un langage muni d'une sémantique formelle. Le formalisme est

[48] La décomposition d'un problème en des problèmes plus simple à résoudre permet de mieux maîtriser sa complexité (voir la section 2.1 du Chapitre I). Cependant, les solutions à ces problèmes doivent être composées pour obtenir une solution globale au problème initial.

[49] Le terme de formalisme souligne bien le caractère formel et outillé du langage cible.

défini par un métamodèle appelé métamodèle cœur (du processus de composition). Le métamodèle coeur représente la syntaxe abstraite du formalisme. Un formalisme permet la résolution d'une classe de problèmes (via un ensemble de techniques d'analyse formelle). Le choix d'un formalisme détermine cette classe de problèmes (et les techniques d'analyses pouvant être appliquées en aval du processus de composition).

Définition – *modèle global* : résultat d'un processus de composition, expression du formalisme et instance du métamodèle coeur.

La composition de modèle partage avec les approches de comparaisons structurelles (cf. section 3.2.2 du Chapitre I) le besoin d'identifier les *superpositions sémantiques* entre modèles d'entrée. Ces superpositions reflètent des *redondances* dans la représentation fragmentée de l'information. Pour obtenir un modèle global, ces redondances doivent être résolues de sorte que toute information n'est représentée qu'une et une seule fois. La différence entre composition et comparaison réside dans la nécessité pour la première de résoudre les superpositions sémantiques identifiées. Nous donnons ci-dessous une définition de la notion de superposition sémantique[50] ainsi qu'une définition de la notion de *résolution d'une superposition sémantique*.

Définition – *Superposition sémantique* : Une superposition sémantique est une situation où un ensemble de fragments de modèles représentent une même information. Une superposition reflète une redondance d'information.

Définition – *Résolution (d'une superposition sémantique)* : La résolution d'une superposition sémantique est une transformation homéomorphe de l'information visant à supprimer la redondance d'information reflètée par cette superposition. L'information après résolution est sémantiquement équivalente à l'information avant résolution ; seule la syntaxe est modifiée.

Certains travaux fournissent des mécanismes semi-automatiques pour la détection de superpositions sémantiques (par exemple Theme [44] et Kompose [78]) alors que d'autres considèrent cette activité comme entièrement manuelle (le travail de Sabetzadeh [40] par exemple). Ces mécanismes sont sujets à caution à cause des situations de collisions [105] (voir définition ci-dessous). Les collisions sont le résultat de divergences conceptuelles entre parties prenantes (cas des collisions structurelles), ou plus simplement de différence dans le choix des termes employés pour désigner les éléments du domaine (collisions terminoloiques et dénotationnelles). Quoi qu'il soit, il convient de garder à l'esprit que la détection de superpositions sémantique ne peut être entièrement automatisée puisqu'une même information peut être représentée de différentes manières.

Définition – *Collision* : Une collision est une situation responsable d'une erreur lors de la détection des superpositions sémantiques. On distingue trois types de collisions :

Collision terminologique : Situation où deux termes distincts désignent le même élément du domaine.

Collision dénotationnelle : Situation où un même terme désigne deux éléments distincts du domaine.

Collision structurelle : Situation où une même information est représentée par des fragments de modèle syntaxiquement différents. Cette notion est une généralisation de la notion de collision terminologique.

Pamela Zave et Mickael Jackson ont introduit dans [41] les principes et problèmes principaux de la composition de modèles, quels que soient les langages d'entrée utilisés (composition multi-formalisme). Zave et Jackson définissent la composition comme la conjonction logique des modèles d'entrée, préalablement traduits dans un même formalisme logique (le domaine sémantique). La traduction des modèles à composer vers le domaine sémantique est réalisée par des fonctions (appelées *fonctions d'interprétations*) mettant en relation la syntaxe des langages d'entrée et le domaine sémantique (ce sont donc des relations sémantiques dans notre terminologie IDM). Zave insiste sur le fait que ces fonctions peuvent être variables dès lors que le langage concerné est semi-formel. En effet, celui-ci peut comporter des variations sémantiques (voir section 1 du Chapitre II).

La Figure 18 présente un processus de composition comme une transformation de modèles. Cette transformation réalise les relations sémantiques des langages d'entrée (nous n'utilisons pas le terme de

[50] Cette définition est plus généraliste que celle donnée en section 3.2.2 du Chapitre I.

« fonctions sémantiques » car un langage d'entrée n'est pas nécessairement formel). Cette figure respecte la terminologie définie plus haut. On retrouve les deux manières de réaliser la sémantique d'un langage en IDM (cf. section 1.3 du Chapitre IV) : (i) à l'aide d'une transformation homéomorphe vers un métamodèle représentant un domaine ou (ii) à l'aide d'une sémantique opérationnelle embarquée. Ce dernier cas est symbolisé par une fonction sémantique dont la source et la cible sont identiques (le formalisme est alors son propre domaine). La représentation de la sémantique des langages d'entrée est conforme au premier cas alors que celle du formalisme peut être conforme à l'un ou l'autre.

Notation – *processus de composition φ* : On notera φ un processus de composition. C'est une transformation acceptant en entrée un ensemble de modèles d'entrée conformes aux métamodèles d'entrée du processus et produisant en sortie le modèle global.

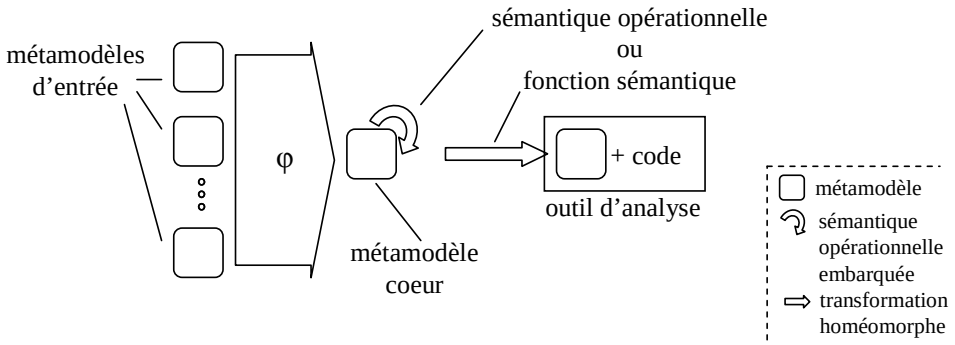


Figure 18 – Un processus de composition vu comme une transformation de modèles.

4.2 Approches de composition de modèles

Seuls Pamela Zave et Mickael Jackson considèr la problématique de composition de modèles dans le cas général [41], c'est-à-dire dans un cadre multi-langage. Ils discutent des principes et problèmes principaux de la composition de modèles, quels que soient les langages d'entrée utilisés (la Figure 18 de la section précédante est conforme à leur vision d'un processus de composition). Cependant, le travail de Zave et Jackson n'apporte pas de solutions concrètes à la composition de modèles. En particulier, il n'aborde pas le problème de résolution des superpositions sémantiques (suppression des redondances). Les travaux présentés dans la suite proposent des solutions de composition de modèles instances d'un même langage d'entrée.

Ainsworth propose [42] une méthode de composition de spécifications partielles exprimées en Z [128]. Cette approche est donc mono-langage et ne requiert pas de traduction préalable des modèles vers le domaine sémantique. Les relations entre les spécifications à composer sont décrites manuellement par des invariants fonctionnels et des obligations de preuves peuvent être menées à bien pour vérifier l'exactitude de la composition. Cette approche est plus une méthode qu'un mécanisme automatique de composition. L'approche consiste en trois étapes : la première vise à identifier les superpositions sémantiques et restructurer les spécifications pour faciliter la composition ; la deuxième a pour but de déceler et résoudre les collisions (cf. section Chapitre I3.1) éventuelles entre spécifications ; la dernière étape est l'étape de composition des spécifications.

Dans [43], les auteurs décrivent une plate-forme pour produire une spécification HOL [148] (pour *high order logic*) à partir de modèles décrits dans différentes notations (machines à états, tables de décisions et HOL). La spécification résultante peut ensuite être analysée avec des outils dédiés à HOL. La sémantique d'un langage d'entrée est définie à l'aide de fonction HOL. Il est alors possible de combiner ces fonctions pour décrire la sémantique d'un nouveau langage d'entrée que l'on désire ajouter à la plate-forme. L'intérêt de cette approche réside en la possibilité d'ajouter de nouveaux langages d'entrée en réutilisant des parties de la sémantique des langages déjà intégrés. Cependant, les auteurs ne précisent pas comment traduire une expression d'un langage d'entrée en une spécification HOL. Comme [42], cette approche est donc fastidieuse car demandant un traitement manuel des modèles avant composition.

Sabetzadeh et Easterbrook présentent une approche de composition basée sur le concept de co-limite de la théorie des catégories [40]. L'intérêt de cette approche est le fait qu'un modèle global peut être obtenu même lorsque les modèles composés sont statiquement incohérents, ce qui est de première importance lorsque les modèles capturent des exigences. Cette approche est mono-langage et est illustrée avec le langage i* [129]. Les modèles à composer sont représentés par des graphes orientés. L'opérateur de composition (co-limite) est formellement défini et des liens de traçabilité entre les modèles d'entrée et le modèle global résultant sont automatiquement inférés. La résolution des superpositions sémantiques est automatique mais les superpositions sémantiques doivent être identifiées manuellement par l'analyste.

On trouve dans le domaine de l'AOSD (cf. section 2.1 du Chapitre I2.1) des travaux portant sur la composition de modèles, puisque la décomposition d'une information en une collection d'aspects (pour mieux appréhender un problème) nécessite ensuite leur composition (pour résoudre le problème). Certains travaux [44-46] à la frontière entre les domaines de l'ingénierie des exigences et l'AOSD décrivent des mécanismes de composition. Dans le cas de [45-46], ces mécanismes sont largement informels, car la notion de superposition est réduite à l'égalité entre des annotations associées manuellement aux exigences dans le but de les classer plus finement que ce que permet l'approche classique par points de vue. Deux travaux issus de l'AOSD sont toutefois intéressants dans le contexte de cette section: Theme [44] et Kompose [78].

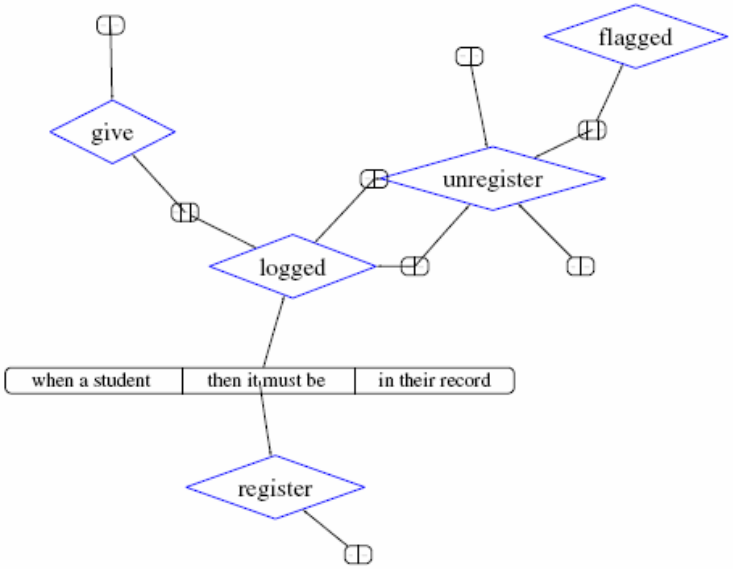


Figure 19 - Une vue d'actions de l'outil Theme/Doc (extrait de [44]).

Theme [44] est une approche de développement par aspects s'appliquant aux étapes d'analyse des exigences et de conception. Au niveau des exigences, l'approche s'intéresse plus particulièrement aux exigences fonctionnelles. Elle vise à étudier les relations entre exigences. Une exigence fonctionnelle est identifiée comme aspect potentiel si elle est sémantiquement liée à un grand nombre d'autres exigences. Pour ce faire, les exigences sont décrites en langage naturel et l'utilisateur doit fournir un ensemble de mots clés représentant des concepts du domaine et apparaissant dans le texte de la spécification. Une fois les mots clés fournis, Theme/Doc (l'outil associé à l'approche) produit un graphe représentant les liens sémantiques entre les exigences en fonction de l'occurrence des mots clés dans celles-ci. La Figure 19 présente un extrait du graphe obtenu pour une spécification d'exigences d'un système de gestion de cours universitaires. Dans cette figure, le graphe est centré sur l'exigence "When a student registers then it must be logged in their record". Ce mécanisme de composition ne produit pas un modèle précis de la sémantique de la spécification d'exigences mais offre néanmoins une vue fonctionnelle utile à l'analyse, en particulier la détection d'incohérences de type collisions (cf. section Chapitre I3.1).

Kompose [78] est un outil de composition d'aspects au niveau conception. C'est une extension du langage de métamodélisation Kermeta [146] (cf. section 3.1). Kompose est une approche automatique de composition. Les aspects sont représentés par des métamodèles. Pour les composer, des règles sont

définies au niveau des métaclasses du MOF. Ces règles identifient les superpositions sémantiques entre métaclasses des métamodèles à composer. Elles spécifient aussi comment résoudre ces superpositions. A titre d'exemple, la règle associée à la classe ATTRIBUTE du MOF spécifie que deux attributs sont égaux s'ils partagent le même nom et sont attributs de la même métaclasse. La Figure 20 présente le résultat (c) de la composition de deux modèles (a) et (b) par Kompose.

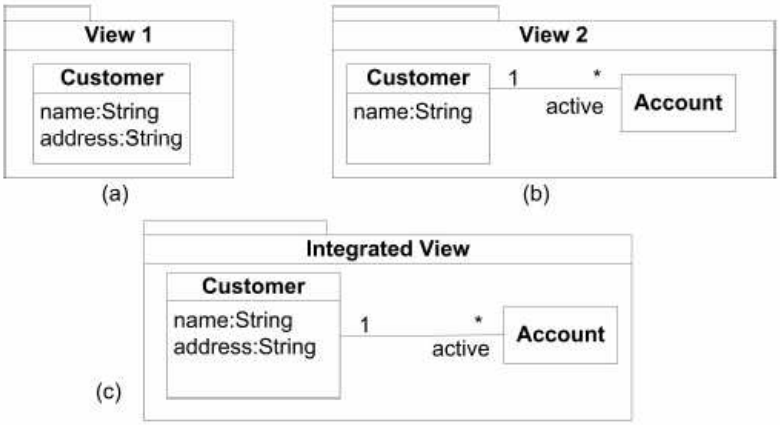


Figure 20 - deux modèles (a) et (c) et le résultat de leur composition (c) par Kompose (extrait de [78]).

Kompose est une approche entièrement automatique. Contrairement à [40], l'identification des superpositions n'est pas manuelle, ce qui pose un problème si les modèles à composer comportent des collisions. Pour pallier ce problème, Kompose propose un langage simple de pré- et post-directives de composition. Les pré-directives permettent de modifier les métamodèles à composer pour supprimer les collisions éventuelles. Les post-directives permettent de modifier le résultat de la composition après composition. Ces directives sont spécifiques aux métamodèles composés.

5 Synthèse

Ce chapitre propose un cadre conceptuel de métamodélisation pour la manipulation de spécifications partielles et hétérogènes d'exigences. Ce cadre conceptuel se base sur les normes IDM pour l'essentiel. Il est utilisé dans le reste de ce manuscrit. Concernant la composition de modèles, nous avons vu qu'il n'existe pas de solution dans le cas général. Les solutions actuelles sont spécifiques à un seul langage d'entrée et sont pour la plupart très peu automatisées. Le langage d'entrée est toujours formel et les problèmes liés à l'ambigüité des spécifications ne sont pas traités. Enfin, seuls Sabetzadeh et Easterbrook [40] proposent un processus de composition de modèles à priori incohérents, ce qui est de première importance si l'on désire composer des exigences.

Chapitre III Problématiques, plate-forme de recherches et spécifications étudiées

Résumé

La privatisation progressive des grands groupes télécoms et leur mise en concurrence au niveau international a profondément bouleversé les stratégies de développement logiciel au sein du groupe France Télécom. L'externalisation progressive des activités d'implémentations pour ne garder in situ que les étapes d'analyses amont et les étapes de validation requièrent une meilleure maîtrise de la qualité du cahier des charges en tant que document contractuel. Pour ce faire, France Télécom s'intéresse à la mise en place d'un processus IE intégrant les techniques de vérifications et valisations (V&V) modernes afin de mieux fiabiliser les exigences de ses produits (services télécoms pour la LiveBox, mais aussi projets R&D). L'objectif de cette thèse est la définition d'une approche de fiabilisation des exigences afin d'en faire des artefacts de développement productif. Le but a terme est la production d'une plate-forme fédérant les activités V&V au niveau exigences et les activités de développement aval liées aux exigences (production d'une spécification logicielle, test système …).

Les deux chapitres précédant fournissent un état de l'art des travaux en génie logiciel utiles pour la mise en place d'un tel processus. Le Chapitre I traite de la capture précise des exigences afin de pouvoir mesurer la cohérence d'une spécification d'exigences. Le Chapitre II offre un cadre conceptuel de métamodélisation adapté (i) à la manipulation d'un ensemble de spécifications partielles, hétérogènes et potentiellement incohérentes, (ii) à la composition de modèles dans son acceptation générale. Ce chapitre a pour but de présenter conjointement les problématiques abordées, l'approche proposée et les spécifications d'exigences étudiées. Il présente la plate-forme R2A, prototype de recherches réalisant notre approche. Cette plate-forme a été développée durant cette thèse et implémente les contributions présentées dans la deuxième partie de ce manuscrit.

La section 1 discute des raisons expliquant le manque de transfert technologique vers l'industrie des technologies issus des travaux en IE. Quatre besoins sont identifiés, puis raffinés en trois objectifs. La section 2 présente l'approche proposée. Elle présente aussi la plate-forme R2A et ses fonctionnalités. La section 3 propose une spécification informelle des exigences d'un système de réunion virtuelle. Cette spécification est analysée afin de concrétiser une partie des notions présentées dans les deux premiers chapitres. Elle est utilisée comme exemple illustratif tout au long de la deuxième partie de ce manuscrit.

Ce chapitre clôt la première partie du manuscrit. La deuxième partie débute par une introduction générale des contributions. Cette dernière fait le lien entre les objectifs identifiés dans ce chapitre et les contributions présentées plus loin. En particulier, elle positionne les contributions dans le contexte de l'approche proposée et plus concrètement au sein de la plate-forme R2A.

1 Problématique

Fiabiliser les exigences est une tâche ardue. Une spécification d'exigences est une collection de spécifications partielles (quelle que soit la ou les technique(s) de décomposition utilisée(s)). Ces spécifications sont initialement informelles et donc non analysables. Un processus IE vise leurs raffinements jusqu'à obtention d'une spécification opérationnelle, nécessaire à leur analyse formelle. Ce travail est effectué en collaboration avec l'ensemble des parties prenantes, et supportée par des méthodes d'identification des exigences et l'utilisation de technologies V&V. Seules les informations analysables par ces technologies sont formalisées. Le processus IE se termine lorsque la spécification est considérée suffisamment cohérente pour servir de point d'entrée aux phases de développement aval (en premier lieu pour la production d'une spécification logicielle satisfaisant les exigences ainsi capturées). Ce processus est schématisé par la Figure 21.

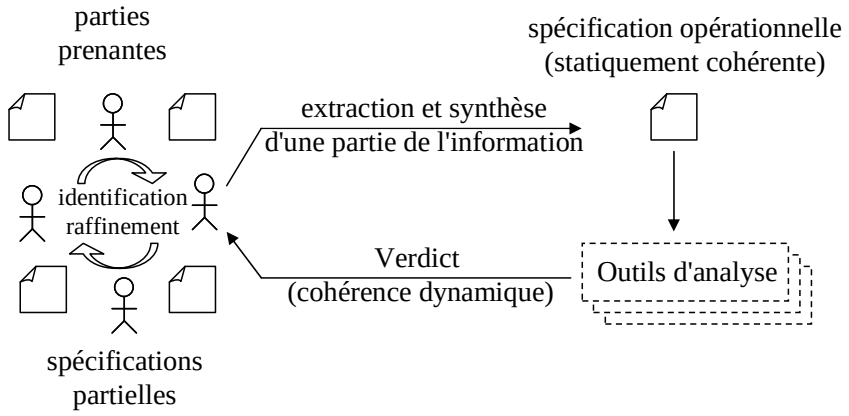


Figure 21 - Processus classique de fiabilisation des exigences.

Bien que les technologies d'analyses soient actuellement matures, plusieurs travaux dans la littérature font état d'un manque de transfert dans l'industrie [7-10]. Ce manque de transfert peut être imputable aux coûts de mise en place et de fonctionnement d'un tel processus. En effet, la production d'exigences opérationnelles, l'utilisation d'outils d'analyse, la gestion de spécification de grandes tailles tout au long du cycle de vie du logiciel et l'emploi de personnels experts a un coût. Ce manque de transfert peut aussi provenir du peu de confiance des industriels quant aux gains escomptés (amélioration de la productivité et de la qualité des logiciels produits). Cette raison est compréhensible car la mise en place d'un processus d'ingénierie des exigences est complexe. Une telle mise en place peut avoir, comme toute modification significative d'un processus de production, des répercussions négatives durant une période de transition difficile à évaluer. Il existe cependant des raisons plus techniques freinant ce transfert de technologies. Nous en avons identifié au moins quatre. La liste ci-dessous les énumère et fait apparaître quatre besoins :

La difficulté d'obtenir un modèle global des exigences. Les outils d'analyse n'acceptent en entrée qu'un modèle global des exigences représentant la spécification opérationnelle [40][51]. Ce modèle doit capturer les informations initialement dispersées dans une collection de spécifications partielles. La production manuelle d'un tel modèle est un travail fastidieux et coûteux (synthèse de spécifications formelles). En outre, ce modèle doit être statiquement cohérent. Un modèle n'est simulable que s'il est conforme au langage accepté par l'outil visé. Cette cohérence statique peut être évaluée par les techniques de comparaison structurelle [33-38]. Néanmoins, ces techniques ne permettent pas la détection d'un manque d'information, et encore moins la production d'un modèle global.

⇨ ***1er besoin : obtention d'un modèle global des exigences par composition de spécifications partielles.***

[51] Ou deux modèles (un modèle des exigences fonctionnelles et un modèle des exigences non-fonctionnelles), suivant les outils.

La nature des langages d'entrée des outils. Les langages acceptés par les outils d'analyse sont des formalismes[52], ce qui limite la participation de la majorité des parties prenantes à une description informelle des exigences [47]. Il devient dès lors difficile de valider les résultats de l'analyse (et par conséquent d'affirmer que les exigences reflètent correctement les besoins), d'autant plus que le raffinement des spécifications n'est effectué que par les analystes et est source d'erreurs. Les approches à base de patrons de spécification et de langages naturels contraints [8, 52-53, 149], ou les travaux de formalisation de langages semi-formels graphiques comme UML [147, 150] tendent à pallier ce problème. Cependant, ces approches imposent tout comme les outils d'analyse un ensemble de langages d'entrée (rarement plus de deux) difficilement modifiables selon les besoins des parties prenantes d'un contexte industriel particulier.

⇨ ***2ème besoin : flexibilité des langages de modélisation en entrée d'un processus IE.***

La diversité des outils et leur non-interopérabilité. La mise en place d'un processus d'ingénierie IE implique le choix préalable d'une batterie d'outils adaptés à la nature du projet logiciel et au degré d'automatisation escompté. Chaque outil est adapté à l'analyse de certains aspects d'une spécification d'exigences et nécessite une spécification exprimée dans un ensemble imposé de langages. Ces outils sont généralement implémentés de manière ad hoc, et ne sont pas interopérables. De plus, leurs langages d'entrée sont généralement différents, ce qui implique de multiples traductions si l'on désire utiliser une batterie d'outils. Cette situation réduit fortement l'intérêt des outils d'analyse et l'adaptabilité des processus IE pour l'industrie. Au travail fastidieux de synthèse exposé précédemment s'ajoute un travail de traduction coûteux.

⇨ ***3ème besoin : intégration et découplage des outils d'analyse et de leurs formalismes au sein d'un environnement homogène.***

La séparation stricte entre le processus d'ingénierie des exigences et le processus de développement logiciel. Les processus d'ingénierie des exigences et de développement logiciel sont dans la pratique deux activités bien distinctes. Comme l'a souligné Sommerville [30], cette séparation n'est plus tenable face aux contraintes de temps imposées par le marché et les exigences évoluent durant les phases de développement aval. Le maintien des liens de raffinement entre exigences et artefacts logiciels (jeux de test, spécification logicielle, COTS utilisés lors de développement par configuration …) est un enjeu crucial. Certains travaux sur les aspects au niveau exigences visent la production et la gestion de liens de traçabilité entre exigences et artefacts de développement aval [44-46, 79, 151]. Cependant, ces liens restent très informels et ne sont pas suffisants pour propager ne serait-ce qu'en partie l'évolution des exigences vers les artefacts logiciels. Il n'existe donc pas de continuité entre les exigences et les spécifications logicielles comme il en existe entre PIM et PSM dans le cadre d'un processus de développement dirigé par les modèles.

⇨ ***4ème besoin : continuité (même partielle) entre exigences et artefacts de développement aval.***

Nous avons identifié trois objectifs participant à la satisfaction des besoins cités plus haut. Nous pensons que la réalisation de ces objectifs favorise le transfert des technologies IE dans l'industrie et par conséquent la fiabilisation des exigences. Ces objectifs sont listés ci-dessous :

- ***Offrir un mécanisme d'import de spécifications d'exigences hétérogènes produisant un modèle global d'exigences et adaptable à des contextes industriels variés (1er et 2ème besoins)***. Cet objectif peut être raffiné en deux sous-objectifs : (i) synthèse d'un ensemble d'information hétérogènes et (ii) variabilité des langages d'entrée proposés pour la description des exigences.

[52] Rappelons que la syntaxe d'un formalisme n'a pas pour but d'être accessible à des non mathématiciens. Au contraire, elle reflète de manière précise et concise la logique représentée.

- ***Offrir une plate-forme d'ingénierie des exigences intégrant une batterie variable d'outils d'analyse des exigences (3ème besoin)***. Cet objectif concerne la variabilité des outils d'analyse. Il vise à faciliter la création de plates-formes d'ingénierie des exigences spécialisées pour des contextes industriels et projets logiciels différents. Cet objectif est lui aussi décomposable en deux sous-objectifs : (i) faciliter au niveau conception l'identification des artefacts relatifs à une préoccupation d'analyse particulière et (ii) faciliter l'évolution du formalisme de la plate-forme, puisque ce dernier dépend de la nature des outils d'analyses supportés.
- ***Coupler les exigences opérationnelles aux artefacts de développement logiciel (4ème besoin)***. Cet objectif concerne la synchronisation des phases amont et aval du développement logiciel. Plus précisément, l'idée est de rendre les exigences productives au sein du processus de développement logiciel comme les artefacts de conception tendent à l'être dans un contexte d'ingénierie dirigée par les modèles.

2 Approche proposée et réalisation concrète

Le projet de la plate-forme R2A est né dans le cadre du projet collaboratif MUTATION (pour Modélisation UML pour l'automatisation de la production de test), associant en particulier l'équipe Triskell de l'IRISA et TAS (Thalès Airborne System). Ce projet fait partie intégrante d'un projet de recherche plus large (CARROLL). L'objectif du projet MUTATION était le transfert de méthodes académiques de génération automatique de tests (voir les travaux de Clémentine Nebut [55, 152]) dans le cadre du processus de développement pour les systèmes embarqués de grande échelle de TAS. La plate-forme a ensuite été étendue pour la modélisation de services télécoms à France Télécom (voir le site de la plate-forme R2A pour plus de détails [153]) dans le but de fiabiliser les spécifications d'exigences produites, générer des tests système et extraire des ébauches de spécifications système.

Cette section présente d'abord en section 0 l'approche de développement logiciel que nous proposons. Elle vise à fiabiliser les spécifications d'exigences et plus généralement à améliorer la qualité des logiciels produits, dans un contexte où les exigences restent les seuls artefacts de développement entièrement maîtrisés par les équipes de développement FT. La section 0 décrit ensuite la version actuelle de la plate-forme R2A, réalisation concrète de l'approche.

2.1 Description générale de l'approche

La Figure 1 présente une vue générale de notre approche au sein d'un processus de développement. Cette approche est généraliste et peut être appliquée dans des contextes industriels très variés. Elle est indépendante des méthodologies de classement choisies. Nous avons opté pour un classement par points de vue pour son application chez France Télécom (voir Chapitre V). Notons que la mise en place d'une autre méthodologie nécessite une modification du formalisme, puisque ce dernier capture les concepts de la méthodologie retenue (voir la vue classification du formalisme RM en Annexe E). Dans tous les cas, il existe des superpositions sémantiques implicites entre spécifications partielles. Ces liens sont détectés par le processus de composition comme nous le verrons en Chapitre VI. L'approche est aussi indépendante du modèle de processus de développement choisi (le « cycle en V » par exemple, comme dans la Figure 1). Enfin, elle est indépendante des méthodes et techniques utilisées durant les étapes préliminaires d'analyse du domaine et d'identification des objectifs (cf. Annexe B).

L'approche est structurée autour d'un modèle global des exigences obtenu par composition de spécifications partielles. Les spécifications partielles produites durant les phases de développement amont représentent les exigences identifiées par les parties prenantes. Ces spécifications partielles sont spécifiées à l'aide d'un ensemble de langages appelés *langages d'entrée*[53]. Elles représentent uniquement les informations nécessaires à l'application des outils d'analyse visés. Les autres

[53] Cette figure ne présente que les artefacts de développement, c'est-à-dire des expressions. Les langages (langages d'entrée et formalisme) ne sont pas présentés.

informations restent informelles. Toutes sont consignées dans le cahier des charges. Le modèle résultat de la composition des spécifications partielles est une instance d'un formalisme représenté par un métamodèle dit *métamodèle cœur*. Une fois le modèle global obtenu, les outils embarqués dans la plate-forme peuvent être utilisés. La Figure 1 classe ces outils dans trois grandes catégories : outils V&V, outils d'extraction d'informations (pour la génération d'artefacts de développement aval par exemple[54]) et outils de génération de tests système. Enfin, l'approche suppose le maintien de liens de traçabilité entre les spécifications informelles, les spécifications partielles produites par les parties prenantes et le modèle global obtenu par composition.

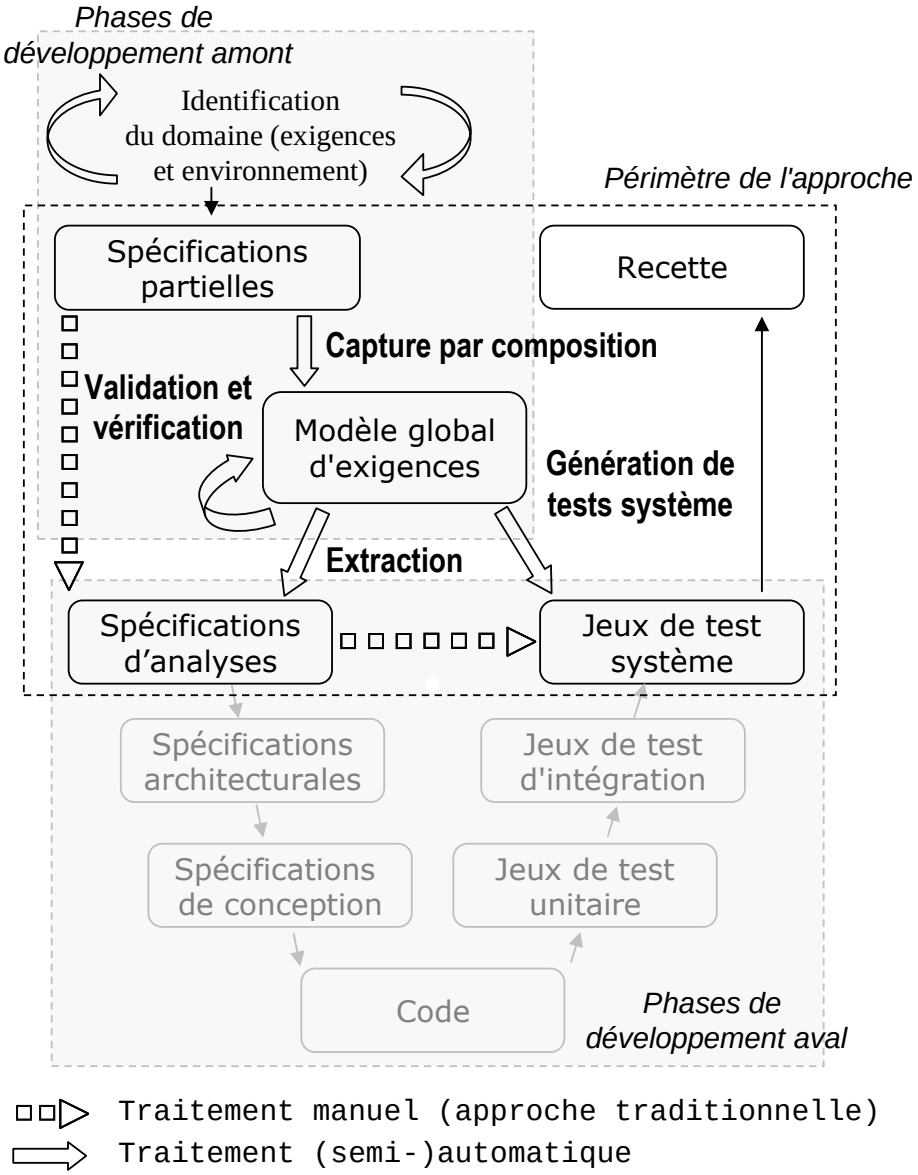


Figure 22 – Vue générale de l'approche proposée.

Le métamodèle cœur doit être défini par les ingénieurs en exigence avec le plus grand soin. Le formalisme doit en effet être en mesure de capturer les préoccupations nécessaires à l'application des outils d'analyse visés. Pour ce faire, les *ingénieurs processus* identifient en premier lieu les activités du processus d'ingénierie des exigences que les responsables du projet logiciel souhaitent automatiser. Cette décision dépend en grande partie de la qualité du processus de développement exigée par le(s) client(s) (et le budget qu'ils souhaitent y consacrer …). Les ingénieurs processus choisissent ensuite les outils d'analyse adéquats, ce qui détermine les préoccupations à capturer. Enfin, ils choisissent en

[54] Les artefacts produits sont évidemment partiels puisque l'implémentation d'un logiciel est une succession de raffinements (donc d'ajouts d'informations).

concertation avec les parties prenantes les langages d'entrée de sorte que ces préoccupations puissent être exprimées.

L'approche proposée est entièrement dirigée par les modèles. Les raisons de ce choix sont les suivantes :

- ***Evolutivité des langages***. Une approche par métamodélisation permet une grande souplesse des langages d'entrée comme du formalisme. Plusieurs travaux concernés par la modélisation des exigences adoptent une approche par métamodélisation et font état de ses avantages (RML [93-94], Telos [95]) : ajout de nouveaux concepts, de nouvelles contraintes, ajout de nouvelles préoccupations à capturer sans modifier en profondeur les transformations déjà définies sur le langage existant. En outre, le développement à l'aide de métamodèles et de transformation favorise une meilleure encapsulation des préoccupations implémentées (fonctionnalités et types d'informations manipulées).
- ***Compatibilité avec les technologies IDM***. Les technologies IDM prônent l'utilisation des modèles, métamodèles et des transformations de modèle pour automatiser au mieux le processus de développement logiciel. La plate-forme doit être compatible avec ce type de technologie et produire des artefacts IDM. De cette manière, les techniques IDM définies pour les phases de développement aval peuvent être directement appliquées (homogénéité du cadre conceptuel).

2.2 La plate-forme R2A : une plate-forme de recherche dirigée par les modèles

La plate-forme R2A a été entièrement développée à l'aide de technologies MDE (EMF [154], Kermeta [146]) pour les besoins de FRANCE TELECOM. En tant que tel, c'est un prototype de recherche et une étude de cas pour l'application systématique de méthodes de conception dirigée par les modèles. C'est aussi une opportunité pour la mise à l'épreuve des outils de l'IDM en cours de maturation comme Kermeta. L'intérêt d'une conception de ce type réside dans la flexibilité de l'architecture de la plate-forme. Cette flexibilité facilite l'adaptabilité de la plate-forme à différents contextes industriels. Enfin, la plate-forme R2A vise à étendre l'approche de développement dirigée par les modèles type MDA aux premières étapes de développement logiciel, durant le processus d'ingénierie logicielle.

La plate-forme R2A (Requirements To Analysis) réalise concrètement l'approche proposée plus haut au sein d'un même environnement technologique. La version courante est spécialisée pour le contexte industriel de France Télécom et a été entièrement développée à l'aide de technologies MDE (voir l'architecture en Annexe D). Elle supporte actuellement la simulation fonctionnelle des exigences (validation), la détection d'incohérences (vérification), l'aide au diagnostic via la visualisation des modèles d'exigences (validation), ainsi que la gestion des liens de traçabilité précis entre exigences et artefacts de développement. Utilisés conjointement, ces fonctionnalités offrent un environnement semi-automatisé pour un processus IE.

La Figure 23 donne une vue globale des fonctionnalités de la plate-forme R2A. Le formalisme RM forme le cœur de la plate-forme (voir le Chapitre IV), dont le modèle global d'exigences est une instance. La plate-forme comporte aussi des langages d'entrée et de sortie de la plate-forme ainsi qu'un ensemble de transformation liant ces langages au formalisme. La transformation d'import est particulière car elle réalise la composition des spécifications partielles ; les autres produisent différents artefacts de développement logiciel. Ces transformations réalisent les fonctionnalités de la plate-forme. Elles peuvent être adaptée et de nouvelles peuvent être implémentées. Les fonctionnalités de la plate-forme dans son état actuel sont :

(1) Import de spécifications partielles. L'import des spécifications partielles d'exigences est réalisé par le mécanisme de composition présenté dans le Chapitre VI. Ce mécanisme est multi-langage. Actuellement, les langages d'entrée de la plate-forme sont :

- Des expressions RDL (voir Chapitre IV),

- Des expressions logiques textuelles RM (voir Chapitre IV),
- Des diagrammes de classes UML,
- Des diagrammes d'activités UML.

(2) Simulation fonctionnelle des exigences. La plate-forme offre un mécanisme de simulation fonctionnelle des exigences. La simulation des exigences offre un moyen de valider intuitivement les exigences avec les parties prenantes. C'est un moyen d'évaluer si les exigences correspondent aux besoins réels et d'évaluer la complétude des exigences [155].

(3) Inspection du modèle global d'exigences. L'obtention d'un modèle global des exigences est utile pour la validation des exigences. La plate-forme offre un mécanisme d'édition du modèle global par requêtes. Les vues obtenues peuvent servir de support à la discussion et la négociation.

(4) Génération de machines à états. La plate-forme génère une machine à état utilisable par des outils de vérification formelle (voir Chapitre IV). Une partie de l'analyse peut ainsi être déléguée à des outils de model-checking, pour confronter exigences fonctionnelles et non-fonctionnelles de qualité de service, par exemple.

(5) Génération d'une ébauche de spécifications système. La plate-forme permet d'extraire un ensemble de modèles pouvant servir de base à la production d'une première spécification système. Ces modèles regroupent un diagramme de cas d'utilisation UML et un modèle métier. Le premier décrit les agents du système et les principales fonctions que le système leur fournit. Le deuxième capture l'ensemble des agents du système, les concepts métiers, leurs propriétés et relations. Il est exprimé sous la forme un diagramme de classe UML.

(6) Génération d'objectifs de test système. La génération de cas de tests système (appelés objectifs de tests système car n'étant pas raffinés pour être directement évalués) et utilisés pour évaluer si le logiciel implémenté satisfait les exigences. Cette fonctionnalité limite le coût des activités de validation système [55].

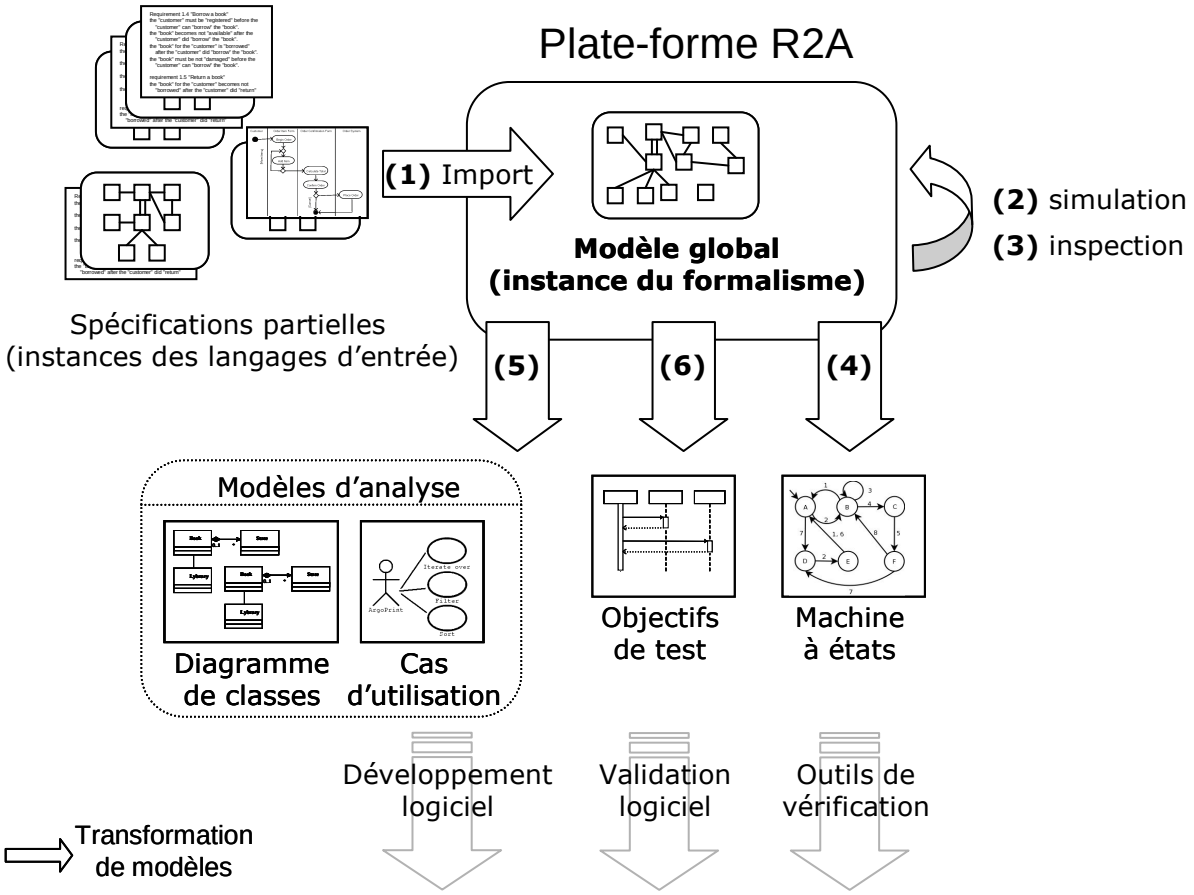


Figure 23 – Architecture de la plate-forme R2A, illustrant les fonctionnalités supportées.

3 Spécifications d'exigences étudiées

Cette section présente les spécifications d'exigences étudiées. La section 3.1 décrit brièvement ces spécifications. La section 3.2 détaille une spécification utilisée comme illustration dans la suite.

3.1 Etudes de cas réalisées

Nous avons étudié plusieurs études de cas provenant de deux industriels et d'un partenaire académique. Nous présentons brièvement ces études de cas, dans l'ordre chronologique :

- ***Thalès***. Les spécifications d'exigences étudiées portaient essentiellement sur le système de navigation des armements de la dernière génération de l'aviation de combat (Mirage et Rafale), dont TAS développe le processeur principal et les logiciels d'application. Ces spécifications ont permis de valider une première version du RDL et l'outil de génération de test système de la plate-forme R2A, fruit du travail de thèse de Clémentine Nebut [54-58].
- ***France Télécom***. Nous avons tout d'abord modélisé les étapes d'initialisation de la Livebox[55] et avons étudié les spécifications de deux services télécoms : la « Live-radio Baracoda » développée au sein du Livebox Lab[56] et un prototype de réunion virtuelle. Ce dernier est à l'origine de l'exemple fourni dans ce manuscrit et présenté en section suivante. C'est l'exemple le plus complet et le plus réaliste que nous ayons étudié à France Télécom.
- ***Ensieta***. Nous avons spécifié un système réactif d'alarme résidentielle (système domotique). Cette spécification regroupe des exigences représentatives de celles rencontrées au sein des spécifications confidentielles d'Airbus confiées à l'Ensieta[57]. Cette étude de cas n'est pas encore aboutie, mais elle nous a permis de nous faire une idée de ce qu'est une spécification opérationnelle d'un système réactif. Dans ce contexte, les exigences fonctionnelles du système sont décrites en RDL et en SDL [86] ; les exigences non-fonctionnelles en RDL.

3.2 Exemple d'une spécification d'exigences : le système SRV

L'exemple utilisé tout au long de ce manuscrit est un système de réunions virtuelles (SRV). Cette spécification est fournie en section 3.2.1. La section 3.2.2 fournit une analyse de cette spécification.

3.2.1 Spécification informelle du système SRV

Cette spécification est donnée sous la forme de six paragraphes en langage naturel. Cette spécification est représentative des premières spécifications d'exigences : elle est peu détaillée, l'information n'est pas classée et très imprécise. Voici le texte :

P_1 00 « Un système de réunion virtuelle permet aux employés de la compagnie de mettre en
01 place des réunions de travail via internet. Le système héberge un ensemble de réunions.
02 Chaque participant peut avoir une caméra branchée à sa machine, ce qui lui permet
03 quand celle-ci est allumée, d'être vu des participants des réunions auxquelles il prend
04 part. Le système propose un mécanisme de vote, géré par un modérateur au sein d'une
05 réunion. Le système propose deux types de réunions. Des réunions publiques ouvertes à
06 l'ensemble des participants. Des réunions privées, auxquelles un participant ne peut
07 accéder que si le modérateur l'y a autorisé. »

P_2 08 « Pour pouvoir participer à une réunion, un participant doit d'abord se connecter au
09 serveur du système, puis consulter la liste des réunions disponibles. Il peut alors choisir
10 une réunion et y entrer. Pour prendre la parole, un participant doit la demander au

[55] Le modem vendu par France Télécom, support des offres télévisuelles, téléphonie fixe et internet.

[56] Incubateur de produits et services internet pour le « home network ».

[57] Parmi celles-ci, on trouve par exemple une spécification de l'AFN (*Aircraft Facilities Notification*), un protocole assurant la prise de contact et l'échange de messages entre les avions et les centres de contrôle aérien.

11 modérateur de la réunion. Lorsqu'un participant a la parole, il peut envoyer un ensemble
12 de messages aux autres participants. Il peut aussi rendre la parole à tout moment. Un
13 participant peut quitter une réunion à tout moment. Le système doit garantir que le temps
14 de chaque prise de parole par un participant n'excède pas trois minutes. »
P_3 15 « Tout participant peut planifier une réunion. Dans ce cas, il est considéré comme le
16 gestionnaire de la réunion. En tant que tel, il peut définir un modérateur pour cette réunion
17 et définir le type de la réunion. Un gestionnaire peut ouvrir les réunions qu'il a planifiées.
18 Une réunion doit être ouverte avant qu'un participant puisse y entrer. Le modérateur peut
19 donner la parole à un participant mais aussi la lui retirer à tout moment. Il est aussi
20 responsable de la fermeture d'une réunion. Dans ce cas, les participants sortent de cette
21 réunion et ne peuvent plus y entrer. »
P_4 22 « Le système de réunion doit offrir un haut niveau de sécurité, car les décisions votées au
23 sein des réunions peuvent être cruciales pour la compagnie. Les messages manipulés
24 par le système doivent être cryptés par clé RSA. De plus, l'identité des participants doit
25 être strictement vérifiée. Pour ce faire, chaque participant doit, une fois connecté à un
26 serveur du système, se connecter à un serveur externe pour identification avant de
27 pouvoir entrer dans une réunion. Le serveur externe est mis en place par une société
28 privée et le temps de connexion est au maximum de 30 secondes. De plus, un participant
29 ne peut voter que s'il a une caméra et que si celle-ci est allumée. Ces règles de sécurité
30 ne s'appliquent que dans le cas de réunions privées. »

P_5 31 « Une réunion ne peut être ouverte si un modérateur ne lui a été affecté. De plus, une
32 réunion ne peut comporter plus d'un modérateur. Un modérateur peut fermer les réunions
33 dont il est responsable. Il doit être impossible à deux participants d'avoir la parole en
34 même temps. Pour des raisons de sécurité, la caméra d'un utilisateur doit être
35 automatiquement éteinte lorsqu'un participant quitte une réunion. Le temps de connexion
36 sur le serveur prend en moyenne quinze secondes. »
P_6 37 « Les responsables des réunions sont généralement les assistant(e)s de projets. Le
38 besoin de réunions virtuelles étant important au sein de la compagnie (nécessité de
39 réduire les coûts de déplacements), la création de réunions doit être rapide et aisée, afin
40 de limiter la charge de travail supplémentaire pour les assistant(e)s. L'interface doit être
41 intuitive de sorte que la planification d'une réunion soit possible en moins de 30
42 secondes. Le temps entre la demande de planification d'une réunion faite à un assistant
43 et l'ouverture de cette dernière doit être inférieur à une minute (ordre de grandeur). »

3.2.2 Analyse de la spécification

La spécification SRV est une spécification d'exigences obtenue au tout début d'un processus d'ingénierie des exigences. C'est une spécification informelle mélangeant exigences fonctionnelles et non fonctionnelles du SRV ainsi que des objectifs décrivant les besoins initiaux des clients. Dans cette section, nous illustrons les notions introduites dans les Chapitre I et Chapitre II.

Ambiguïtés. Cette spécification comporte de nombreuses ambiguïtés. Certaines peuvent être dues à la nature informelle du langage naturel. Par exemple, il est impossible de déterminer syntaxiquement à quel mot se réfère « celle-ci » dans la phrase du premier paragraphe[58] : « Chaque participant peut avoir une caméra branchée à sa machine, ce qui lui permet quand celle-ci est allumée … » (lignes 2 à 4). Cette ambiguïté a des chances d'être résolue intuitivement et de la même manière par l'ensemble des parties prenantes (néanmoins, il est possible qu'une partie prenante comprenne que c'est de la machine dont il est question).

Une autre cause d'ambiguïté est le manque de précision et par conséquent l'incomplétude de la spécification (ce que l'usage du langage naturel favorise). Voici quelques exemples de manques d'information au sein de la spécification SRV :

- rien ne précise dans la spécification qu'un gestionnaire de réunion puisse participer à celle-ci. Autrement dit, un gestionnaire peut-il être un participant ?

[58] En français, « celle-ci » représente toujours le dernier mot nommé, ici « machine ». Si on avait voulu désigner « caméra », il aurait fallu employer « celle-là ». En fait, il n'y a pas d'ambiguïté, même s'il est vrai que dans le langage courant, on tient rarement compte de cette règle de grammaire.

- il n'est pas non plus spécifié si un gestionnaire peut se définir lui-même comme modérateur.
- il n'est pas précisé si un participant doit être en possession d'une caméra pour pouvoir prendre part à une discussion privée. En effet, un vote a-t-il encore un intérêt dans le cadre d'une réunion privée où seul un participant est en possession d'une caméra (donc en mesure de voter) ?
- la référence au « serveur » ligne 36 est ambiguë car il est impossible de déterminer si ce serveur correspond au serveur externe (pour l'identification sécurisée, cf. ligne 26 par exemple) ou au serveur du système (cf. ligne 9 par exemple).

Incohérences. La spécification SRV comporte plusieurs incohérences. Elle contient tout d'abord des incohérences de type sous-spécification comme nous avons vu en section précédente (incomplétude), ce qui est source d'ambiguïté. Elle contient aussi d'autres types d'incohérences :

- ***Contradiction logique statique*** : par exemple, l'expression « au serveur du système » (ligne 9) suppose qu'il existe un seul serveur pour la connexion des participants au système. Or, cette information est contredite par l'expression « à un serveur du système » (ligne 25-26). Cette incohérence est statique car elle porte uniquement sur une connaissance des états possibles du domaine.
- ***Contradiction logique dynamique*** : par exemple, la spécification indique que le temps entre la connexion au serveur interne et l'ouverture d'une réunion doit être inférieur à une minute. Cette exigence peut être satisfaite par le scénario suivant: (i) connexion du gestionnaire au serveur du système en 15 secondes (cf. ligne 36), (ii) planification en 30 secondes (cf. ligne 41-42) puis (iii) ouverture (le temps est a priori négligeable puisqu'aucune durée n'est précisée). Cette exigence ne peut cependant pas être satisfaite si la réunion est privée, car il faut alors ajouter 30 secondes pour l'identification via le serveur externe (cf. ligne 28).
- ***Collision*** : par exemple, il est question dans la spécification SRV de responsables (ligne 37) et gestionnaires de réunions (e.g. ligne 16). Les termes « responsables » et « gestionnaires » désignent en fait la même notion, mais désignée par deux mots différents. C'est un cas de collision terminologique (cf. section 4.1 du Chapitre II). Le premier paragraphe contient en outre un exemple de collision dénotationnelle : le terme « employés » désigne en fait le même concept que « participant ». Les collisions structurelles ne peuvent être détectées au sein d'une spécification informelle car elles consistent en une différence dans la manière de modéliser une information complexe (plusieurs notions liées).

Nature des informations. La spécification SRV est une spécification d'exigences de haut niveau. En tant que telle, elle capture des informations mélangeant un grand nombre de types d'informations. On trouve pêle-mêle :

- ***(A1) Des objectifs***. Par exemple, la phrase « Le système de réunion doit offrir un haut niveau de sécurité car les décisions votées au sein des réunions peuvent être cruciales pour la compagnie. » (lignes 22-23) décrit un objectif de confidentialité. Les deux phrases suivantes sont des objectifs raffinant ce dernier. En effet, le cryptage des messages échangés durant une réunion et l'identification des participants contribue à la confidentialité des informations échangées dans le système. La première partie du dernier paragraphe décrit aussi un objectif, relatif à la rapidité d'utilisation du système.
- ***(A2) Des descriptions des états possibles du domaine***. Les exigences décrivent aussi un aspect statique du domaine. Ce type d'informations comporte des informations sur les concepts du domaine, leurs relations et les cardinalités de celles-ci. Par exemple, la spécification SRV porte sur les concepts de « participant », de « réunion », de « caméra », de « serveur du système » … Elle spécifie que la machine d'un participant peut être reliée à une caméra (ligne 2), que plusieurs participants peuvent être connectés au serveur du système (ligne 25-26) …
- ***(A3) Des contraintes sur les états du système***. La spécification du SRV contient aussi des contraintes statiques comme le fait qu'il ne puisse y avoir plus d'un modérateur par réunion (ligne 32), ou qu'il soit impossible à deux participants d'une même réunion d'avoir la parole en même temps (ligne 33-34).

- ***(A4) Des descriptions d'actions et liens de causalité***. La spécification SRV comporte un ensemble de descriptions d'actions pouvant être exécutées par les agents du système, comme « se connecter au serveur du système », « prendre la parole » ou encore « entrer dans une réunion ». Quatre passages de la spécification capturent des informations relatives à l'action « entrer dans une réunion » (lignes 10, 18, 21 et 27), dont deux (lignes 18 et 27) décrivant des conditions sur son exécution (liens de causalité). Ce type d'informations est dynamique (au sens de Mylopoulos [62]).
- ***(A5) Des informations sur la durée des actions***. Certaines parties de la spécification SRV décrivent la durée d'une action. Par exemple, les durées des actions de connexion au serveur interne et externe (lignes 36 et 28 respectivement).
- ***(A6) Des contraintes de durée sur un ensemble de scénarios***. La dernière phrase du paragraphe 6 est un bon exemple de ce type d'informations. C'est une condition sur la durée maximum des scénarios visant à ouvrir une réunion en un temps minimum.
- ***(A7) Des contraintes techniques***. Le seul exemple dans la spécification SRV est l'imposition d'une technique de cryptage par clé RSA (lignes 23-24).

D'autres types d'informations pourraient figurer dans la spécification SRV comme des préoccupations de coûts de fonctionnement, des stratégies politiques (e.g. contraintes sur les compagnies candidates pour la gestion du serveur externe d'identification) ou encore des informations organisationnelles. Nous avons vu que la décomposition et le classement de l'information est une problématique importante en IE (cf. section 2.1 du Chapitre I). La spécification d'exigences SRV est un exemple typique de spécification non classée. Les informations appartenant aux différents types d'informations présentés en section précédente sont fortement entrelacées.

Deuxième partie : Contributions

Préambule

Les chapitres précédents présentent les domaines de recherche étudiés pendant cette thèse. Le Chapitre I décrit l'ingénierie des exigences, domaine de recherche portant sur la maîtrise de la qualité logicielle via des techniques et méthodes V&V opérant dès les premières étapes d'un projet logiciel. Ces techniques sont particulièrement efficaces une fois une spécification opérationnelle des exigences obtenue. Cette spécification est particulièrement difficile à obtenir vu sa nature. Cette spécification est fragmentée en un ensemble de spécifications partielles, hétérogènes et potentiellement incohérentes. Pour faire face à cette complexité, nous avons choisi d'adopter le cadre conceptuel de métamodélisation de l'ingénierie dirigée par les modèles, présenté dans le Chapitre II. Ce cadre conceptuel est adapté à la production de processus de développement fortement automatisés, mettant en jeu un grand nombre de langages et algorithmes.

Cette partie présente les contributions. Nous avons présenté dans le Chapitre III une approche intégrant au sein d'un même environnement des outils V&V, dont l'élément principal est un processus de composition de modèles. Nous proposons une solution pour ce processus, utilisé dans deux contextes bien distincts : (i) lorsque ces modèles représentent des exigences, le modèle global obtenu représente une vue unifiée de ces exigences ; (ii) lorsque ces modèles représentent des composants de conception représentant des types d'information et fonctionnalités pour l'analyse d'exigences opérationnelles, le modèle global obtenu représente le formalisme de la plate-forme R2A. Nous proposons de plus deux langages supportés par la plate-forme.

Organisation

Le Chapitre IV présente ces deux langages de modélisation, à savoir le formalisme RM et le langage RDL. Le premier est le formalisme actuel de la plate-forme. Il définit la nature des informations pouvant être extraites au sein des spécifications partielles et capturées par le modèle global obtenu après composition. Le langage RDL est le langage d'entrée principal de la plate-forme. C'est un langage naturel contraint à base de patrons de spécification. Il offre une grande expressivité et est le plus accessible des quatre langages d'entrée actuellement supportés. Ses expressions peuvent décrire des propriétés fonctionnelles et non-fonctionnelles (que ce soit des exigences ou des assertions environnementales). Le RM et le RDL sont utilisés dans la suite de ce manuscrit pour illustrer les processus de composition : (i) processus de composition simple (Chapitre V et Chapitre VI), (ii) processus de composition double (Chapitre VII).

Le Chapitre V fournit une analyse des caractéristiques d'un processus de composition de spécifications partielles. Il discute des problématiques d'interprétation, d'ambiguïté, de variations sémantiques et des stratégies possibles pour résoudre les ambiguïtés. Il illustre aussi les problèmes pouvant être rencontrés lors de l'identification de superpositions sémantiques, en particulier dans le cas de collisions structurelles. Ces caractéristiques peuvent être considérées comme des contraintes que tout processus de composition d'exigences doit satisfaire.

Le Chapitre VI présente le processus de composition simple, respectant les contraintes précédemment citées. Ce processus compose des spécifications partielles d'exigences, artefacts de niveau M1 puisque le métamodèle cœur (niveau M2) est instance du MOF (niveau M3). Il autorise une séparation des préoccupations au niveau instance, ce qui facilite le travail des parties prenantes (particulièrement les rédacteurs et les analystes). Les exigences sont décrites par une collection d'expressions de notation différentes et sont classées à l'aide de points de vue : les points de vue servent à classer les exigences par préoccupations systèmes, les notations permettent une description

multi-langages des exigences et les modèles d'entrée une description fragmentée. Le processus de composition simple satisfait le premier objectif et participe au troisième objectif de la plate-forme R2A (voir la section 1 du Chapitre III) : l'obtention d'un modèle global à partir de spécifications partielles hétérogènes (1er objectif) et le couplage entre exigences et artefacts de développement grâce au mécanisme de traçabilité (3ème objectif).

Le processus de composition simple permet de composer des spécifications partielles incohérentes. Il transforme un problème de détection d'incohérences inter-modèles en un problème intra-modèle. Il fournit dès lors une vue globale des incohérences et permet de détecter des manques d'information. Des règles de cohérence peuvent être appliquées sur le modèle non conforme au métamodèle cœur et le résultat est présenté de manière compréhensible pour les parties prenantes. Ce processus fournit à terme un modèle global statiquement cohérent pour les outils de détection d'incohérences dynamiques. Le processus est outillé d'un mécanisme natif de traçabilité produisant automatiquement un modèle de traçabilité entre spécifications partielles et modèle global. Ces liens sont utilisés par les parties prenantes pour déterminer les spécifications potentiellement mises en cause par la détection d'une incohérence, qu'elle soit statique ou dynamique. L'utilisation conjointe du processus présenté ici et des outils de vérification formelle dépasse le cadre de cette thèse.

Le Chapitre VII présente le processus de composition double. C'est une extension du processus de composition simple. Il autorise une séparation des préoccupations au niveau conception. Il participe à la réalisation du deuxième objectif de la plate-forme R2A, à savoir son adaptabilité (offrir une plate-forme d'ingénierie des exigences intégrant une batterie variable d'outils d'analyse des exigences).

Chapitre IV Le formalisme RM et le langage RDL

Résumé

Le formalisme RM est décrit en section 1. Il est construit sur la base d'une comparaison entre les formalismes utilisés en ingénierie des exigences et plus généralement en représentation des connaissances. Il permet la capture des aspects statiques et dynamiques du domaine d'un système (temporel et temps réels) : assertions environnementales et exigences, propriétés fonctionnelles et non-fonctionnelles. En outre, il comprend un ensemble de constructions propres à la classification des exigences, à la métamodélisation et à l'analyse d'une spécification. En tant que tel, le formalisme RM constitue les fondements d'un environnement de fiabilisation des exigences dont la plate-forme R2A est une implémentation.

Le langage RDL est décrit en section 2. C'est un langage naturel contraint à base de patrons de spécification : c'est un langage formel muni d'une syntaxe facilitant la rédaction des exigences par les parties prenantes (ce qui n'est pas le cas d'un formalisme comme RM). Il permet de décrire un grand nombre de type d'informations : conditions d'activation et effet d'une action, durée d'une action, lien de causalités entre états du système, configuration d'un système à un instant donné. Il intègre aussi des patrons de spécifications temps réels facilitant la descritpion de propriétés non-fonctionnelles.

1 Le formalisme RM

Le formalisme RM est un cadre mathématique représenté par le métamodèle RM. Il s'inspire du cadre conceptuel de Jackson décrit en Annexe B, de KAOS [97], de IF [90], de l'approche CREWS-SAVRE [156-157], des cas d'utilisation UML [158] et des approches de classification de l'information par points de vue (par exemple VOSE [16]). Nous distinguons comme Jackson propriétés indicatives et optatives car la traduction d'un modèle RM vers le langage d'un outil de vérification en dépend. Tout comme KAOS, RM combine une logique du premier ordre avec une logique modale. Les propriétés fonctionnelles sont décrites à l'aide d'actions munies de pré- et de post-conditions. La durée de ces actions peut aussi être spécifiée. Les propriétés non-fonctionnelles (temporelles et temps réel) sont capturées par un sous-ensemble du langage IF. RM intègre aussi la notion de scénarios (UML, CREWS-SAVRE …), les actions correspondant dans notre cas aux cas d'utilisations. Enfin, Il est outillé pour la gestion de points de vue (rédaction mais aussi extraction de points de vue et traçabilité de l'information). RM adresse trois préoccupations :

- La capture d'une spécification opérationnelle.
- L'application de techniques V&V.
- Le classement et la traçabilité de l'information pour faciliter la séparation des préoccupations, la communication et la négociation entre parties prenantes.

Cette section détaille le métamodèle RM. La section 1.1 présente les différentes vues du RM, les trois niveaux de description supportés et les choix d'implémentations dans un contexte IDM (niveaux de modélisation). Chaque vue participe à la réalisation d'une des préoccupations citées plus haut. Les deux sections suivantes présentent de manière factuelle la sémantique du langage. La section 1.2 présente la sémantique statique du RM et donne quelques exemples de modèles RM. La section 1.3 définit ensuite la sémantique dynamique et illustre l'intégration d'outils V&V au sein de la plate-forme R2A. Nous présentons rapidement l'implémentation de cette sémantique avec (i) Kermeta [146] pour la simulation des exigences fonctionnelles [155], la production de tests systèmes [55, 57] et avec (ii) IF [90] pour vérifier formellement la satisfaction des exigences non-fonctionnelles.

1.1 Vues, niveaux de description et niveaux de modélisation

	vues \ niveaux	processus	produit	instance
domaine	conceptuelle		X	
	fonctionnelle		X	
	non-fonctionnelle		X	
	logique		X	
V&V	instanciation			X
	analyse	X		
Organisat.	classification	X		
	traçabilité	X		
	métamodèlisation	X		

Tableau 4 - Classification des vues suivant le niveau de description représenté.

La Tableau 4 présente les neuf vues RM et les trois niveaux de modélisation supportés. Les vues sont regroupées au sein de trois groupes (domaine, V&V et organisationnel). Les vues du domaine fournissent l'ensemble des concepts et relations utiles à la description d'une spécification opérationnelle. Les vues V&V sont dédiées à l'application de techniques V&V. Enfin les vues

organisationnelles structurent les spécifications partielles ainsi que la spécification opérationnelle obtenue par le mécanisme de composition simple (voir Chapitre VI). Les vues de formalisme RM capturent des informations appartenant à trois niveaux de description (processus, produit et instance). Les vues du domaine représentent des informations de niveau produit. La vue instanciation capture des informations appartenant au niveau instance. Les vues analyse et classification capturent des informations de niveau processus, c'est-à-dire utile au bon déroulement du processus IE. Ces vues sont brièvement présentées ci-dessous. Elles sont détaillées dans la suite de ce chapitre ou en Annexe E :

- ***Vue conceptuelle*** : capture l'aspect statique de l'information
- ***Vue logique*** : représente la logique du premier ordre.
- ***Vue fonctionnelle*** : représente les propriétés fonctionnelles du domaine.
- ***Vue non-fonctionnelle*** : représente les propriétés non-fonctionnelles du domaine.
- ***Vue instanciation*** : capture les instances d'un domaine (configuration) et les scénarios.
- ***Vue analyse*** : fournit un cadre pour paramétrer les preuves effectuées.
- ***Vue classification*** : représente les moyens de classification d'une spécification opérationnelle et capture les liens de traçabilité entre spécifications partielles et la spécification opérationnelle globale obtenue par composition.
- ***Vue Traçabilité*** : représente les liens sémantiques entre les spécifications partielles et la spécification globale obtenue après composition.
- ***Vue métamodélisation*** : capture une partie du cadre conceptuel de l'IDM (cf. Chapitre II).

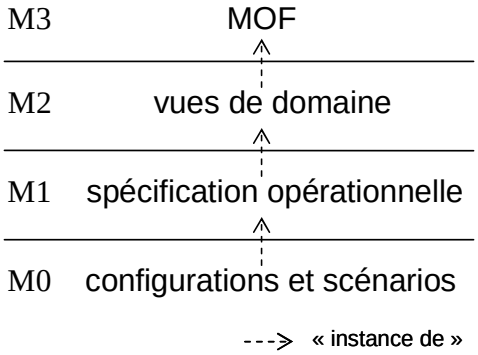


Figure 24 - Niveaux de modélisation théoriques du formalisme R2A dans le cadre du MDA.

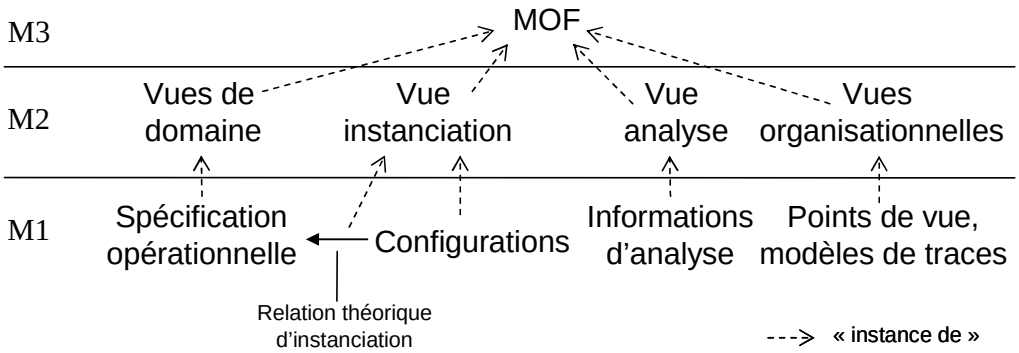


Figure 25 – Implémentation des niveaux de modélisation du formalisme RM dans le contexte MDA.

Dans le contexte de métamodélisation MDA [126], le formalisme RM s'articule autour de quatre niveaux de modélisation, soit trois niveaux d'instanciation (cf. Figure 24). Les instances de la vue instanciation sont instances de la spécification opérationnelle, elle-même instance des vues domaines. Les vues de domaine sont instances du MOF (les autres vues ne sont pas présentées dans la Figure 24). Cependant, les transformations en IDM ne peuvent manipuler que des artefacts de niveau M1, M2 et M3 car elles sont spécifiées sur les éléments d'un métamodèle (niveau M2 ou M3, le MOF étant aussi un métamodèle). Pour ces raisons techniques, les informations manipulées par les transformations définies attachées au métamodèle RM ne peuvent être des artefacts de niveau M0 [155], ce qui rend

impossible le traitement des instances de la spécification opérationnelle. Pour remédier à ce problème, nous avons décidé de représenter ces instances à l'aide d'artefacts de niveau M1. La Figure 25 schématise ce choix d'implémentation et présente l'ensemble des vues du formalisme RM dans le contexte MDA. La relation d'instanciation entre les niveaux théoriques M0 et M1 de la Figure 24 est donc explicitement modélisée par le métamodèle RM.

1.2 Sémantique statique

Cette section détaille les cinqs premières vues du métamodèle RM (vue conceptuelle, fonctionnelle, non-fonctionnelle, logique et instanciation). Cette section présente en outre des expressions RM. Ces expressions sont présentées à l'aide de différentes syntaxes : (i) une syntaxe textuelle formelle adaptée à la représentation des propriétés fonctionnelles (proche de celle de KAOS), (ii) une syntaxe FSA pour les propriétés non-fonctionnelles et (iii) sous la forme de modèles comme ceux présentés dans le Chapitre II. Nous présentons uniquement la sémantique statique dans cette section, la sémantique dynamique étant détaillée en section suivante.

Les spécifications d'exigences que nous avons étudiées dans les contextes industriels de THALES et France TELECOM décrivent pour la plupart les conditions nécessaires à l'activation de services et les effets sur le système si ceux-ci sont déclenchés. Les exigences non-fonctionnelles, lorsqu'elles ne sont pas informelles, sont décrites comme des durées attribuées à l'activation d'une action ou des durées séparant deux phénomènes du système. En somme, ces observations corroborent celles de Jackson [85, 159] : les exigences décrivent un ensemble de liens de causalités (temporisés ou pas) entre les phénomènes d'un système. Nous avons défini le métamodèle RM en nous basant sur ces observations et l'étude des langages de modélisation des exigences proposés dans littérature (voir section 2.2 du Chapitre I).

```
01 | Action "connect to the external server" (p : Participant)
   | duration : [] 30
   | pre ∃ s1 : Server, ∃ s2 : System, of(s1, s2) ∧ connected(p, s1)
   | post identified(p)
05 |
   | Constraint : (a)
   | ∀m : Meeting, ∀ p : Participant, ∃ c : Camera, entered(p, m) ∧ private(m) ∧ of(c, p)
   |                    => state(c) = setOn
   |
10 | Assertion : (b)
   | ∀ p : Participant, ¬∃ s : Server, connected(p, s) => ¬identified(p)
   |
   | Constraint : (c)
   | ∀ p : Participant, ∀m : Meeting, private(m) => (¬identified(p) => ¬entered(p, m))
```

Figure 26 - Un modèle RM sous sa forme textuelle.

La Figure 26 présente un extrait d'une spécification opérationnelle RM présentée avec la syntaxe concrète textuelle par défaut. Cette spécification capture quelques informations de la spécification SRV. Elle comporte trois éléments différents : une description de l'action de connexion au serveur externe (*action*), une propriété non-fonctionnelle optative (*constraint*) et une propriété non-fonctionnelle indicative (*assertion*). Cette section décrit les quatre vues de domaine du métamodèle RM nécessaire à la capture d'une spécification opérationnelle RM.

1.2.1 Vue conceptuelle

La vue conceptuelle représente les notions du domaine et les relations mises en jeu dans la spécification opérationnelle. Cette vue décrit l'aspect statique d'une spécification et appartient au niveau produit. La vue conceptuelle est présentée par la Figure 27 et la Figure 28. La première figure présente une partie de la vue, similaire à la partie du métamodèle UML capturant les diagrammes de

classes (le MOF comporte aussi des notions et relations similaires). La deuxième figure présente une partie du métamodèle RM représentant les trois mécanismes d'abstraction supportés par la vue conceptuelle : l'héritage, la possession et le rôle.

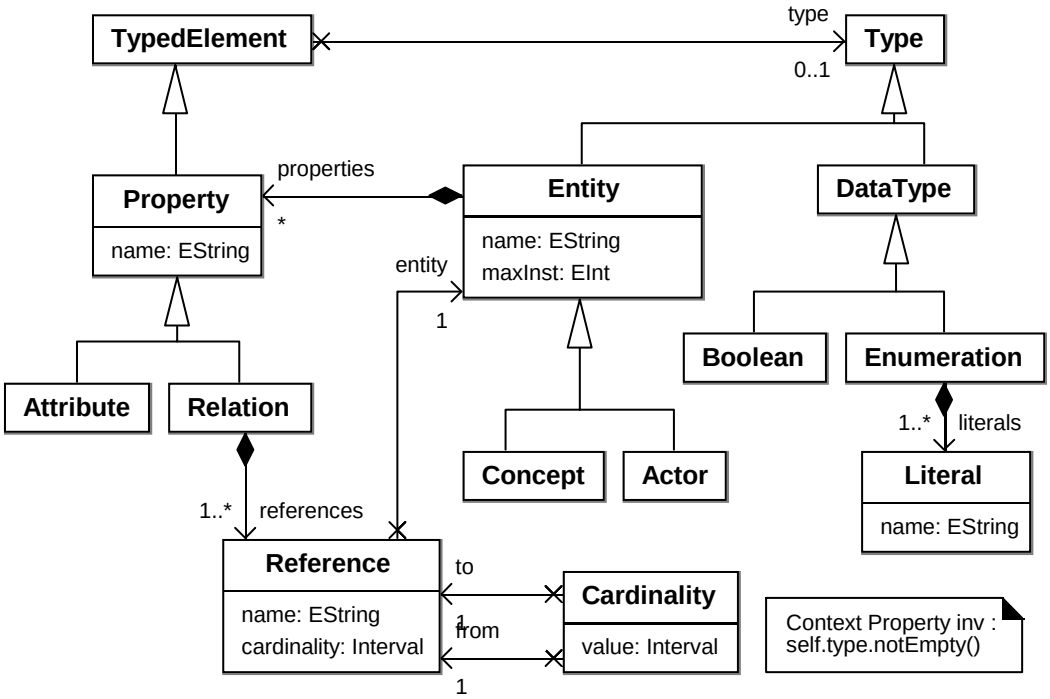


Figure 27 – Vue conceptuelle du métamodèle RM (entités et propriétés).

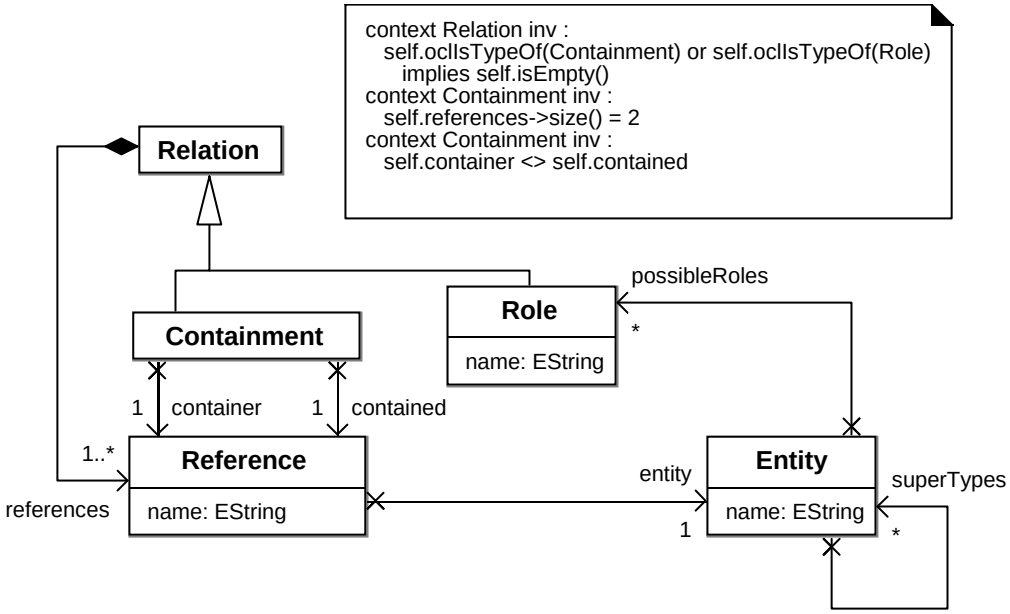


Figure 28 – Vue conceptuelle du métamodèle RM (mécanismes d'abstraction).

Entité (ENTITY). Une *entité* est un *concept métier* (CONCEPT) ou un *acteur du système* (ACTOR). Un acteur est une personne humaine interagissant avec le système, une machine extérieure au système ou toute autre entité de l'environnement dont les actions peuvent modifier l'état du système. Une entité contient un ensemble de propriétés.

Propriété (PROPERTY). Une *propriété* représente une *relation* entre plusieurs entités (RELATION) ou un *attribut* (classe ATTRIBUTE), c'est-à-dire un état d'une entité. Une propriété a un type (TYPE), comme une valeur booléenne ou une énumération (BOOLEAN et ENUMERATION). Une énumération est un ensemble fini de *littéraux* (LITERAL) dont la valeur (*name*) est une chaîne de caractères (les entiers et les intervalles peuvent être représentés par une chaîne de caractères). Les types de données

complexes ne sont pas supportés actuellement car (i) nous n'en avons pas eu besoin dans les études de cas menées et (ii) ce type d'informations ne fait pas a priori partie des exigences car trop détaillées (niveau conception). Il est néanmoins possible d'étendre RM si nécessaire.

Rôle (ROLE). Une entité peut jouer différents *rôles* au sein d'une spécification d'exigences (*possibleRoles*). Un rôle est un type particulier de relation. Un rôle peut être considéré à la fois comme une relation et comme une classe d'entité. La notion de rôle est dépendante du mécanisme d'abstraction héritage (représenté par la relation *superTypes*), au sens où les sous-types d'une entité e peuvent jouer les rôles possibles définis pour e ou ses super-types. Un rôle n'a pas de type comme le précise la première contrainte OCL.

Possession (CONTAINMENT). Cette notion est identique à la notion d'agrégation définie par le diagramme de classe UML ou MOF. C'est une relation binaire (voir la deuxième contrainte OCL).

1.2.2 Vue fonctionnelle

La vue fonctionnelle capture la description des actions d'un système (proche de la notion d'action de KAOS et de la notion des cas d'utilisation UML. Il est possible de spécifier la durée d'une action, suivant l'état du système lors de l'activation par un acteur. Cette vue capture donc les propriétés fonctionnelles du domaine.

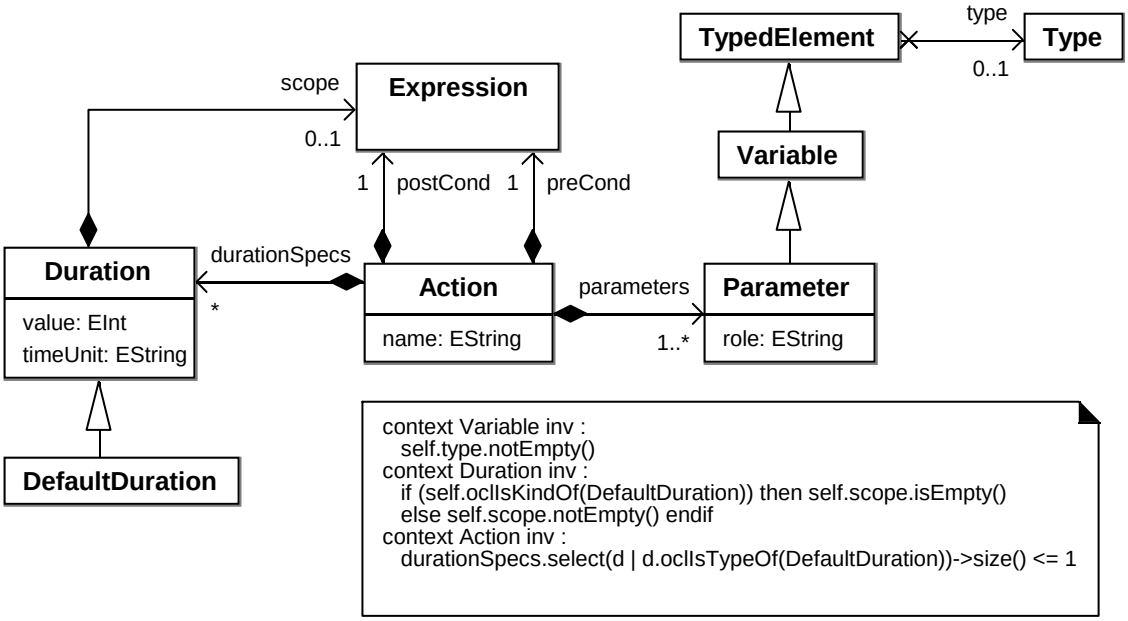


Figure 29 – Vue fonctionnelle du métamodèle RM.

Action (ACTION). Une action est une spécification par contrats des interactions entre acteurs du système. Une action comporte une *pré-condition* et une *post-condition* (ce sont des contrats [24]). Ces conditions sont des expressions logiques portant sur l'état du système (EXPRESSION). La pré-condition est toujours satisfaite par l'état courant du système au moment où l'action est activée. La post-condition spécifie l'effet de l'activation de l'action sur l'état du système. Une action comporte un ensemble de paramètres (PARAMETER), désignant les entités observées ou impactées lors de son activation (similaires aux paramètres formels d'une fonction). Un de ces paramètres désigne l'acteur responsable de l'activation de l'action (l'attribut *role* vaut alors « actor »). Pour une expression RM présentée textuellement, l'acteur correspond au premier paramètre (p est l'acteur de l'action présentée dans la Figure 26).

Durée d'une action (DURATION). Une action peut être instantanée ou durer pendant un intervalle de temps (*value*) dont la valeur dépend de l'état courant du système au moment de l'activation. Plusieurs durées possibles peuvent être spécifiées (*durationSpecs*). La durée retenue pour une action correspond à celle dont le champ d'application (*scope*) est vrai lors de l'activation. Lorsqu' aucun champ d'application n'est satisfait, la durée de l'action est définie par une durée par défaut (DEFAULTDURATION). Dans le cas contraire, l'action est considérée instantanée. Seule une durée par

défaut peut être spécifiée (cf. troisième contrainte OCL). Une durée par défaut ne comporte pas de champ d'application comme le spécifie la deuxième contrainte OCL

1.2.3 Vue non-fonctionnelle

La vue non-fonctionnelle permet de spécifier les propriétés non fonctionnelles. Ces propriétés sont décrites à l'aide d'automates FSA proches des automates du langage IF ou de ceux utilisés comme domaine du langage à base de patrons de spécification de Schmidt [53]. Les automates ont pour avantage d'être plus compréhensibles et expressifs que des logiques modales comme TCTL [91].

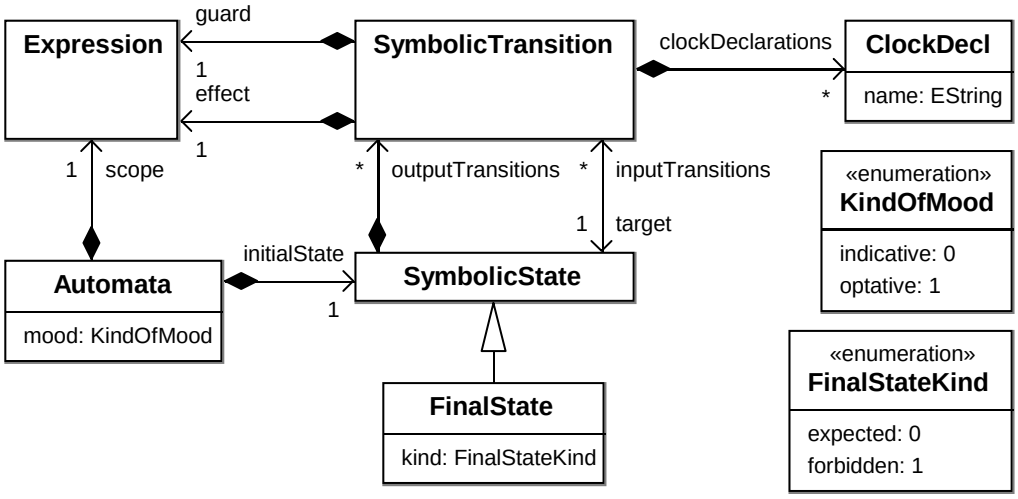


Figure 30 – vue non-fonctionnelle du métamodèle RM.

Automate (AUTOMATA). Un automate représente soit une assertion environnementale (*mood* = « indicative »), soit une exigence non fonctionnelle (*mood* = « optative »). Une assertion est représentée par un système de transitions, c'est-à-dire un ensemble de transitions (SYMBOLICTRANSITION) et d'états (SYMBOLICSTATE)[59]. Lorsqu'une assertion est optative, elle contient au moins un état final (FINALSTATE). Il existe deux sortes d'états finals : les états attendus (*kind* = « expected ») et les états interdits (*kind* = « forbidden »). La sémantique de ces états et de manière plus générale la sémantique d'une propriété décrite par un automate n'est pas présentée ici. De manière informelle, l'arrivée dans un état attendu d'une assertion signifie que l'assertion a été validée, et que l'arrivée dans un état interdit signifie que l'assertion a été violée. Le temps est représenté par des *horloges* (CLOCKDECL) pouvant être interrogées et attribuées par les gardes et les effets.

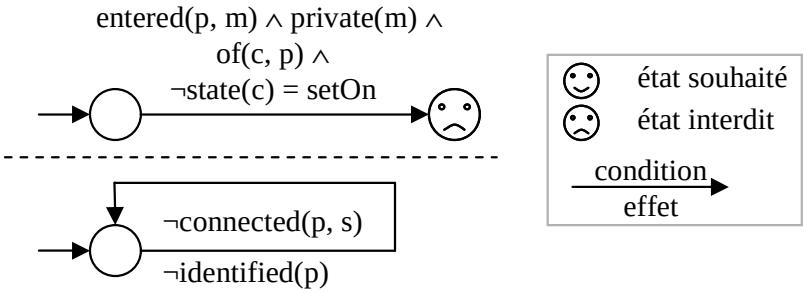


Figure 31 – Les deux propriétés non-fonctionnelles de la Figure 26, instances de la vue non-fonctionnelle du formalisme RM (syntaxe graphique du langage IF).

[59] On qualifie ici les états et les transitions de « symboliques » pour les distinguer des états (STATE) et transitions (TRANSITION) présentés dans la vue instanciation. Le terme « symbolique » signifie qu'un état de la vue non-fonctionnelle représente un ensemble d'états de la vue instanciation (et une transition représente un ensemble de transitions). A ne pas confondre donc avec la notion de système de transitions symboliques, où les transitions peuvent être étiquetées par des ensembles de valeurs (voir par exemple le model-checker NuSMV).

La Figure 31 fournit deux exemples d'instances de la vue non-fonctionnelle (et de la vue logique par dépendance). Ces exemples sont présentés avec la syntaxe du langage IF. Le premier FSA représente l'exigence non-fonctionnelle (contrainte) de la Figure 26 (lignes 06-07). Partant de l'état initial, le fait que la caméra d'un participant est éteinte alors que ce dernier est dans une réunion privée provoque le déclenchement de la transition symbolique et provoque l'arrivée dans un état interdit ; dans ce cas, la contrainte est violée. Le deuxième FSA représente l'assertion environnementale de la Figure 26 spécifiant qu'un participant non connecté n'est pas identifié. La condition déclenche la transition symbolique dont l'effet est répercuté sur le système.

1.2.4 Vue logique

La vue logique représente la logique du premier ordre spécialisée pour les particularités de la vue statique (rôle, énumération …). Les vues fonctionnelle et non-fonctionnelle dépendent de la vue logique car elles décrivent des relations de causalité (pré- et post-conditions des actions, garde et effet d'une transition symbolique).

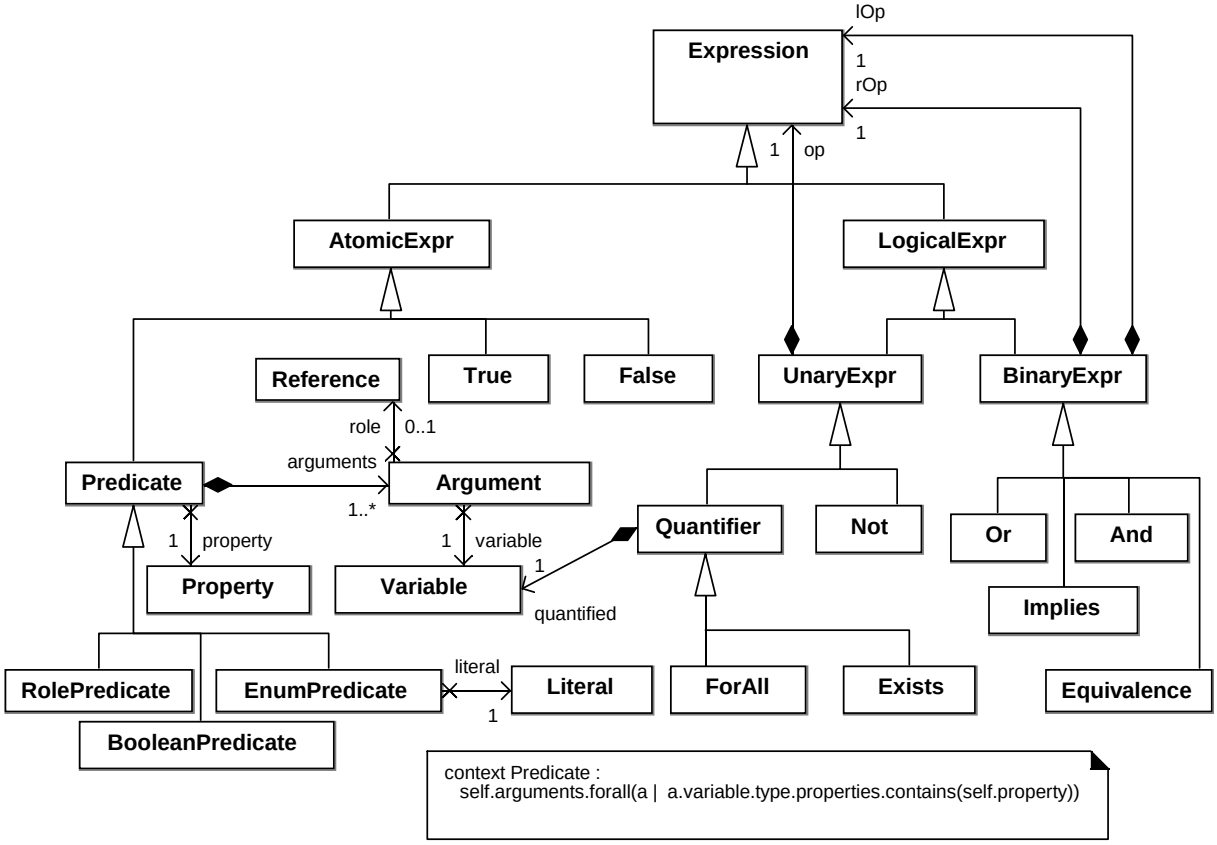


Figure 32 – Vue logique du métamodèle RM.

Expression (EXPRESSION). Une expression est une combinaison d'opérateurs logiques, de comparateurs booléens et de propositions atomiques. Les opérateurs logiques sont la conjonction (AND), la disjonction (OR), la négation (NOT), l'implication (IMPLIES) et les quantifieurs universels (FORALL, EXISTS). Les propositions atomiques sont les constantes booléennes (TRUE, FALSE) et les prédicats (PREDICATE) représentant les valeurs des propriétés des entités.

Prédicat (PREDICATE). Un prédicat représente la valeur d'une propriété d'une entité dans le cadre d'une expression. C'est la donnée d'une propriété (*property*) et d'un ensemble d'arguments (ARGUMENT) représentant les variables (VARIABLE) mises en jeu (le *role* d'un argument permet de spécifier le rôle d'une variable dans une relation).

1.2.5 Vue instanciation

La vue instanciation permet de représenter (i) les *configurations* possibles du système et (ii) des *scénarios* (séquences d'actions). La production de configurations est nécessaire pour spécifier un état

initial si l'on désire simuler le système. Les scénarios sont utiles pour la validation et la production d'objectifs de test système [55]. Ces scénarios peuvent être utilisés pour modéliser des tests systèmes ou encore sauvegarder les scénarios produits lors de la simulation des exigences fonctionnelles. Les configurations comme les scénarios peuvent être utilisés pour restreindre la portée d'une preuve formelle comme nous le verrons dans la section suivante.

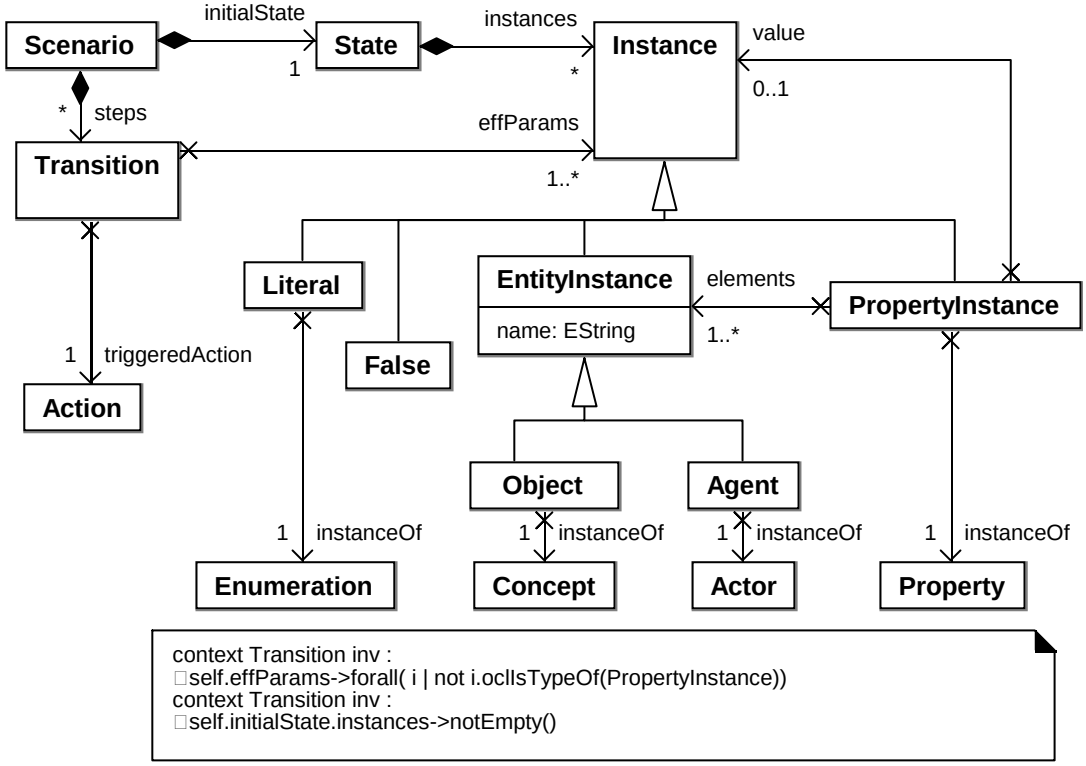


Figure 33 - vue instanciation du métamodèle RM.

Etat (STATE). Un état du système est la donnée d'un ensemble d'instances des propriétés et des entités du domaine. Un état du système représente une *configuration* du système à un instant donné.

Instance (INSTANCE). Une instance peut être un objet (OBJECT, instance d'un concept métier), un agent (AGENT, instance d'un acteur du système), une propriété d'une instance d'entité (ENTITYINSTANCE), ou encore un littéral. Par convention, on ne modélise les instances des attributs booléens que si leur valeur est à « vrai ». Les relations *instanceOf* permettent d'associer à chaque instance son type. Ces relations modélisent la relation d'instanciation entre les niveaux de modélisation théorique M0 et M1 (cf. section 1.1).

Scénario (SCENARIO). Un scénario est une séquence d'états (STATE) liés un à un par une transition (TRANSITION). Un scénario débute par un état initial (*initialState*). Chaque transition représente l'activation d'une action (*triggeredAction*) avec pour paramètres effectifs (*effParams*) une liste d'instances des éléments du domaine (autres que les instances de propriétés comme le spécifie la première règle OCL). Ces paramètres effectifs sont conformes aux paramètres formels de l'action activée. Seul l'état initial représente un état du système dans son intégralité, les autres états pouvant être calculés par simulation du scénario. La deuxième expression OCL contraint la propriété *instances* dans ce sens. Néanmoins, rien n'empêche de représenter dans leur intégralité l'ensemble des états.

1.3 Sémantique dynamique

Nous détaillons la sémantique dynamique du formalisme RM. Une description informelle est fournie en section 1.3.1. La section 1.3.2 suivante présente l'implémentation de la sémantique de RM à l'aide d'une sémantique opérationnelle pour la simulation des exigences fonctionnelles.

1.3.1 Description informelle

Etant donné un état du système S, les contrats d'une action spécifient si elle est applicable et quels sont les impacts sur S s'il est exécuté. En outre, les assertions décrivent aussi des liens de causalité modifiant l'état du système. Il est donc possible de créer de manière incrémentale des scénarios valides d'actions satisfaisant les exigences fonctionnelles et l'environnement. La construction d'une séquence valide est définie par l'algorithme suivant :

(i) Instanciation des actions avec des valeurs possibles pour chacun des paramètres, étant donné l'état courant du système.

(ii) Obtention d'un ensemble d'actions instanciées L dont la pré-condition est satisfaite par l'état du système.

(iii) Choix d'une action A à exécuter parmi les éléments de L.

(iv) Mise à jour de S par application de la post-condition de A.

(v) Identification des assertions environnementales pouvant être exécutées et mise à jour de S par application de ces dernières.

(vi) Deux variantes :

- On cherche à vérifier si les exigences fonctionnelles satisfont les exigences non-fonctionnelles. Dans ce cas, on identifie les assertions environnementales pouvant être exécutées et on met à jour S par application de ces dernières.
- On cherche à valider par simulation des exigences. Dans ce cas, on identifie les propriétés non-fonctionnelles (assertions et contraintes) pouvant être exécutées et on met à jour S par application de ces dernières.

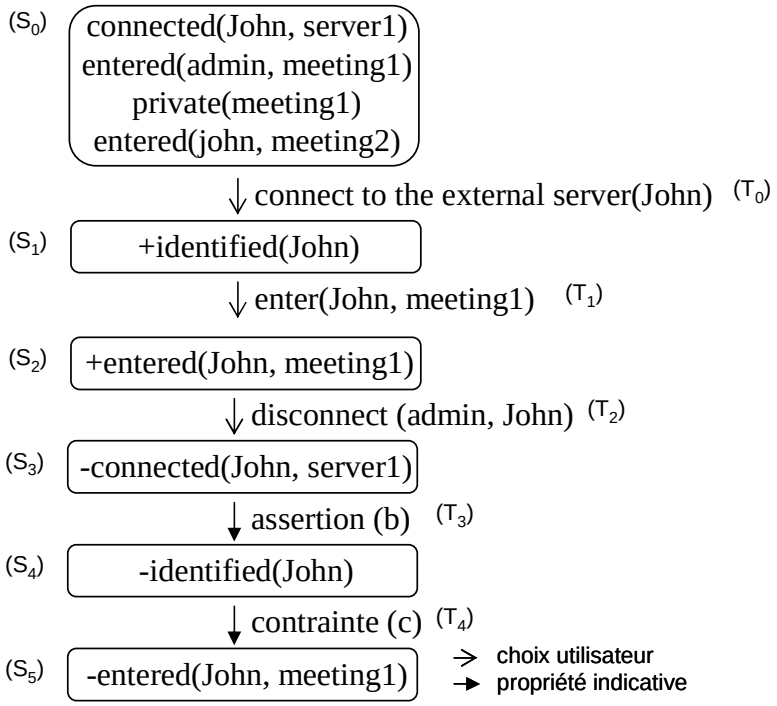


Figure 34 – Un exemple de scénario produit par simulation

En appliquant systématiquement cet algorithme, il est possible de générer un graphe fini et exhaustif représentant tous les comportements envisageables du système. La Figure 34 présente un extrait d'exécution de cet algorithme. C'est un scénario satisfaisant la sémantique dynamique. Il est composé d'une séquence de transitions liant deux à deux des états du système. L'état initial S_0 représente un état du système où un participant « John » est dans une réunion publique « meeting2 » et un autre participant (un administrtaeur) est dans une réunion privée « meeting1 ». Chaque transition représente soit une action effectuée par un acteur du système (transitions T_0 à T_2), soit l'exécution

automatique d'une propriété non-fonctionnelle (T_3 exécute l'assertion (b) de la Figure 26 et T_4 exécute la contrainte (c)). Par exemple, la connexion au serveur externe de John (transition T_0) modifie l'état du système en ajoutant le fait que « John » est désormais identifié par le système de sécurité (le serveur externe).

La Figure 34 est un exemple de scénario produit en choisissant la variante 6b de l'algorithme. En effet, la contrainte (c) est non pas évaluée mais exécutée. Dans ce cas, ce scénario représente les besoins des parties prenantes (cas d'une simulation pour validation). La variante 6a est appropriée si l'on cherche à vérifier que le logiciel satisfait ses exigences (cas du test système par exemple, ou du model-checking). Dans ce cas, un exemple de scénario pourrait être le même scénario que la Figure 34, excepté la dernière transition. A la place de cette dernière, il conviendrait d'ajouter une évaluation de la contrainte (c) sur le logiciel. Le couple scénario et évaluation forme un cas de test système (voir les travaux de Clémentine Nebut [57]).

1.3.2 Simulation par sémantique opérationnelle embarquée

Nous présentons ici l'implémentation de la sémantique RM à l'aide d'une sémantique opérationnelle embarquée[60]. Cette implémentation est spécialisée pour la simulation des exigences. Elle est implémentée en Kermeta, comme nous avons vu en section 3.2.2 du Chapitre II. La sémantique opérationnelle est donc spécifiée avec un ensemble d'opérations, de relations et de métaclasses étendant le métamodèle RM. La Figure 35 présente un extrait du métamodèle RM étendu. Elle comporte des métaclasses décrites précédemment dans ce chapitre et appartenant à différentes vues. Les éléments de la sémantique opérationnelle comme la métaclasse Simulator n'augmente pas la l'expressivité du formalisme RM ; ils servent uniquement à la spécification de la fonctionnalité de simulation. Nous verrons dans le Chapitre VII que le formalisme RM est en fait le résultat de la composition de composants ontologiques (les vues RM résultent de la fusion de composants ontologiques) et de composants fonctionnels (par exemple un composant de simulation).

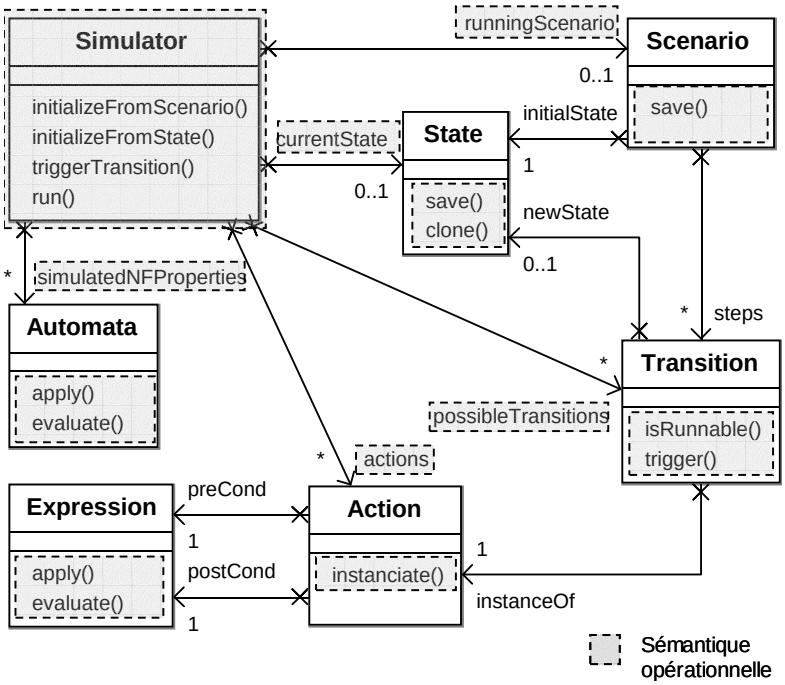


Figure 35 – Extrait du métamodèle RM étendu pour la simulation des exigences avec une sémantique opérationnelle embarquée Kermeta.

La métaclasse SIMULATOR définit quatre opérations utilisées pour piloter la simulation. L'opération *run* orchestre la boucle de simulation : calcul des transitions possibles (*possibleTransitions*), attente du choix de l'utilisateur, et enfin application des propriétés non-

[60] L'implémentation à l'aide d'un homéomorphisme vers le langage IF pour la vérification des exigences non-fonctionnelles n'est pas présentée.

fonctionnelles simulées (*simulatedNFProperties*). La simulation peut être initialisée avec un état du système fourni par l'utilisateur (*initializeFromState*) ou avec un scénario (*initializeFromScenario*) ayant été créé et sauvegardé (*save*) au préalable par simulation. Cette initialisation affecte l'état courant du simulateur (*currentState*). Les transitions possibles sont produites en affectant aux paramètres des actions l'ensemble des instances d'entités existantes dans l'état courant (*instanciate*), puis en vérifiant si ces actions satisfont les pré-conditions (c'est-à-dire si l'état courant implique logiquement la pré-condition). Cette vérification est réalisée par la méthode *isRunnable* faisant elle-même appel à la méthode *evaluate* de la pré-condition. L'utilisateur choisit la transition a tirée en exécutant l'opération *triggerTransition*. Cette dernière procède en deux étapes :

- Dans un premier temps, la transition est exécutée (trigger), ce qui a pour effet d'appeler la méthode *clone* sur l'état courant, puis d'appeler la méthode *apply* de l'expression post-condition sur l'état cloné. Cet état devient le nouvel état courant.
- Dans un deuxième temps, les propriétés non-fonctionnelles connues du simulator (*simulatedNFProperties*) sont appliquées une à une. A chaque application, l'état courant est modifié en conséquence.

2 Le langage RDL : un langage naturel contraint pour exprimer le domaine d'un système

Cette section présente le langage RDL, un langage naturel contraint à base de patrons de spécification initialement développé pour les besoins de Thalès [152], puis étendu pour ceux de France Télécom. Le RDL est le langage d'entrée principal de la plate-forme R2A. Il a fait l'objet d'un effort particulier afin de proposer une syntaxe adaptée à la manière dont sont rédigées les spécifications d'exigences chez nos partenaires industriels. Le RDL est un des langages d'entrée de la plate-forme R2A. Il propose une syntaxe accessible à l'ensemble des parties prenantes, permettant d'instancier de manière uniforme le métamodèle RM (son domaine). Il peut être considéré comme une syntaxe supplémentaire du formalisme RM dans son ensemble (les autres langages d'entrée de la plate-forme comme les diagramme de classe UML ou les diagrammes d'activités sont moins expressifs).

La section 2.1 discute de l'intérêt du RDL dans un processus IE et donne un exemple d'expression RDL. La section 2.2 caractérise l'expressivité du langage et fournit une description des patrons de spécification supportés. Enfin, la section 2.3 propose une description plus technique du langage. Elle décrit le métamodèle RDM, représentant la syntaxe abstraite du RDL. Cette description est utile pour illustrer le processus de composition simple du Chapitre VI. En effet, le mécanisme de composition de la plate-forme R2A comporte entre autre un ensemble de règles d'interprétation utilisées pour spécifier la sémantique des langages d'entrée (et en particulier le RDL, utilisé comme illustration).

2.1 Intérêt et Principes et du RDL

Le langage RDL est un langage naturel contraint anglais et est construit à base de patrons de spécification similaires à ceux de Dwyer [8, 51-52], Schmidt [53] et Cheng [52] (voir la section 2.3 du Chapitre I). Le choix de l'anglais résulte d'une exigence du partenaire industriel Thalès pour lequel le RDL a été initialement développé. Le caractère « naturel » du RDL rend ses expressions lisibles et compréhensibles par tout un chacun (les expressions conformes à la grammaire RDL sont aussi conformes à la grammaire anglaise). Son caractère « contraint » restreint les manières d'exprimer une même information et permet de lui attacher une sémantique formelle. Cette sémantique formelle est réalisée par une fonction sémantique définie à l'aide de règles d'interprétations et de fusion comme nous le verrons dans le Chapitre VI. L'adaptation du RDL à d'autres langues ne pose pas de difficulté particulière. Certes, l'anglais est une langue ne comportant pas autant de cas particulier et de règles orthographiques que le français par exemple. Le choix de l'anglais limite en ce sens la complexité du traitement syntaxique, mais ne modifie en rien la complexité du traitement sémantique étudié dans ce manuscrit (spécification de la relation sémantique).

L'utilisation du RDL implique une traduction manuelle par les parties prenantes d'une partie de la première spécification informelle des exigences. Cet effort de réécriture a semblé raisonnable aux ingénieurs de chez Thalès et France Télécom. Il est même apparu souhaitable pour deux raisons : (i) l'appropriation d'un recueil d'exigences est de toute façon nécessaire pour la production des artefacts de développement aval (modèles de conception, tests systèmes …) et (ii) la traduction du langage naturel vers un langage naturel contraint facilite la désambiguïsation des exigences. Enfin, notons que l'utilisation du RDL requiert une légère formation des parties prenantes. La perte de temps est cependant négligeable pour un industriel si l'on considère le retour sur investissement : composition automatique d'expressions compréhensibles par tous et obtention d'une spécification opérationnelle dont l'analyse est grandement assistée par la machine. Ce retour sur investissement est encore plus important si la spécification d'exigences est productive durant les activités de développement aval.

(1) A "participant" can be the "moderator" of at most three "meeting".
(2) A "participant" must set the "type" of a "meeting" before he can "plan" this "meeting".
(3) A "participant" becomes the "manager" of a "meeting" after he does "plan" this "meeting" .
(4) It takes 30 seconds for a "participant" to "connect" to the "external server".
(5) It is always the case that a "participant" is a "speaker" of a "meeting" for less than 3 minutes.
(6) A "meeting" has a "type" which can be "private" or "public".
(7) "Anne" is "connected" to the "cisco secure access control server" and is "entered" in the "full session 2008".
(8) When the "type" of a "meeting" is "private", a "participant" must be "identified" before he can "enter" in this "meeting".
(9) A "participant" can "plan" a "meeting".
(10) When the "type" of a "meeting" is "private", it is always the case that a "participant" did "connect" to the "external server" before he does "enter" in this "meeting".
(11) A "meeting" is "closed" if and only if it is not "opened".
(12) There is a "meeting" called "full session 2008".
(13) When a "participant" does "speak" in a "meeting", it is always the case that this "participant" is the "speaker" of this "meeting".
(14) A "participant" becomes the "speaker" of a "meeting" after the "moderator" of this "meeting" does "designate as speaker" this "participant".

Figure 36 - Exemples de phrases RDL.

Le choix d'un langage naturel comme le RDL est le résultat d'un compromis avec nos partenaires industriels. Nous aurions pu choisir de développer un mécanisme de traitement automatique de la langue naturelle. Cependant, un tel traitement est difficilement envisageable et l'obtention automatique d'une représentation formelle à partir d'un texte en langue naturelle est encore aujourd'hui de la « science fiction ». Au mieux, un tel mécanisme nécessiterait un dialogue entre machine et utilisateurs pour disqualifier toute ambiguïté. En outre, la spécification formelle obtenue devrait être inspectée de toute manière, puisque la qualité du logiciel est en jeu. En somme, le traitement du langage naturel nécessite quand même une implication forte des parties prenantes.

Une spécification RDL consiste en un ensemble de phrases anglaises courtes et grammaticalement simples décrivant le domaine du système (assertions environnementales et exigences). La Figure 36 présente une spécification RDL comportant 14 phrases, issues du raffinement de la spécification SRV. La grammaire du langage RDL est fournie en Annexe F. On peut remarquer qu'un grand nombre de termes sont systématiquement entourés de guillemets. Cette particularité syntaxique se justifie pour deux raisons :

- ***la volonté de mettre en exergue les notions du domaine***. Ces notions référencent en effet des concepts du domaine métier et font donc partie d'un vocabulaire devant être commun à l'ensemble des parties prenantes (généralement défini dans un document appelé dictionnaire des concepts métiers).

- ***la nécessité de pouvoir désigner un concept par une expression (séquence de mots)***. C'est le cas de la chaîne « designate as speaker » dans la phrase (14) de la Figure 36. Dans ce cas, cette chaîne est considérée comme un concept et non comme le verbe « designate » suivi d'un complément circonstanciel « as speaker ».

2.2 Patrons de spécification RDL et expressivité du langage

Le langage RDL peut être considéré comme une syntaxe accessible pour le formalisme RM. Le RDL permet de capturer plusieurs types d'informations. Plus précisément, il permet d'instancier les vues de niveau produit (M1) et instance (M0) du métamodèle RM (cf. Tableau 4). Il est spécialisé pour la spécification du domaine d'un système d'une part (exigences et assertions environnementales) et l'instanciation de *configurations* possibles du système (états du système) d'autre part. Il offre ainsi une expressivité suffisante pour les activités V&V de simulation et de vérification formelle. Bien que contraint, le RDL accepte une grande variété de phrases comme l'illustre la Figure 36. Dans son état actuel, le langage RDL autorise la description des types d'informations suivants (chaque type d'informations est associé à une ou plusieurs vue(s) du métamodèle RM, donnée entre parenthèses) :

Informations relatives aux entités du domaine, aux états possibles de ces entités et leurs relations (vue statique). Toutes les phrases de la Figure 36 comportent des informations de ce type. A titre d'exemples : la phrase (1) définit l'existence des concepts de « participant », de « modérateur » et de « meeting », le deuxième étant un rôle pouvant être joué par le premier ; la phrase (2) définit la notion de « type » comme une propriété du concept de « meeting » ; la phrase (6) précise des valeurs possibles pour cette propriété.

Informations relatives aux actions pouvant être effectuées par les acteurs du domaine (vue fonctionnelle et vue logique). A titre d'exemples : la phrase (9) définit l'action de planification d'une réunion par un participant ; la phrase (4) décrit la durée de l'action de « connexion au serveur externe » ; les phrases (2, 3, 8) définissent des conditions d'applications et effets (liens de causalité) des actions « planifier une réunion » et « entrer dans une réunion ».

Informations contraignant le comportement du domaine (vue non-fonctionnelle et vue logique). A titre d'exemples : la phrase (5) spécifie qu'un participant ne peut avoir la parole plus de 3 minutes d'affilée dans une réunion ; la phrase (13) spécifie qu'une personne s'exprimant dans une réunion doit au préalable avoir la parole (ce qui n'est possible que si le modérateur la lui a donnée, cf. phrase (14)).

Informations sur une instance du domaine, c'est-à-dire un état particulier du domaine (vue configuration). Par exemple, la phrase (7) fait état de l'existence d'un individu Anne et décrit la valeur de deux de ses relations. La phrase (12) décrit l'existence d'une réunion et lui donne un nom.

Information précisant si les propriétés non-fonctionnelles spécifiées sont indicatives ou optatives (vue analyse). Ce type d'informations permet de distinguer ce qui décrit l'environnement de ce qui décrit les exigences. A titre d'exemple : l'utilisation de verbes modaux tels que « must » ou « shall » souligne le caractère optatif de l'information.

Chaque phrase RDL est instance d'un des patrons de spécification RDL, présentés dans le Tableau 5 (un exemple de phrase RDL est donné sous chaque patron). Ces patrons sont classés dans quatre catégories différentes : patron fonctionnel, non-fonctionnel, statique et de configuration. Cette classification correspond aux types d'informations cités plus haut (le dernier type d'informations n'est pas représenté car orthogonal aux autres[61]). Les trois premières catégories sont utiles pour la description du domaine, la dernière pour l'instanciation de celui-ci dans un but de validation par simulation par exemple. Les patrons non-fonctionnels sont un sous-ensemble des patrons proposés par Dwyer [8, 51-52] et Cheng [52]. Dans la suite, on appellera *type d'une phrase RDL* le patron dont elle est instance. Le Tableau 6 précise le type des phrases RDL de la Figure 36. Tous les patrons

[61] Plus précisément, le caractère indicatif/optatif est orthogonal au caractère fonctionnel/non-fonctionnel (on a affaire à la tyrannie d'une décomposition, d'un partitionnement dans ce cas …).

fonctionnels et non-fonctionnels peuvent être instanciés avec un des champs d'application proposés par Dwyer.

Cat.	Nom	Description
FONCTIONNELLE	ACTIONDECLARATION	Définit une action, son acteur et les concepts et/ou acteurs mis en jeu.
	« A "participant" can "ask the word" to a "moderator". »	
	ACTIONEFFECT	Spécifie un effet d'une action sur l'état du domaine une fois l'action effectuée.
	« A "participant" is the "manager" of a "meeting" after he did "plan" it. »	
	ACTIONCONDITION	Spécifie une condition sur l'état du système devant être satisfaite pour autoriser un acteur à effectuer une action.
	« When a meeting is private, a "participant" must be "connected" before he can "enter" in a "meeting". »	
	ACTIONDURATION	Spécifie la durée d'une action.
	« It takes 30 seconds for a "participant" to "connect" to the "external server. »	
STATIQUE	CARDDESCRIPTION	Spécifie les cardinalités possibles d'une relation entre entités du domaine.
	« When a "meeting" is "private", a "participant" must be "identified" before he can "enter" in this "meeting". »	
	ENUMDESCRIPTION	Spécifie les valeurs possibles d'une propriété statique.
	« The "state" of a "camera" can be "setOn" or "setOff". »	
CONFIGURATION	INSTANCIATION	Fait état de l'existence d'une instance d'une entité du domaine.
	« There is a "participant" called "Jacques". »	
	INSTANCESETTING	Définit l'état d'une instance.
	« The "type" of "meeting1" is "private". »	
	ACTIONTRIGGERING	Déclare l'activation d'une action.
	« "Jacques" does "give the word" to "Jean". »	
NON-FONCTIONNELLE	INVARIANT	Spécifie une relation de causalité indicative entre états du domaine.
	« A "participant" must not be "entered" in a "meeting" if this "meeting" is "closed". »	
	RESPONSE *	Spécifie le fait que si une condition (ou l'activation d'une action) est satisfaite, celle-ci est suivie de la satisfaction d'une autre condition (ou activation d'une action).
	PRECEDENCE *	Spécifie le fait que si une condition (ou l'activation d'une action) est satisfaite, celle-ci est précédée de la satisfaction d'une autre condition (ou activation d'une action).
	EXISTENCE *	Spécifie le fait qu'une condition sur l'état du domaine (ou l'activation d'une action) est satisfaite au moins une fois.
	ABSENCE *	Spécifie le fait qu'une condition sur l'état du domaine (ou l'activation d'une action) n'est jamais satisfaite.
	« A "moderator" of a "meeting" must not be the "manager" of this "meeting". »	
	UNIVERSALITY *	Spécifie le fait qu'une condition sur l'état du domaine est toujours vraie, pour un champ d'application donné.
	« When a "meeting" is "private", the "state" of the "camera" of a "participant" must be "setOn" if this "participant" is "entered" in this "meeting". »	
	BOUNDEDRESPONSE **	Spécifie la durée maximum séparant la satisfaction de deux conditions sur l'état du domaine (ou l'activation d'une action).

	BOUNDEDINVARIANCE **	Spécifie la durée minimum séparant la satisfaction de deux conditions sur l'état du domaine (ou l'activation d'une action).
	MINIMUMDURATION **	Décrit la durée minimum pendant laquelle une condition sur l'état du domaine doit être satisfaite.
	MAXIMUMDURATION **	Décrit la durée maximum pendant laquelle une condition sur l'état du domaine doit être satisfaite.

* : patron de spécifications de Dwyer, ** : patron de spécifications de Cheng

Tableau 5 - Les patrons de spécification RDL.

Phrase	Type	Phrase	Type
(1)	CARDINALITYDESCRIPTION	(8)	ACTIONCONDITION
(2)	ACTIONCONDITION	(9)	ACTIONDECLARATION
(3)	ACTIONEFFECT	(10)	PRECEDENCE
(4)	ACTIONDURATION	(11)	STATEINVARIANT
(5)	MAXIMUMDURATION	(12)	INSTANCIATION
(6)	ENUMERATIONDESCRIPTION	(13)	UNIVERSALITY
(7)	INSTANCESETTING		

Tableau 6 - Types des phrases RDL de la Figure 36.

Tout comme pour les patrons de spécification de Dwyer et Cheng, une sémantique formelle est associée à chacun des patrons de spécification RDL. Cette sémantique est une instance du RM puisque ce dernier est le formalisme de la plate-forme R2A. Plus précisément, on peut représenter cette sémantique comme un modèle RM abstrait où certains fragments ne sont pas encore instanciés (ces fragments sont représentés par des variables). Ces modèles sont appelés *motifs*. La Figure 37 présente les motifs associés à trois patrons RDL (deux fonctionnels et un non-fonctionnel), présentés comme une expression de la syntaxe formelle du formalisme RM. Les variables des motifs sont en italiques. Le premier motif correspond à la sémantique du patron ACTIONCONDITION.

ActionCondition : action ***name*** (***Parameter***)
pre : ***scope*** $\wedge$ ***condition***
post :

ActionEffect : action ***name*** (***Parameter***)
pre :
post : ***scope*** => ***effect***.

Invariant : ***invariant*** | ***constraint*** :
Quantifiers, ***scope*** => (***logicalExpr1*** => ***logicalExpr2***)

Figure 37 - Motifs de trois patrons RDL.

On peut donc déterminer à partir des motifs associés aux patrons RDL la sémantique de chacune des phrases RDL de la Figure 36. Pour ce faire, on procède à une affectation de chacune des variables. Par exemple, la phrase (8) spécifie une condition sur l'activation de l'action « entrer dans une réunion » et cela lorsque cette réunion est privée (cette dernière condition est le champ d'application). C'est une phrase de type ACTIONCONDITION. Après affectation des variables du motif associé, on obtient la formule (a) de la Figure 38 représentant la sémantique de la phrase (8). La phrase (3) stipule que la planification d'une réunion par un participant signifie que celui-ci en devient le gestionnaire. Cette phrase est de type ACTIONEFFECT. Sa sémantique est donnée par la formule (b). La formule (c) est obtenue de la même manière par instanciation du motif associé au patron INVARIANT. Elle représente la sémantique de la phrase (11).

On peut remarquer que la formule (b) n'est pas exactement l'instance de la sémantique formelle du patron ACTIONEFFECT donnée en Figure 37. Premièrement, cette différence est due au fait que la phrase (3) ne définit pas de champ d'application, ce qui revient à considérer la propriété toujours vraie

(équivalant à un champ d'application « Globally » pour les patrons de Dwyer et Cheng). Deuxièmement, cette différence est due à l'utilisation du verbe « become » dans la phrase (3), ce qui signifie que le participant ne peut planifier une réunion s'il en est déjà le gestionnaire. On remarque ici que tout comme les patrons de Smith [53], chaque patron RDL comporte un ensemble de variations dépendant dans notre cas de la nature des valeurs affectées aux variables (dans le cas des patrons de Schmidt, les variations sont précisées par un ensemble de phrases). Enfin, on remarque que le RDL permet de décrire les phénomènes du système dans un langage naturel contraint, contrairement aux patrons de spécification proposés dans la littérature.

(8) « When the "type" of a "meeting" is "private", a "participant" must be "identified" before he can "enter" in this "meeting". »

(a) **action** enter(p : Participant, m : Meeting)
pre : type(m)= private ∧ identified(p)

(3) « A "participant" becomes the "manager" of a "meeting" after he does "plan" this "meeting" . »

(b) **action** plan(p : Participant, m : Meeting)
pre : ¬manager(p, m)
post : manager(p, m)

(11) «A "meeting" is "closed" if and only if it is not "opened". »

(c) **assertion**
∀ m : Meeting, closed(m) <=> ¬opened(m).

Figure 38 - Sémantiques des phrases (8,3,11) de la Figure 36, obtenues à l'aide des sémantiques associées aux patrons de spécification RDL.

motif : action ***name*** (***Parameter***)
pre :
post : ***scope*** => ***effect***.
1 - action **name** (***Parameter***)
pre :
post : <u>true</u> => ***effect***.
2 - action ***name*** (***Parameter***)
pre :
post : ***effect***.
3 - action <u>plan</u> (<u>p : Participant, m : Meeting</u>)
pre :
post : <u>manager(p, m)</u>.
4 - action <u>plan</u> (<u>p : Participant, m : Meeting</u>)
pre : <u>¬manager(p, m)</u>
post : manager(p, m).

Figure 39 - instanciation manuelle, étape par étape, du patron ACTIONEFFECT pour la phrase (3) de la Figure 37.

La Figure 39 montre une manière d'instancier manuellement le patron ACTIONEFFECT pour obtenir la sémantique de la phrase (3), soit la formule (b) de la Figure 38. Cette instanciation est présentée en quatre étapes. La modification du motif effectuée pendant une étape est soulignée dans le texte. La première étape consiste à affecter la variable *scope*, la deuxième à simplifier l'expression logique. La troisième étape consiste en l'affectation des variables *action* et *effect* et la dernière en la

mise à jour de l'expression pour prendre en compte la variation associée à l'utilisation du verbe « become ».

2.3 Description du langage RDL : le métamodèle RDM

Toute spécification RDL peut être représentée par une instance du métamodèle RDM (pour *Requirements Description Metamodel*). Ce métamodèle représente la syntaxe abstraite de la grammaire RDL. Le RDM définit une grande partie des contraintes sur la nature des expressions anglaises conformes au langage RDL. Les autres contraintes sont définies par la grammaire RDL encodée avec l'outil Syntaks [160] (voir Annexe D). Les sémantiques des phrases RDL sont instances d'un autre métamodèle présenté en section 1. Les patrons de spécifications RDL introduits dans la section précédente sont représentés par le métamodèle RDM (voir Figure 89 fournie en Annexe F). On retrouve les quatre catégories de patrons représentées par les métaclasses FUNCTIONALPATTERN, STATICPATTERN, CONFIGURATIONPATTERN et NONFUNTIONALPATTERN. Cette dernière catégorie comporte deux sous-catégories, TEMPORALPATTERN et REALTIMEPATTERN. Un patron non fonctionnel fait partie de la première s'il ne fait aucune référence à une durée, au second sinon.

Nous avons vu qu'un motif comporte un ensemble de variables. Les valeurs pouvant être affectées à ces variables sont des *fragments* de phrases. Ces variables sont représentées par des relations vers des métaclasses spécifiques du métamodèle RDM permettant de capturer ces informations. La Figure 40 présente ces relations pour les patrons fonctionnels. Les trois variables du patron ACTIONEFFECT sont représentées par des relations du même nom (*scope*, *action* et *effect*) pointant vers les métaclasses FRAGMENT et CLAUSE.

Le Tableau 7 présente les fragments de quatre phrases RDL de la Figure 36. Nous avons vu en section précédente que la phrase (3) est une instance du patron ACTIONEFFECT. Ce patron comporte trois variables, à savoir un champ d'application (*scope*), la description de l'activation d'une action (*action*) et la description d'un effet sur l'état du domaine lorsque l'action est déclenchée (*effect*). La phrase (3) ne comporte pas de champ d'application comme nous l'avons vu en section précédente (cf. Figure 39). Les deux fragments de la phrase (3) donnés dans le Tableau 7 correspondent aux valeurs des variables *action* et *effect* respectivement.

Phrase	Fragments
(3)	« A "participant" becomes the "manager" of a "meeting" » « he does "plan" this "meeting" »
(5)	« a "participant" is a "speaker" of a "meeting" », « less than 3 minutes. »
(6)	« A "meeting" », « "type" », « "private" or "public" »
(8)	« the "type" of a "meeting" is "private" », « a "participant" must be "identified" », « he can "enter" in this "meeting" »

Tableau 7 - Exemples de fragments RDL.

Les fragments RDL sont instances de la partie du métamodèle RDM fournie en Figure 41. Cette partie du métamodèle RDM décrit une grammaire simple du langage naturel. On retrouve les notions traditionnelles de proposition (CLAUSE), sujet (relation *subject*), verbe (VERB), complément (COMPLEMENT), groupe nominal (NOMINALGROUP), préposition (PREPOSITION), pronom (PRONOM) … On trouve aussi les opérateurs logiques de conjonction (AND), disjonction (OR) et négation (NOT). Un groupe nominal est composé d'un déterminant (DETERMINANT) et d'un nom (NOUN), celui-ci pouvant accessoirement être muni d'un complément du nom (relation *nounCpt*).

Determinant	this, the, a, an, one, all, several, …	subjectPronoun	he, they, …
Preposition	to, from, in, by, with, on, …	disjunctivePronoun	him, it, them, …

Tableau 8 - Mot-outils de la grammaire RDL.

Le RDL accepte un choix restreint de déterminants, prépositions et pronoms (appelés mots-outils en linguistique). Le Tableau 8 présente une partie des mots-outils autorisés par la grammaire RDL. Le RDL accepte des déterminants définis (le, tous) et indéfinis (un, plusieurs). Ces déterminants permettent d'exprimer des quantités. Un grand nombre de prépositions sont autorisées (on peut en ajouter si besoin). Concernant les pronoms, on distingue les pronoms du sujet (SUBJECTPRONOUN) des

pronoms disjonctifs (DISJUNCTIVEPRONOUN) utilisés au sein d'un complément. Ces derniers permettent de référencer un concept préalablement introduit dans le sujet. On peut remarquer que certains mots-outils sont relatifs (e.g. « this » et « he ») et permettent de référencer dans une phrase RDL une notion précédemment citée dans la phrase en cours.

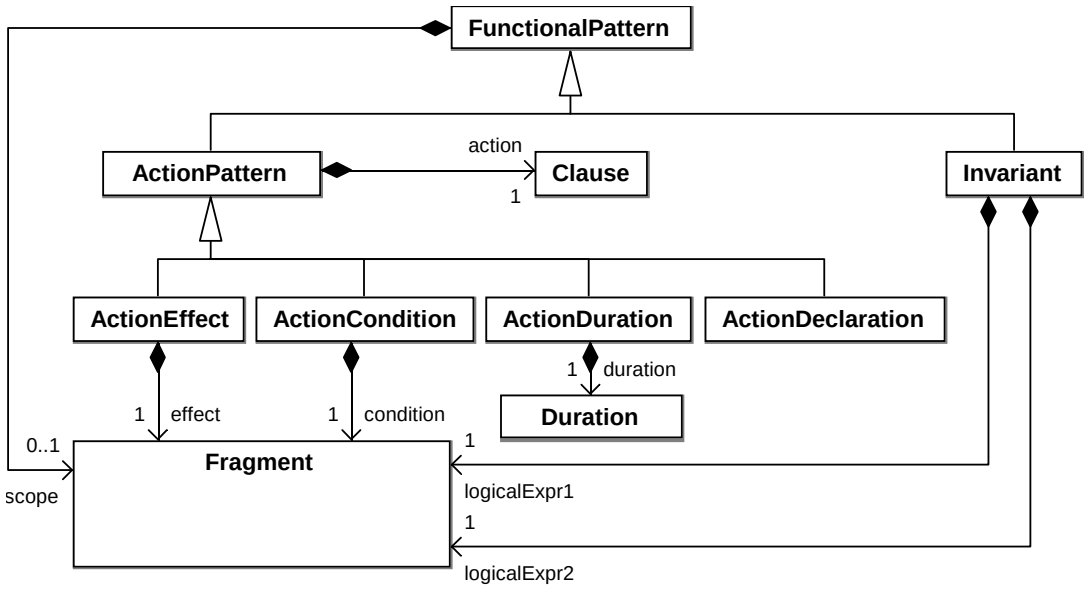


Figure 40 - Les patrons RDL fonctionnels et la représentation de leurs variables (extrait du métamodèle RDM).

Nous avons utilisé Syntaks pour générer une instance RDM à partir d'une spécification RDL. La Figure 90 en Annexe F présente un modèle conforme à RDM. C'est un modèle correspondant à la phrase (8) de la spécification RDL donnée en Figure 36. On trouve une instance de la métaclasse ACTIONCONDITION représentant le patron de spécifications RDL du même nom, ainsi que trois fragments (les arbres ayant pour racines des instances de CLAUSE). Ces fragments représentent les valeurs des trois variables *condition*, *action* et *scope* du patron ACTIONCONDITION (ces variables sont représentés par des liens du même nom).

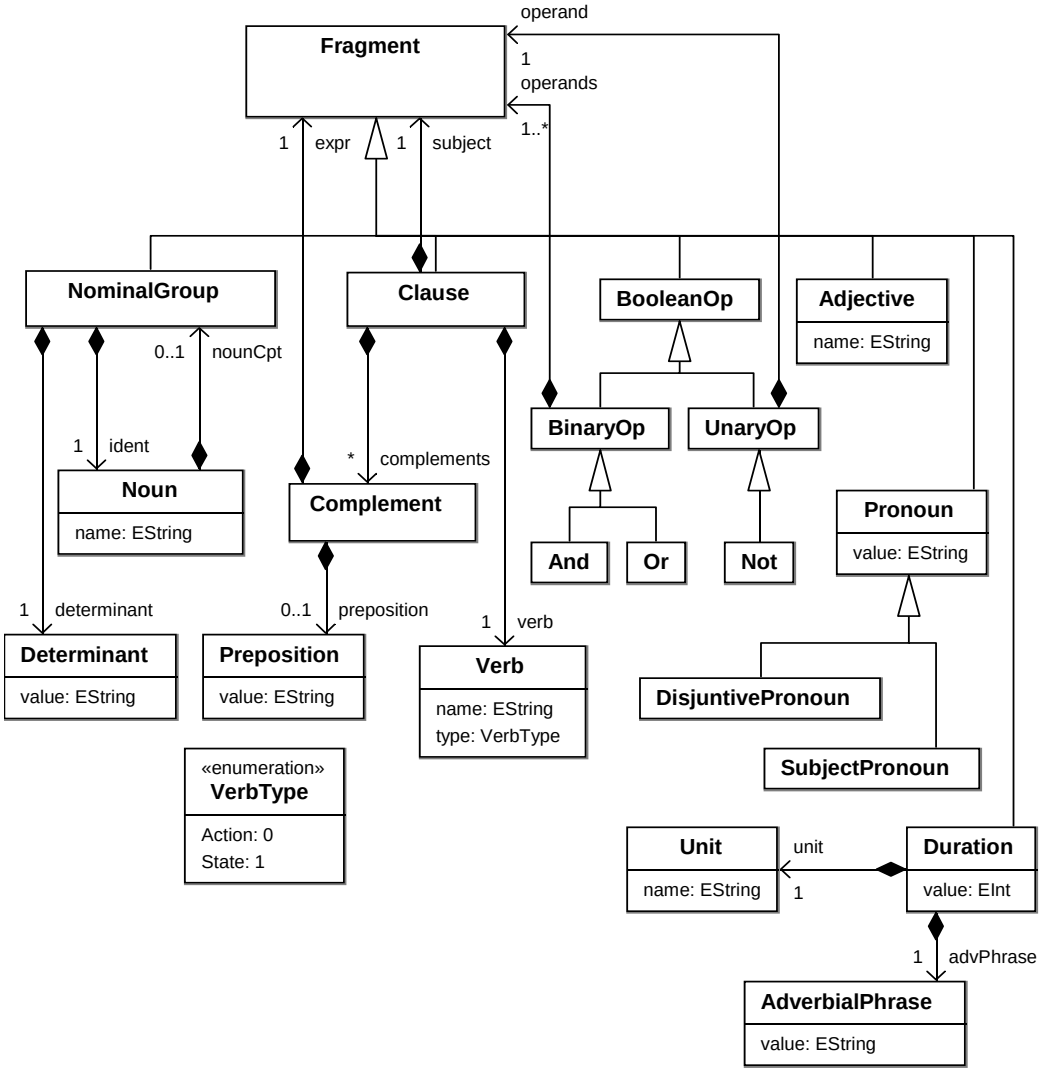


Figure 41 – Extrait du métamodèle RDM.

Chapitre V Caractérisation d'un processus de composition de spécifications partielles hétérogènes

Résumé

Ce chapitre propose une analyse de la problématique de composition de spécifications partielles et hétérogènes d'exigences. Il introduit et justifie les grandes lignes du processus de composition simple présenté dans le chapitre suivant. Ce chapitre peut être vue comme une analyse préliminaire aboutissant à un ensemble d'exigences dont le Chapitre VI donne une solution technique satisfaisante. Le vocabulaire utilisé est conforme à la terminologie définie en section 4 du Chapitre II concernant la composition de modèles. Outre l'obtention d'un modèle global des exigences, nous nous intéressons à la détection d'incohérences. La détection d'incohérences dynamiques n'est pas abordée. Nous traitons uniquement de la détection d'incohérences statiques, étant entendu que la résolution de ces dernières précède nécessairement l'analyse dynamique d'une spécification opérationnelle d'exigences.

La section 1 présente des exemples concrets de points de vue, instance de la vue classification du métamodèle RM (cf. Annexe E). Ces points de vue capturent une partie de la spécification informelle SRV fournie en section 3.2 du Chapitre III. Ils regroupent des spécifications partielles opérationnelles, conformes à différentes notations. Ces spécifications partielles regroupent elles-mêmes des modèles d'entrée conformes aux langages d'entrée supportés par la version actuelle de la plate-forme R2A. Ces points de vue sont représentatifs des spécifications rédigées par les parties prenantes après raffinement de la spécification informelle du SRV. Ils sont utilisés dans ce chapitre pour illustrer la problématique de composition ; ils servent aussi à illustrer le fonctionnement du processus de composition simple du Chapitre VI.

La section 2 fournit une analyse détaillée de la problématique en partant d'abord de considérations générales (section 2.1) pour ensuite entrer dans les détails et fournir des exemples concrets (sections 2.2 et 2.3). La section 2.1 introduit une terminologie pour le reste du chapitre et introduit les différentes préoccupations qu'un processus de composition se doit de considérer : symétrie, interprétation, traitement des superpositions, analyse et traçabilité. En outre, cette section souligne la limitation des techniques de comparaison structurelle pour la détection d'incohérences (cf. section 3.2.2 du Chapitre I) dans un contexte multi-langage. La section 2.2 détaille la problématique d'interprétation. Elle présente le problème de *variation sémantique* inhérent à toute interprétation et discute de la notion d'*ambiguïté*. Ces ambigüités doivent être résolues si l'on désire obtenir une spécification formelle. La section 2.3 détaille la problématique de traitement des superpositions sémantiques (identification et résolution).

Enfin, la section 3 résume les points importants abordés tout au long de ce chapitre. Elle fournit une liste de contraintes devant être satisfaites par un processus de composition de spécifications partielles et hétérogènes d'exigences comme le processus de composition simple du Chapitre VI.

1 Exemple d'une spécification d'exigences opérationnelles

La formalisation d'une spécification d'exigences est un processus itératif de raffinement et de modélisation à l'aide de langages munis d'une sémantique formelle. Ce processus se termine lorsque la spécification est suffisamment précise pour appliquer les techniques d'analyse (vérification, validation) ou de génération visées (test systèmes, artefacts de développements). La spécification SRV présentée dans le Chapitre III est une première version des exigences produites par les parties prenantes impliquées au sein du projet SRV. Elle n'est pas analysable à l'aide de techniques de vérification formelle car trop informelle. La Figure 42 fournit un extrait d'une spécification d'exigences opérationnelle pour le SRV, obtenue durant un processus IE supporté par la plate-forme R2A. La spécification proposée est intentionnellement incohérente et incomplète. Elle est représentative d'une spécification opérationnelle en cours de raffinement. C'est une instance de la vue classification du métamodèle RM : c'est une collection de modèles d'entrée regroupés au sein de spécifications partielles conformes à un ensemble de notations et classées par points de vue.

La spécification de la Figure 42 comporte cinq points de vue (PV1 à PV5) regroupant un ensemble de spécifications partielles (SP1 à SP6). Chaque point de vue comporte un ensemble de critères de classification (description succincte des exigences capturées, date de création du point de vue, version, partie(s) prenante(s) impliquée(s) lors de la rédaction). Par exemple, les critères de classification dans le cas du point de vue PV3 spécifient qu'il a été rédigé par Jane le 04 août 2008, a été révisé une fois (version 1.01) et capture des informations initialement décrites dans les paragraphes P4 et P5 de la spécification SRV initiale. Ces critères sont utiles pour l'organisation et la gestion des exigences. Certains travaux en IE s'intéressent à la composition de ces critères, et en particulier à ceux relatifs aux préoccupations [45-46] (par exemple la description du point de vue PV5 détermine une préoccupation fonctionnelle, à savoir l'action *leave*). Ces travaux facilite l'analyse manuelle d'une spécification mais ne fournissent pas un modèle global et opérationnel des exigences. Nous nous intéressons pour notre part à ce dernier point. Notons que notre approche est complémentaire aux approches de composition centrées sur les critères de classification.

Chaque spécification partielle respecte une notation, et à ce titre comporte un ensemble de valeurs conformes à un des champs de la spécification (cf. Annexe E). La valeur d'un champ peut être un modèle ou une valeur informelle (les modèles sont entourés d'un cadre en pointillés dans la Figure 42). Quatre notations sont utilisées dans la spécification d'exigences opérationnelles du SRV : le diagramme de classes UML, le diagramme d'activité UML[62], la spécification RDL et la version de Cockburn [161] du cas d'utilisation. A titre d'exemple, la spécification partielle SP6 du cinquième point de vue respecte la notation des cas d'utilisation de Cockburn. Cette notation comporte six champs fixes (*name*, *goal*, *primary actor(s)*, *secondary actor(s)*, *preconditions*, *postconditions*) et un ensemble fini de champs capturant chacun un scénario nominal ou exceptionnel (*nominal scenario* et *exceptional scenario*). Les quatre premiers champs sont typés informellement (du texte) ; les autres ont pour type le métamodèle RDM. Plus précisément, le type des champs pré- et post-conditions est le métamodèle RDM restreint à l'ensemble des patrons de spécification RDL fonctionnels (catégorie FUNCTIONALPATTERN, cf. section 2.2 du Chapitre IV). Le type des champs de scénarios est le métamodèle RDM restreint à l'ensemble des patrons de spécification RDL de configuration (catégorie CONFIGURATIONPATTERN).

Les notions propres à la vue classification (point de vue, notation et champ) ne servent pas uniquement à organiser et normaliser la spécification d'exigences. Elles jouent aussi un rôle dans les activités V&V, et plus généralement dans le traitement des informations capturées. Voici pêle-mêle quelques exemples de rôles joués par ces notions :

[62] On remarquera que le nom des activités est paramétré, ce qui permet de décrire une information plus précise. En fait UML décrit une famille de langages et l'usage que l'on fait d'un diagramme dépend du contexte industriel ou de la méthode employée.

VP1 : modeling of paragraph P2 in the SRV first specification.
Author and Date : George, 2008-08-05
Version : 1.0.0

-> SP1 : RDL specification

1- A “participant” is “connected” after he does “connect” to the “server” of the “system”.
2- A “participant” must be “connected” before he can “consult the meetings”.
3- A “participant” shall “consult the meetings” before he can “enter” in a “meeting”.
4- A “participant” can “ask for word” to a “moderator”.
5- A “participant” must be the “speaker” of a “meeting” before he can “send messages” to another “participant”.
6- A “participant” must be “entered” in a “meeting” before another “participant” can “send messages” to him.
7- A “participant” is not the “speaker” of a “meeting” after he does “return the word”.
8- A “participant” must be the “speaker” of a “meeting” before he can “return the word”.
9- A “participant” is not “entered” in a “meeting” after he does “leave” this “meeting”.
10- It is always the case that a “participant” is the “speaker” of a “meeting” for less than 3 minutes. (a)

VP2 : modeling of paragraph P3 in the SRV first specification.
Author and Date : Jane, 2008-08-04
Version : 1.0.0

-> SP2 : RDL specification

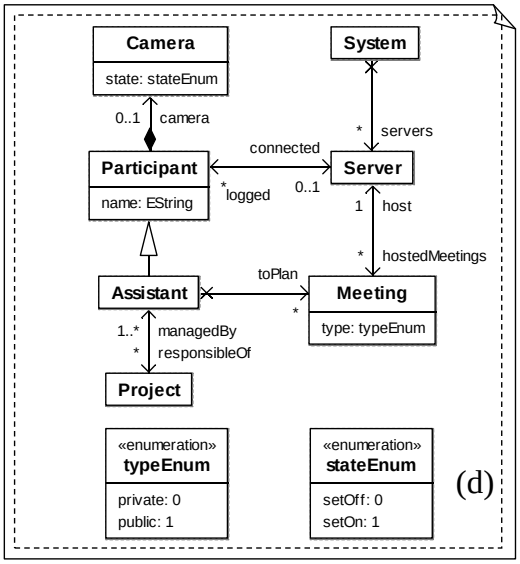
1- A “participant” is the “manager” of a “meeting” after he did “plan” it. (b)
2- A “meeting” must be “opened” before a “participant” can “enter” in this “meeting”.
3- A “participant” must be the “manager” of a “meeting” before he can “open” this “meeting”.
4- A “participant” must be “entered” in a “meeting” before the “moderator” of this “meeting” can “give the floor” to this “participant”.
5- A “participant” is not “entered” in a “meeting” if this “meeting” is “closed”.
6- It does not exist a “speaker” of a “meeting” before the “moderator” of this “meeting” can “give the floor” to a “participant”.

VP3 : modeling of paragraph P4 and P5 in the SRV first specification.
Author and Date : Jane, 2008-08-04
Version : 1.0.1

• **SP3 : RDL specification**

1- A “participant” must be “connected” to the “server” of the “system” before he can “connect” to the “external server”.
2- When a “meeting” is “private”, a “participant” must be “identified” before he can “enter” in this “meeting”.
3- When a “meeting” is “private”, a “participant” must have a “camera” and the “state” of this “camera” must be setOn before this “participant” can “enter” in this “meeting”.
4- It takes 30 seconds for a “participant” to connect to the “external server”.
5- When a “participant” does “speak” in a “meeting”, it is always the case that this “participant” is the “speaker” of this “meeting”.
6- When a “participant” is the “speaker” of a “meeting”, it is never the case that it exists another “speaker” of this “meeting”. (c)

• SP4 : UML **Class diagram**



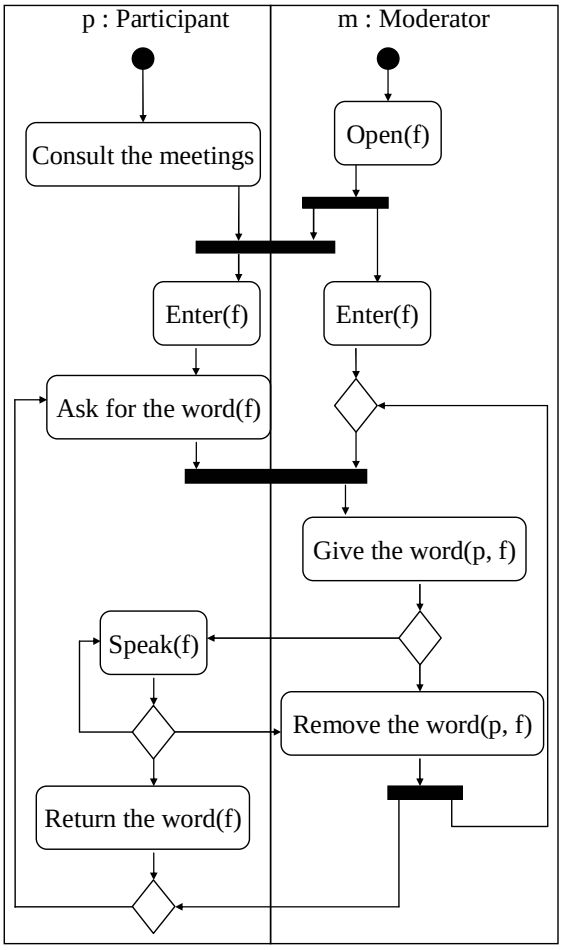
VP4 : Interaction between a participant and a moderator concerning speaking.
Author and Date : Bob and Anne, 2008-08-14
Version : 1.0.0

-> SP5 : UML Activity diagram

- scope :

1- p is “connected”.
2- m is “connected”.
3- There exists a “forum” called “f”. (e)

- diagram **:**



(f)

VP5 : Action “leave a meeting”
Author and Date : Jim, 2008-08-23
Version : 1.0.1

-> SP6 : Cockburn use case

- name : Leave

- goal : A participant who takes part in a meeting can leave it at any moment.

- primary actor : Participant

- secondary actor(s) : Meeting

- preconditions :

1- A “participant” must be “entered” in a “meeting” before he can “leave” this “meeting”. (g)

- postconditions :

1- A “participant” is not “entered” in a “meeting” after he does “leave” this “meeting”.
2- A “meeting” is “closed” after the “moderator” of this “meeting” did “leave” it.
3- The “state” of the “camera” of a “participant” must be “setoff” after he does “leave” a “meeting”. (h)

- nominal scenario 1 :

1- There is a “meeting” called “full session 2008”
2- There is a “meeting” called “Saturday night”.
3- There is a “participant” called “Jean”.
4- “Jean” is the “speaker” of “Saturday night”.
5- There is a “moderator” called “Jacques”.
6- “Jacques” is the “moderator” of “full session 2008”.
7- “Jacques” does “gives the word” to “Jean”.
8- “Jean” does “speak” in “full session 2008”.
9- “Jean” does “return the word” in “full session 2008”.
10- “Jean” does “speak” in “Saturday night”.
11- “Jean” does “leave” “full session 2008”. (i)

- exceptional scenario 1 : (j)

1- There is a “meeting” called “full session 2008”
2- There is a “participant” called “Jean”.
3- There is a “moderator” called “Jacques”.
4- “Jacques” does “gives the word” to “Jean”.
5- “Jacques” does “leave” “full session 2008”.
6- “Jean” does “speak” in “full session 2008”.

point de vue (PV)
spécification partielle (SP)
modèle

Figure 42 – Cinq points de vues regroupant un ensemble de spécifications partielles représentant des exigences du système SRV présenté informellement en section 3.2.

N.B. : Dans la suite, nous désignerons les phrases RDL de la Figure 42 par la référence du modèle, un tiret, suivi des numéros de phrases séparés par des barres. Par exemple, « a-1 » désigne la première phrase RDL du modèle (a), « a-1|2 » les deux premières phrases du même modèle.

- Les valeurs des champs *scénario* de la notation des cas d'utilisation de Cockburn peuvent être validées par une simulation fonctionnelle. Le scénario nominal 1 du cas d'utilisation l*eave* (modèle (i)) décrit en effet une configuration initiale du domaine et une succession d'actions. Une simulation fonctionnelle de ce scénario permettrait de détecter une incohérence, puisque le participant Jean ne peut prendre la parole dans une réunion (i-10) après avoir rendu la parole (i-9) d'après le diagramme d'activité du point de vue VP4.
- Les valeurs du champ *diagram* de la notation diagramme d'activité UML peuvent servir à la description d'un ensemble de contraintes sur le comportement du domaine ou peuvent être utilisées pour générer des scénarios (ou des objectifs de test système).
- Le champ *scope* de la notation diagramme d'activité UML détermine pour quels états du domaine les contraintes décrites par le diagramme sont vraies (on a ici un exemple de relation logique entre deux champs). Ce champ est similaire dans sa fonction au champ d'application d'un patron de spécifications : il définit un contexte de validité. Les valeurs du champ *scope* peuvent aussi servir à la création d'états initiaux pour la simulation du diagramme. On peut aussi vérifier leur existence.
- Un champ peut aussi être utile pour la détection d'incohérences. Par exemple, le fait qu'il ne soit fait mention d'une action ou d'un état uniquement dans un scénario ou dans le champ *scope* d'un diagramme d'activité (en somme dans une information de niveau instance) peut être considéré comme une incohérence de niveau processus. Dans ce cas, la règle de cohérence associée fait explicitement référence à un champ particulier (le champ *scenario* joue ici un rôle de filtre de l'information).

2 Analyse de la problématique

Nous présentons dans cette section une analyse de la problématique de composition de spécifications partielles d'exigences. La section 2.1 décrit les grandes lignes d'un processus de composition de spécifications partielles d'exigences. Les sections suivantes détaillent et illustrent les problématiques identifiées lors de l'élaboration du processus de composition simple proposé en Chapitre II. La section 2.2 se focalise sur l'*interprétation* des informations capturées par les spécifications partielles. Les notions d'*ambiguïté* et de *variations sémantiques* sont discutées et illustrées. La section 2.3 discute des liens sémantiques entre des spécifications partielles. La notion de *superposition sémantique* est illustrée et les problèmes causés par les *collisions structurelles* sont évoqués.

2.1 Description générale d'un processus de composition

Nous caractérisons dans cette section les préoccupations propres à un processus de composition de spécifications partielles d'exigences. Ces préoccupations sont illustrées à l'aide d'un exemple de composition fourni par la Figure 43. Cet exemple illustre la composition de neuf modèles d'entrée, provenant des spécifications partielles de la spécification SRV de la Figure 42 : sept phrases RDL, un extrait du diagramme d'activité de la spécification partielle SP5 et un extrait du diagramme de classes de la spécification partielle SP4. Le modèle global RM résultant est présenté de manière textuelle. Ce modèle ne représente pas les incohérences détectées ni les différentes interprétations possibles des neufs modèles d'entrée. Il correspond au modèle global obtenu après application du processus de composition simple, comme nous le verrons en en Chapitre VI.

Nous avons identifié cinq préoccupations propres à un processus de composition de spécifications partielles d'exigences : (i) la symétrie, (ii) l'interprétation des informations à composer (les modèles d'entrée), (iii) le traitement des superpositions sémantiques entre éléments d'une information morcelée, (iv) l'analyse d'une information unifiée (le modèle global) potentiellement incohérente et (v) la traçabilité. Voici une première description de ces préoccupations (les préoccupations (ii), (iii) et (iv) sont plus étudiées dans les sections suivantes) :

Symétrie. Un processus de composition doit être symétrique, c'est-à-dire que le résultat doit être identique pour un ensemble de spécifications partielles, quel que soit l'ordre de composition de ces modèles. La symétrie est importante car un processus de composition des exigences est itératif et peut être distribué [162]. Concrètement, cela signifie qu'un processus de composition ne doit pas être sensible à l'ordre de composition des spécifications partielles, ces dernières pouvant être décrites à différents moments du développement par des équipes travaillant ne se concertant pas constamment. Plus formellement, on doit avoir pour tout processus de composition φ et pour tous modèles d'entrée a, b et c les égalités suivantes :

réflexivité : $\varphi(a, b) = \varphi(b, a)$

associativité : $\varphi(a, \varphi(b, c)) = \varphi(\varphi(a, b), c)$

Interprétation. Un processus de composition est une transformation réalisant autant de fonctions sémantiques qu'il y a de langages d'entrée. Dans le cas général, le formalisme est un langage différent des langages d'entrée. En conséquence, un processus de composition effectue la traduction des informations capturées par les modèles d'entrée dans les termes du formalisme. Cette activité est appelée interprétation [41]. Tout comme les fonctions sémantiques des langages d'entrée, l'interprétation doit être homéomorphe puisqu'un processus de composition conserve le sens des informations composées.

L'interprétation d'une information capturée par un modèle d'entrée peut fournir plusieurs informations relatives à des préoccupations diverses. Ces informations peuvent être représentées à des endroits différents du modèle global. Par exemple, le fragment de la phrase (b) « the *moderator* of this *meeting* » signifie (i) qu'il existe un concept métier *meeting*, (ii) un rôle *moderator* dépendant de *meeting* et (iii) qu'une réunion ne peut avoir qu'un modérateur (puisque l'on a le déterminant *the*). Inversement, une même information au sein du modèle global peut être décrite par plusieurs fragments de modèles d'entrée. Par exemple, l'acteur *participant* est désigné dans les modèles (a, c, g, h) de la Figure 43 (*participant* est aussi désigné deux fois par le modèle (d) puisque c'est un extrait de l'activité d'une instance de *participant*).

L'interprétation d'une information est réalisée dans un contexte. Ce contexte peut être local. C'est le cas du modèle (a) par exemple puisque ce modèle seul est suffisant pour interpréter l'information qu'il capture. En revanche, le modèle (d) doit être interprété dans un contexte plus large car un diagramme d'activité est défini pour un champ d'application donné (champ *scope*). Dans ce cas, l'interprétation dépend aussi de la notation sous-jacente et des relations logiques entre champs (ce contexte reste néanmoins local). L'interprétation peut nécessiter une connaissance de l'ensemble de la spécification (on parle alors de contexte global). En effet, un modèle d'entrée capture une information partielle par définition. Le fait que *moderator* soit un rôle n'est pas déductible de la seule connaissance du modèle (a) par exemple. Cette connaissance globale n'est pas toujours possible car (i) le processus IE peut est distribué (effectué en parallèle par différentes équipes) et (ii) un processus IE est itératif (l'information est identifiée et composée au fur et à mesure).

Enfin, il peut exister plusieurs interprétations possibles pour une même information. A titre d'exemple, l'information capturée par le modèle d'entrée (d) ne précise pas si l'activation de l'action *Ask for the word* doit être immédiatement précédée de l'activation *Enter*. Il peut donc exister des variations sémantiques (et donc des ambiguïtés), ce qui doit n'est pas compatible avec l'obtention d'une spécification opérationnelle (c'est-à-dire formelle). Ce problème est analysé en détails dans la section 2.2.

Traitement des superpositions. Certaines informations capturées par le modèle global proviennent de l'interprétation de plusieurs informations provenant de modèles d'entrée distincts. On a alors affaire à une superposition sémantique (cette notion est précisément définie en sections 2.3). Par exemple, les fragments « a *moderator* » de la phrase (a) et « the *moderator* of this *meeting* » de la phrase (b) de la Figure 43 se superposent sémantiquement puisqu'ils désignent le même élément du domaine, à savoir le rôle de modérateur. Quelle que soit la manière de composer les modèles d'entrée, ces superpositions doivent être identifiées de sorte que le modèle global ne représente qu'une et une seule fois un élément du domaine (le modèle global ne doit pas contenir de redondances, contrairement à une information morcelée). Deux approches peuvent être envisagées pour détecter des superpositions sémantiques :

- ***Comparaison avant interprétation***. Cette approche consiste en la comparaison des modèles d'entrée. Cette approche est similaire aux techniques de comparaison structurelle. Dans ce cas, les modèles comparés sont des expressions des langages d'entrée.

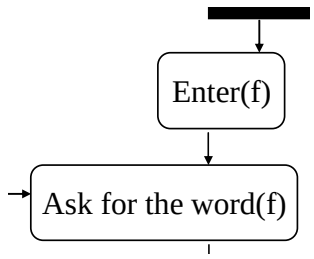
Ψ

A *participant* can "*ask for the word*" to a *moderator*. (a)

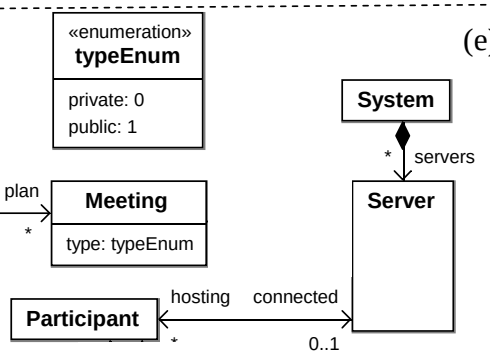
A *meeting* becomes *closed* after the *moderator* of this *meeting* did *leave* it. (b)

The *state* of the *camera* of a *participant* must be *setOff* after he does *leave* a *meeting*. (c)

(d)



(e)



A *moderator* must be *connected* before he can *enter* in a *meeting*. (f)

A *meeting* must be *opened* before a *participant* can *enter* in this *meeting*. (g)

A *participant* must be *connected* to the *server* of the *system* before he can *connect* to the "*external server*". (h)

When a *meeting* is *private*, a *participant* must be "*authorized to moderate*" in this *meeting* if he is a *moderator* of this *meeting*. (i)

=

Action *ask for the word*(p_1 : *Participant*, p_2 : *Participant*, m : *Meeting*)
pre p_2 is *Moderator*(m) ∧
connected(p_1) => *didEnter*(p_1, m)
post ¬ *didEnter*(p_1, m)
Action *leave* (p : *Participant*, m : *Meeting*)
pre ∃ c : *Camera*, of(c, p) ∧ (p is *Moderator* => *closed*(m))
post (p is *Moderator*(m) => *closed*(m)) ∧ *state*(c) = *setOff*
Action enter (p : *Participant*, m : *Meeting/Forum*)
pre *connected*(p) ∧ *opened*(m)
∧ (p is *Moderator* => ∃ s : *Server*, *connected*(p, s))
post *connected*(p) => *didEnter*(p, m)
Assertion :
∀ p : *Participant*, ∀ s : *Server*, *connected*(p, s) <=> *hosting*(s, p)
Assertion :
∀ p : *Participant*, ∀ s : *Server*, ¬*connected*(p, s) <=> ¬*hosting*(s, p)

Figure 43 – Exemple de composition de neuf modèles d'entrée de la spécification SRV.

- ***Comparaison après interprétation***. Cette approche consiste en la comparaison des modèles résultant de l'interprétation. Dans ce cas, les modèles comparés sont des expressions du formalisme. Cette approche nécessite l'application des fonctions sémantiques de chacun des langages d'entrée, activité de tout processus de composition comme on l'a vu précédemment.

Nous choisissons la deuxième approche car le nombre d'expressions nécessaires à la détection d'un type de superpositions ne dépend pas du nombre de langages d'entrée supportés par le processus de composition. A titre d'exemple, supposons que l'on veuille détecter les superpositions sémantiques relatives à l'action *enter* pour les modèles de la Figure 43. Dans le cas de la première approche, il est nécessaire de spécifier autant d'expressions qu'il existe de paires de langages d'entrée pour détecter les superpositions relatives aux actions puisque la représentation d'une même information dépend de la syntaxe du langage utilisé[63]. A titre d'exemple, si l'on considère uniquement le RDL et le diagramme d'activité UML, il faut spécifier (i) une expression pour deux modèles instances du RDL, (ii) une deuxième pour deux modèles instances du diagramme d'activité et (iii) une troisième pour deux modèles instances du RDL et du diagramme d'activité respectivement. Cette dernière permet la détection de la superposition entre l'activité *Enter* du modèle (d) et le verbe *enter* du modèle (f).

	nombre d'expressions pour la détection d'un nouveau type de superpositions	nombre d'expressions à rédiger pour l'ajout d'un langage d'entrée
avant interprétation	$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$	$m(n+1)$
après interprétation	1	1

avec : - n le nombre de langages d'entrée
- m le nombre de type de superpositions détectables

Tableau 9 – Différence entre comparaison avant et après interprétation.

En extrapolant pour n langages d'entrée, le nombre d'expressions pour identifier une superposition relative à un type d'éléments du domaine (type de superpositions) croît de manière exponentielle avec le nombre de langages d'entrée. Sans compter que l'ajout d'un langage d'entrée implique la rédaction de nouvelles expressions de sorte que les superpositions déjà identifiables puissent l'être aussi. Dans le cas de la deuxième approche, il suffit de ne spécifier qu'une seule expression quel que soit le nombre de langages d'entrée. Les formules du Tableau 9 proposent une comparaison quantifiée de la différence entre les deux approches en fonction d'un nombre de langages d'entrée *n* ; la colonne de gauche quantifie le nombre d'expressions nécessaires pour la spécification d'un type de superpositions, la colonne de droite quantifie le nombre d'expressions supplémentaires devant être rédigées si l'on ajoute un langage d'entrée au processus (avec *m*, le nombre de types de superpositions pouvant être détectables).

Analyse du modèle global. La construction d'un modèle global révèle les liens sémantiques implicites entre informations composées et fournit une représentation unifiée de la spécification d'exigences. Cette représentation unifiée facilite la compréhension des exigences et la détection d'incohérences (détection manuelle ou par algorithme). Manuellement, il est plus aisé d'étudier l'action *leave* en inspectant le modèle global de la Figure 43 qu'en étudiant les modèles d'entrée, par exemple (c'est un des avantages de Theme/doc [44], bien que ce dernier ne produise pas une spécification opérationnelle). D'un point de vue algorithmique, une représentation unifiée est gage d'une moindre complexité.

Considérons un algorithme particulièrement simple comme une règle de cohérence vérifiant si toute action comporte un paramètre jouant le rôle d'acteur. Cette règle de cohérence vise la détection d'une sous-spécification (absence d'un acteur pour une action), ce que ne permettent pas les techniques de comparaison structurelle [33-38] puisqu'elles ne comparent que deux à deux les

[63] On remarquera d'ailleurs que les approches de comparaison de modèles ne sont jamais illustrées dans la littérature avec plusieurs langages de spécifications d'exigences.

modèles d'entrée. La règle peut être spécifiée par une simple expression déclarative comme l'expression OCL (a) ci-dessous s'il existe un modèle global. En revanche, le même algorithme est beaucoup plus complexe s'il est appliqué sur une collection d'instances RM résultant de l'interprétation de modèles d'entrée où les redondances n'ont pas été supprimées. Dans ce cas, la spécification est impérative et nécessite de naviguer l'ensemble des instances de la métaclasse ACTION (voir l'implémentation en Kermeta[64] (b)).

(a)
```
∀ a : Action, ∃ p : Parameter | p ∈ a.parameters ∧ p.role = "actor"
```

(b)
```
var h : Hashtable<String, Set<Action>> init Hashtable<String, Set<String>>.new()
Action.allInstances.each{a |
        var s : Set<Action> init h.getValue(a.name)
        if (s != void) then s.add(a) else h.put(a.name, Set<Action>.new().add(a)) end
}
from var it : Iterator<Set<Action>> init h.valueIterator
until it.isOff loop
        var cpt : Integer init 0
        it.next.each{a |
                a.parameters.each{p |
                        if (p.role == "actor") then cpt:= cpt + 1 end
                }
                if (cpt < 1) then raise Exception end
        }
end
```

Le choix de comparer les modèles d'entrée après interprétation et de spécifier les règles de cohérence sur un modèle global implique de devoir construire un modèle potentiellement non-conforme au formalisme. Nous verrons dans la section 2.3 que la composition de modèles d'entrée incohérents entre eux provoque la production d'un modèle global non-conforme au formalisme mais sur lequel les règles de cohérences doivent être évaluées.

Traçabilité. Il est important de maintenir des liens de traçabilité entre les modèles d'entrée et le modèle global. En effet, le diagnostic résultant de l'analyse d'une spécification d'exigences porte initialement sur les éléments du modèle global. Ce dernier étant difficilement compréhensible par les parties prenantes, le diagnostic d'une analyse doit être décrit dans les termes des langages d'entrée. La connaissance des liens de traçabilité entre éléments des modèles d'entrée et les éléments du modèle global permet de localiser les parties des spécifications partielles incriminées lors du diagnostic. Ces liens de traçabilité sont de plus importants si le modèle global est lié aux artefacts de développement logiciel (approche de développement IDM), en particulier si les exigences évoluent souvent durant la phase de développement logiciel.

2.2 Interprétation

Un processus de composition nécessite l'application d'un ensemble de fonctions sémantiques traduisant les modèles d'entrée dans les termes du métamodèle coeur[65]. Nous montrons en section 2.2.1 que cette traduction n'est pas entièrement automatisable car elle comporte généralement un ensemble d'*ambiguïtés*. Ces ambiguïtés résultent de *variations sémantiques* [41, 114, 118-119] au niveau de la sémantique des langages d'entrée ou résultent de la décomposition de la spécification

[64] Kermeta favorise la concision du code grâce aux clôtures transitives « à la OCL ». Un code exprimée dans un langage plus généraliste comme Java serait encore plus long.

[65] Le formalisme RM utilisé comme langage d'entrée (sous sa forme textuelle) est un cas particulier. Dans ce cas, il n'est pas nécessaire de l'interpréter.

d'exigences en une collection de spécifications partielles. Ces variations sémantiques ont pour conséquence une variété d'interprétations différentes pour une même collection de spécifications partielles. Nous présentons et illustrons en section 2.2.2 les causes possibles de variations sémantiques.

2.2.1 Notion de d'ambiguïté, d'interprétation et de variation sémantique

Les langages utilisés pour exprimer les premières exigences d'un système sont généralement des langages de haut niveau d'abstraction, ce qui facilite leur utilisation par les parties prenantes et limite la taille de la spécification. Les langages à base de patrons de spécification comme le RDL en sont un bon exemple. Cependant, ce haut niveau d'abstraction ainsi que la décomposition d'une spécification en une collection de spécifications partielles favorisent la production de spécifications ambiguës. Ces ambiguïtés conduisent à plusieurs interprétations acceptables pour une même spécification, suivant la compréhension de chacune des parties prenantes. Dans ce contexte, une spécification définit une variété de spécifications d'exigences possibles, de même qu'une ligne de produits représente un ensemble de solutions pour un produit. Le raffinement de cette spécification permet de limiter ce nombre de spécifications possibles, jusqu'à l'obtention d'une seule spécification opérationnelle. Nous définissons ci-dessous les notions d'ambiguïté et d'interprétation d'une spécification.

Définition – *Ambiguïté* : Source de divergence quant à l'interprétation d'une spécification par les parties prenantes. Une ambiguïté reflète un manque d'information impliquant l'existence possible de plusieurs interprétations possibles d'une même spécification.

Définition – *Interprétation* : résultat possible (c'est-à-dire raisonnable) d'un processus de composition pour une collection de spécifications partielles. Une interprétation est donc un modèle global (dans notre cas une spécification opérationnelle d'exigences) satisfaisant chacune des spécifications partielles composées.

f (A "participant" is not "entered" in a "meeting" after he does "leave" this "meeting".)

= leave (p : Participant, m : Meeting)
post not entered(p, m).

Figure 44 - Exemple d'application d'une fonction sémantique du langage RDL où le domaine sémantique est le langage RM de la plate-forme R2A.

La Figure 44 présente le résultat de l'application d'une fonction sémantique f : RDL → RM de la phrase RDL h-1 de la Figure 42. Dans ce cas, c'est aussi le résultat du processus de composition, puisque appliquée à un seul modèle d'entrée (une phrase RDL). Dans cet exemple, la phrase RDL n'est pas ambiguë car son sens ne porte pas à confusion et l'instance RM résultante représente la seule interprétation acceptable. En effet, la phrase RDL signifie précisément que le fait qu'un participant quitte une réunion a pour conséquence immédiate le fait que ce participant n'est plus dans cette réunion (ce que décrit formellement l'instance RM présentée textuellement dans la Figure 44). Si l'on reprend la définition d'une fonction sémantique proposée dans la section 1 du Chapitre II, on a donc card (f(h-1)) = 1.

Ce n'est pas le cas de la phrase RDL a|3 de la Figure 42, instance du patron de spécifications PRECEDENCE. La Figure 45 donne une représentation textuelle de deux instances de RM correspondant aux interprétations possibles pour cette phrase dans le domaine sémantique RM. La phrase RDL a|3 ne précise pas si un participant doit préalablement consulter les réunions à chaque fois qu'il entre dans une réunion, ou s'il doit le faire une fois pour toute. Il existe deux interprétations possibles par la fonction sémantique f : RDL → RM. On a card(f(a-3)) = 2. La première interprétation correspond au modèle RM avec la post-condition de l'action *enter*, la deuxième sans cette post-condition. Pour cet exemple, on a donc une ambiguïté causée par une variation sémantique relative à la post-condition. La notion de variation sémantique est définie ci-dessous :

Définition – *Variation sémantique* : Elément variable d'une fonction sémantique f : A → B. Les modèles de A dont l'interprétation dépend d'une variation sémantique sont ambigus par définition. Un langage comportant une ou plusieurs variation(s) sémantique(s) n'est formel que s'il est possible de

déterminer automatiquement quelle variation choisir pour toute interprétation (dans le cas contraire, certaines expressions sont ambiguës).

Une variation sémantique pour une syntaxe implique l'existence d'une variété de fonctions sémantiques pour cette syntaxe et donc l'existence d'un ensemble de langages différents partageant cette dernière. Ces langages contiennent les mêmes expressions (car partageant la même syntaxe), mais conduisent à la production d'interprétations différentes pour certaines de leurs expressions. Dans la mesure où nous visons la production d'une spécification opérationnelle, les ambiguïtés doivent être résolues, c'est-à-dire qu'un choix doit être effectué pour chaque variation sémantique impliquée dans l'interprétation d'un modèle d'entrée ambiguë. Ces choix font partie intégrante d'un processus de raffinement effectué durant le processus de composition.

f ⟨ A "participant" shall "consult the meetings" before he can "enter" in a "meeting". ⟩

= enter(p : Participant, m : Meeting)
pre didConsultTheMeetings(p)
post ¬ didConsultTheMeetings(p).

consultTheMeetings(p : Participant)
post didConsultTheMeetings(p).

optionnel

Figure 45 - Exemple d'une spécification ambiguë due à une sémantique semi-formelle.

La section suivante classifie les causes de variations sémantiques que nous avons identifiées au sein des spécifications d'exigences étudiées. La section 2.2.3 énumère les techniques possibles pour raffiner une spécification ambiguë.

2.2.2 Cause d'une variation sémantique d'un langage

Cette section présente les quatre causes de variations sémantiques que nous avons identifiées. La première cause est un *flou sémantique* dû à la nature semi-formel d'un langage d'entrée. La deuxième cause est une *imprécision syntaxique*, c'est-à-dire un manque de précision de la syntaxe d'un langage d'entrée. La troisième cause est l'utilisation d'une même syntaxe dans des contextes différents (*interprétation contextuelle*), en particulier lorsqu'un langage est utilisé au sein de notations différentes. Enfin la dernière cause est une *décomposition de l'information* ne respectant pas les mécanismes de composition supportés par le formalisme. Dans tous les cas, l'existence de variations sémantiques est le résultat d'un manque d'information. Ces causes sont décrites et illustrées ci-dessous :

Flou sémantique. Certains langages de modélisation comportent intentionnellement des variations sémantiques. C'est le cas des langages semi-formels comme UML pour lequel l'OMG spécifie la sémantique à l'aide d'une instance du MOF et d'un ensemble de contraintes OCL. Cette spécification décrit la sémantique statique UML, mais sa sémantique dynamique reste en grande partie informelle. Chauvel [163] donne plusieurs exemples de variations sémantiques pour le diagramme d'états UML. La Figure 45 fournit un exemple d'interprétations multiples ayant pour cause un flou sémantique. Un langage comportant des variations de ce type peut être considéré comme une famille de langages partageant la même syntaxe mais différant pour une partie de leurs fonctions sémantiques[66].

Imprécision syntaxique. Le manque de précision de la syntaxe d'un langage peut être à l'origine de variations sémantiques. C'est le cas du RDL par exemple. La phrase RDL (c) de la Figure 43 illustre ce cas : il est impossible de déterminer automatiquement si *state* est une entité ou une propriété de l'entité *camera* (pour s'en convaincre, on peut remplacer *state* par *button*). La Figure 48 présente une petite partie du modèle RM obtenu après interprétation de cette phrase suivant que *state* est interprété comme une entité ou comme une propriété. La différence entre une imprécision sémantique et flou

[66] Ces fonctions sémantiques sont similaires en de nombreux points puisqu'elles portent sur une même syntaxe

sémantique est subtile. Elle tient dans le fait que seule une des interprétations est raisonnable dans le cas d'une imprécision syntaxique contrairement à un flou sémantique.

Interprétation contextuelle. Dans certains cas, l'interprétation d'un modèle d'entrée peut dépendre d'autres modèles d'entrée. A titre d'exemple, la notation respectée par la spécification partielle SP5 de la Figure 42 (notation diagramme d'activité) définit une relation sémantique entre ses champs *scope* et *diagram* : le premier définit dans quelles configurations du système les informations capturées par le deuxième sont correctes. L'interprétation du modèle d'entrée (e) de la Figure 42 est donc liée à l'interprétation du modèle d'entrée (f) de la même figure puisque les phrases RDL de (e) désignent des éléments de (f) (voir les terminaux « p », « m » et « f »). La même phrase est interprétée différemment si elle fait partie d'un champ *scenario* de la notation cas d'utilisation de Cockburn.

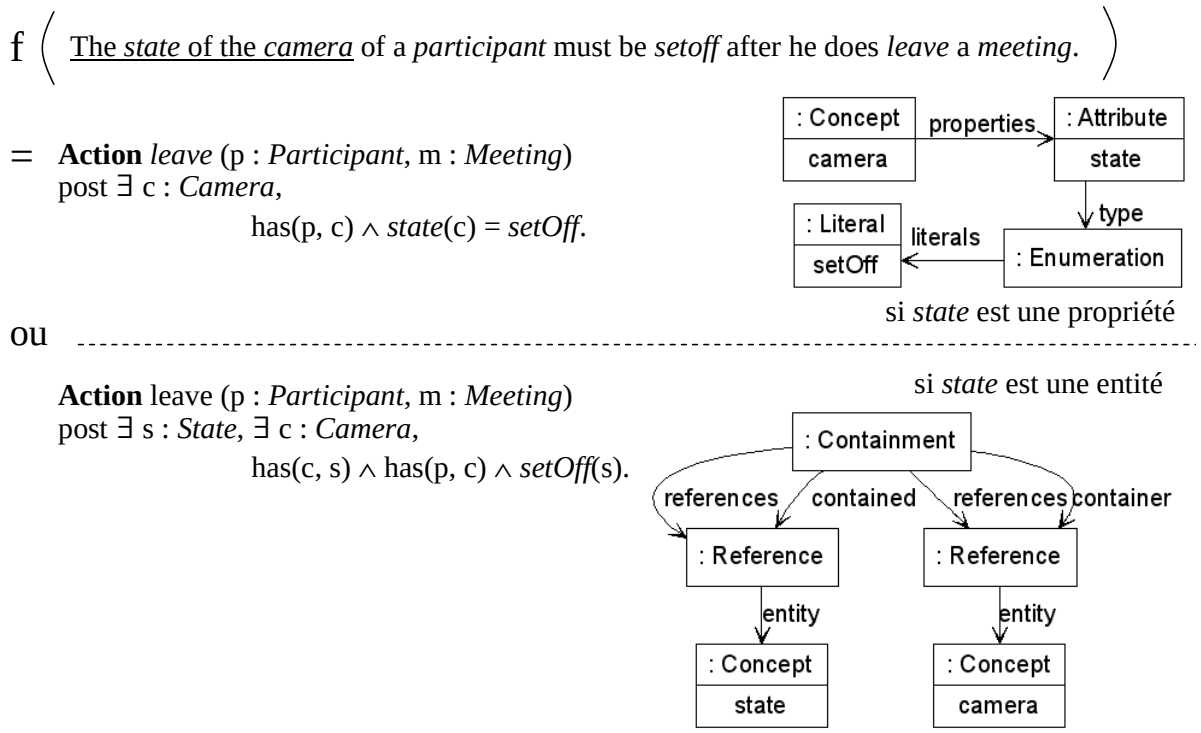


Figure 46 – Illustration d'une ambiguïté ayant pour cause une imprécision syntaxique.

Décomposition de l'information. Nous avons vu en section 2.1 du Chapitre I que les langages de modélisation offrent un ensemble de mécanismes d'abstraction pour décomposer et classer l'information. Ces mécanismes permettent de spécifier formellement une spécification à l'aide d'une collection de spécifications partielles. L'utilisation de langages d'entrée offrant des mécanismes d'abstraction différents du formalisme peut être à l'origine de variations sémantiques. A titre d'exemple, il est possible de spécifier deux actions différentes dans deux spécifications partielles RM distinctes, mais chaque action doit être entièrement décrite dans une même spécification partielle. Ce n'est pas le cas du RDL qui favorise la description d'une action à l'aide de plusieurs phrases RDL. L'interprétation de ces modèles produit un ensemble de spécifications RM partielles portant sur une même action, ce qui n'est pas supporté par le RM (le RM ne supporte pas de mécanisme de composition systématique pour les actions). Dans ce cas, la description de cette action est incomplète car il existe une ambiguïté sur le résultat de la composition des spécifications RM obtenues par interprétation.

La Figure 43 fournit deux exemples de variations sémantiques causées par la décomposition de l'information ; un concernant l'aspect dynamique de l'information (au sens de Mylopoulos [62]) et l'autre l'aspect statique. La pré-condition de l'action *enter* est décrite par deux modèles d'entrée distincts (les modèles (f) et (g)). Le résultat de leur composition est une conjonction comme le montre l'instance RM produite (ligne 9). Cette conjonction ne peut être déduite de ces modèles d'entrée (le choix d'une disjonction satisfait aussi les deux modèles). Cet exemple de variation sémantique concerne l'aspect dynamique de l'information. Un autre exemple concerne la notion de modérateur.

Pris dans son ensemble, il est possible de déduire de la spécification de la Figure 43 que *moderator* est un rôle de participant. Cependant, ce fait n'est pas déductible des modèles d'entrée pris séparément. En effet, l'analyse du modèle (a) suggère que *moderator* est un concept et l'analyse du modèle (b) suggère que *moderator* est un rôle mais ne précise pas de quelle entité. La Figure 47 présente deux extraits de modèles RM résultant de l'interprétation séparée des modèles (a) et (b). Ces fragments représentent l'interprétation de la notion de *moderator* pour ces deux modèles. Cet exemple de variation sémantique concerne l'aspect statique de l'information.

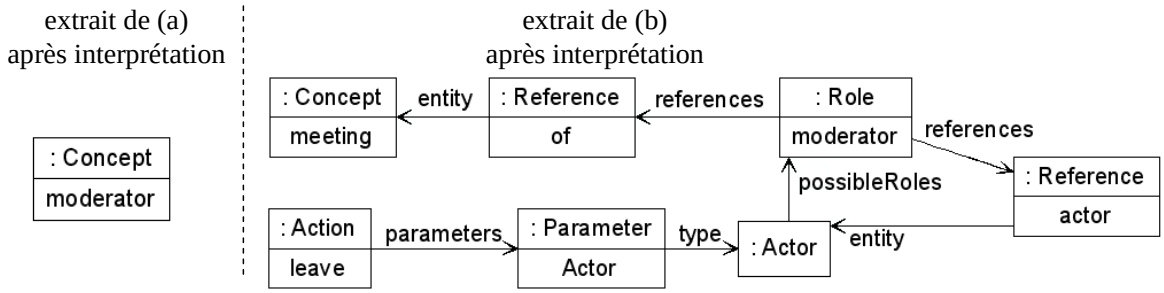


Figure 47 – Fragments de modèles résultant de l'interprétation de la notion de modérateur pour les modèles d'entrée (a) et (b) de la Figure 43.

2.2.3 Résolution d'une ambiguïté

Une ambiguïté a pour origine une variation sémantique. Les variations sémantiques ayant pour origine un flou sémantique, une imprécision syntaxique ou une décomposition de l'information provoquent des ambiguïtés. Cependant, une variation sémantique ne provoque pas nécessairement une ambiguïté. C'est le cas des variations sémantiques ayant pour cause une interprétation contextuelle ; dans ce cas, le choix de la variation sémantique lors de l'interprétation dépend du contexte du modèle d'entrée interprété (en l'occurrence la notation dans laquelle il s'inscrit et la sémantique de ses champs). Dans cette section, nous présentons les différentes stratégies de résolution des ambiguïtés durant un processus de composition. Les variations sémantiques ayant pour origine une interprétation contextuelle ne sont donc pas concernées.

L'obtention d'une spécification opérationnelle nécessite de résoudre les ambiguïtés identifiées durant l'interprétation des modèles d'entrée, c'est-à-dire de faire un choix parmi les variations sémantiques possibles pour chaque ambiguïté. Plusieurs stratégies peuvent être considérées pour résoudre une ambiguïté : l'*extension de la syntaxe du langage d'entrée*, la *restriction de son expressivité* ou l'*interaction homme-machine*. En outre, il est important de déterminer quand cette résolution doit être effectuée, c'est-à-dire le contexte de résolution de l'ambiguïté (local ou global). Le reste de cette section décrit ces deux points.

Le contexte de résolution d'une ambiguïté est local si la résolution peut être effectuée lors de l'interprétation du seul modèle d'entrée comportant l'ambiguïté. Autrement dit, il est local si le calcul des interprétations ne nécessite pas une vision globale de la spécification. Il est global sinon. Le contexte de résolution est local pour les variations sémantiques causées par un flou sémantique ou une imprécision syntaxique. En effet, le choix d'une variation concerne uniquement le modèle interprété. Le contexte de résolution d'une ambiguïté causée par une décomposition de l'information est global puisque seul une connaissance de l'ensemble des spécifications permet de détecter l'ambiguïté et de proposer les interprétations possibles. Les deux exemples donnés pour illustration plus haut sont explicites.

Nous avons identifié plusieurs stratégies pour la résolution des ambiguïtés suivant leur cause. Elles sont décrites ci-dessous et résumés dans le tableau plus loin :

- ***Extension de la syntaxe du langage d'entrée***. Dans ce cas, il est possible de spécifier explicitement l'information manquante. L'extension des patrons de Dwyer à l'aide d'options (les patrons de Schmidt [53]) vont dans ce sens. Ces options sont clairement identifiées et peuvent être résolues en augmentant la précision de la syntaxe. Par exemple, il est possible de désambiguïser la fonction sémantique f de la Figure 45 en ajoutant explicitement une option dans la représentation textuelle du patron de spécifications RDL PRECEDENCE (ajout d'une

séquence optionnelle de terminaux « each time » avant le terminal « before » par exemple). Cette stratégie est envisageable dans le cas des variations sémantiques ayant pour origine une imprécision syntaxique ou un flou sémantique.

- ***Restriction de l'expressivité du langage d'entrée***. Dans ce cas, il convient de se mettre d'accord sur une seule interprétation possible dans le cas d'une variation sémantique. Par exemple, on statue que seule la première interprétation de la phrase RDL de la Figure 45 est correcte. Dans ce cas, le langage d'entrée perd en expressivité mais devient formel. Cette solution n'est viable que si les parties prenantes ont connaissance de ces choix et maîtrisent la sémantique des langages d'entrée. Cette stratégie est envisageable dans le cas des variations sémantiques ayant pour origine un flou sémantique ou la décomposition de l'information.
- ***Interaction homme machine***. Nous avons identifié deux approches pour interagir avec les parties prenantes : (i) inspection des interprétations possibles ou (ii) élimination d'interprétations possibles par simulation ou inspection de scénarios générés. Dans le premier cas, on demande explicitement de lever l'ambiguïté en proposant aux parties prenantes l'ensemble des interprétations possibles. Dans le deuxième cas, la simulation fonctionnelle permet d'éliminer progressivement les ambiguïtés. Le deuxième cas ne s'applique que pour les ambiguïtés relatives à l'aspect dynamique de l'information[67]. Cette stratégie est envisageable dans tous les cas.

caractérisation de la résolution / origine de la variation	contexte de résolution	stratégies de résolution envisageables		
		extension de la syntaxe	restriction de l'expressivité	interaction homme-machine
flou sémantique	local	X	X	X
imprécision syntaxique	local	X		X
décomposition	global			X

Dans le Chapitre VI, nous illustrons uniquement la stratégie « interaction homme-machine ». En outre, nous ne considérons pas de variations sémantiques ayant pour origine un flou sémantique. En d'autres termes, nous considérons les langages d'entrée comme des langages formels. Notons ici que les notations pourraient être utilisées comme une stratégie pour résoudre une ambiguïté résultant d'un flou sémantique : un même diagramme d'activité pourrait être interprété différemment suivant la notation dans laquelle il s'inscrit. Une autre approche pourrait consister en la production de l'ensemble des interprétations possibles pour toutes les variations sémantiques, puis une disqualification des interprétations incorrectes par simulation (cette approche n'est applicable que pour l'aspect dynamique de l'information). Cette dernière approche est similaire à la deuxième approche proposée plus haut dans le cas d'une stratégie de résolution par interaction homme-machine.

2.3 Traitement des superpositions sémantiques

Un processus de composition ne consiste pas uniquement en une juxtaposition des fonctions sémantiques des langages d'entrée. En effet, si l'on se contente d'appliquer les fonctions sémantiques des langages d'entrée sur les modèles d'entrée, on obtient un ensemble de modèles RM. L'application des fonctions sémantiques traduit les informations dans les termes du formalisme, mais ne résout en rien le problème de la séparation de l'information. A titre d'exemple, la Figure 48 présente deux modèles RM obtenus en appliquant la fonction sémantique du langage RDL (noté f dans la figure) sur deux phrases RDL distinctes (c-2 et k-2 de la Figure 42). Ces modèles sont fournis dans deux syntaxes concrètes différentes : la syntaxe textuelle RM et le diagramme d'objet UML. Le résultat est une collection de deux modèles n'ayant aucuns liens représentés (ou deux spécifications textuelles séparées) et dont certains fragments se superposent sémantiquement.

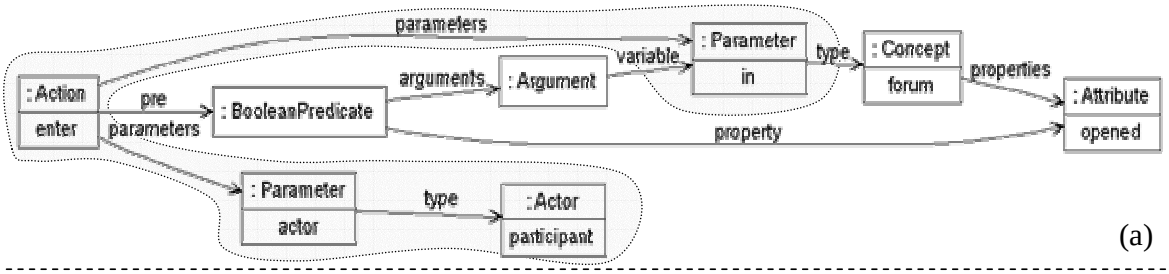
[67] Encore une fois, une analyse dynamique comme la simulation n'est possible que si le modèle global est statiquement cohérent.

L'ensemble des deux fragments de modèles grisés dans la Figure 48 est un exemple de superposition sémantique. Ces fragments représentent la même information, à savoir l'action *enter* et ses paramètres. Ces superpositions doivent être résolues de sorte que le modèle global ne représente pas deux fois un même élément du domaine. Un processus de composition comporte donc une étape d'analyse des relations sémantiques entre des informations initialement séparées mais portant sur des éléments identiques du domaine. Cette étape a pour but (i) l'identification des superpositions sémantiques [32] entre les informations à composer et (ii) la résolution de ces superpositions. Comme introduit dans la section 2.1, il est plus avantageux de comparer les modèles d'entrée une fois interprétés. Nous donnons ci-dessous une définition de la notion de superposition sémantique, étant entendu que les modèles d'entrée ont été interprétés au préalable :

Définition – *superposition sémantique* : Ensemble de fragments de modèles appartenant aux modèles d'entrée après interprétation et représentant une même information (un ou plusieurs éléments du domaine).

f ⟨A *forum* must be *opened* before a *participant* can *enter* in this *forum*.⟩

= | action *enter* (p : *Participant*, m : *Forum*)
pre *opened*(m)



(a)

(b)

f ⟨When a *meeting* is *private*, a *participant* must be *identified* before he can *enter* in this *meeting*.⟩

= | action *enter* (p : *Participant*, m : *Meeting*)
pre *private*(m) => *identified*(p)

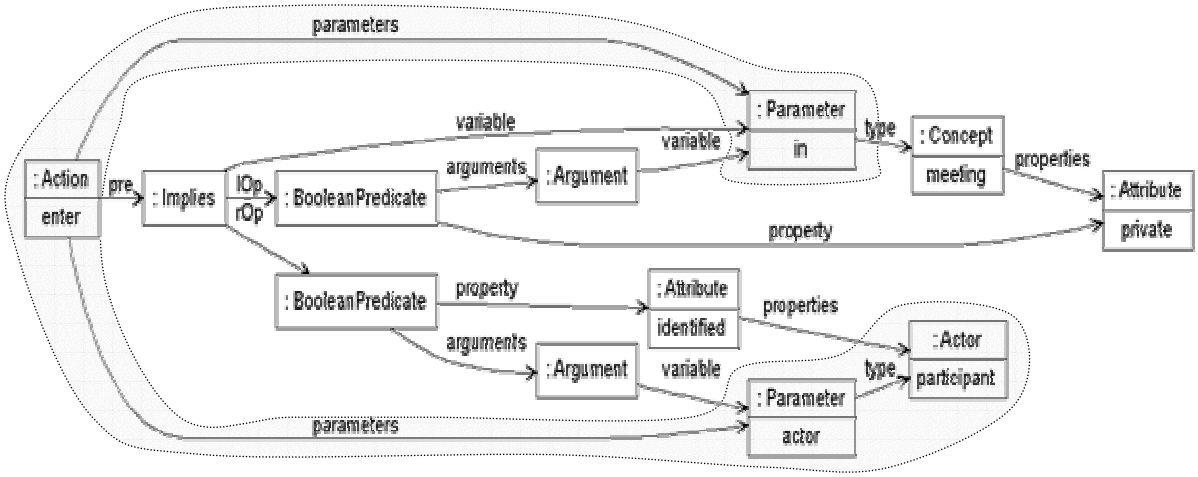


Figure 48 – Exemple de deux modèles RM obtenus par application de la fonction sémantique du RDL.

La Figure 49 présente un extrait du modèle global obtenu après résolution de la superposition sémantique identifiée dans la Figure 48. Le fragment grisé de la Figure 49 représente l'ensemble des éléments impactés par la résolution. Tous ces éléments ont en commun le fait de représenter une

information initialement redondante dans la Figure 48. Le résultat de la résolution des superpositions est un modèle ne comportant plus de redondances. Ce modèle représente désormais la synthèse des informations capturées par les deux modèles d'entée de la Figure 48, ce qui offre une vision unifiée des informations composées. On peut remarquer que la résolution peut faire apparaître des violations de conformité. Par exemple, le paramètre *in* a deux types (les concepts *forum* et *meeting*), ce qui n'est pas conforme au métamodèle RM. De même, l'action *enter* a deux pré-conditions. Ces violations peuvent refléter une incohérence au niveau des exigences capturées par les modèles composés ou une malformation syntaxique qu'il convient de résoudre durant la composition. Le premier exemple de violation reflète une incohérence alors que le deuxième reflète une malformation syntaxique (le modèle de la Figure 49 est en cours de composition). La détection d'incohérence après résolution est discutée en section suivante.

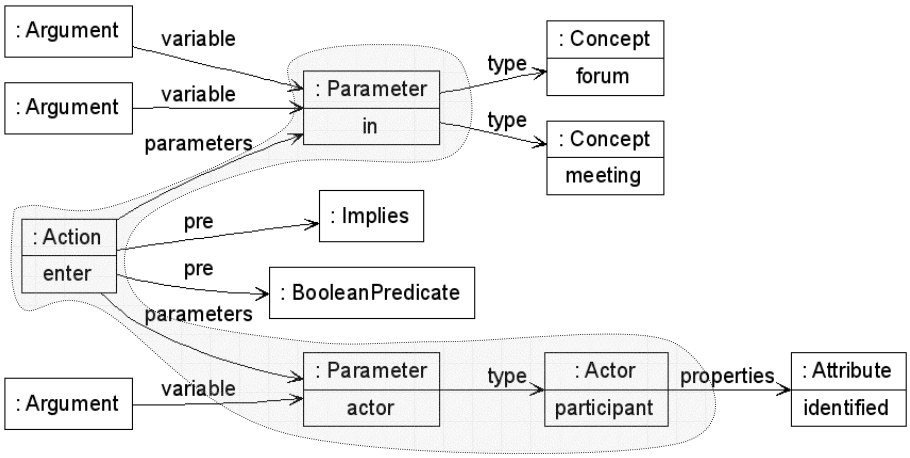


Figure 49 – Illustration de la résolution de superposition sémantique (extrait du modèle obtenu par composition des deux modèles d'entrée de la Figure 48.

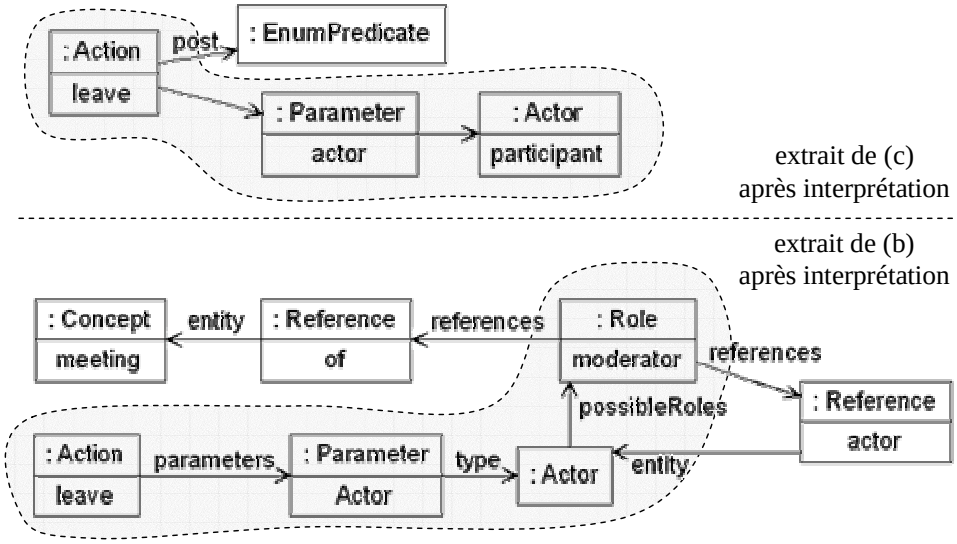


Figure 50 – Exemple de superposition sémantique complexe (collision structurelle).

Cet exemple de superposition sémantique et sa résolution sont particulièrement simples car les fragments sémantiquement équivalents le sont aussi syntaxiquement. Dans ce cas, la résolution consiste en la somme des fragments comme Sabetzadeh et Easterbrook [40] la définissent. Cependant, l'identification et la résolution de superpositions peuvent être plus complexes. Par exemple, deux propriétés de même nom appartenant à deux modèles d'entrée différents ne désignent pas toujours le même élément du domaine ; c'est le cas si les entités les contenant se superposent aussi. Prenons les éléments *connected* des modèles d'entrée (e) et (f) de la Figure 42. Ces deux éléments représentent deux attributs de même nom. Ils désignent le même élément du domaine si les éléments *moderator* et *participant* se superposent, ce qui est le cas pour cet exemple, puisque *moderator* est un rôle de

participant. Dans le cas contraire, ces deux attributs ne se superposent pas bien qu'ils soient syntaxiquement équivalents. L'identification d'une superposition entre deux éléments peut donc dépendre d'autres superpositions (généralement, ces superpositions portent sur des éléments adjacents).

Enfin, l'identification de superposition peut être encore plus complexe si la syntaxe de deux fragments superposés diffère de manière importante. On a alors affaire à une collision structurelle [98] (cf. section 4 du Chapitre II). La Figure 50 illustre ce cas. Les modèles présentés sont des extraits des modèles d'entrée (c) et (b) après interprétation. Les fragments grisés forment une superposition sémantique. Ils représentent tous deux l'action *leave* et son acteur mais chacun capture une information différente sur ce dernier : le fragment issu de (c) signifie que l'acteur de l'action *leave* est un participant ; le fragment issu de (b) signifie que l'acteur est un acteur dont le rôle est d'être modérateur d'une réunion. La résolution d'une superposition de ce type est plus complexe que les cas précédemment présentés.

3 Résumé des caractéristiques d'un processus de composition d'exigences

Les sections précédentes permettent de dégager un ensemble de caractéristiques (pour ne pas dire exigences) qu'un processus de composition de spécifications partielles hétérogènes doit comporter :

- ***Composition symétrique***. Un processus de composition est itératif. L'ordre dans lequel les informations sont composées ne doit pas avoir d'impact sur le résultat.
- ***Invariance sémantique***. Le modèle global est une expression du formalisme de la plate-forme. Le sens de l'information capturée par le modèle global après composition doit être équivalent au sens des informations éparpillées dans les spécifications partielles avant composition. Dans le cas contraire, un diagnostic obtenu par analyse du modèle global n'a pas de sens.
- ***Composition multi-langages***. Un processus de composition doit être multi-langages puisque les parties prenantes ne connaissent pas toujours ou ne sont pas à l'aise avec les mêmes langages. Autrement dit, le processus doit s'adapter à leurs habitudes et connaissances, et non l'inverse.
- ***Comparaison après interprétation.*** L'identification des superpositions sémantiques et des incohérences après interprétation limite l'impact de l'évolution des langages d'entrée (ce n'est pas le cas pour le formalisme).
- ***Variation sémantique***. Une syntaxe peut être munie de plusieurs sémantiques possibles. Un processus de composition doit faciliter la description de variations sémantiques.
- ***Détection des ambiguïtés***. Une ambiguïté doit être résolue si l'on désire obtenir un modèle global. Un processus de composition doit supporter leur détection et leur résolution, avec l'aide des parties prenantes si nécessaire (avec interface homme-machine).
- ***Traçabilité***. Le processus doit maintenir des liens de traçabilité entre les spécifications partielles et le modèle global. Dans le cadre d'un processus IE, un diagnostic sur le modèle global n'est utile que s'il peut être compréhensible par les parties prenantes. Dans un contexte de développement IDM, la relation entre artefacts de développement logiciel et exigences est cruciale si ces dernières évoluent durant la phase de développement.
- ***Flexibilité des langages supportés***. La plate-forme R2A doit être adaptable à différents contextes industriels. La comparaison après interprétation agit dans ce sens. En outre, La plate-forme R2A devant être adaptable à différents contextes industriels, son processus de composition doit être flexible de façon à pouvoir facilement modifier le formalisme, puisque le processus de composition est spécifique à un ensemble prédéfini de langages (langages d'entrée et formalisme). Ce langage doit être spécifique (DSL) pour limiter la complexité[68] des spécifications.

[68] Comprendre complexité accidentelle uniquement ; la complexité conceptuelle est, elle, inévitable.

Chapitre VI Obtention d'une spécification opérationnelle globale : processus de composition simple

Résumé

Ce chapitre présente le *processus de composition simple*. Ce processus produit un *modèle global* reflétant des exigences exprimées par les parties prenantes à l'aide de spécifications partielles hétérogènes. Il est itératif dans le contexte d'un processus IE : les exigences sont composées à mesure qu'elles sont rédigées et le diagnostic guide la rédaction de nouvelles exigences. Le processus de composition simple réalise le mécanismse d'import de la plate-forme R2A. Il est illustré dans ce contexte à l'aide de l'exemple de la spécification SRV et plus particulièrement les modèles d'entrée de la Figure 43 du chapitre précédant. Les modèles acceptés en entrée du processus sont donc instances des trois langages de modélisation des exigences actuellement supportés (RDL, diagramme de classes UML, diagramme d'activités UML)[69].

La section 0 présente le processus dans son ensemble. Ce processus consiste en trois étapes séquentielles : l'*interprétation*, la *fusion* et l'*analyse*. Les deux premières étapes réalisent les relations sémantiques des langages d'entrée. L'étape d'interprétation est présentée en section 2. Elle traduit chaque modèle d'entrée dans les termes du formalisme, en l'occurrence le RM. L'interprétation aboutit à la création d'une collection de modèles appelés *modèles intermédiaires*. L'étape de fusion fait l'objet de la section 3. Elle identifie et résout les superpositions sémantiques entre les modèles intermédiaires. La résolution a pour effet d'expliciter les liens sémantiques entre modèles intermédiaires jusqu'à obtention d'un modèle global représentant l'ensemble des informations capturées par les modèles d'entrée. L'étape d'analyse a pour but la détection d'incohérences au niveau du modèle global. Elle produit un *diagnostic* associant aux incohérences une description informelle et une localisation précise des modèles d'entrée incriminés.

Le processus de composition simple est entièrement construit à base de technologies IDM. Les étapes du processus sont réalisées par des transformations de modèles. Chaque étape est spécifiée à l'aide d'un langage spécifique (DSL [14]) à base de règles. Ces langages sont présentés dans ce chapitre pour illustrer le processus. Ils ne doivent pas être considérés comme une contribution en tant que telle ; l'important tient dans les principes qu'ils mettent en œuvre. Ces langages supportent un mécanisme de traçabilité natif (section 4.2). Ce mécanisme produit automatiquement un modèle de traçabilité associant les objets des modèles d'entrée composés aux objets du modèle global sémantiquement équivalents. Ce mécanisme est essentiel dans un processus IE impliquant les parties prenantes ; le modèle de traçabilité est exploité durant l'étape d'analyse afin d'associer aux incohérences détectées au niveau du modèle global un sens au niveau des modèles d'entrée rédigés par les parties prenantes. Enfin, le caractère IDM du processus facilite son adaptabilité à des contextes industriels différents puisque ses étapes sont spécifiées par des transformations de modèles[70] et les langages par des métamodèles.

[69] On peut ajouter à cette liste le formalisme RM lui-même, jouant le rôle de métamodèle cœur.

[70] Les langages utilisés pour capturer des exigences et le langage cible d'un processus de composition dépendent étroitement du contexte industriel (us et coutume des parties prenantes, logiciels utilisés, culture de l'entreprise, nature du logiciel développé …).

1 Un processus itératif en trois étapes

La section 1.1 présente dans son ensemble le processus et les différents rôles joués par les parties prenantes. Elle souligne la nature orientée modèle du processus et introduit la notion de métamodèle relâché, nécessaire pour produire un modèle global statiquement incohérent. La section 1.2 décrit plus précisément chacune des étapes du processus. Elle précise comment ces étapes participent à la composition de l'information, la résolution des ambiguïtés et la détection des incohérences. Enfin, la section 1.3 discute de l'intérêt d'une approche symétrique de composition et souligne le caractère itératif du processus de composition dans le cadre d'un processus IE.

1.1 Vue générale du processus

La Figure 51 présente le processus de composition dans son ensemble. Ce processus est constitué de trois étapes : l'*interprétation*, la *fusion* et l'*analyse*. Ces trois étapes sont brièvement décrites en section 1.2 et détaillées dans la suite de ce chapitre. L'interprétation et la fusion visent la production du modèle global. Ces étapes sont responsables de la composition des modèles d'entrée. Le résultat de ces deux premières étapes est la production d'un modèle global représentant l'ensemble des informations capturées par les modèles d'entrée. Elles produisent de plus un *modèle de traçabilité* décrivant précisément les liens sémantiques entre les modèles d'entrée et le modèle global. Le modèle de traçabilité est caractérisé en section 4.1. L'analyse vise la détection au sein du modèle global des incohérences statiques et dynamiques. Elle produit un *verdict* répertoriant l'ensemble des incohérences détectées.

Le processus de composition est séquentiel. Les *rédacteurs* (clients, prestataires, utilisateurs finals …) produisent avec l'aide des *analystes* un ensemble de modèles d'entrée capturant les exigences et les assertions environnementales devant être analysées (initialement décrites de manière informelle). Les étapes d'interprétation et de fusion sont ensuite exécutées. Les ambiguïtés sont détectées et résolues durant ces étapes. Le modèle global est ensuite analysé pendant l'étape d'analyse. Les langages d'entrée et le formalisme sont choisis par les *ingénieurs processus*. Les langages d'entrée sont choisis suivant les compétences des parties prenantes, la culture de l'entreprise et la nature du formalisme. Le formalisme est choisi suivant le degré d'automatisation des activités V&V souhaité mais aussi en fonction des outils formels visés. Les *développeurs processus* sont responsables de l'implémentation des trois étapes du processus.

Le processus proposé est entièrement construit en suivant une approche orientée modèle comme la Figure 51 le souligne (modèles, métamodèles et transformations de modèles). Le cœur du processus est le formalisme choisi pour la composition, représenté par le métamodèle cœur muni d'une sémantique opérationnelle. Dans la version actuelle de la plate-forme R2A, ce métamodèle est le RM muni d'une sémantique opérationnelle implémentée en Kermeta pour la simulation fonctionnelle et un homéomorphisme pour la vérification des propriétés non-fonctionnelles (transformation vers le langage IF). Chaque langage d'entrée du processus est représenté par un métamodèle d'entrée, associé à une grammaire lorsque le langage est textuel (cas du RDL), et muni d'une fonction sémantique réalisée par les étapes d'interprétation et de fusion. Le métamodèle de traçabilité correspond à la vue traçabilité du métamodèle RM (cf. vue traçabilité en Annexe E).

Les trois étapes du processus sont réalisées à l'aide de transformations de modèles. Ces transformations sont spécifiées à l'aide de langages dédiés à base de règles : IRL (*Interpretation Rule Language*) pour l'interprétation, FRL (*Fusion Rule Language*) pour la fusion et IDL (*Inconsistency Detection Language*) pour l'analyse. Ces langages sont présentés tout au long de ce chapitre[71]. Nous avons choisi une approche à base de règles pour deux raisons principales. En premier lieu, cela permet

[71] Notons que ces langages sont eux-mêmes spécifiés à l'aide d'un langage à base de règles. Ils partagent en outre un noyau commun (environ 90% des règles sémantiques et syntaxiques sont identiques). Ces trois langages constituent une étude de cas intéressante concernant la réutilisation de spécifications pour la production de DSL. Ce point n'est pas abordé dans cette thèse.

une contextualisation du code suivant les types d'objets manipulés (la notion de règle est un mécanisme de classification). Il est alors possible de localiser rapidement l'ensemble des règles concernant un ensemble de métaclasses des métamodèles d'entrée et du métamodèle cœur. Cette caractéristique est essentielle pour la maîtrise et la maintenance de transformations de grande taille. En deuxième lieu, une approche par règle permet de produire automatiquement un modèle de traçabilité précis entre les objets des spécifications partielles et les objets du modèle global (voir la section 4.1).

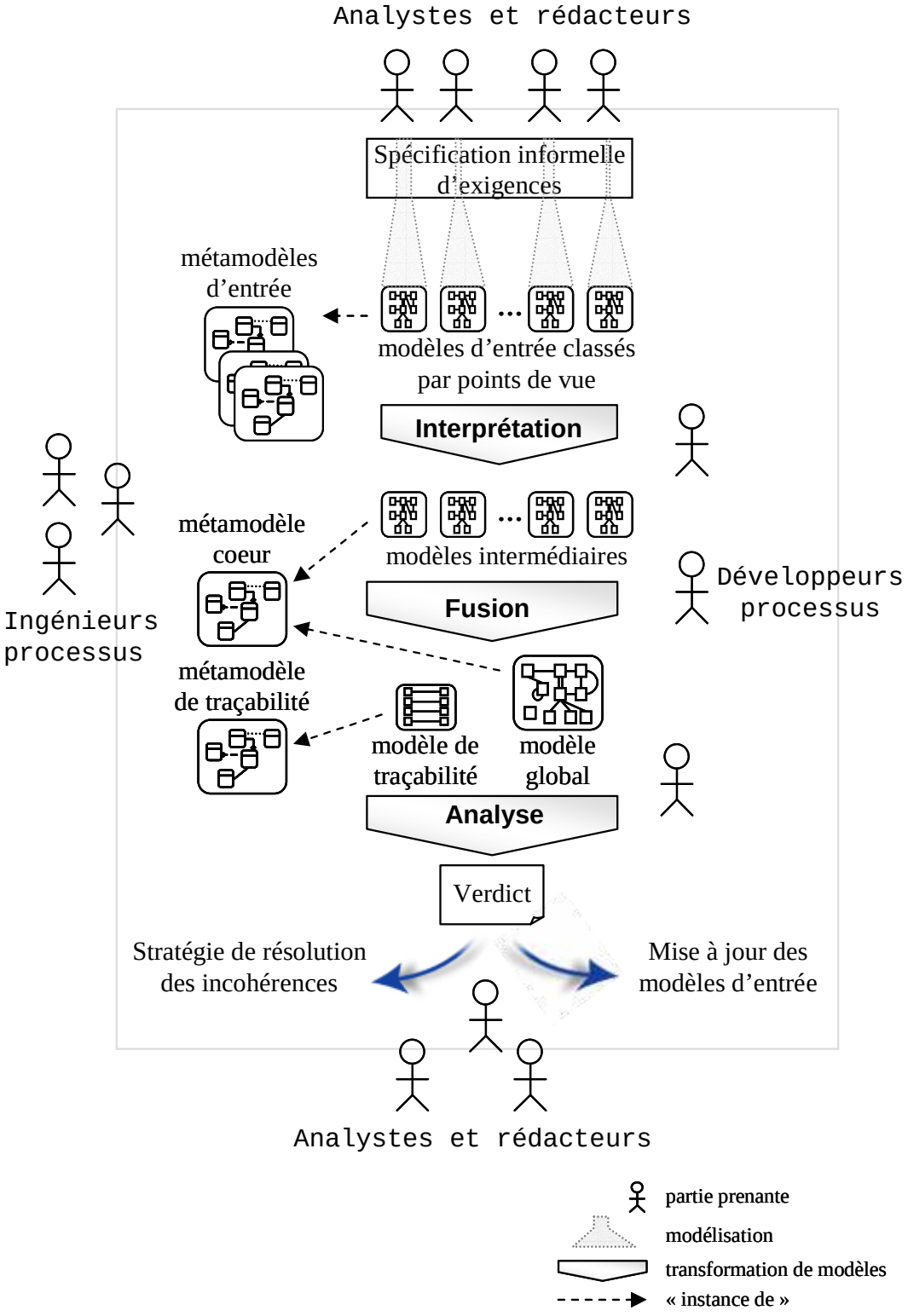


Figure 51 – Vue globale du processus de composition.

Enfin, nous avons vu que la composition de modèles d'entrée incohérents implique la représentation de ces incohérences au niveau du modèle global. La représentation d'une contradiction logique est particulièrement problématique car elle peut induire la violation de la borne supérieure d'une cardinalité (voir l'instanciation de la relation *type* du paramètre *in* de la Figure 49). Dans ce cas, une spécification définie sur le métamodèle cœur n'est pas exécutable (par exemple, une contrainte OCL). Nous introduisons la notion de métamodèle relâché pour contourner ce problème (voir la section 1.4). Les langages IRL, FRL et IDL manipulent des instances d'une version relâchée du métamodèle cœur de manière transparente : les spécifications IRL, FRL et IDL sont définies sur le métamodèle cœur initial.

1.2 Description des trois étapes du processus

L'obtention du modèle global est conjointement réalisée par l'étape d'interprétation et l'étape de fusion. Cette séparation des deux préoccupations propres à un processus de composition (traduction vers le formalisme et traitement des superpositions sémantiques) limite la complexité des spécifications réalisant les fonctions sémantiques des langages d'entrée. On peut faire ici une comparaison avec un processus de compilation : la traduction des spécifications vers un langage bas niveau et l'édition de liens sont deux étapes bien distinctes. En outre, la séparation de la composition en deux étapes distinctes permet de factoriser une partie du code implémentant les fonctions sémantiques des langages d'entrée. En effet, le traitement des superpositions sémantiques est effectué sur les modèles d'entrée, une fois interprétés (les *modèles intermédiaires* dans la Figure 51) ; ce traitement est donc indépendant des langages d'entrée. Autrement dit, la transformation réalisant l'étape de fusion est peu couplée aux langages d'entrée.

La fonction ci-dessous définit formellement la composition de modèles d'entrée comme une fonction φ résultant de la composition (fonctionnelle) des fonctions implémentant l'étape d'interprétation et l'étape de fusion (ces fonctions sont décrites plus bas). Les trois étapes du processus de composition sont décrites en détails dans la suite de ce chapitre et sont brièvement introduites dans le reste de cette section.

$$\begin{aligned} \varphi : \mathcal{P}(M) &\rightarrow F \\ \{m_1, \ldots, m_n\} &\rightarrow \psi(\rho(\{m_1, \ldots, m_n\})) \end{aligned}$$

avec :
- ρ est la fonction implémentant l'étape d'interprétation.
- ψ est la fonction implémentant l'étape de fusion.
- M l'ensemble des modèles conformes à un des langages d'entrée.
- F le formalisme.
- $\mathcal{P}(X)$ est l'ensemble des parties de l'ensemble X.

Interprétation. L'interprétation consiste à appliquer les fonctions sémantiques des langages d'entrée sur les modèles d'entrée. Le terme d'interprétation est conforme à la définition proposée en section 2.2.1. Chaque fonction sémantique est réalisée par une transformation de modèles spécifiée avec le langage IRL. Une spécification IRL est une collection de *règles d'interprétation*. Chaque règle vise l'extraction d'une partie de l'information capturée par ce modèle, puis sa traduction dans les termes du formalisme. Pour chaque modèle d'entrée, l'étape d'interprétation produit une instance du métamodèle cœur relâché. Ces modèles sont appelés *modèles intermédiaires* (cf. Figure 51). Ils ne sont pas nécessairement conformes au métamodèle cœur puisqu'ils capturent une information partielle par définition. Le résultat de l'interprétation d'une collection de modèles d'entrée est donc une collection de spécifications partielles, expressions du formalisme. L'étape d'interprétation participe à la résolution des ambiguïtés pouvant être résolues dans un contexte local (ambiguïtés dues à une imprécision syntaxique ou un flou sémantique). Plus formellement, l'interprétation d'un ensemble de modèles d'entrée est définie par la fonction ρ suivante :

$$\rho : \mathcal{P}(M) \rightarrow \mathcal{P}(F)$$
$$\{m_1, \ldots, m_n\} \rightarrow \{ f_{lan(m_i) \rightarrow F}(m_i) \mid i = 1 .. n \}$$

avec :
- M l'ensemble des modèles conformes à un des langages d'entrée.
- F le formalisme et $\mathcal{L}$ l'ensemble des langages d'entrée.
- $f_{L \rightarrow F}$: $L \rightarrow F$ la fonction sémantique du langage d'entrée L définie sur F.
- $\mathcal{P}(X)$ est l'ensemble des parties de l'ensemble X.
- lan : $M \rightarrow \mathcal{L}$ la fonction associant à chaque modèle d'entrée son langage d'entrée.

Fusion. La fusion vise l'identification des superpositions sémantiques entre modèles intermédiaires, la résolution de ces superpositions et la normalisation de l'information. Nous avons vu en section 2.2.3 que la résolution d'une superposition consiste en la suppression des redondances identifiées de sorte que toute information au sein du modèle global produit est exprimée une seule fois. La résolution des superpositions explicite les liens sémantiques implicites entre les modèles intermédiaires représentant des informations partielles. Le modèle global produit après résolution des superpositions sémantiques est sémantiquement correct (il représente l'ensemble des informations capturées par les spécifications partielles), mais syntaxiquement incorrect dans la plupart des cas. La normalisation de l'information vise à pallier ce problème (voir section 2). L'étape d'interprétation participe à la résolution des ambiguïtés pouvant être résolues dans un contexte global (ambiguïtés dues à une décomposition de l'information). Plus formellement, l'étape de fusion d'un ensemble de modèles intermédiaires est définie par la fonction ψ suivante :

$$\psi : \mathcal{P}(F) \rightarrow F$$
$$\{m_1, \ldots, m_n\} \rightarrow m_n$$

avec :
- F le formalisme.
- $\mathcal{P}(X)$ est l'ensemble des parties de l'ensemble X.

Analyse. L'étape d'analyse regroupe l'analyse statique et l'analyse dynamique du modèle global. L'analyse statique est effectuée sur le modèle global obtenu après interprétation et fusion. Cette approche transforme une comparaison inter-modèles en une comparaison intra-modèle. Nous avons vu en section 2.1 du Chapitre V que la comparaison des modèles d'entrée après interprétation comporte plusieurs avantages importants pour un passage à l'échelle industrielle. Le fait de détecter les incohérences après composition (i) limite la complexité des spécifications de règles de cohérence, (ii) rend leur nombre indépendant du nombre de langages d'entrée et (iii) limite l'effort de spécification lors de l'ajout d'un langage d'entrée. L'analyse statique consiste en l'application des règles de cohérences sur le modèle global. Le résultat de l'analyse est un verdict associant à chaque incohérence détectée (i) les modèles d'entrée incriminés et (ii) une description de l'incohérence compréhensible par l'ensemble des parties prenantes. L'analyse dynamique consiste en l'utilisation de techniques de vérification formelle. Nous ne nous intéressons pas à ce type d'analyse ici. Le verdict sert de support à la négociation et à la discussion entre parties prenantes : certaines incohérences reflètent des conflits, des divergences ; d'autres permettent d'identifier de nouvelles exigences [34]. Nous verrons en section 3 un exemple d'incohérence permettant l'identification d'une nouvelle exigence.

1.3 Itération et symétrie du processus

Le processus composition est itératif puisque les exigences sont identifiées progressivement, la détection d'incohérences favorise l'identification de nouvelles exigences et les exigences peuvent être décrites séparément par des équipes différentes. Une fois les modèles d'entrée mis à jour par les parties prenantes (suivant la stratégie de résolution adoptée), le modèle global est mis à jour par réitération du processus. Si un modèle d'entrée est ajouté, le modèle global courant est considéré

comme un modèle d'entrée, ce qui diminue fortement le temps de calcul puisque les modèles d'entrée préexistants n'ont pas à être à nouveau composés. Cependant, ce scénario n'est applicable que lorsque les modèles d'entrée déjà traités n'ont pas été modifiés. Dans le cas contraire, le modèle global ne peut être réutilisé et la composition doit être appliquée sur l'ensemble des modèles d'entrée (anciens et nouveaux), ce qui demande un temps de calcul d'autant plus important que la spécification d'exigences est précise et complète.

Pour pallier ce problème, nous proposons de profiter de la nature symétrique du processus de composition. Il est en effet possible de fragmenter la composition des modèles d'entrée de sorte que la modification d'un modèle d'entrée ne nécessite pas la recomposition intégrale de tous les modèles d'entrée. La Figure 52 illustre une fragmentation possible pour huit modèles d'entrée (a-h). Les modèles sont composés deux à deux de manière récursive jusqu'à obtention du modèle global (o). Les modèles générés (i-n) peuvent être sauvegardés de façon à limiter l'effort de composition à la prochaine itération. A titre d'exemple, supposons que le modèle (c) de la Figure 52 ait été modifié et que l'on ajoute un nouveau modèle d'entrée (p). Le nouveau modèle global est alors ψ(i, c, d, n, p) au lieu de ψ(a, b, c, d, e, f, g, h, p) sans fragmentation[72].

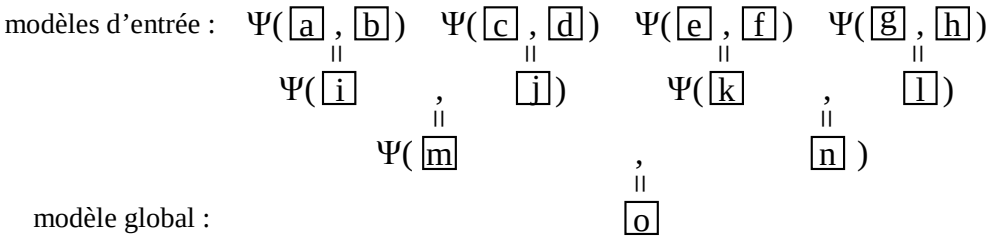


Figure 52 – Exemple de fragmentation de la composition.

1.4 Relâchement du métamodèle cœur

Contrairement aux approches de composition proposées dans la littérature (excepté le travail de Sabetzadeh [40]), nous considérons les modèles d'entrée potentiellement incohérents entre eux. Cela implique que les modèles manipulés durant le processus de composition ont des chances d'être structurellement incohérents. La représentation de ces incohérences implique l'existence de violations des cardinalités du métamodèle global. Lorsque ces violations concernent les bornes supérieures des cardinalités, les spécifications définies sur le métamodèle global ne sont plus exécutables. Pour pouvoir manipuler des modèles structurellement incohérents, il est nécessaire de redéfinir le métamodèle coeur, de sorte que les contraintes structurelles puissent être violées durant la composition. Nous proposons pour cela la notion de version relâchée d'un métamodèle, définie ci-dessous :

Définition – *version relâchée d'un métamodèle* : une version relâchée d'un métamodèle X est un métamodèle noté X', identique à X mais débarrassé de toutes contraintes structurelles[73]. Plus précisément, X' est identique à X, à l'exception des points suivants :

(i) X' ne comporte pas de contraintes OCL.
(ii) Les métaclasses abstraites dans X sont concrètes dans X'.
(iii) Les cardinalités des relations de X' sont relâchées ; elles sont égales à [0, *[(* représente l'infini).
(iv) Les relations de composition dans X sont des relations simples dans X'.

[72] Cette technique de fragmentation pourrait être optimisée si l'on disposait d'heuristiques déterminant la probabilité de modification d'un modèle d'entrée. En outre, elle pourrait être utile si l'on cherchait à calculer l'ensemble des interprétations possibles en fonction des variations sémantiques causées par un flou sémantique. Dans ce cas, les fragments de modèles obtenus par interprétation n'étant pas impactés par une variation sémantique pourraient être composés une fois pour toutes, ce qui limiterait le temps de calcul de chacune des interprétation possibles.

[73] On peut remarquer ici que l'on a une relation de type de métamodèle entre un métamodèle et sa version relâchée. En effet, l'ensemble des modèles conformes à un métamodèle est inclus dans l'ensemble des modèles conformes à sa version relâchée.

La Figure 53 présente la version relâchée MM' d'un métamodèle MM. Les cinq contraintes structurelles définies par MM sont absentes dans MM' : les trois cardinalités des associations, la composition pour l'association *a* et la contrainte OCL. On peut remarquer que l'absence des contraintes structurelles interdit l'exécution de code définie sur MM. A titre d'exemple, l'interprétation de la règle OCL de la Figure 53 est impossible si *self.c* retourne une liste (quel élément choisir si cette liste comporte plus d'un élément ? Doit-on vérifier la contrainte OCL pour chaque élément ?). Plus généralement, ce problème est vrai pour tout code défini sur MM et dont l'interprétation doit naviguer des modèles instances de MM'. Nous verrons dans le reste de ce chapitre que les langages FRL et IDL de sorte que leurs expressions (les règles de fusion et de détection d'incohérences) soient définies sur le métamodèle cœur bien que les modèles manipulés soient instances du métamodèle cœur. Cette particularité favorise la concision des spécifications FRL et IDL.

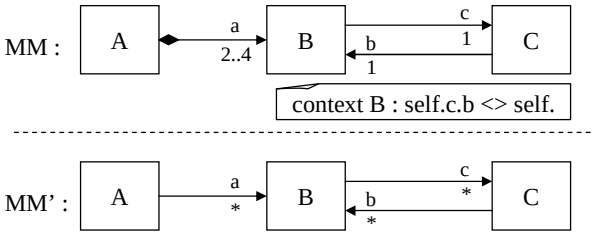


Figure 53 - Illustration de la notion métamodèle relâché et de la relation de type de métamodèle.

2 Interprétation

Nous présentons dans cette section la première étape du processus de composition. Cette étape a pour but (i) l'extraction des informations capturées par les modèles d'entrée ayant un sens dans le domaine sémantique du formalisme et (ii) la traduction de ces informations sous la forme de modèles instances du métamodèle coeur. La section 2.1 donne une vue générale de l'étape d'interprétation. La section 2.2 présente ensuite de la notion de règle d'interprétation. Nous discutons des principes de l'approche à base de règles, de leur implémentation dans le langage IRL et de la gestion des ambiguïtés syntaxiques dans ce cadre. L'exécution de règles d'interprétation IRL est finalement illustrée. Enfin, la section 2.3 discute de la nature des modèles intermédiaires produits.

2.1 Vue générale

L'interprétation consiste en l'application des fonctions sémantiques des langages d'entrée sur les modèles d'entrée. Le type d'un modèle d'entrée détermine la fonction sémantique appliquée. Chaque modèle d'entrée représente une spécification partielle et est conforme à un métamodèle d'entrée représentant un langage d'entrée. L'interprétation d'un modèle d'entrée dépend de trois facteurs :

(i) le sens que l'on attribue à une syntaxe,

(ii) le type d'informations que l'on désire extraire (fonction du formalisme et des outils d'analyse visés),

(iii) le contexte dans lequel s'inscrit le modèle interprété (notations et patrons).

La Figure 54 schématise l'étape d'interprétation. Dans le contexte de la plate-forme R2A, les fonctions sémantiques des langages d'entrée sont spécifiées dans un langage dédié appelé IRL (pour *Interpretation Rule Language*). Ce langage est basé sur une approche d'identification de *motif* comme le langage ATL [141]. C'est un langage à base de règles. Une fonction sémantique est représentée par un code IRL, conforme au métamodèle IRL. Le résultat de l'application d'une fonction sémantique sur un modèle d'entrée est un ensemble de fragments de modèles, instances du métamodèle cœur relâché et regroupés au sein d'un modèle intermédiaire. Une fonction produit aussi un modèle de traçabilité et un modèle d'équivalences. Le modèle de traçabilité capture une partie des traces générées automatiquement pendant la composition. Le modèle de traçabilité final résulte de la composition de ce modèle de traçabilité avec le modèle de traçabilité produit par l'étape de fusion (cf. section 4.2). Le

modèle d'équivalences capture les équivalences détectées entre éléments des modèles intermédiaires et ne pouvant être détectées durant l'étape de fusion (voir section 2.3).

Dans la Figure 54, l'interprétation accepte un ensemble de modèles mais les spécifications textuelles ne sont pas représentées. En fait, l'interprétation d'une spécification textuelle comme une expression RDL nécessite l'application d'une étape précédant l'interprétation, appelée *analyse grammaticale*. Cette étape est spécifique aux spécifications textuelles et n'est pas appliquée dans le cas de spécification partielle graphique, puisque les outils de modélisation graphique génèrent directement un modèle. Elle implémente un mécanisme classique d'analyse syntaxique et le modèle obtenu n'est ni plus ni moins qu'un arbre syntaxique abstrait. Cette étape est réalisée à l'aide de l'outil Syntaks [160] de Kermeta ou à l'aide d'outils classiques comme Antlr [164]. Elle convertit chaque spécification textuelle en un modèle d'entrée sémantiquement équivalent. Ce modèle d'entrée est instance d'un métamodèle d'entrée représentant la syntaxe abstraite du langage textuel (par exemple, RDM pour le RDL). L'intérêt de l'analyse grammaticale est de fournir à l'étape d'interprétation une information homogène quelle que soit sa représentation initiale (texte, graphique, tableau …).

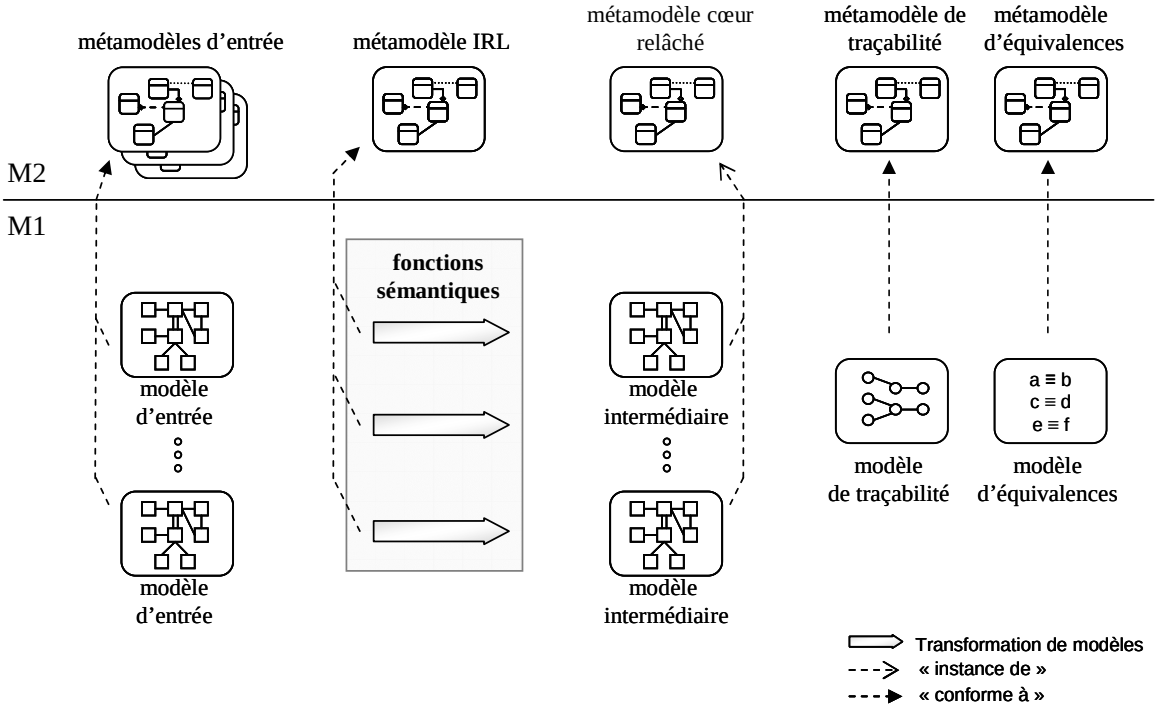


Figure 54 - Schématisation de l'étape d'interprétation.

L'étape d'interprétation participe à la résolution d'une partie des ambiguïtés identifiées durant la composition. Les ambiguïtés dues à une imprécision syntaxique sont résolues par un dialogue avec les parties prenantes. Dans certains cas, ces ambiguïtés peuvent être résolues par des heuristiques, spécifiées au sein des règles d'interprétation. En revanche, nous n'avons pas étudié en détails la résolution des ambiguïtés dues à un flou sémantique. Nous considérons que chaque langage d'entrée est un langage formel ; la fonction présentée ci-dessus illustre ce choix puisqu'une fonction sémantique d'un langage d'entrée produit un seul modèle intermédiaire (dans le cas contraire, une fonction sémantique devrait être définie sur $\mathcal{P}$(F)). Enfin, les ambiguïtés dues à la décomposition de l'information sont traitées durant l'étape de fusion car leur résolution nécessite une vue globale de l'information.

2.2 Notion de règle d'interprétation

2.2.1 Principes

Un programme IRL représente la fonction sémantique f : N → F d'un des langages d'entrée, où N est le métamodèle d'entrée, c'est-à-dire la syntaxe du langage d'entrée ; F est le métamodèle cœur

(représentant le formalisme). L'étape d'interprétation est donc spécifiée par autant de programme IRL qu'il y a de langages d'entrée. Un programme IRL regroupe un ensemble de *règles d'interprétations*. Une règle d'interprétation comprend un *motif* et une *production*. Un motif est une expression logique identifiant un type d'informations pouvant être extrait d'un modèle d'entrée. Il représente un ensemble de fragments instances de N ; une production est aussi un type d'informations. Tout comme un motif, c'est une expression logique. Elle vise la production d'un fragment instance de S. Une règle d'interprétation associe donc deux ensembles de fragments : des fragments instances de N identifiés par le motif et des fragments instances de S sémantiquement équivalents, générés par la production. Les notions de règle d'interprétation, de motif et de production sont définies ci-dessous :

Définition – *motif* : Expression logique définissant un ensemble de fragments des modèles d'entrée pouvant être extraits et traduits par une règle d'interprétation.

Définition – *production* : Expression logique définissant l'ensemble des fragments des modèles intermédiaires pouvant être produits par une règle d'interprétation. Le fragment produit par une production est fonction du fragment identifié par le motif de la règle d'interprétation.

Définition – *règle d'interprétation* : Plus petit élément de spécification d'une fonction sémantique, constitué d'un motif et d'une production. Une règle d'interprétation est responsable de l'extraction et de la traduction d'un type d'informations. Elle produit pour chaque fragment identifié par son motif au sein d'un modèle d'entrée un fragment sémantiquement équivalent instance du métamodèle coeur. Le fragment produit est inclus dans le modèle intermédiaire résultat de l'interprétation du modèle d'entrée.

La sémantique d'une règle d'interprétation est simple. La production d'une règle est exécutée pour chaque fragment identifié par son motif. Ainsi, l'exécution d'une règle associe à chaque fragment identifié au sein d'un modèle d'entrée un fragment sémantiquement équivalent instance du métamodèle coeur. Nous verrons en section suivante que le modèle intermédiaire résultant d'une interprétation est une combinaison de fragments produits par les règles d'interprétation exécutées. Une règle sémantique est en fait un constituant élémentaire d'une (ou plusieurs) fonction(s) sémantique(s). Dans la suite de cette section, nous illustrons par l'exemple les notions de motif et de production ; des exemples de règles d'interprétation sont fournis dans la section suivante.

La Figure 55 donne un exemple d'interprétation d'un type d'informations dans le contexte d'une fonction sémantique f : RDL → RM. Cet exemple se focalise sur l'interprétation de l'information relative à la notion de RELATION entre concepts métiers, au sens du formalisme RM. Le modèle en haut de la figure est un fragment de modèle RDM représentant une partie de la phrase RDL (g) de la Figure 43 (la phrase est reproduite sous le modèle et le texte souligné correspond à l'information représentée par le modèle). Le modèle du bas de la Figure 55 est un des fragments RM produit par f. Ce fragment capture la relation *connected* désignée par la phrase RDL interprétée. Si l'on cherche à caractériser la partie de la fonction sémantique f responsable de l'interprétation du type d'informations relatif à la notion de relation, on peut faire les remarques suivantes :

- La notion de RELATION est identifiable au sein d'une proposition *p* (CLAUSE) comportant (i) un verbe d'état (l'objet (a) dans la Figure 55 par exemple), (ii) un attribut du sujet, c'est-à-dire un complément contenant un adjectif (l'objet (b)) et (iii) au moins un complément autre que l'attribut du sujet (l'objet (c)). Cette condition constitue un motif suffisant pour détecter une relation au sein d'un modèle RDM.
- Outre la production d'une instance de RELATION, la détection d'une relation implique aussi la production d'au moins deux instances de REFERENCE. Les entités liées à ces références sont celles désignées par le sujet de *p* et les compléments de *p* autres que l'attribut du sujet (« A participant » et « to the server » dans la phrase RDL). Deux références sont produites dans l'exemple de la Figure 55. La première est liée à un Concept dont le nom provient du sujet (d). La deuxième est liée à un Concept dont le nom provient du complément (c), plus précisément de l'objet (e). On remarque ici que la production d'une REFERENCE dépend aussi du motif ayant permis l'identification de la RELATION liée.
- Enfin, on peut remarquer que deux motifs peuvent partager des points communs. Par exemple le motif identifiant un ATTRIBUTE partage les points (i) et (ii) du motif identifiant une

RELATION. Seul le point (iii) change : dans le cas d'un ATTRIBUTE, il ne doit pas exister d'autre complément que l'attribut du sujet. A titre d'exemple, le terme *connected* désigne un ATTRIBUTE dans le texte RDL suivant : « A *participant* must be *connected* before … ». Notons cependant qu'un Attribute peut aussi être identifié pour des configurations différentes comme par exemple « the *state* of the *camera* must be *setOff* » (dans ce cas, c'est *camera* et non *state* qui est l'entité attribuée).

A *participant* must be *connected* to the *server* of the *system* before he can *connect* to the *external server*.

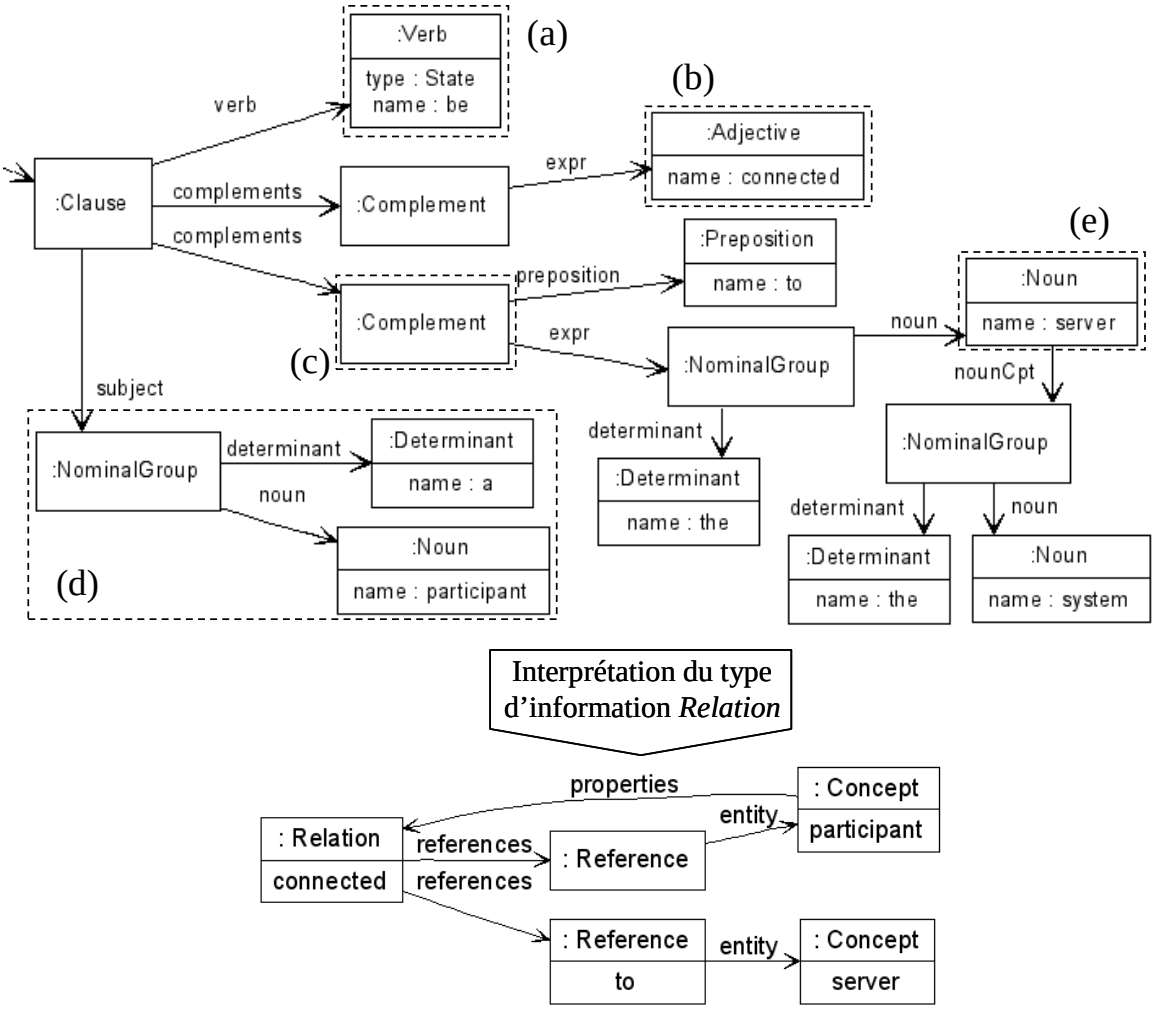
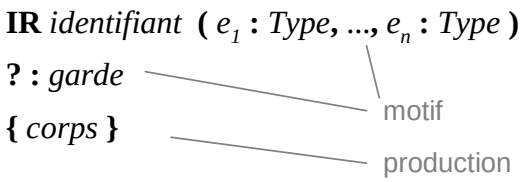


Figure 55 – Illustration de l'interprétation du type d'informations RELATION au sein d'un modèle RDM.

2.2.2 Description IRL des règles d'interprétation

Nous présentons ici la partie du langage IRL permettant la description des règles d'interprétation. Le schéma ci-dessous présente la syntaxe textuelle générique d'une règle d'interprétation (voir la grammaire IRL en Annexe G pour plus de détails sur la syntaxe IRL). Une règle d'interprétation (IR pour *Interpretation Rule*) a un nom unique (*identifiant*), un ensemble de *paramètres* (e_1,…, e_n) en lecture seule, une *garde* et un *corps*. Le motif d'une règle d'interprétation est définie par les paramètres sources et la garde. Cette dernière est une expression booléenne portant sur les paramètres sources ; elle retourne vrai si le motif a été identifié. La production d'une règle d'interprétation est définie par le corps. Ce dernier est un ensemble d'instructions produisant un fragment du modèle intermédiaire chaque fois qu'un fragment est identifié par le motif. Les instructions au sein du corps ont accès aux paramètres (dans le cas contraire, la traduction serait impossible).



La Figure 56 présente un extrait de la fonction sémantique f : RDM → RM (le code complet de cette fonction sémantique est disponible en Annexe H). Cet extrait est suffisant pour décrire comment spécifier des règles d'interprétation dans le langage IRL. Le langage RDL supporte trois constructions dédiées à la spécification des règles d'interprétation (*IR*) : des fonctions (*function*), des requêtes (*query*), des fragments de motifs (*pattern*). Les fonctions sont similaires aux fonctions proposées par les langages de programmation usuels ; elles permettent la factorisation d'un code (le code de la Figure 56 ne comporte pas d'exemple de fonction). Une requête est une fonction portant uniquement sur des éléments du modèle d'entrée (similaire aux requêtes QVT [142]) ; elle est évaluée une fois pour toutes lors de son premier appel [74]. Un fragment de motif est une requête retournant obligatoirement une valeur booléenne ; il autorise la factorisation d'un code partagé par plusieurs motifs.

Le code de la Figure 56 comporte deux règles d'interprétation (*conceptOrActor* et *relation*) responsable respectivement de l'interprétation des informations relatives aux entités (ENTITY) et aux relations (RELATION). Le motif de la règle *conceptOrActor* identifie tout fragment RDM contenant une instance *e* de ENTITYREF telle que si *e* est un groupe nominal (donc n'est pas un pronom), alors *e* ne doit désigner ni une propriété, ni une contenance (appel de *designateProperty*), ni un rôle (appel de *designateRole*, non fourni dans la Figure 56). On remarquera que les motifs des deux règles d'interprétation de la Figure 56 partagent l'appel au fragment de motif *designateProperty* ; directement pour le motif de la règle *conceptOrActor* et indirectement pour le motif de la règle *relation* (voir le code des fragments de motifs *relation* et *property*). Le motif de la règle *relation* implémente la logique d'identification d'une RELATION donnée informellement dans la section précédente (la requête *getCptsFromSubjectAttribute* retourne l'ensemble des compléments d'une proposition, mise à part l'attribut du sujet contenant l'adjectif fourni en paramètre).

Une fois un fragment identifié par le motif d'une règle, la production est exécutée pour produire le fragment du métamodèle cœur équivalent. Par exemple, la production de la règle *actorOrConcept* crée une entité : un acteur (ligne 18) si le paramètre *e* joue le rôle de sujet dans une proposition définissant une action (où le verbe est un verbe d'action) ou un concept sinon (ligne 19). La propriété *name* de l'entité créée est ensuite attribuée suivant le nom de *e* si ce dernier est un groupe nominal (ligne 20). Dans le cas où *e* est un pronom (lignes 22-24), la valeur affectée est déterminée par le nom du groupe nominal référencé dans la phrase. Cette valeur est fournie par l'appel à la méthode *getReferencedName*. Par exemple, la valeur retournée par cette méthode pour le pronom « he » dans la phrase « A participant is connected after he … » est « participant ». La production de la règle relation produit un fragment similaire au modèle du bas de la Figure 55. La production d'une règle d'interprétation produit un fragment inclus dans le modèle intermédiaire. Elle peut aussi créer des liens entre le fragment qu'elle produit et un ou plusieurs fragments produits par d'autres règles. Par exemple, la production de la règle *relation* crée des instances de REFERENCE liées à des entités créées par la production de la règle *conceptOrActor*. En somme, le modèle intermédiaire produit par interprétation est une combinaison des fragments de modèles produits par les règles d'interprétation.

[74] Question de performance.

```
query getCptsFromSubjectAttribute(a : Adjective) : List<<Complement>> {
        var o : Complement init getComplementFor(a);
        return a.clause().complements.select(c | c != o); }
pattern subject(e : EntityRef) = e.subject~ != void;
pattern relation(a : Adjective) = property(a) and getCptsFromSubjectAttribute(a).size() > 0;
pattern property(a : Adjective) = a.cpt() != void and and a.clause().verb.isStateVerb() and
        a.clause().subject instanceof NominalGroup =>
        not designateProperty(a.clause().subject#NominalGroup);
pattern designateProperty (e : NominalGroup) = e.noun.nounCpt != void and
        ((subject(e) => e.clause().subjectAttributeCpt() != void) or (complement(e)
        => e.clause().verb.isSetVerb())) and Dico.isProperty(e);
//------------ ENTITY -------------
IR conceptOrActor(e : EntityRef)
? : e instanceof NominalGroup => (not designateProperty(e#NominalGroup) and
        not designateRole(e#NominalGroup));
{
        var entity : Entity;
        if (subject(e) and e.subject~.verb.isActionVerb()) entity := Actor.new;
        else entity := Concept.new;
        if (e instanceof NominalGroup) c.name := e#NominalGroup#.noun.name;
        else {
                var l : List<String> init Resolver.getReferencedName(p);
                if (l.size() >= 1)
                entity.name  := l.get(0);
        }
}
//------------ RELATION -------------
IR relation(a : Adjective)
? : relation(a);
{
        var r : Relation init Relation.new;
        r.name := a.name;
        Entity@(a.clause()..subject).properties.add(r);
        var ref : Reference init Reference.new;
        r.references.add(ref);
        ref.entity := Entity@(a.clause().subject);
        getCptsFromSubjectAttribute(a).forall(c : Complement) {
                ref := Reference.new ;
                r.references.add(ref);
                ref.entity := Entity@(c.expr);
                ref.name := c.preposition.name;
        }
        var b : Boolean init Boolean.new;
        r.type := b;
}
```

Figure 56 – Extrait du code IRL de la fonction sémantique f : RDL → RM.

Pour spécifier les liens entre ces fragments, il est nécessaire de fournir un mécanisme permettant de référencer les objets créés durant l'exécution des règles d'interprétation. Pour cela, IRL offre une instruction dédiée (représentée par le caractère « @ ») appelée *référencement*. Un référencement fournit les objets d'un type particulier et créé durant l'exécution de la production d'une règle, étant donné un ensemble d'objets identifiés par les patrons. Par exemple, le référencement *Entity@(e)* (ligne 36) fournit une instance de la métaclasse ENTITY créée par une règle d'interprétation produisant un objet de ce type et dont *e* est un paramètre effectif lors de l'exécution de la règle. Cette instruction interroge en fait le modèle de traçabilité produit durant l'interprétation (cf. section 4.1). Un référencement retourne une instance ou une liste d'instances suivant le nombre d'objets trouvés. Une exception est levée si aucun objet n'est trouvé ou si les objets trouvés proviennent de l'exécution de règles distinctes. Notons que l'ordre dans lequel les règles sont exécutées dépend du code des productions (référencements et type des objets créés)[75]. Notons enfin qu'il est possible de restreindre explicitement le référencement à un ensemble de règles particulières. Par exemple, le référencement *Entity@[conceptOrActor](e)* limite la recherche aux objets crées par la règle *conceptOrActor*.

La section suivante décrit la manière dont les ambiguïtés syntaxiques sont traitées dans un programme IRL. La section 2.2.4 fournit ensuite une illustration de l'exécution du code IRL de la fonction sémantique f : RDL → RM. Le reste de cette section décrit brièvement quelques aspects du langage IRL nécessaires pour comprendre le code IRL fourni par la Figure 56 :

- La syntaxe du langage IRL est proche de celle de Kermeta (voir la syntaxe du langage en Annexe G pour plus de détails). Elle allie une syntaxe impérative classique (contrôle de flot, affectation) et une syntaxe déclarative « à la OCL » (navigations des propriétés des métaclasses comme *complements* ligne 3, clôtures lexicales comme *select* ligne 3, …).
- IRL intègre nativement Java et Kermeta. Une classe java peut être importée, instanciée, et ses propriétés statiques ou dynamiques manipulées (exécution des méthodes, accès aux attributs). La méthode *getReferencedName* (ligne 22) est par exemple une méthode statique de la classe java *Resolver* et la méthode *size* (ligne 23) est une méthode de l'interface standard *List*. Lorsque les modèles manipulés sont instances de métamodèles spécifiés avec Kermeta, les actions Kermeta sont aussi exécutables. Par exemple, la méthode *clause* (ligne 16) est une action Kermeta définie sur l'ensemble des métaclasses du métamodèle RDM et retournant la proposition contenant l'objet sur lequel la méthode est appelée[76] (voir l'Annexe F pour une description des actions Kermeta ajoutées à RDM).
- IRL offre un « sucre syntaxique » pour naviguer une association « à l'envers », ce qui limite l'ajout d'associations superflues au sein des métamodèles[77]. Cette navigabilité inverse est désignée par le caractère « ~ »[78] : l'expression *e.subject~* de la ligne 7 retourne donc (si elle existe) une instance o de la métaclasse CLAUSE telle que *o.subject* est égal à *e*.
- La valeur de retour de l'exécution d'un motif est « faux » si une exception est déclenchée[79]. En revanche, l'interprétation est arrêtée si une exception est déclenchée durant l'exécution d'une production.

2.2.3 Gestion des ambiguïtés locales

Les ambiguïtés dues à une imprécision syntaxique ou à un flou sémantique peuvent être résolues de manière locale (cf. section 2.2.2). Elles peuvent donc être traitées dès l'étape d'interprétation. Nous

[75] Dans certains cas, il est possible d'avoir des dépendances circulaires entre règles concernant leur priorité d'exécution. Ces dépendances sont détectables statiquement et peuvent être évités par la création de nouvelles règles.

[76] Les actions Kermeta des métaclasses du métamodèle d'entrée sont considérées comme des requêtes (elles ne sont exécutées qu'une fois).

[77] L'ajout d'associations conceptuellement inutiles dans un métamodèle dans le seul but d'autoriser la navigabilité « pollue » le métamodèle et rend plus difficile sa compréhension.

[78] Il peut y avoir indéterminisme si une métaclasse est pointée par deux associations de même nom. Dans ce cas, on ajoute le nom de la métaclasse contenant l'association pour lever l'ambiguïté. Par exemple, o.b~A retourne les objets instances de A et liés à o par un lien de type b.

[79] Ce choix favorise la concision du code en limitant la gestion des exceptions.

illustrons dans cette section la résolution d'une ambiguïté due à une imprécision syntaxique du langage RDL en suivant une stratégie de résolution homme-machine[80]. L'exemple de variation sémantique issu de f : RDL → RM a pour origine l'impossibilité dans certains cas de déterminer automatiquement le sens des fragments RDL du type « X of Y » comme ceux soulignés dans les trois phrases RDL ci-dessous. Ces trois phrases RDL font parties de l'exemple de composition de la Figure 43 (cette variation sémantique a été illustrée en section 2.2.2 pour la phrase (c)). En effet, un fragment de ce type peut être interprété de trois manières différentes dans le cas général : (i) X est une propriété de Y, (ii) il existe une relation de contenance entre Y et X, (iii) il existe un rôle X lié à Y.

(b) A *meeting* becomes *closed* after the *moderator* of this *meeting* did *leave* it.

(c) The *state* of the *camera* of a *participant* must be *setoff* after he does *leave* a *meeting*.

(h) A *participant* must be *connected* to the *server* of the *system* before he can *connect* to the *externalServer*.

Le choix d'une interprétation parmi celles possibles est effectué à l'aide d'heuristiques portant sur la sémantique du langage d'entrée (en l'occurrence le RDL), à l'aide d'informations complémentaires et à l'aide d'un dialogue homme machine avec les parties prenantes si nécessaire. La solution adoptée pour notre exemple marie les trois approches. Nous avons défini des heuristiques basées sur les fonctions grammaticales des éléments de la phrase RDL interprétée. Par exemple, il est impossible que X désigne une propriété de Y si le fragment fait partie d'un complément[81] (cas des phrases (b) et (h)) ou si X ne se situe pas au début d'un sujet ou d'un complément (cas de « the *camera* of a *participant* » dans la phrase (c)). Les heuristiques ne sont pas toujours suffisantes ; c'est le cas pour la phrase (b) par exemple car il est impossible de déterminer automatiquement si *moderator* est un rôle ou un concept contenu par un *meeting*. Dans ce cas, la fonction sémantique fait appel à un dictionnaire élémentaire (informations complémentaires) mis à jour au fur et à mesure de l'interprétation de phrases RDL.

Ce dictionnaire qualifie les noms rencontrés dans les phrases RDL (propriété, humain, chose inerte, chose vivante). Ce dictionnaire est mis à contribution si les heuristiques ne suffisent pas et une question est posée aux parties prenantes en dernier recours (dialogue homme-machine). Dans ce cas, le dictionnaire est mis à jour. La Figure 57 fournit deux des trois fragments de motifs responsables de la résolution de l'ambiguïté illustrée ; le troisième est le fragment de motif *designateProperty* fourni par la Figure 56. La classe java *Dico* implémente le dictionnaire. La résolution d'une ambiguïté à l'aide de ces fragments de motif est illustrée dans la section suivante.

```
01  pattern designateContainment(ng : NominalGroup) = ng.noun.nounCpt != void
        and not designateProperty(ng) and Dico.isThing(ng);
    pattern designateRole(ng : NominalGroup) = ng.noun.nounCpt != void
        and not designateProperty(ng) and not designateContainment(ng);
```

Figure 57 – Deux des trois fragments de motif jouant un rôle dans la résolution d'une ambiguïté syntaxique du RDL.

2.2.4 Illustration

La Figure 58 fournit le modèle intermédiaire résultant de l'exécution de f : RDL → RM pour la phrase RDL (h) de la Figure 43 (les modèles intermédiaires des modèles de cette figure sont tous fournis en Annexe K). Les fragments RM produits par les deux règles d'interprétation de la Figure 56 sont grisés : le fragment (1) a été produit par la règle *relation* et les fragments (2) à (6) ont été produits par la règle *conceptOrActor*. La Figure 58 comporte aussi la phrase RDL interprétée, présentée sous forme textuelle (la Figure 55 fournit une partie de sa représentation sous forme d'un modèle RDM). Les fragments RDL identifiés lors de la production des fragments RM grisés sont identifiés au sein de la phrase RDL avec le même identifiant. Par exemple, le fragment (1) a été produit après identification du fragment RDL « A *participant* must be *connected* to the *server* ».

[80] Nous choisissons cette stratégie car (i) nous désirons conserver l'aspect langage naturel du RDL (l'extension de la syntaxe n'irait pas dans ce sens) et (ii) nous ne désirons pas restreindre l'expressivité du RDL.

[81] Excepté si le verbe est un verbe de paramétrage ; nous n'entrons pas dans les détails.

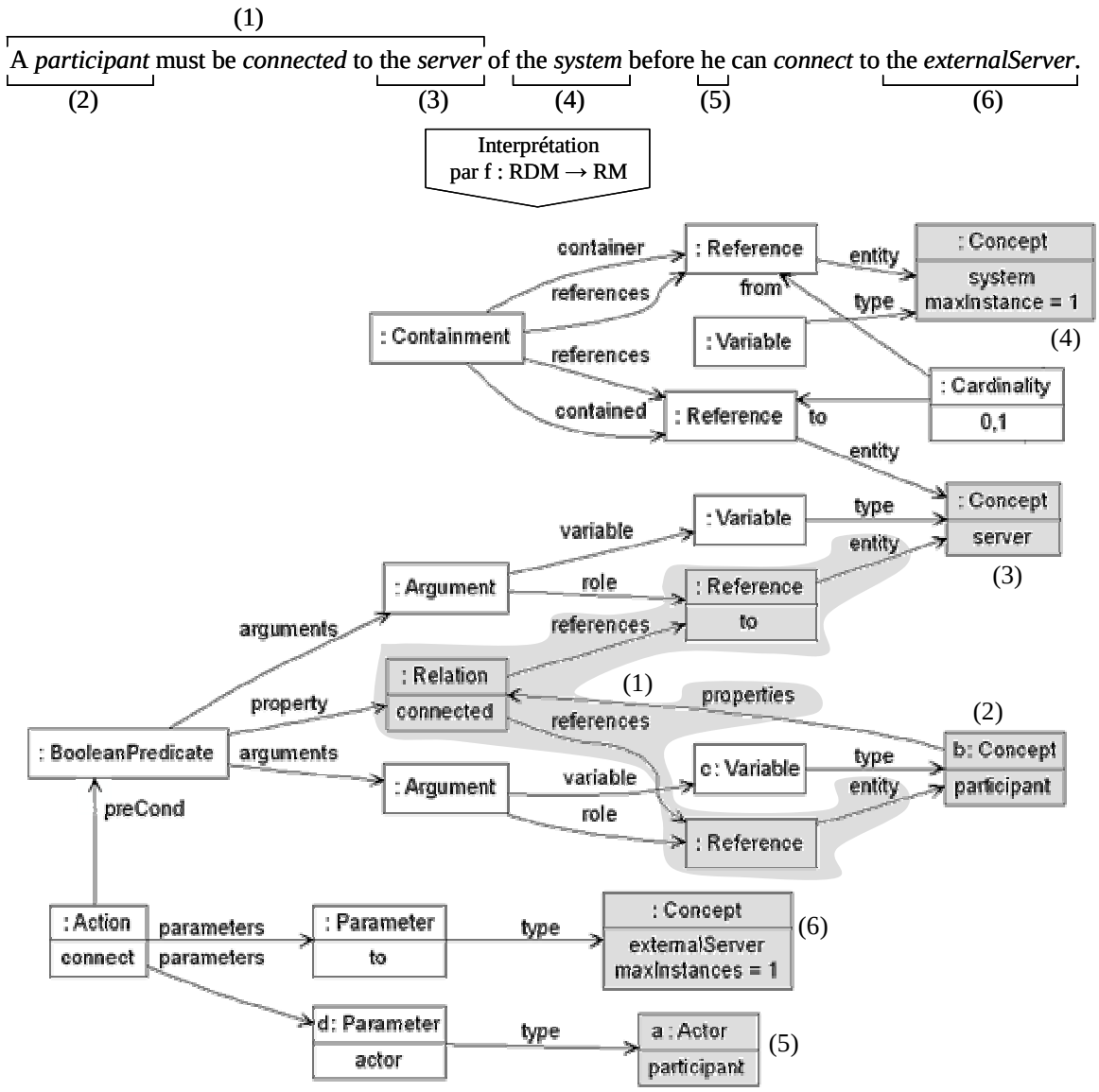


Figure 58 – Modèle intermédiaire obtenu après interprétation de la phrase RDL (h) de la Figure 43 par la fonction sémantique f : RDL → RM.

Le texte de la Figure 59 présente le dialogue homme-machine pendant l'interprétation[82]. Les trois premières questions résultent de l'appel de la méthode *getReferencedName* lors de l'interprétation du pronom « he » par la règle *conceptOrActor*. Cette méthode sélectionne l'ensemble des noms pouvant être référencés par le pronom (*participant*, *server* et *system* dans l'exemple). Puisque le pronom fourni est « he », ce dernier désigne un humain. Les trois premières questions ont pour but de disqualifier les entités susceptibles d'être référencées. Le résultat de la méthode *getReferencedName* pour « he » est donc « participant » d'où le nom de l'instance de ACTOR du fragment RM (5). La quatrième question résulte de l'appel au fragment de motif *designateContainment*, appelant la méthode *isThing* du dictionnaire (voir la Figure 57, ligne 1). Cette dernière question permet d'écarter la possibilité que *server* désigne un rôle.

La Figure 58 illustre bien le fait que chaque règle d'interprétation est responsable de la traduction d'un type d'informations donné. Les fragments sont reliés entre eux grâce aux référencements. Par exemple, les liens *type* des instances de REFERENCE sont instanciés par la règle *relation*, puisque cette règle n'est pas responsable de la traduction du type d'informations relatif aux entités. On peut aussi remarquer qu'un même objet peut être le résultat de l'application de plusieurs règles ; par exemple, les

[82] On se place dans un contexte où les informations du dictionnaire ne permettent pas de résoudre les ambiguïtés syntaxiques (c'est le cas lors de la première interprétation par exemple).

valeurs de l'attribut *name* des instances d'ENTITY sont définies par la règle *conceptOrActor* mais pas celles de la propriété *maxInstance* (ce type d'informations est sous la responsabilité de la règle *maxInstance*). On peut faire un constat similaire pour l'extraction des informations relatives à la cardinalité d'une relation (l'instance de CARDINALITY spécifiant le nombre de serveur pouvant être contenu par le système n'est pas de la responsabilité de la règle *containment*).

```
Does "participant" refer to a human ? (y/n).      > y
Does "server" refer to a human ? (y/n).           > n
Does "system" refer to a human ? (y/n).           > n
Does "server" refer to a thing ? (y/n).           > y
```

Figure 59 - Exemple de dialogue homme-machine pendant l'interprétation.

2.3 Nature des modèles intermédiaires produits

Le langage IRL est conceptuellement proche d'un langage à base de règles comme ATL [141]. Cependant, la spécification d'une fonction sémantique avec IRL diffère substantiellement d'une spécification avec ATL car le modèle produit a pour vocation d'être traité durant l'étape de fusion présentée en section suivante. Une fonction sémantique est en fait implémentée par un code IRL et la spécification de fusion commune à l'ensemble des fonctions sémantiques. En effet, la fusion permet de résoudre les redondances entre les modèles intermédiaires, de résoudre certaines ambiguïtés et de supprimer les malformations syntaxiques comme nous le verrons en section 3.

L'existence d'une étape de fusion diminue de fait la complexité des spécifications implémentant l'étape d'interprétation puisqu'une partie des spécifications des fonctions sémantiques est factorisée par le code implémentant l'étape de fusion. En outre, le traitement des superpositions sémantiques effectué par l'étape de fusion autorise la création de modèles intermédiaires (i) non-conformes au métamodèle coeur, (ii) comportant des redondances et (ii) des malformations syntaxiques. Nous décrivons dans cette section la nature des modèles intermédiaires produits par l'étape d'interprétation. La section 2.3.1 illustre la non-conformité des modèles intermédiaires et la section 2.3.2 souligne la redondance de l'information générée par l'interprétation.

2.3.1 Non-conformité et malformation syntaxique

On peut remarquer en observant la Figure 58 qu'un modèle intermédiaire comporte des malformations syntaxiques :

(i) L'action *connect* n'a pas de post-condition.

(ii) Il existe une instance de VARIABLE n'étant liée à aucune expression.

(iii) Il existe une instance de CONTAINMENT n'est pas liée à la description de l'action *connect*.

Certaines malformations sont détectables grâce à la relation de conformité. C'est le cas de la malformation (i), puisque l'absence d'une instance du lien *postCond* pour une ACTION constitue une violation de cardinalité en regard du métamodèle RM. D'autres comme (ii) et (iii) ne peuvent être détectées de cette manière. Il est envisageable de définir des contraintes structurelles supplémentaires OCL dans cette optique. Cependant, nous avons vu que dans le cas général, certaines contraintes structurelles ne sont pas exprimables. En outre, il n'est pas raisonnable de définir toutes les contraintes structurelles possibles. L'absence d'une durée pour l'action *connect* pourrait aussi être une malformation, bien que dans le cas du RM, on considère par convention qu'une action sans durée définie est considérée comme instantanée. Quoi qu'il en soit, ces malformations reflètent un manque d'information caractéristique des modèles intermédiaires puisqu'ils représentent une information partielle par nature (les modèles d'entrée). Ces malformations doivent être résolues durant l'étape de fusion.

Enfin, on peut remarquer qu'un modèle intermédiaire vérifie un ensemble de contraintes structurelles propres à la nature des fragments produits par les productions des règles d'interprétation puisqu'une production spécifie un type d'informations. A titre d'exemple, il est impossible q'un

modèle intermédiaire contienne une instance de Parameter non lié à une action car cet objet n'aurait alors aucun sens. De même, une instance de Property est forcément lié à une entité (via une instance de la propriété properties) et à un type. Dans le cas contraire, soit le code implémentant l'interprétation comporte une erreur, soit le modèle d'entrée n'est pas conforme à son métamodèle d'entrée (par exemple, un diagramme de classe où une partie prenante a omis de préciser le type d'un attribut d'une classe). Dans le deuxième cas, l'incohérence est détectable avant composition, mais rien n'empêche d'appliquer l'étape de fusion ; l'incohérence sera aussi détectée sur le modèle global.

2.3.2 Redondance de l'information

L'étape de fusion résout les redondances entre les modèles intermédiaires. Cependant, elle peut aussi résoudre des redondances au sein d'un même modèle intermédiaire. Cette caractéristique permet de limiter la complexité des spécifications IRL. C'est le cas de la spécification IRL f : RDL → RM. Le modèle intermédiaire de la Figure 58 illustre la création artificielle de redondance par f. A titre d'exemple, on peut remarquer deux superpositions sémantiques produit lors de l'interprétation d'un seul modèle d'entrée :

(i) Le concept métier *participant* est représenté deux fois ; par une instance de la métaclasse ACTOR (a) et par une instance de la métaclasse CONCEPT (b).

(ii) Le paramètre de l'action *connect* de type *participant* est représenté par les objets (c) et (d) (respectivement instances des métaclasses VARIABLE et PARAMETER).

La première redondance reflète la double référence à la notion de *participant* au sein de la spécification RDL textuelle. En effet, ce concept est représenté par le groupe nominal « A participant » et par le pronom « he ». La résolution de cette redondance est laissée à la charge de l'étape de fusion, ce qui limite la complexité du code IRL. Plutôt que de vérifier si une entité a déjà été créée pour représenter cette notion (cette entité pouvant être une instance d'Actor ou de Concept), la fonction f crée systématiquement une entité. La deuxième redondance reflète comme la première la double référence à la notion de participant. La fonction f génère systématiquement une instance de Variable pour toute entité, si cette dernière n'est pas liée à un paramètre d'une action. Cependant, il n'est pas possible dans ce cas de laisser à la charge de la fusion le soin d'identifier une superposition sémantique entre les objets (c) et (d). Alors qu'il est possible de statuer que les objets (a) et (b) désignent le même élément du domaine, il est impossible de déterminer si les objets (c) et (d) sont équivalents. Seul la connaissance de la spécification RDL permet d'être conscient de cette équivalence (grâce au référencement du pronom « he »). Il est donc nécessaire dans ce cas de préciser que (c) et (d) sont équivalents.

```
01  IR variableEquivalence(e : EntityRef)
    ? : variableOrParameter(e);
    {
        var l : List<<NominalGroup>> init Resolver.getReferencedNominalGroups(e);
05      if (l.size() >= 1)
                Tracer.declareAsEquivalent(Variable@(e), Variable@(l.get(0)), "variable");
    }
```

Figure 60 - Déclaration d'équivalences entre objets d'un modèle intermédiaire durant l'étape d'interprétation.

Le langage IRL permet de déclarer des équivalences devant être résolues durant l'étape de fusion grâce à une instruction native. La Figure 60 fournit la règle d'interprétation déclarant l'équivalence entre des instances de VARIABLE. Cette règle ne produit pas de fragments ; elle se contente de déclarer équivalent les variables (instances de PARAMETER ou VARIABLE) produits pour des ENTITYREF différents mais désignant la même instance d'ENTITY (par exemple, « A participant » et « he » dans la phrase de la Figure 58), grâce à l'appel de la méthode *declareAsEquivalent* (ligne 6 de la Figure 60). Les équivalences déclarées durant l'étape d'interprétation sont capturées par un modèle traité durant l'étape de fusion, présentée en section suivante. Ce mécanisme est aussi utilisé pour définir des équivalences entre modèles iintermédiaires lorsqu'il existe des relations sémantiques entre des champs

d'une notation. A titre d'exemple, les objets produits par interprétation et représentant les instances *p*, *f* et *m* dans les modèles d'entrée (e) et (f) (champs scope et diagram de la notation diagramme d'activité) sont déclarés équivalents durant l'interprétation en utilisant la méthode *declareAsEquivalent*.

3 Fusion

Nous avons vu en section précédente l'étape d'interprétation et la nature des modèles intermédiaires produits. Ces modèles capturent dans les termes du formalisme les informations décrites par les parties prenantes à l'aide des langages d'entrée et des notations supportés par le processus de composition. L'interprétation ne résout pas la décomposition initiale des informations sur les exigences d'un système rédigées par une multitude de parties prenantes. C'est le but de l'étape de fusion, c'est-à-dire la production d'une vue globale des exigences représentée par le modèle global. En outre, l'étape de fusion est aussi responsable de la résolution des ambiguïtés nécessitant un contexte global Nous détaillons dans cette section les principes de l'étape de fusion.

Cette section est organisée comme suit. La section 3.1 offre une vue générale de l'étape de fusion. La section 3.2 présente un ensemble de fragments de modèles intermédiaires résultant de l'interprétation des modèles d'entrée de la Figure 43 et caractérise l'étape de fusion par l'exemple. La section 3.3 détaille techniquement l'approche adoptée. Le langage FRL est présenté dans ses grandes lignes et l'étape de fusion est illustrée avec les fragments de modèles de la section 3.2. Nous présentons les deux types de règles de fusion proposées, à savoir les règles d'équivalences et les règles de normalisation, et discutons de l'ordonnancement de leur exécution. Nous montrons de plus la manière dont les ambiguïtés résultant de la décomposition de l'information sont traitées et résolues. Enfin, la section 3.4 donne une illustration de l'exécution de l'étape de fusion sur les fragments de modèles de la section 3.2.

3.1 Vue générale

L'étape de fusion a pour but (i) l'identification de superpositions sémantiques au sein des modèles intermédiaires et (ii) la résolution de ces superpositions sémantiques de sorte qu'il ne reste plus une seule redondance. Nous avons brièvement illustré cette problématique en section 2.3. L'identification de superpositions sémantiques repose sur une comparaison syntaxique des modèles intermédiaires[83]. Ces superpositions concernent des objets appartenant à des modèles intermédiaires, mais aussi dans certains cas à un même modèle intermédiaire puisque l'interprétation peut produire des superpositions artificielles (voir la section 2.3.2). L'identification d'une superposition porte généralement sur la comparaison d'objets de même type et ayant des similitudes concernant les valeurs de leur propriété. L'approche classique pour réaliser une fusion de modèles est de comparer les objets des modèles intermédiaires deux par deux et de les combiner sur la base d'équivalences syntaxiques. Cependant ce cas est particulier et l'équivalence entre deux objets ne peut être systématiquement déterminée par égalité syntaxique. En outre, l'efficacité de cette approche est limitée par les phénomènes de collisions. Nous avons aussi vu que l'identification d'une superposition peut dépendre d'une autre identification et plus généralement de la sémantique du métamodèle coeur.

La résolution d'une superposition décrit le remplacement d'un ensemble d'objets équivalents par un seul nouvel objet. Les propriétés de cet objet sont affectées de manière à ce qu'il reflète correctement l'information capturée par les objets remplacés. Dans certain cas, une résolution d'équivalence revient à une simple union des valeurs des propriétés des objets remplacés (c'est le seul cas considéré par Sabetzadeh et Easterbrook [40]). Cependant, une résolution d'équivalence est plus complexe dans le cas général car (i) les objets remplacés peuvent être de types différents, (ii) l'obtention de la valeur d'une propriété du nouvel objet peut nécessiter un calcul autre qu'une simple union et (iii) elle peut nécessiter la création de nouveaux fragments d'objets. Par exemple, dans le cas

[83] La structure d'un modèle (la syntaxe) représente le sens d'un modèle, étant donné la sémantique de son métamodèle.

d'un ensemble d'objets e instances de PROPERTY et désignant une même propriété, l'entité contenant cette propriété correspond à l'entité la plus générale dans la hiérarchie de type (relation *superTypes*) liant les entités contenant les objets de e.

Les modèles intermédiaires sont instances du métamodèle cœur mais pas toujours conformes (voir section 2.3.1). Le modèle global est conforme au métamodèle cœur relâché[84] puisque la résolution peut faire apparaître des violations portant sur les bornes supérieures des cardinalités. Ces violations peuvent refléter des incohérences au niveau des exigences ; elles sont analysées durant l'étape d'analyse. Cependant, d'autres peuvent refléter des malformations syntaxiques. Par exemple, une action ne comportera pas de post-condition au niveau du modèle global après résolution des superpositions si aucun modèle d'entrée ne spécifie d'informations en ce sens. Cette malformation syntaxique (détectée dans ce cas par la relation de conformité du métamodèle RM) doit être supprimée de telle sorte qu'une incohérence détectée dans le modèle global après fusion corresponde effectivement à une incohérence au niveau des exigences. Ce traitement est appelé normalisation et est à la charge de l'étape de fusion.

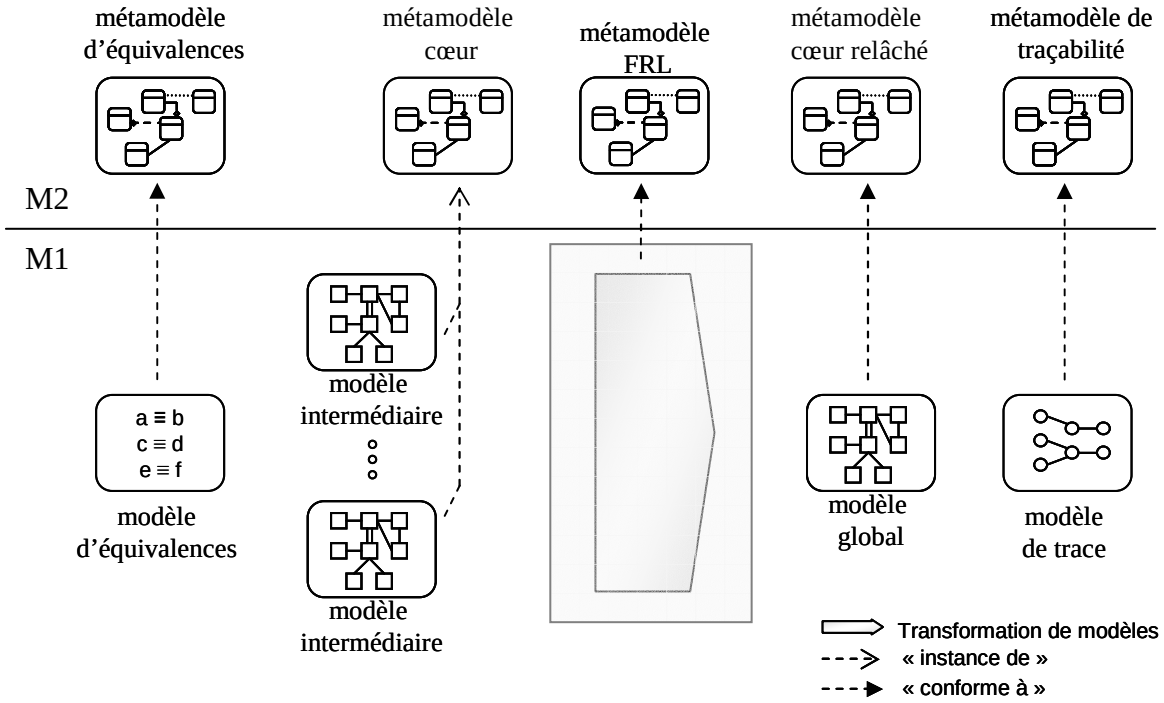


Figure 61 - Schématisation de l'étape de fusion.

La Figure 61 schématise l'étape de fusion dans une perspective IDM. Elle est implémentée par une transformation de modèle spécifiée par un ensemble de règles de fusion (*règles d'équivalences* et *règles de normalisation*, voir la section 3.3.1). Dans le cadre de la plate-forme R2A, ces règles sont spécifiées par un code FRL (pour *Fusion Rule Language*). Cette transformation est commune à l'ensemble des fonctions sémantiques des langages d'entrée. Elle accepte en entrée l'ensemble des modèles intermédiaires produits par interprétation ainsi qu'un *modèle d'équivalence* décrivant les superpositions détectées durant l'interprétation et non détectable par la seule analyse des modèles intermédiaires. Le modèle global est construit progressivement par une application itérative des règles de fusion (la première version du modèle global rassemble l'ensemble des modèles intermédiaires au sein d'un seul modèle). L'ordre d'exécution de ces règles est partielle et explicitement défini (voir la section 3.3.2). Le langage FRL supporte un mécanisme de traçabilité natif, tout comme IRL. L'étape de fusion produit donc en plus du modèle global un modèle de traçabilité, qui, associé au modèle de traçabilité produit durant l'étape d'interprétation, permet d'associer sémantiquement les objets du modèle global aux objets des modèles d'entrée (voir section 4.2).

[84] Il ne peut être que conforme puisqu'un métamodèle relâché ne comporte aucune contrainte structurelle par définition.

L'étape de fusion porte sur l'ensemble des modèles intermédiaires, ce qui permet de déterminer quelles variations sémantiques choisir lorsque ces dernières ont pour cause la décomposition de l'information statique. En revanche, les variations sémantiques dynamiques du même type doivent être résolues par un dialogue avec les parties prenantes. Nous avons vu en section 2.2.3 que cette résolution peut être effectuée de deux manières possibles : (i) par proposition exhaustive des alternatives possibles ou (ii) par simulation ou génération de scénarios. Nous illustrons la première possibilité et discutons des problèmes liés à la deuxième en section 3.4.

3.2 Exemples de fragments à fusionner et caractérisation de l'étape de fusion

La Figure 62 fournit quelques exemples de fragments des modèles intermédiaires obtenus après interprétation des modèles d'entrée de la Figure 43. Chaque fragment est nommé par une lettre correspondant au modèle d'entrée dont il est issu (par exemple, le fragment (e) est un extrait du modèle intermédiaire produit par interprétation du modèle d'entrée (e)). Les modèles intermédiaires complets sont fournis en Annexe K. Nous illustrons ci-dessous les observations faites dans le paragraphe précédant :

- On peut identifier plusieurs superpositions sémantiques simples dans la Figure 62. Par exemple la notion de *server* est représentée par plusieurs instances de CONCEPT parmi les fragments (e) et (h). Dans ce cas, la résolution consiste en une fusion simple de ces instances. Elles sont remplacées par une unique instance de CONCEPT où les valeurs des propriétés sont obtenues par une *stratégie de combinaison* des valeurs primitives similaire à celle proposée par Sabetzadeh et Easterbrook [40]. Cette stratégie de combinaison supprime les redondances (par exemple, on a {*server*} + {*server*} = {*server*, *server*} = {*server*}).
- Certaines superpositions sont plus complexes car elles dépendent de l'identification d'autres superpositions. Par exemple, les paramètres *actor* des fragments (d) et (g) sont équivalents car les instances de ACTION de ces fragments désignent la même action.
- D'autres superpositions sont plus complexes car elles mettent en jeu des instances de métaclasses différentes. Par exemple, la notion de *participant* est représentée par une instance de CONCEPT dans le fragment (e) et une instance de ACTOR dans les fragments (g) et (f).
- Les superpositions les plus complexes mettent en jeu des fragments de modèles. Elles ont pour cause une collision structurelle due à un manque d'information au sein des modèles d'entrée lors de l'interprétation. A titre d'exemple, l'ensemble des objets du fragment (e) désigne la même information que l'ensemble constitué des instances de CONCEPT et D'ATTRIBUTE du fragment (f), à savoir la relation *connected* entre un *participant* et un *server*. La notion de *moderator* est un autre exemple de notion pouvant impliquer des collisions structurelles et donc des superpositions complexes. En effet, le fragment (f) représente la notion de *moderator* à l'aide d'une instance de ACTOR alors que le fragment (b) la représente comme un rôle. Dans ce cas, la résolution de cette superposition est particulièrement complexe car elle a un impact sur la pré-condition de l'action *enter* du fragment (f). Ce point est détaillé en section 3.4.
- Enfin, on peut remarquer que la fusion ne se limite pas à la résolution de superpositions sémantiques. Nous avons vu en section 2.2.2 que la composition de modèles d'entrée peut révéler des ambiguïtés compositionnelles devant être résolues pour obtenir un modèle global. Par exemple, la résolution de la superposition sémantique entre les instances de ACTION des fragments (e) et (f) produit une instance d'ACTION liée à deux instances BOOLEANPREDICATE via la relation *preCond*. Cette résolution provoque la violation de la cardinalité de la relation *preCond*, ce qui dans ce cas reflète une ambiguïté compositionnelle. Une violation peut aussi refléter une malformation syntaxique devant être corrigée. Par exemple, la résolution des superpositions des fragments de la Figure 62 produit une instance de ACTION ne comportant pas de post-condition, ce qui n'est pas conforme au métamodèle RM.

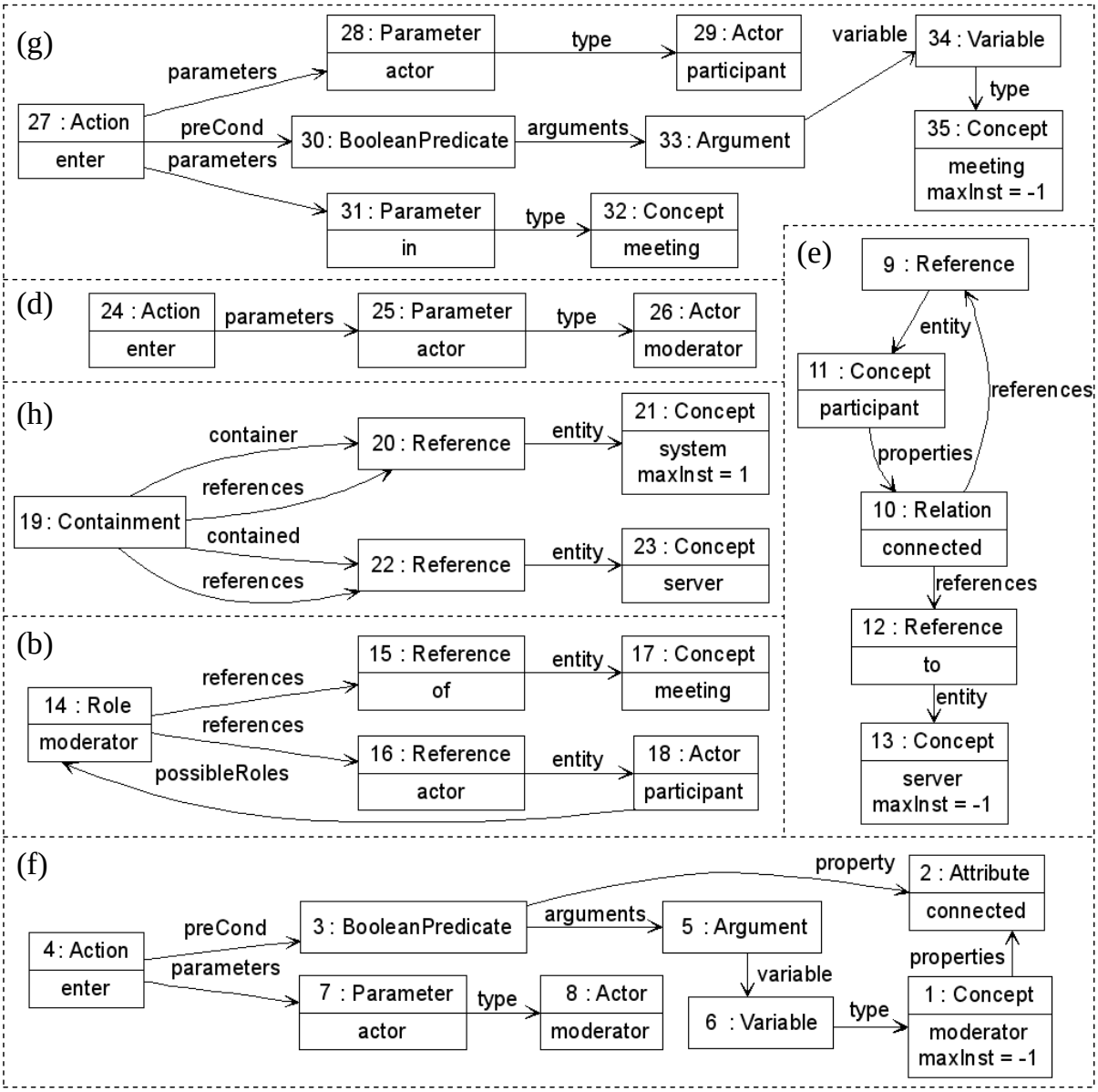


Figure 62 - Quelques fragments des modèles intermédiaires obtenus après interprétation des modèles d'entrée de la Figure 43.

3.3 Approche adoptée et implémentation avec le langage FRL

Cette section décrit le langage FRL. Ce langage a été construit en tenant compte des observations faites précédemment concernant l'identification des équivalences et la résolution des modèles intermédiaires. La section 3.3.1 présente les règles de fusion FRL et la section 3.3.2 la stratégie d'ordonnancement adoptée concernant leur exécution. L'exécution d'un code FRL est illustrée en section 3.4.

3.3.1 Notion de règle de fusion

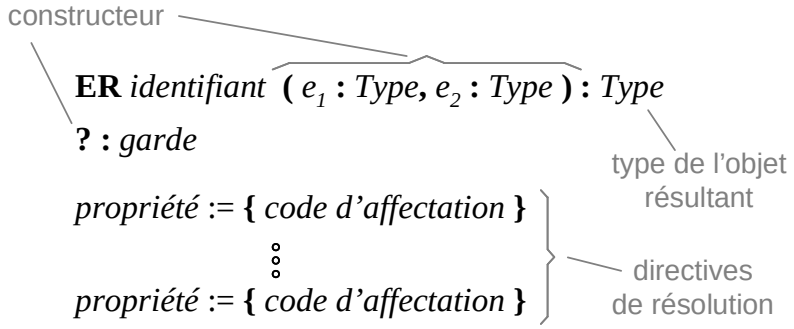
Nous avons défini le langage FRL (*Fusion Rule Language*) pour spécifier l'étape de fusion d'un processus de composition. FRL partage un grand nombre de points communs avec IRL : FRL intègre Java et Kermeta, supporte les fonctions, la navigation et l'interrogation des modèles « à la OCL »[85] et est aussi basé sur une approche à base de règles. Un code FRL définit une *sémantique*

[85] Nous avons déjà souligné qu'ils partagent en fait une grande partie de leur spécification.

compositionnelle pour un métamodèle[86], le métamodèle cœur dans le cadre d'un processus de composition d'exigences. Une sémantique compositionnelle est spécifiée à l'aide de *règles de fusion*. Nous décrivons dans cette section le principe des règles de fusion, leur sémantique, ainsi que leur syntaxe en FRL. L'exécution de règles de fusion est illustrée en section 3.4. Les règles de fusion sont basées sur la reconnaissance de motif comme les règles d'interprétation. On distingue deux types de règles :

- Les *règles d'équivalences* (ER pour *equivalence rule*) : une règle d'équivalences spécifie la manière d'identifier et de résoudre un type de superposition sémantique. Elle a pour effet de fusionner un ensemble d'objets du modèle global et de le remplacer par un nouvel objet sémantiquement équivalent (appelé *objet résultant*). Une règle d'équivalences peut aussi créer de nouveaux objets si nécessaire.
- Les *règles de normalisations* (NR pour *normalization rule*) : une règle de normalisation spécifie une modification particulière du modèle global en cours de construction. Une règle de normalisation peut être utile pour résoudre les collisions structurelles de sorte que les règles d'équivalences puissent être appliquées. Elle peut aussi être utile pour résoudre les malformations syntaxiques. Une règle de normalisation peut aussi bien créer ou supprimer des objets au sein du modèle global.

Le schéma ci-dessous présente la syntaxe textuelle générique d'une règle d'équivalences spécifiée dans le langage FRL (voir la grammaire FRL en Annexe H pour plus de détails). Une règle d'équivalences est constituée d'un *constructeur de classes d'équivalences* (*constructeur*), d'une *stratégie de résolution* et d'une référence vers une métaclasse du métamodèle cœur définissant le type de l'*objet résultant* après résolution. Le constructeur d'une règle d'équivalences est représenté par deux paramètres typés (*e1* et *e2*) et une *garde*. Il permet, tout comme le motif d'une règle d'interprétation, d'identifier les objets du modèle global sur lesquels la règle est appliquée. La garde est une expression booléenne[87] acceptant en entrée une pair d'objets du modèle global satisfaisant les types du motif et retournant vrai s'ils sont considérés équivalents par la règle (c'est-à-dire si les éléments du domaine qu'ils désignent se superposent). Le constructeur d'une règle produit donc un ensemble de pairs d'objets équivalents. Ces pairs sont ensuite combinés dans le but d'obtenir un ensemble de *classes d'équivalences*.



Une classe d'équivalence est un ensemble d'objets du modèle global ayant été identifiés comme équivalents deux à deux par une règle d'équivalences. Une classe d'équivalence est construite par transitivité à partir des résultats de l'application du constructeur d'une règle sur l'ensemble des paires d'objets satisfaisant les types du motif. La notion de classe d'équivalence est définie en termes mathématiques ci-dessous. A titre d'exemple, la règle *concept* définit comme équivalent les instances de CONCEPT ayant le même nom (valeur de la propriété *name*). Les trois pairs d'objets (17, 32) et (17, 35) et (32, 35) de la Figure 62 sont donc identifiées comme équivalents par cette règle puisqu'elles sont toutes des concepts désignant la notion de *meeting*. Par transitivité, on obtient la classe

[86] Cette notion est définie dans la section 1 du Chapitre VII.

[87] FRL supporte les expressions déclaratives ou impératives pour l'ensemble de ses constructions comme les gardes, les directives, l'initialisation de variables …). Dans l'exemple de la Figure 63, les constructions sont toutes spécifiées de manière déclarative.

d'équivalence $\{26, 56, 62\}_{concept}$. L'intérêt des classes d'équivalences est la spécification des stratégies de résolution dans un contexte global.

Définition – *Classes d'équivalences détectées par une règle d'équivalences* : Soit O l'ensemble des objets du modèle global et X une règle d'équivalences, l'ensemble des classes d'équivalences détectées par X est :

$$\{[a] \subset O \mid a\ R_X{}^*\ b\}$$

avec :
- $R_X = \{ (a, b) \in E \times E \mid X.constructor(a, b) \}$
- R^* la fermeture transitive de R.
- X.constructor le constructeur de X (fonction booléenne).

Notation - On note $\{a_1, \ldots, a_n\}_X$ une classe d'équivalence identifiée par la règle d'équivalences X et contenant les objets $a_1, \ldots, a_n$ du modèle global.

Une classe d'équivalence identifiée est résolue en suivant la stratégie de résolution de la règle d'équivalences l'ayant identifiée. Une stratégie de résolution spécifie les valeurs de chacune des propriétés de l'objet résultant de sorte que ce dernier soit sémantiquement équivalent à l'ensemble des objets de la classe d'équivalence résolue. Une stratégie de résolution associe à chaque propriété de l'objet résultant une *directive de résolution*. Cette dernière est un code retournant la valeur à affecter à l'objet résultant en fonction des objets de la classe d'équivalences (la classe d'équivalence en cours de résolution est un ensemble accessible par le mot clé *equRange*). Les directives de résolution sont optionnelles. Si aucune directive n'est définie pour une propriété, une stratégie par défaut est adoptée. Elle consiste à faire l'union des valeurs de la propriété pour l'ensemble des objets de la classe d'équivalence en cours de résolution. Dans le cas où la propriété est de type primitif, le résultat de l'union est normalisé de telle sorte qu'il n'existe pas de doublon dans l'ensemble résultat (stratégie similaire à celle adoptée systématiquement par Sabetzadeh [40]).

```
ER concept (c1 : Concept, c2 : Concept) : Concept
    ? : c1.name == c2.name;

ER Parameter(p1 : Parameter, p2 : Parameter) : Parameter
    ? : p1.parameters~ == p2.parameters~ and p1.role != void and p1.role == p2.role;

ER attributeAndRelation(r : Relation, a : Attribute) : Relation
    var e1 : Entity init r.properties~[0];
    var e2 : Entity init a.properties~[0];
    ? : r.name == a.name
        and (e1 == e2 or Algo.linkedBySuperType(e1,  e2));
    properties~ := Algo.fcs(equRange.collect(a | a.properties~[0]).flatten);
```

Figure 63 – Trois règles d'équivalences extraites de ψ_{RM}.

La Figure 63 fournit trois exemples de règles d'équivalences extraites de ψ_{RM}, la sémantique compositionnelle du métamodèle RM supportée par la version actuelle de la plate-forme R2A (la version complète de ψ_{RM} est fournie en Annexe K). La règle d'équivalences *concept* spécifie que deux concepts ayant le même nom sont équivalents. Cette règle produit donc la classe d'équivalence $\{26, 56, 62\}_{concept}$ donnée en exemple plus haut. La règle *concept* ne donnant aucune directive de résolution particulière, la stratégie de résolution par défaut est adoptée pour toutes les propriétés de l'instance de CONCEPT résultante. Plus complexe, la règle *attributeAndRelation* spécifie qu'une relation et un attribut sont équivalents s'ils ont le même nom et sont la propriété d'une même entité ou de deux entités liées par une relation de super-type (dans ce cas, la méthode java *linkedBySuperType* retourne vrai). La règle *attributeAndRelation* comporte une directive de résolution pour la propriété *properties* entre la métaclasse ENTITY et PROPERTY. Cette directive choisit pour la valeur de cette propriété l'entité la plus haute dans la hiérarchie super-type pour les relations et attributs de la classe

d'équivalence en cours de résolution (cette valeur est calculée par la méthode java *fcs*[88]). La stratégie de résolution par défaut est adoptée pour les autres valeurs de propriétés de la RELATION résultante.

```
var treatedParametersInRole : List<Parameter> init ArrayList<Parameter>.new;
NR entityDesignateRole(role : Role, e : entity)
var p : Parameter init e.type~[0]#Parameter;
var a : Action init p.parameters~[0];
? : role.name == e.name;
! : {
    e.name := void;
    p.type.add(role.references.select(r | r.name == "actor")[0].entity);
    if (not treatedParametersInRole.contains(p)) {
        var v : Variable init Variable.new;
        v.role := role.references.select(r | r.name == "of")[0].entity;
        var exp : Expression init a.preCond;
        if (expr == void) a.preCond := createExists(v, createRolePredicate(role, e.type~[0], v));
        else a.preCond := createExists(v,
            createImplies(createRolePredicate(role, e.type~[0], v), expr));
        exp := a.postCond;
        if (expr != void)
            a.preCond := createImplies(createRolePredicate(role, e.type~[0], v), expr);
                    treatedParametersInRole.add(p);
    }
}
NR superTypeAndTypeRelations(e1 : Entity, e2 : Entity, v : Variable)
? : v.type!.contains(e1) and v.type!.contains(e2) and Algo.linkedBySuperType(e1, e2);
! : {
        if (Algo.isSuperType(e1,e2)) v.type!.remove(e1);
        else v.type!.remove(e2);
}
NR preCondVoid(a : Action)
? : a.preCond == void;
! : {a.postCond := True.new;}
```

Figure 64 – Trois règles de normalisation extraite de ψ_{RM}.

Le schéma ci-dessous présente la syntaxe textuelle générique d'une règle de normalisation FRL. Les règles de normalisation assistent les règles d'équivalences. Elles sont similaires aux règles d'interprétation, mis à part le fait que les objets identifiés par le motif et les objets créés par le *code de normalisation* (la production pour les règles d'interprétation) sont instances du même métamodèle (le métamodèle cœur dans notre cas) et font partie du même modèle. On peut considérer une règle de normalisation comme une règle de réécriture. Une règle de normalisation comporte un motif et un code de normalisation. Le motif consiste en un ensemble de paramètres typés et une garde. Un fragment du modèle global est identifié par une règle de normalisation s'il comporte un ensemble d'objets distinct instances des types des paramètres et si ces objets satisfont à la garde. Le code de normalisation est appliqué pour chaque fragment identifié. On peut classer les règles de normalisation en trois grands groupes suivant leur rôle au sein de l'étape de fusion :

[88] Acronyme de *First Common Supertype*.

(i) Les règles de normalisation peuvent transformer le modèle global dans le but de résoudre les collisions structurelles de sorte que les règles d'équivalences puissent identifier correctement des superpositions complexes.

(ii) Elles peuvent servir à normaliser la structure du modèle global de manière à limiter la complexité des règles d'équivalences.

(iii) Elles peuvent servir à résoudre des malformations syntaxiques provoquant des violations de conformité mais ne reflétant pas une incohérence au niveau des exigences.

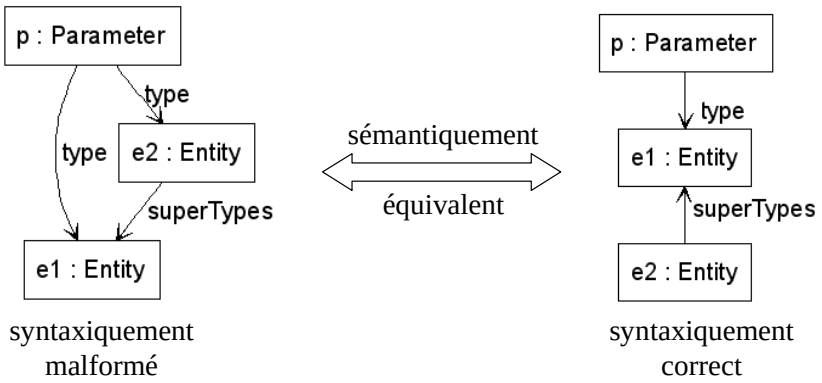
NR *identifiant* **(** e_1 **:** *Type*, ..., e_n **:** *Type* **)**

? : *garde* motif

! : { *code de normalisation* **}**

La Figure 64 fournit trois exemples de règles de normalisation extraites de ψ_{RM}. Chacune illustre un des groupes introduits plus haut. L'exécution de ces règles est illustrée en section 3.4. La règle *entityDesignateRole* vise à transformer le modèle global lorsqu'une entité désigne en fait un rôle (la Figure 50 illustre cette situation). Elle vise la résolution d'une collision structurelle complexe (premier groupe). Lorsqu'une entité et un rôle ont le même nom et que l'entité est un type de paramètre d'une action, cette règle met à jour le type du paramètre en fonction du type du rôle et met à jour les pré- et post-conditions de l'action pour prendre en compte le fait que le paramètre est un rôle.

La règle *superTypeAndTypeRelation* est représentative du deuxième groupe. Elle met à jour les propriétés *type* des VARIABLE (une règle de normalisation similaire existe pour les propriétés *entity* des REFERENCE) de sorte que la sémantique du mécanisme d'abstraction de classification (représentée par la relation *superTypes* de la métaclasse ENTITY) soit respectée. Par exemple, un paramètre lié à deux entités *e1* et *e2* par la propriété type est équivalent sémantiquement à ce même paramètre uniquement lié à *e1*, si *e1* est un super-type de *e2* (voir le schéma ci-dessous). La règle *superTypeAndEntityRelation* veille à ce que la structure du modèle global vérifie le deuxième cas. Cette règle limite la complexité des règles d'équivalences dont le motif porte sur la propriété type de VARIABLE. Enfin, la règle *preCondVoid* veille à ce que la représentation d'une action n'ayant pas de pré-condition ait une instance de la métaclasse TRUE pour valeur de la propriété *preCond*. Cette règle appartient au troisième groupe.



3.3.2 Exécution et ordonnancement des règles

Contrairement à l'interprétation, les fragments identifiés par les règles de fusion et les fragments modifiés par ces mêmes règles appartiennent au même modèle (le modèle global). L'application d'une règle de fusion a donc un impact sur les fragments pouvant être identifiés une fois la règle exécutée. De ce fait, les règles de fusion sont implicitement ordonnées : l'application d'une règle n'est possible que lorsqu'une ou plusieurs autres règles ont été appliquées. Par exemple, la règle d'équivalences *parameter* identifie deux paramètres comme équivalents si les actions correspondantes ont été préalablement identifiées comme équivalente et fusionnées (via la règle d'équivalences *action*). Le langage FRL permet de définir un ordre partiel d'exécution des règles de fusion. La Figure 65 présente l'ordonnancement suivi par les règles de fusion de ψ_{RM}.

L'ordonnancement est décrit dans un code FRL par une expression régulière simple mettant en jeu des ensembles de règles spécifiés entre accolades. Par exemple, la ligne 12 spécifie un ensemble de deux règles et signifie que ces deux règles doivent être exécutés. Il est possible de spécifier une application itérative d'un ensemble de règles de fusion (présence du terminal *)[89]. Par exemple, l'exécution des règles du deuxième ensemble (lignes 4-6) est itérative. Le critère d'arrêt d'une itération est satisfait lorsque aucune règle de l'ensemble n'a été appliquée pour la durée d'un cycle entier. La section suivante illustre l'ordonnancement présenté par la Figure 65. Le support d'un ordre partiel limite la complexité des gardes des règles de fusion.

```
01  //Combinaison
    { entityDesignateRole }
    { boolean, actorAndConcept, actor, concept, actorAndConceptVoid, conceptVoid, actorVoid,
      attribute, relation, attributeAndRelation, action, enumeration,  literal, reference, role, parameter,
05    superTypeAndEntityRelations, superTypeAndTypeRelations, oneEntityWithNoName}*
    //Complétion
    { variable, deleteExists, addExists }
    { multiPreCond, multiPostCond}
    { normalizeNot, normalizeBinaryExpr }*
10  { preCondVoid, postCondVoid }
```

Figure 65 - ordonnancement des règles de fusion de ψ_{RM}.

Nous avons identifié trois grandes phases durant l'étape de fusion :

- ***Résolution des équivalences initiales*** : Ces équivalences sont données en entrée de la fusion. Elles sont générées par l'étape d'interprétation (ou définies par l'utilisateur dans le cadre de la composition de modèles de conception, voir Chapitre VII).
- ***Combinaison*** : La composition de l'information en tant que telle. Les superpositions sémantiques sont identifiées et résolues.
- ***Complétion*** : Le modèle global après combinaison peut comporter des malformations syntaxiques dues à un manque d'information. Ces malformations doivent être résolues car elles ne reflètent pas une incohérence.

On peut faire une analogie avec les trois phases identifiées par France [78]. Cependant, les modèles d'entrée ont été produits par l'interprétation. Leurs structures est donc « contrôlée » ; les modèles composés par Kompose (modèles de conception) sont crées par des humains. Le code de la première phase de Kompose est donc spécifique aux modèles d'entrée, ce qui n'est pas notre cas.

3.4 Illustration

Cette section illustre l'exécution de l'étape de fusion des modèles d'entrée fournis par la Figure 43. En particulier, nous nous focalisons sur la fusion des fragments de modèles intermédiaires donnés dans la Figure 62 (il est difficile de présenter des modèles de grande taille car le format A4 n'est pas très adapté …). Les objets des figures qui suivent portent le même identifiant (un numéro) que celui qu'ils portent dans la Figure 62 ; les objets résultant de l'application d'une règle d'équivalences ou ceux créés par une règle de fusion portent un numéro différent de ceux utilisés dans la Figure 62 (à partir du numéro 35).

Le Tableau 10 présente une trace détaillée de l'exécution de ψ_{RM} pour les fragments de la Figure 62. Pour chaque règle de fusion appliquée, le tableau donne les objets identifiés par le motif, l'objet résultant si la règle est une règle d'équivalences, et les objets créés, modifiés ou supprimés[90]. Les

[89] Il est aussi possible d'imbriquer les ensembles (ce qui n'est pas le cas dans la version courante de ψ_{RM}). Par exemple, on pourrait avoir une expression du type {{x, y}, {z, t}*}*.

[90] Dans le cas d'une règle d'interprétation, l'objet résultant est comptabilisé dans les objets créés et les objets identifiés sont comptabilisés dans les objets supprimés.

lignes pointillées symbolisent le début d'une nouvelle itération. La première phase de ψ_{RM} (phase de résolution des équivalences initiales de la Figure 65) consiste en la résolution des équivalences définies par ρ_{RM} capturées par le modèle d'équivalence. La Figure 66 présente un extrait du modèle global une fois la première phase exécutée et la la règle *entityDesignateRole* exécutée. Seuls les fragments impactés par cette phase sont présentés, c'est-à-dire les fragments (g), (d), (b) et (f). Ces fragments proviennent des modèles d'entrée de même nom. Les fragments (g), (d) et (b) sont délimités par des cadres en pointillés ; les objets en dehors de ces cadres appartiennent au fragment (f). Les objets créés ou modifiés sont grisés.

	règle appliquée	objets identifiés	objet résultant	objets		
				modifiés	créés	supprimés
rés.	1- variable	32, 34	36		36	32, 34
	2- variable	6, 7	37		37	6, 7
combinaison	3- entityDesignateRole	14, 1		1, 37	38, 39, 40, 41, 42, 43	
	4- entityDesignateRole	14, 8		8		
	5- entityDesignateRole	14, 26		26, 25		
	6- actorAndConcept	11, 18, 29	44		44	11, 18, 29
	7- concept	17, 32, 35	45		45	17, 32, 35
	8- concept	13, 23	46		46	13, 23
	9- action	4, 24, 27	47		47	4, 24, 27
	10- parameter	25, 28, 37	48		48	25, 28, 37
	11- actorAndConceptVoid	1, 8, 26	49		49	1, 8, 26
	12- oneEntityWithNoName	49		49		
	13- actor	44, 49	50		50	44, 49
	14- attributeAndRelation	2, 10	51	40	51, 52, 53, 54	2, 10
complétion	15- variable	36, 39			55	
	16- deleteExists	38		55		38
	17- multiPreCond	47			56	
	18- postCondVoid	47			57	

Tableau 10 – Trace détaillée de l'exécution de ψ_{RM} pour les fragments de la Figure 62.

Le modèle d'équivalence comporte deux classes d'équivalence dans le contexte des fragments de modèle de la Figure 62 : $\{32, 34\}_{variable}$ et $\{6, 7\}_{variable}$. Leur résolution donne respectivement les objets 36 et 37 de la Figure 66. L'application de la règle de normalisation *entityDesignateRole* est plus complexe. Elle supprime la valeur de la propriété *name* des objets 1, 8 et 26 (ayant pour valeur *moderator*) car cette valeur désigne le rôle de *moderator* (information donnée par l'objet 14). Elle ajoute de plus un lien type entre les objets 37 et 18 afin de préciser qu'un modérateur est un participant (information donnée par le fragment (b)). Enfin, elle modifie la pré-condition de l'action *enter* dans le contexte du fragment (f) uniquement (le fragment (g) désigne l'action *enter* mais ne comporte pas de référence au rôle *moderator*). Les objets 38-43 sont créés dans ce but.

Le Tableau 10 illustre bien le caractère itératif de l'étape de fusion ; les règles de la phase de combinaison sont appliquées durant trois itérations. On remarque par exemple que la règle *attributeAndRelation* n'est appliquée qu'à la troisième itération (exécution 14). La Figure 67 présente trois fragments du modèle global incluant les objets 2 et 10 susceptibles d'être identifiés comme équivalents par le motif de la règle *attributeAndRelation*. Le fragment de gauche est un extrait du modèle global après application des huit premières règles dans l'ordre chronologique donné par le Tableau 10 ; le fragment du milieu correspond au même extrait après application des trois règles suivantes (jusqu'à la onzième) et le dernier fragment après application de la treizième. Seul le fragment de droite permet d'identifier les objets 2 et 10 comme équivalents par la règle *attributeAndRelation* car ces objets sont liés par la relation *properties* à la même entité (l'objet 50).

Cette entité est elle-même le résultat de plusieurs exécutions (en particulier les exécutions 11, 12 et 13 si l'on part du modèle global obtenu après l'exécution 8).

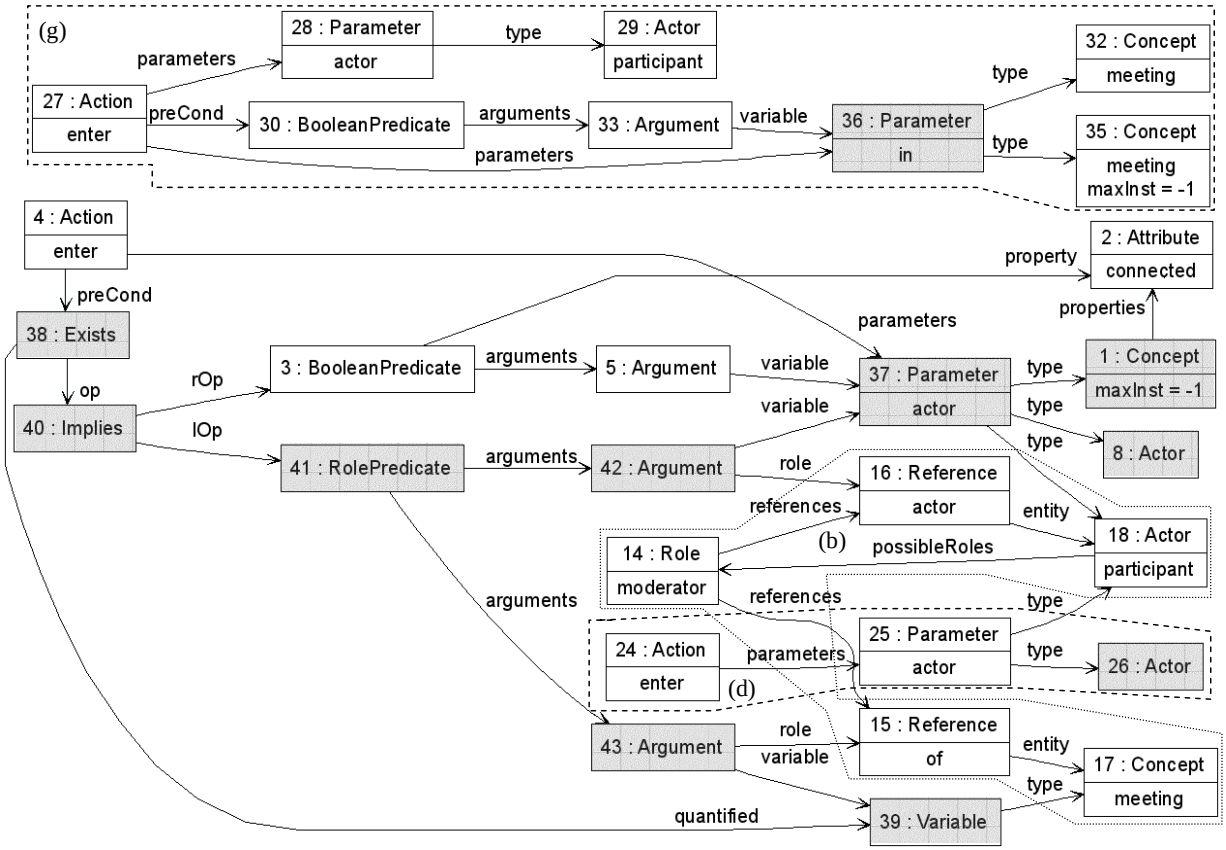


Figure 66 – Extrait du modèle global obtenu après la phase d'initialisation de ψ_{RM} pour les fragments (g), (d), (b) et (f) de la Figure 62.

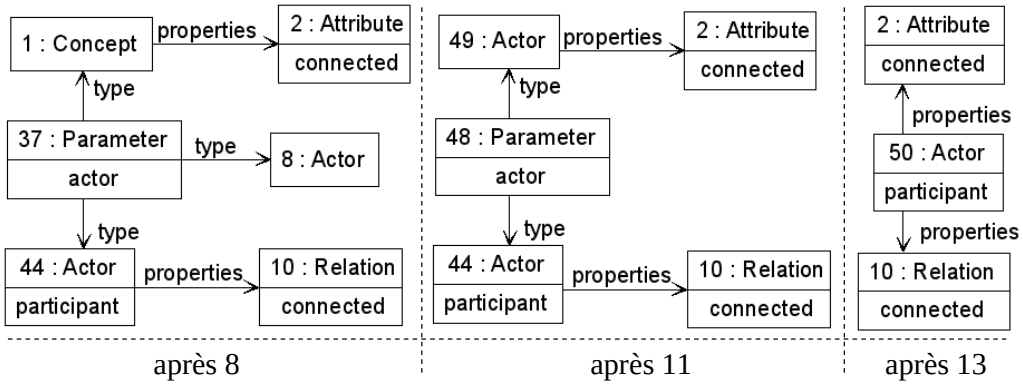


Figure 67 – Trois extraits du modèle global en cours de fusion, montrant des fragments susceptibles d'être identifiés par le motif de la règle attributeAndRelation.

La phase de complétion comporte les règles de fusion nécessitant une vision globale de l'information composée, et en particulier celles visant à résoudre les ambiguïtés compositionnelles. La phase de complétion de ψ_{RM} comprend trois règles de ce type, à savoir *variable*, *multiPreCond* et *multiPostCond*. Ces règles implémentent toute une stratégie de résolution par interaction homme-machine. Une VARIABLE *v* peut désigner le même objet qu'un PARAMETER *p* si *v* et *p* ont le même type (ou si l'un des types est super-type de l'autre) et font partie de la description d'un contrat d'une même action. La règle *variable* vise à résoudre ce type d'équivalence. Sa garde a pour effet une interaction homme-machine posant explicitement la question aux parties prenantes impliquées dans

leur description[91]. Les contracts des actions (pré- et post-conditions) sont décrits par une collection de modèles d'entrée, ce qui implique des ambiguïtés compositionnelles. Les règles *multiPreCond* et *multiPostCond* visent à résoudre ce type d'équivalence. Elles proposent aux parties prenantes les combinaisons logiques possibles des pré- ou post-conditions définies séparément.

Dans notre exemple, la règle variable est exécutée pour les objets 36 et 39 (voir la Figure 66 ; entre temps, les objets 17, 32 et 35 ont été fusionnés par la règle concept). La règle *multiPreCond* est ensuite appliquée car l'Action *enter* comporte deux liens de type *preCond* (depuis l'exécution 9). Cette règle propose aux parties prenantes deux choix possibles pour la pré-condition de l'action enter (la conjonction ou la disjonction entre les pré-conditions décrites par les fragments (f) et (g)). L'application des règles *variable* et *multiPreCond* conduit à quatre interprétations possibles pour la pré-condition de l'action *enter*, suivant les choix des parties prenantes. Le Tableau 11 présente ces quatre interprétations possibles suivant le choix des parties prenantes. Dans notre exemple, les parties prenantes définissent les objets 36 et 39 comme équivalents, ce qui a pour conséquence de fusionner ces deux objets (voir la règle *variable*), et choisissent la conjonction comme lien logique entre les deux fragments de pré-conditions. La pré-condition résultant correspond donc à la première interprétation du Tableau 11.

choix des variations sémantiques		pré-condition de l'action *enter*
variable	multi preCond	
36 ≡ 39	∧	*opened*(m) ∧ (p is moderator(m) => ∃ s : Server, connected(p, s))
36 ≡ 39	∨	*opened*(m) ∨ (p is moderator(m) => ∃ s : Server, connected(p, s))
36 ≡ 39	∧	*opened*(m) ∧ (∃ n : Meeting, n ≠ m ∧ p is moderator(n) => ∃ s : Server, connected(p, s))
36 ≡ 39	∨	*opened*(m) ∨ (∃ n : Meeting, n ≠ m ∧ p is moderator(n) => ∃ s : Server, connected(p, s))

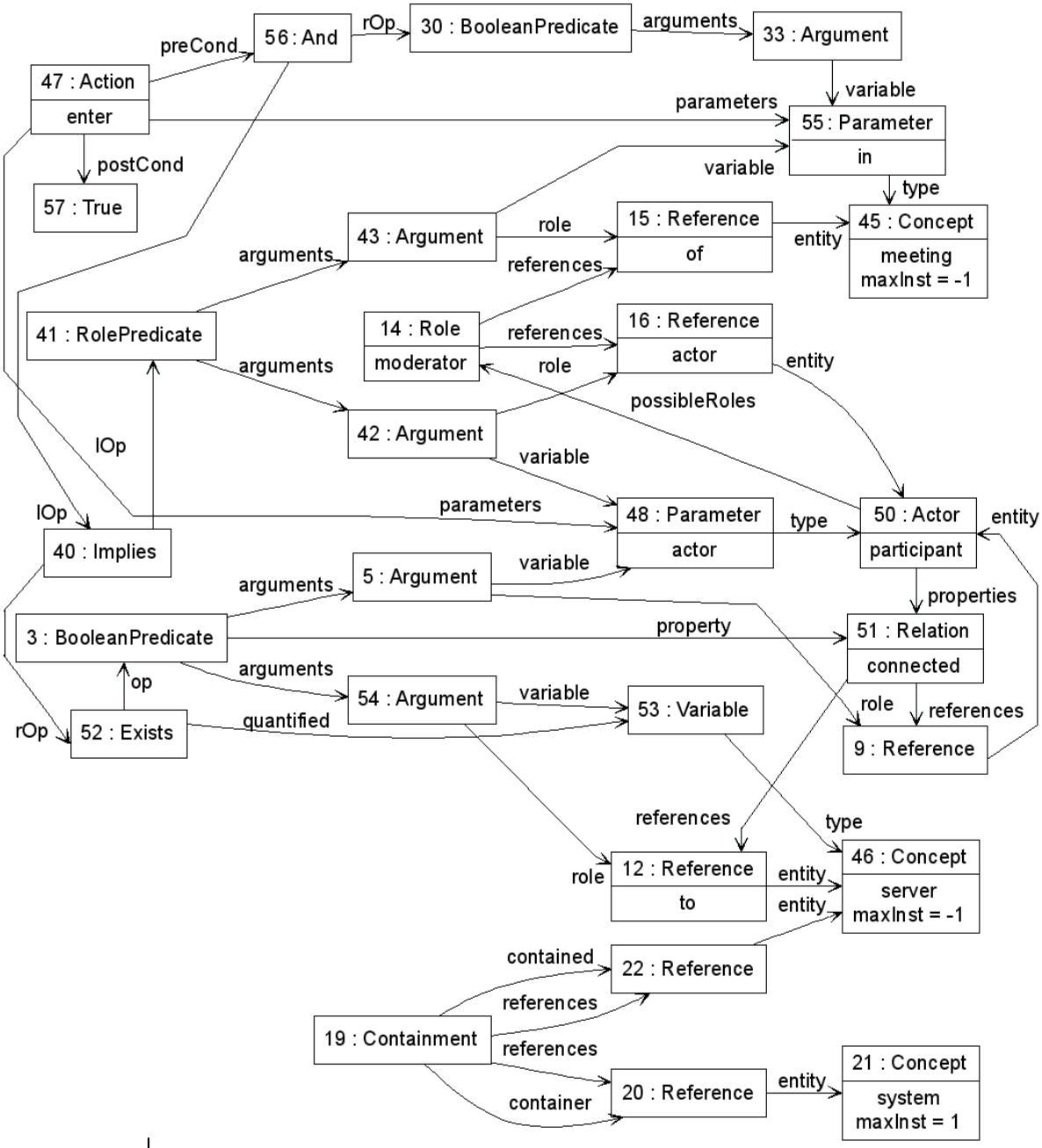
Tableau 11 - Les quatre interprétations possibles de la pré-condition de l'action *enter*, fonctions de deux variations sémantiques compositionnelles.

Une autre technique pourrait être basée sur la recherche de scénarios permettant de disqualifier des interprétations possibles. Par exemple, la variation sémantique ∧ est disqualifiée s'il existe un scénario validé par les parties prenantes tel que lors de l'activation de l'action enter, ce scénario satisfait les formules (2) ou (4) et ne satisfait pas les formules (1) et (3)[92]. Dans ce cas, les interprétations (1) et (3) sont disqualifiées. Il est possible pour chaque formule de définir les conditions suffisantes pour sa satisfaction ou sa non-satisfaction. Par exemple, pour que la formule (1) soit fausse, il suffit qu'un des deux termes de la conjonction soit faux. Par calcul des prédicats, il est possible de statuer qu'il suffit que *opened(m)* soit faux lorsque l'action enter est activée pour disqualifier la variation sémantique ∧. En revanche, cette technique ne fonctionne pas toujours. A titre d'exemple, il est impossible de disqualifier la variation (36 ≡ 39). Nous n'avons pas complètement étudiée cette technique de résolution des ambiguïtés compositionnelles par simulation.

L'application de la règle *variable* implique l'existence d'une instance de la métaclasse EXISTS (objet 38) n'étant pas justifié syntaxiquement puisqu'un paramètre d'une action existe par définition. On a ici un exemple complexe de malformation syntaxique ne reflètant pas une incohérence. La règle *deleteExists* a pour but de remédier à ce type de situation. La dernière règle exécutée est *postCondVoid*. Cette dernière ajoute une instance de la métaclasse TRUE au niveau de la post-condition de l'action enter puisque la cardinalité de la relation *postCond* est de 1 (voir le métamodèle RM). Cette règle vise aussi à supprimer des malformations syntaxiques. La Figure 68 présente le modèle global obtenu après fusion des fragments de modèles donnés dans la Figure 62. Ce modèle est inclus dans le modèle global obtenu par composition des fragments de modèle de la Figure 43.

[91] Ces parties prenantes peuvent être identifiées en recoupant les informations contextuelles capturées par les points de vue et le modèle de traçabilité généré (voir section 4.1).

[92] Si, les formules (1) ou (3) sont aussi satisfaites, le diagnostic est inconclusif.



01 **Action** enter (p : *Participant*, m : *Meeting*)

02 pre *opened*(m) ∧ (p is *Moderator*(m) => ∃ s : *Server*, *connected*(p, s))

Figure 68 - résultat de la fusion des fragments de modèles de la Figure 62.

On peut remarquer que ce modèle n'est pas conforme au métamodèle RM. A titre d'exemple, l'objet 30 n'est pas lié à une PROPERTY (qui devrait être un ATTRIBUTE ayant pour nom *opened*). Cette violation est due ici au fait que ce modèle est un extrait du résultat de la fusion de fragments de modèles intermédiaires. Cependant, dans certains cas, la non-conformité peut révéler des incohérences statiques entre les modèles d'entrée. La détection de ces incohérences est effectuée lors de l'étape d'analyse décrite dans la section suivante.

4 Analyse

Cette section présente la dernière étape du processus de composition. La section 4.1 offre une vue générale de l'étape d'analyse. La section 4.2 présente le mécanisme de traçabilité supporté par les étapes d'interprétation et de fusion, discute de la nature des liens de traçabilité générés et donne une formalisation du modèle de traçabilité produit. Enfin, la section 4.3 présente les règles de détection d'incohérences utilisées durant l'analyse statique et illustre leur utilisation par les parties prenantes.

4.1 Vue générale

L'étape d'analyse guide le processus d'identification des exigences. A chaque itération du processus de composition, elle produit un *diagnostic* identifiant les incohérences détectées au sein du modèle global et localisant précisément les parties des modèles d'entrée incriminées. Ce diagnostic regroupe deux types d'incohérences : les incohérences statiques (manque d'information et contradiction logique structurelle) et les incohérences dynamiques. Les premières sont détectées à l'aide des contraintes structurelles du métamodèle cœur (violations de la conformité) et de règles spécialisées appelées règles *de détection d'incohérences* (IDR). Les deuxièmes sont détectées par utilisation d'outils de vérification formelle tels que IF [90], après projection du modèle global vers les langages d'entrée des outils visés. La détection des premières précède la détection des secondes puisque la cohérence statique du modèle global est un pré-requis à la détection d'incohérences dynamiques [40]. Cependant, il est possible de vérifier la cohérence dynamique d'une partie du modèle global, tant que cette dernière est statiquement cohérente. Dans la suite, nous ne nous intéressons qu'à l'*analyse statique* du modèle global. L'*analyse dynamique* est considérée mais n'est pas traitée en détails.

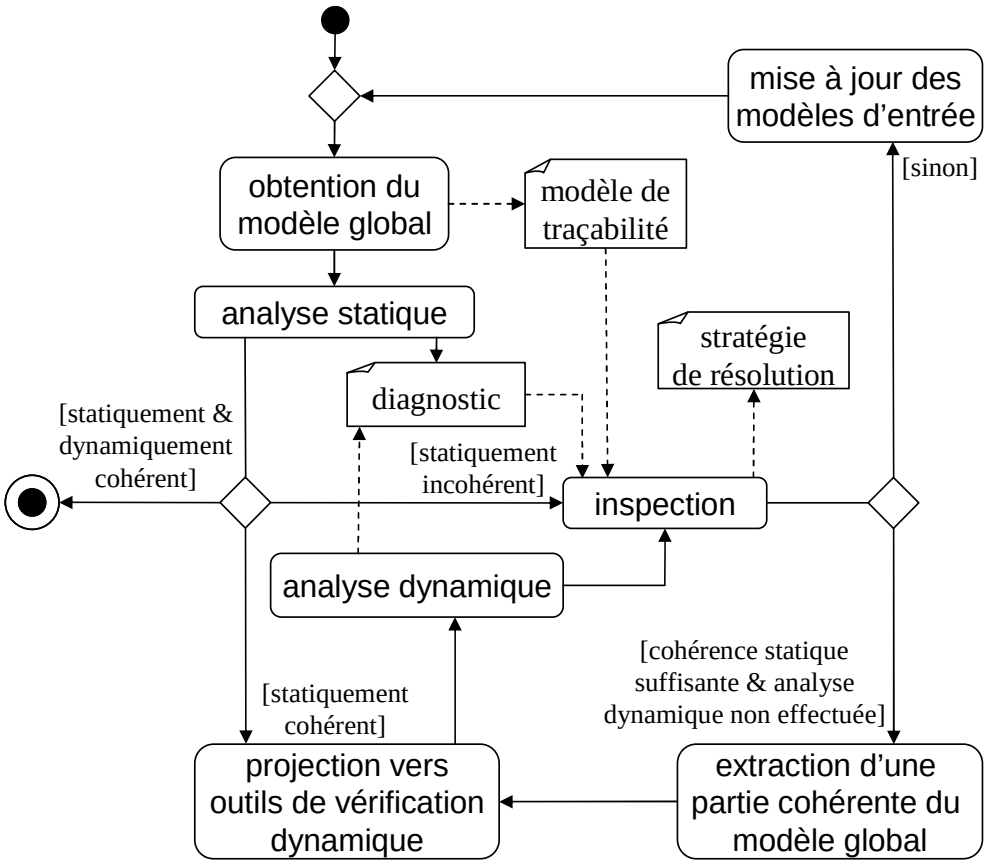


Figure 69 - Vue globale de l'étape d'analyse.

La Figure 69 donne une vue générale de l'étape d'analyse sous la forme d'un diagramme d'activité. L'étape d'analyse débute après obtention du modèle global par interprétation et fusion des modèles d'entrée. La première activité consiste en la détection d'incohérences statiques (analyse statique). Cette activité produit un premier diagnostic regroupant l'ensemble des incohérences portant sur

l'aspect statique de l'information. Si le modèle global est statiquement cohérent, il est transformé dans le ou les langage(s) d'entrée du ou des outil(s) de vérification dynamique et analysé dynamiquement. Dans le cas contraire, le modèle global est inspecté par les parties prenantes (rédacteurs et analystes). L'inspection est effectuée à l'aide du diagnostic et du modèle de traçabilité. L'inspection aboutit à une *stratégie de résolution* produit par les parties prenantes : certaines incohérences sont tolérées, d'autres sont appelées à être résolues en priorité. Une fois une stratégie de résolution définie, les analystes et les rédacteurs peuvent soit mettre à jour les modèles d'entrée (ajout/modification) de façon à résoudre une partie des incohérences statiques, soit décider d'analyser une partie de la cohérence dynamique des exigences. Dans ce cas, les vérifications dynamiques autorisées ne doivent concerner que les parties du modèle global statiquement cohérentes[93]. L'analyse dynamique met à jour le diagnostic ; elle requiert une nouvelle discussion entre parties prenantes (inspection et mise à jour de la stratégie de résolution).

La stratégie de résolution adoptée par les analystes et les rédacteurs influe sur la mise à jour des modèles d'entrée. En effet, le choix des priorités de résolution des incohérences oblige les parties prenantes à se focaliser sur une certaine partie de la spécification. La stratégie de résolution est choisie suivant des impératifs de qualités. A titre d'exemple, les incohérences relatives à une action peuvent être écartées si cette action n'est pas impliquée dans la vérification d'une exigence non-fonctionnelle de première importance. Plus généralement, une stratégie de résolution reflète une ou plusieurs intentions : elle vise la vérification d'exigences prioritaires.

4.2 Support de la traçabilité

Le modèle de traçabilité produit par les étapes d'interprétation et de fusion joue un rôle prépondérant lors de l'analyse du verdict par les parties prenantes. Il permet de localiser précisément les fragments de modèles d'entrée potentiellement responsables des incohérences détectées, autrement dit la(s) source(s) de l'incohérence (et potentiellement du conflit à l'origine). En outre, les critères de classification associés par les points de vue aux fragments de modèles incriminés sont utiles pour le diagnostic (identification des parties prenantes ou des préoccupations concernées).

Le modèle de traçabilité garde aussi une trace des règles exécutées durant l'interprétation et la fusion. Ce modèle peut donc servir à identifier les règles mises en jeu dans la production de chaque élément du modèle global. Cet aspect est important pour la correction des fonctions sémantiques. En effet, la composition de modèles étant une transformation, une incohérence peut dans certains cas refléter une erreur de programmation au niveau de l'interprétation ou de la fusion (le métamodèle cœur est en fait un oracle partiel pour le test de la composition puisque le modèle résultant de cette transformation en est une instance [130, 165]).

Les deux sections suivantes décrivent le mécanisme de traçabilité du processus de composition. La section 4.2.1 présente la nature des liens générés pendant le processus. Elle décrit de plus la manière dont ces liens sont automatiquement générés lors de l'exécution des règles d'interprétation et de fusion. La section 4.2.2 donne une formalisation du modèle de traçabilité et détaille comment les liens de traçabilité sont composés pour associer les objets des modèles d'entrée aux objets du modèle global (et vice versa). Elle fournit de plus une illustration de l'utilisation du mécanisme par les parties prenantes.

4.2.1 Nature des liens de traçabilité générés

Le processus de composition génère automatiquement un modèle de traçabilité[94]. Ce modèle capture l'historique d'exécution de l'ensemble des règles d'interprétation et de fusion appliquées durant la composition. Un modèle de trace est instance de la vue traçabilité du métamodèle RM. Cette vue présentée dans le Chapitre IV est reproduite par la Figure 70 avec une sémantique opérationnelle. Chaque exécution de règle (RULE) est associée aux objets identifiés par le patron de la règle ainsi que

[93] La vue analyse du métamodèle global permet de définir un contexte de vérification ayant pour effet la restriction de la vérification à une partie des informations capturées par le modèle global.

[94] Le développeur n'est pas responsable de cette préoccupation, ce qui limite la complexité accidentelle (tout comme Java supporte la gestion automatique de la mémoire, contrairement à C par exemple).

les objets produits. Cette association est capturée par une structure de donnée appelée *trace unitaire* et représentée par une instance de la métaclasse ATOMICTRACE. Une trace unitaire consiste en un lien vers la règle exécutée (via l'association *executedRule*), un lien vers chaque objet source, c'est-à-dire identifié par le patron de la règle (via l'association *sources*), et un lien vers les objets cibles, c'est-à-dire produits par la règle (via l'association *targets*). L'historique des traces produites durant un processus de composition est représenté par une instance de la métaclasse TRACERECORD, liée à toutes les traces unitaires générées.

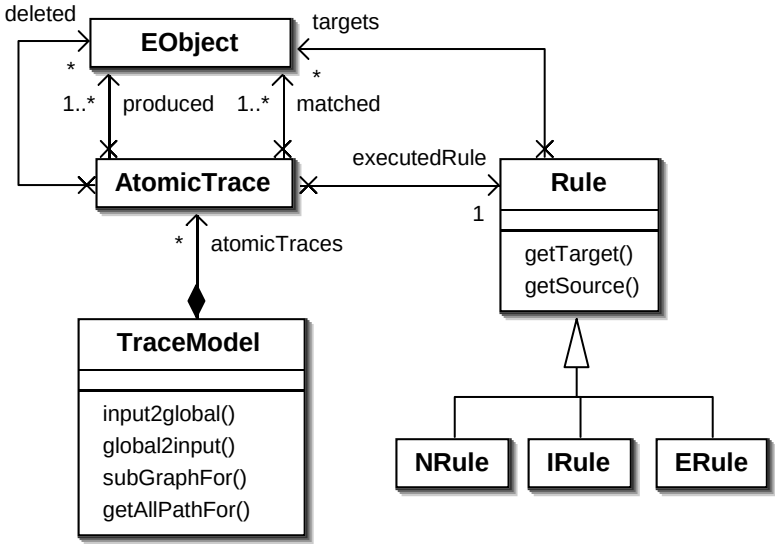


Figure 70 - Reproduction de la vue traçabilité du métamodèle RM.

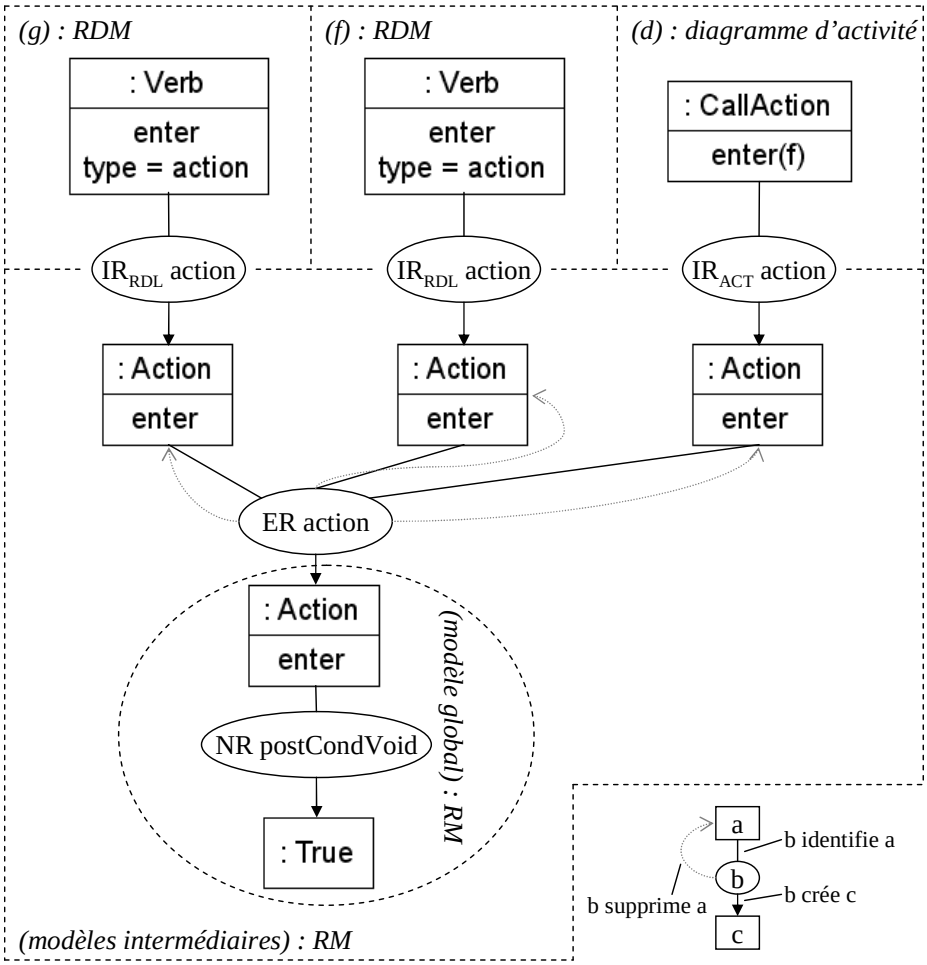


Figure 71 - Extrait de traces unitaires générées pendant la composition des modèles de la Figure 43.

Les objets sources et cibles d'une trace unitaire sont calculés en fonction du type de règle exécutée :

- Règles d'interprétation (IRULE) : les objets sources correspondent aux objets identifiés par le motif et les objets cibles sont les objets crées par la production.
- Règles d'équivalences (ERULE) : les objets sources sont les objets contenus par la classe d'équivalence résolue et objets supprimés ; les objets cibles sont l'objet résultant et les objets créés durant la résolution le cas échéant.
- Règles de normalisation (NRULE) : les objets sources sont ceux identifiés par le motif et les objets supprimés ; les objets cibles ceux produits par la production.

La Figure 71 montre un extrait d'un modèle de traçabilité où les traces unitaires visibles ont été générées par traitement d'informations liées à l'action *enter*. Les objets de cette figure proviennent de quatre modèles différents : deux instances du métmodèle RDM obtenus après analyse grammaticale des modèles d'entrée (f) et (g) (deux des phrases RDL de la Figure 43 retenues pour illustrer la composition), un modèle UML représentant le diagramme d'activité de la Figure 42 (plus précisemment le modèle (d) de la Figure 43) et les modèles intermédiaires produits par interprétation et modifiés par l'étape de fusion. Seule une partie des objets des modèles intermédiaires appartient finalement au modèle global. Plus précisément, les objets du modèle global sont ceux (i) appartenant initialement aux modèles intermédiaires, (ii) créés par une règle de fusion et (ii) n'ayant pas été supprimés par une règle de fusion. La section suivante formalise les liens de traçabilité et les opérations natives *input2global* et *global2input*, proposées par le modèle de traçabilité (TRACEMODEL).

4.2.2 Formalisation et illustration

Les traces unitaires capturent un type particulier d'information, à savoir l'historique de l'exécution des règles. Le modèle de traçabilité est un graphe où les nœuds sont les objets manipulés durant la composition et où les arêtes relient les objets identifiés et produits par l'exécution d'une règle. Plus formellement, un modèle de traçabilité est un graphe valué $\mathcal{G} = (V, E, \sigma)$ où V est l'ensemble des nœuds (les objets sources et cibles des traces unitaires générées), E est l'ensemble des arêtes, et σ est la fonction de valuation. Ce graphe est orienté et acyclique ; il est défini comme suit :

$$\mathcal{G} = \begin{cases} V = \mathrm{ran}(s) \cup \mathrm{ran}(c). \\ E = \{ (a, b) \in V \times V \mid \exists\, t \in T, (a \in s(t) \vee a \in d(t)) \wedge b \in c(t) \}. \\ \sigma : E \rightarrow R = \{ ((a, b), c) \in E \times R \mid \exists\, t \in T, (a \in s(t) \vee a \in d(t)) \wedge b \in c(t) \wedge c = r(t) \}. \end{cases}$$

avec :
- R est l'ensemble des règles d'interprétation et de fusion.
- O est l'ensemble des objets crées durant la composition.
- T est l'ensemble des traces unitaires générés durant le processus de composition.
- $s : T \rightarrow \mathcal{P}(O)$ est la fonction donnant pour chaque trace unitaire ses objets sources.
- $c : T \rightarrow \mathcal{P}(O)$ est la fonction donnant pour chaque trace unitaire ses objets cibles.
- $d : T \rightarrow \mathcal{P}(O)$ est la fonction donnant pour chaque trace unitaire les objets supprimés.
- $r : T \rightarrow R$ est la fonction associant à chaque trace unitaire la règle exécutée.
- ran(X) retourne le co-domaine de la fonction X.

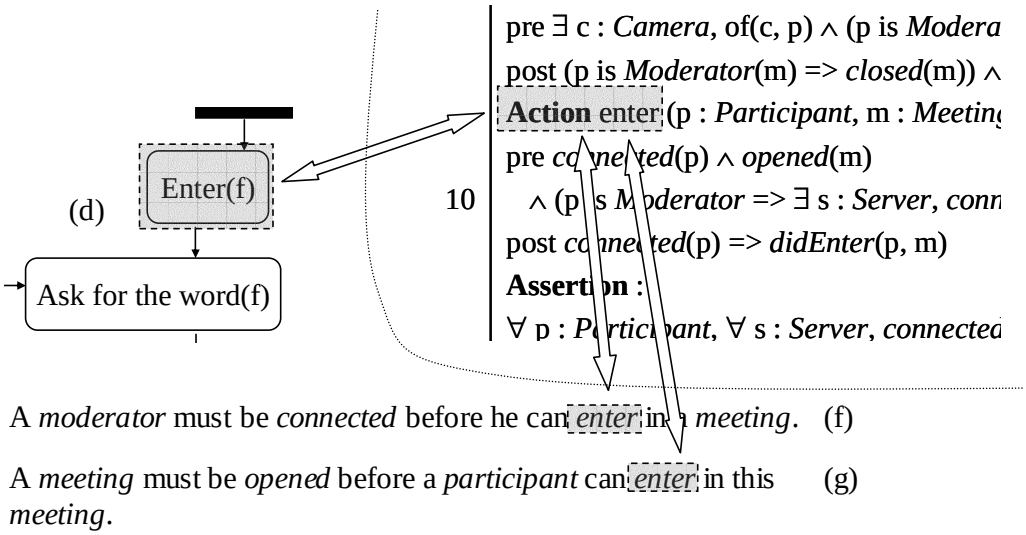
La sémantique opérationnelle de la vue traçabilité offre deux opérations natives décrites plus loin. La première fournit les objets du modèle global capturant l'information représentée par les objets des modèles d'entrée fournis (*input2global*), la deuxième fournit les objets des modèles d'entrée représentant l'information capturée par les objets du modèle global fournis (*global2input*). La première permet la localisation des sources possibles d'une incohérence et la deuxième la localisation des parties du modèle global capturant une information de la spécification. Les deux formules ci-dessous implémentent ces opérations. Le graphe permet de déterminer pour un objet de O quels sont les objets liés sémantiquement (ces objets pouvant appartenir aux modèles d'entrée, aux modèles intermédiaires ou aux modèle global). Ces deux formules sont fonctions des chemins du graphe $\mathcal{G}$.

input2global : $\mathcal{P}(I) \rightarrow \mathcal{P}(G)$
$X \rightarrow \{ y \in G \mid \exists e \in X, e \sim y \}$

global2input : $\mathcal{P}(G) \rightarrow \mathcal{P}(I)$
$X \rightarrow \{ y \in I \mid \exists e \in X, y \sim e \}$

avec : - G est l'ensemble des objets du modèle global.
- I est l'ensemble des objets des modèles d'entrée.
- ~ est la relation tel que o ~ o' ssi il existe un chemin dans $\mathcal{G}$ reliant o à o'.

La Figure 72 illustre l'utilisation du mécanisme de traçabilité. Les parties prenantes peuvent sélectionner un ensemble d'objets des modèles d'entrée ou du modèle global pour connaître les parties des autres modèles désignant la ou les même(s) préoccupation(s). Cependant, les objets identifiés par les traces (le résultat de l'application d'une des deux opérations) ne désignent pas toujours la préoccupation. La qualité du modèle de traçabilité dépend du niveau de granularité des spécifications IRL. On peut remarquer aussi que l'utilisation graphique comme celle illustrée par la Figure 72 nécessite aussi un modèle de traçabilité liant les objets des modèles aux éléments graphiques les représentant. La plate-forme R2A s'appuie pour cela sur les fonctionnalités d'outils comme Syntaks [160] ou TopCased [166].



A *moderator* must be *connected* before he can *enter* in a *meeting*. (f)

A *meeting* must be *opened* before a *participant* can *enter* in this *meeting*. (g)

Figure 72 – Illustration pratique du mécanisme de traçabilité.

4.3 Analyse statique du modèle global

L'analyse statique consiste en la détection de malformations structurelles au sein du modèle global. Nous faisons l'hypothèse qu'une malformation syntaxique reflète une incohérence au sein de la spécification. La phase de normalisation de l'étape de fusion garantit cette hypothèse (dans le cas contraire, le code FRL doit être modifié en conséquence). Certaines malformations structurelles violent des cardinalités des propriétés du métamodèle cœur et peuvent être détectées par vérification de la relation de conformité. D'autres sont complexes et nécessitent des règles spécialisées appelées règles de détection d'incohérences (IDR pour *Inconsistency Detection Rule*).

L'analyse statique est divisée en deux étapes séquentielles. La première vérifie la relation de conformité entre le modèle global et le métamodèle cœur et produit une liste de violations de conformité (voir section 2.4.3.1). La deuxième étape exécute les IDRs sur le modèle global (voir section 4.3.2). Le résultat de ces deux étapes est consigné dans le verdict présenté aux parties prenantes (voir l'illustration de l'étape d'analyse en section 4.4).

4.3.1 Violations de conformité

La première étape de l'analyse statique consiste en la vérification de la relation de conformité entre le modèle global et le métamodèle cœur. Les cardinalités et les contraintes OCL sont vérifiées une à une. Cette étape aboutit à un ensemble de *violations de conformité*. Une violation de conformité peut refléter deux situations :

- Les transformations d'interprétation ou de fusion comportent une ou plusieurs erreurs. Dans ce cas, le métamodèle cœur joue le rôle d'oracle de test [130, 165]. Par exemple, si l'on supprime la règle de normalisation *postCondVoid* de ψ_{RM}, une violation sera détectée durant l'analyse du modèle global de la Figure 68 car l'objet 47 (l'action enter) ne serait pas lié à l'objet 57 (cette instance de TRUE n'aurait pas été créée).
- Une partie de l'information capturée par un ou plusieurs modèles d'entrée est incohérente.

La Figure 73 donne les trois violations de conformité détectées sur le modèle global obtenu après composition des modèles d'entrée de la Figure 43. Ces trois violations sont des violations de la borne supérieure d'une cardinalité (*over-violation*[95]). Par exemple, la violation (1) stipule que l'objet *121*, instance de la métaclasse PARAMETER, comporte deux liens instance de la propriété *type* alors que cette dernière a une cardinalité de [1, 1]. Les éléments soulignés du texte représente des objets et peuvent être « navigués » à l'aide du modèle de traçabilité.

(1) over-violation of property *type* of *Parameter* 121 : 2 vs [1, 1].
(2) over-violations of property *value* of Cardinality 92 : 2 vs [1, 1].
(3) over-violation of property *value* of Cardinality 93 : 2 vs [1, 1].

Figure 73 – Violations de conformité détectées pour le modèle global obtenu après composition des modèles d'entrée de la Figure 43.

4.3.2 Règles de détection d'incohérences

La relation de conformité n'est pas suffisante pour détecter des incohérences. De plus, les règles OCL ne sont pas adaptées pour plusieurs raisons :

- Les règles OCL ne sont pas toujours conclusives lorsqu'elles sont exécutées sur un modèle instance d'une version relâchée du métamodèle cœur. En effet, le code d'une règle OCL dépend de certaines cardinalités du métamodèle cœur. Si ces dernières sont violées par le modèle global, la règle OCL ne peut être exécutée.
- Une règle OCL est définie pour un seul contexte, ce qui rend compliqué la description d'une règle portant sur des objets très éloignés.
- Le langage OCL est hermétique pour la plupart des parties prenantes ; le résultat de l'exécution d'une règle est peu compréhensible (un booléen) et porte sur le modèle global et non sur les modèles d'entrée (ce qui importe le plus pour les parties prenantes).

Nous avons défini le langage IDL pour contourner ces limitations. IDL est un langage à base de règles supporté par la plate-forme R2A pour spécifier des règles de détection d'incohérences. Un code IDL définit un ensemble de règles de détection d'incohérences. Ces règles sont exécutées sur le modèle global après analyse des violations de conformité. La Figure 74 présente quatre IDRs spécifiques au métamodèle RM (d'autres règles sont fournies en Annexe J) et le schéma ci-dessous définit la forme textuelle d'une IDR. Une règle IDR est constituée de trois éléments :

IDR *identifiant* **(** e_1 **:** *Type***,** ...**,** e_n **:** *Type* **) :** *type(s) d'incohérences*
? : *garde*
« *texte paramétré* **»**

motif
description informelle

- Un *motif* : sa sémantique est équivalente à celle des motifs des règles de fusion. Une incohérence est détectée si le motif est satisfait. Une règle OCL peut être considérée comme un motif d'IDR. Cependant, un motif IDR peut comporter plus d'un paramètre (e_1 … e_n dans le

[95] « Under-violation » pour une violation d'une borne inférieure d'une cardinalité et « OCL-violation » pour une violation d'une règle OCL.

schéma) et il peut porter sur un modèle global instance du métamodèle cœur relâché (comme FRL).

- Une *description informelle* : Elle associe à l'incohérence détectée un message compréhensible à l'ensemble des parties prenantes. C'est un texte en langage naturel utile aux non-experts pour comprendre le sens de l'incohérence. Le texte peut comporter des paramètres (expressions évaluables sur les objets identifiés par le motif), de telle sorte que le texte obtenu soit fonction du contexte de l'incohérence détectée. Une fois les paramètres instanciés dans le contexte de l'incohérence détectée, le texte permet d'associer à une incohérence aux objets incriminés au sein du modèle global (et des modèles d'entrée par traçabilité).
- Un ou plusieurs *types d'incohérences* susceptibles d'être détéctés par la règle. Ces types sont qualitatifs. Nous considérons trois types : le manque d'information (*Lack*), la contradiction logique (*Contradiction*) et les collisions structurelles (*SCollision*), terminologiques (*TCollision*) et dénotationnelle (*DCollision*). A cela s'ajoute le type optionnel (Optional) spécifiant que la satisfaction du motif de la règle ne reflète pas toujours l'existence d'une incohérence (cas de l'IDR *collisionBetweenPropertyAndLiteral*).

La première IDR détecte un manque d'information. Elle détecte toute ENUMERATION ne comportant qu'une valeur littérale possible. La résolution de cette incohérence conduit à l'identification de nouvelles exigences (modification des modèles d'entrée). La deuxième détecte toute situation où une valeur littérale et une propriété ont le même nom (si la propriété est égale ou liée par classification à la propriété typée par l'énumération contenant la valeur littérale). Il est en effet possible qu'une partie prenante considère une réunion publique ou privée alors qu'un autre considère le type d'une réunion comme publique ou privée (voir une illustration de ce ce cas en section suivante). Cette règle détecte une possible collision structurelle reflètant une divergence conceptuelle entre parties prenantes. La troisième IDR vérifie si les valeurs littérales de toutes les propriétés énumérées sont référencées par au moins une expression logique. En d'autres termes, un manque d'information est suspecté si aucune expression logique ne fait référence à une valeur littérale. Enfin la quatrième IDR détecte une incohérence si un paramètre a plus d'un type. Elle identifie une contradiction logique.

```
IDR enumWithOneValue (e : Enumeration) : Lack, Processus
? : card(e.literals) == 1;
"the value of the property {e.type~[0].name} can be only {e.literals[0].name}."

IDR collisionBetweenPropertyAndLiteral(l : Literal, a : Property) : Lack, SCollision, Optional
var p : Property init l.literals~.type~[0];
? : l.name == a.name and Algo.linkedBySuperType(p.properties~[0], a.properties~[0]);
"It seems that the property {a.name} designate the same domain element as the literal {l.name} of
the property {p.name}."

IDR noReferenceToLiteral (a : Attribute, l : Literal) : Lack
? : a.type#Enumeration#.literals.contains(l)
            and not a.property~.exists(e : EnumPredicate | e.literal == l);
"The literal {l.name} is never referenced by a logical expression."

IDR severalTypeForParameter (p : Parameter) : Contradiction, TCollision
? : card(r.type) > 1;
"One parameter of the action {p.parameters~.name} has more than one type.
It could be a conceptual disagrement between stakeholders. Perhaps these types {r.entity} are
linked by a super-type relation. In that case, add this information in requirements. It could also be a
terminology clash."
```

Figure 74 - Trois exemples de règles de détection d'incohérences.

On peut remarquer que les motifs des règles *literalWithOneValue* et *severalTypeForParameter* font appel à l'instruction *card* et sont susceptibles d'être satisfaits par un même fragment du modèle global. Deux violations peuvent donc dans certains cas ne reflèter qu'une incohérence. C'est le cas des règles des deux premières IDRs. En outre, l'utilisation de l'instruction *card* au sein d'un motif a une sémantique particulière pour le langeg IDL : les violations de conformité sont ignorées si elles sont identifiées par au moins une IDR. Par exemple, l'instance 121 de PARAMETER est identifée par la relation de conformité (première violation de la Figure 73) mais aussi par l'IDR *severalTypeForParameter*. Dans ce cas, la première violation de la Figure 73 est absente du verdict fourni aux parties prenantes[96].

4.4 Illustration

La Figure 75 présente le diagnostic résultat de l'analyse statique après composition des modèles d'entrée de la Figure 43. Ce diagnostic présente six incohérences détectées. Il ne comporte pas les trois violations de conformité présentées en Figure 73 car ces dernières sont ignorées puisque traitées par les incohérences (6), (4) et (5) respectivement. Chaque incohérence détectée est le résultat de la satisfaction d'un motif d'une IDR par un fragment du modèle global. Le diagnostic fournit aux parties prenantes trois informations pour chaque incohérence détectée : le nom de l'IDR violée, le ou les objets identifiés par le motif (des numéros dans la Figure 75) et le texte instancié dans la description informelle. Les numéros tout comme les parties instanciées de la description informelle sont associés à un objet du modèle global et le ou les objet(s) des modèles d'entrée sémantiquement liés. Ils sont soulignés dans la Figure 75 et peuvent être « navigués » à l'aide du modèle de traçabilité.

Outre le diagnostic, la Figure 75 fournit les modèles d'entrée de la Figure 43 où les éléments identifiés par les règles IDR exécutées sont grisés. Ces éléments ont été localisés à l'aide du modèle de traçabilité. L'utilisation conjointe du diagnostic et du modèle de traçabilité facilite l'inspection des modèles d'entrée. Enfin, notons que les types d'incohérences des IDR peuvent être utiles pour classer les incohérences détectées. A titre d'exemple, il est possible de ne visualiser que les incohérences optionnelles ; dans ce cas, seul les incohérences (1), (2) et (3) seront présentées aux parties prenantes. On peut aussi vouloir traiter dans un premier temps les incohérences pouvant être dues à une collision terminologique ; dans ce cas, seul l'incohérence (6) est affichée.

[96] Les violations ignorées sont visibles pour l'analyste si nécessaire (pour la recherche d'erreurs au niveau des codes IRL et FRL).

STATIC DIAGNOSTIC :

- Conformity violations :

  all catched.

- IDR violations :

  (1) *enumWithOneValue* (112) :

  The value of the property state can be only setOff.

  (2) *collisionBetweenAttributeAndLiteral* (89, 115) :

  It seems that the property private designate the same domain element as the literal private of the property type.

  (3) *noReferenceToLiteral* (101, 110) :

  The literal setOff is never referenced by a logical expression.

  (4) *cardinalityConflict* (92) :

  The cardinality bewteen meeting and moderator has several values : [0,1], [0,-1].

  (5) *cardinalityConflict* (93) :

  The cardinality bewteen system and server has several values : [0, 1], [0, -1].

  (6) *severalTypeForParameter* (55)

  One parameter of the action enter has more than one type. It could be a conceptual disagrement between stakeholders. Perhaps these types (meeting, forum) are linked by a super-type relation. In that case, add this information in requirements. It could also be a terminology clash.

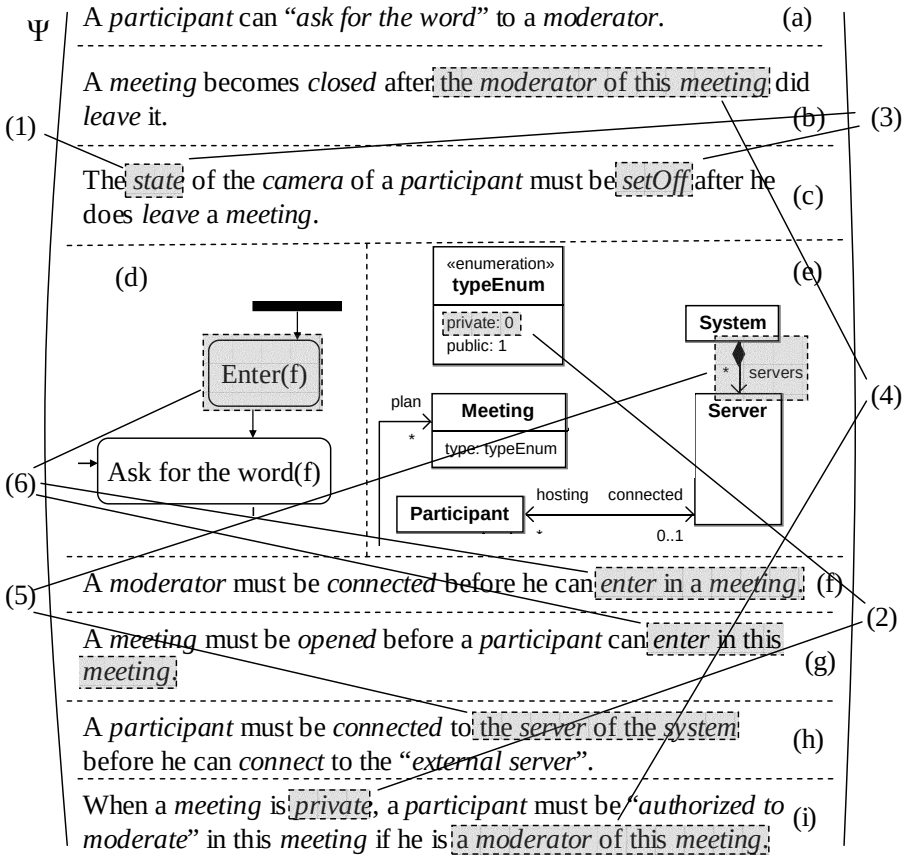


Figure 75 – Exemple de verdict généré pendant l'étape d'analyse statique (résultat pour les modèles de la Figure 43), et visualisation des incohérences au niveau des modèles d'entrée.

Chapitre VII Création de plates-formes spécialisées : processus de composition double

Résumé

Au niveau instance (M1), les connaissances (comme les exigences d'un système) proviennent d'un grand nombre de parties prenantes et sont décrites par une collection de modèles [46, 65, 161][97]. Nous avons proposé en Chapitre VI un processus de composition autorisant une description fragmentée des exigences d'un projet logiciel. Cette fragmentation de l'information est conforme au principe de séparation des préoccupations [18] et de décomposition des problèmes [19]. Au niveau conception (M2), la description de logiciels complexes nécessite aussi d'être proprement découpée, comme prôné par l'approche de développement par aspect (AOSD) [44, 78, 167-169]. La fragmentation de l'information co-existe donc à deux niveaux de modélisation, suivant des méthodes de décomposition adaptées à la nature de l'information considérée.

Dans ce chapitre, nous nous intéressons à un processus de composition au niveau conception, dans le but d'autoriser une séparation des préoccupations considérées par les analystes et développeurs impliqués dans le déploiement de la plate-forme R2A pour un contexte industriel donné. Ce processus de composition, appelé *composition double* pour le différencier du processus de composition simple, participe à l'objectif d'adaptabilité de la plate-forme R2A. Il produit le métamodèle cœur du processus de composition simple (le formalisme de la plate-forme R2A). L'approche proposée est inspirée pour l'essentiel des travaux de Tarr [169] et France [78], portant respectivement sur la décomposition et la composition d'artefacts de conception logicielle. Le métamodèle coeur est décrit par une collection de *composants de conception*, chacun capturant un type d'information (*composant ontologique*) ou tout ou partie d'une fonctionnalité supportée par la plate-forme (*composant fonctionnel*). Le résultat de processus de composition double est un métamodèle monolithique capable de capturer les types d'information et de réaliser les fonctionnalités représentées par les composants. Une fois ce métamodèle obtenu, le processus de composition peut être appliqué et le modèle global produit est outillé pour exécuter ces fonctionnalités (visualisation graphique, simulation, …).

Ce chapitre est organisé comme suit. Le processus de composition double a pour origine les difficultés rencontrées lors du développement de la plate-forme R2A. La section 1 décrit brièvement ces difficultés et motive l'intérêt de la séparation des préoccupations. La section 2 définit la notion de composants de conception et en donne des exemples dans le contexte de la plate-forme R2A. Ces composants embarquent des sémantiques diverses (sémantiques opérationnelles, compositionnelles et de déploiement). La section 3 présente le processus de composition double en tant que tel et l'illustre. Ce processus applique le processus de composition simple à deux niveaux de modélisation. Dans un premier temps, il produit par composition des composants de conception le métamodèle cœur et sa sémantique de composition. Des directives de composition sont utilisées par les ingénieurs pour résoudre manuellement les collisions structurelles entre composants de conception. Cette composition est guidée par une sémantique de composition spécifique au MOF, exprimée avec le langage FRL.

[97] Ces trois références adoptent trois techniques de décomposition de l'information : par cas d'utilisation, par points de vue, ou sans mécanisme de classification imposé (plus général).

1 Motivation

La production d'une plate-forme de vérification et de validation des exigences a un coût significatif. La plate-forme R2A est d'autant plus rentable qu'elle est applicable à un grand nombre de projets logiciel et plus généralement à des contextes industriels différents. L'adaptation de la plate-forme à un contexte donné est susceptible de faire varier trois de ses composantes à savoir : (i) son formalisme (les types d'information pouvant être capturés au sein des spécifications d'exigences), (ii) les langages supportés (aussi bien d'entrée que de sortie) et (iii) ses fonctionnalités. L'adoption d'une approche dirigée par les modèles favorise l'adaptabilité des langages d'entrée et de sortie. Elle favorise aussi l'adaptabilité des fonctionnalités implémentées par une transformation de modèles entre le formalisme et un autre langage (le formalisme d'un model-checker par exemple). Nous nous intéressons ici à l'adaptabilité du formalisme et des fonctionnalités dont l'implémentation est spécifiée par une sémantique opérationnelle embarquée par le formalisme (par exemple, la simulation ou des fonctions IHM).

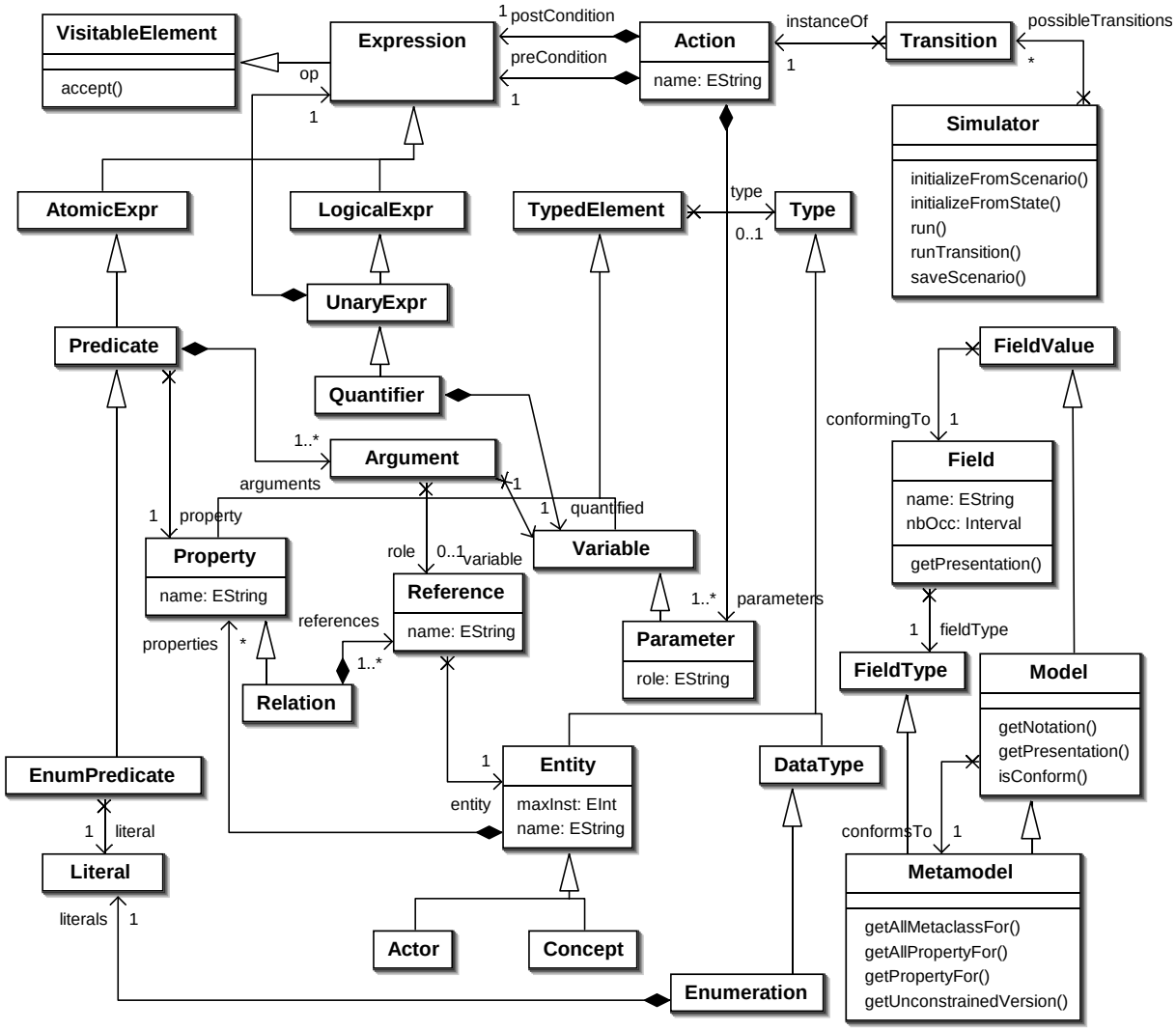


Figure 76 – Extrait du métamodèle RM avec sa sémantique opérationnelle.

Le formalisme actuel de la plate-forme R2A est implémenté par le métamodèle RM. Nous avons présenté ce métamodèle à l'aide d'un ensemble de vues dans le Chapitre IV. Ces vues décrivaient uniquement les types d'informations pouvant être capturés par la plate-forme (les métaclasses propres à l'IHM par exemple, n'étaient pas visibles). De plus, ces vues ne présentaient que l'aspect statique de l'information ; elles faisaient abstraction de la sémantique opérationnelle RM, implémentée par des opérations embarquées par les métaclasses. Cette présentation incomplète et fragmentée avait pour but

de faciliter la compréhension des types d'information pouvant être capturées par le RM. Elle n'est cependant pas réaliste. Le métamodèle RM est en réalité un métamodèle monolithique difficilement compréhensible vu le nombre de concepts repésentés (plus de 80 métaclasses) et le nombre de fonctionnalités implémentées. Il suffit pour s'en rendre compte d'observer la Figure 76. Cette dernière fournit un extrait du métamodèle RM avec une partie de sa sémantique opérationnelle. Elle est représentative du modèle de conception (comprendre la description au niveau M2 de la plate-forme) manipulé par les développeurs de la plate-forme R2A.

Les métaclasses de la Figure 76 comportent des opérations et liens relatifs à des fonctionnalités diverses. Par exemple, les trois opérations de la métaclasse MODEL : les opérations *getNotation* et *getPrésentation* jouent un rôle dans l'IHM (affichage d'un point de vue), l'opération *getNotation* joue aussi un rôle dans l'interprétation des modèles d'entrée, l'opération *isConform* et l'association *conformsTo* sont liées à la description du lien de conformance entre modèle et métamodèle. Certaines métaclasses jouent des rôles différents suivant les préoccupations considérées. A titre d'exemple, la métaclasse METAMODEL est un sous-type de FIELDVALUE si l'on se préoccupe du classement de l'information (voir la vue classification RM) ; elle est aussi un sous-type de MODEL d'un point de vue métamodélisation (voir la vue métamodélisation RM). L'entrelacement de toutes les préoccupations rend la tâche des développeurs compliquée car il est difficile dans ce contexte d'identifier et de se focaliser sur une seule préoccupation.

2 Cadre conceptuel pour l'adaptabilité de la plate-forme R2A

Pour limiter cette complexité, nous avons adopté une approche de séparation des préoccupations inspirée du travail de Tarr [169], basée sur une décomposition d'un modèle de conception en fonction d'au moins deux dimensions [98] (types de préoccupations) : les fonctionnalités [99] et les types d'informations. Cette approche est similaire aux approches de développement par aspects portant sur les modèles de conception (niveau M2) [44, 78, 167-169]. Une telle décomposition facilite la compréhension, l'évolution et la réutilisation de solutions de conception éprouvées. En ce sens, elle favorise l'adaptabilité de la plate-forme.

De ce point de vue, le modèle de conception de la plate-forme est décrit par un ensemble de fragments de modèles de conception. Nous appelons ces fragments des *composants de conception*. La notion de composant revêt différentes définitions dans la littérature. Dans notre cas, un composant regroupe (encapsule) un ou plusieurs métamodèles généralement munis d'une ou plusieurs sémantiques. Les composants de conception peuvent décrire des préoccupations à des niveaux d'abstraction différents. Nous distinguons deux types de composants, suivant ce qu'ils représentent. Un composant peut représenter une fonctionnalité ou un type d'informations. Dans le premier cas, on parlera de *composant fonctionnel*, dans l'autre de *composant ontologique*. Le modèle de conception de la plate-forme R2A est dans ce contexte une composition de composants de conception.

Définition – *Composant de conception* : Un composant de conception est une unité d'encapsulation d'une préoccupation au niveau conception. Un composant de conception est la donnée d'un ou plusieurs métamodèles. Ces métamodèles peuvent être plus ou moins génériques suivant le niveau d'abstraction de l'information représentée. Ils peuvent embarquer des sémantiques diverses, exprimées en fonction des métaclasses de ces métamodèles uniquement.

N.B. : Dans la suite, le terme de « composant » désigne la notion de composant de conception.

Définition – *Composant fonctionnel* : Un composant fonctionnel représente une fonctionnalité. Un composant fonctionnel est muni d'une sémantique opérationnelle réalisant cette fonctionnalité et d'une sémantique.

Définition – *Composant ontologique* : Un composant ontologique représente une connaissance (par exemple le classement de l'information par points de vue, les informations sur les fonctionnalités attendues d'un système ou encore la description de concepts métiers). C'est donc un aspect dans

[98] Une dimension peut être vue comme un type de préoccupations.
[99] « Features » dans la littérature anglophone.

son acceptation la plus large, celle utilisée en ingénierie des exigences (type d'informations). Un composant ontologique est muni d'une sémantique de composition.

Nous avons identifié trois types de sémantiques différentes : *sémantique opérationnelle*, *sémantique compositionnelle* et *sémantique de déploiement*. Une sémantique opérationnelle décrit la sémantique statique du composant ; elle est décrite en Kermeta dans la plate-forme R2A. Une sémantique compositionnelle décrit comment composer des instances du composant. Elle est exprimée en FRL dans la plate-forme R2A. Enfin, une sémantique de déploiement décrit un contrat que toute instance du composant doit respecter. Cette sémantique est décrite en OCL dans la plate-forme R2A, ou avec des règles de normalisation FRL (dans ce cas, le contrat a un effet de bord).

Définition – *Sémantique de déploiement* : Une sémantique de déploiement est associée à un composant fonctionnel. C'est un contrat que toute instance du composant doit satisfaire, de sorte que la fonctionalité puisse correctement fonctionner.

Définition – *Sémantique compositionnelle* : Une sémantique compositionnelle est associée à un composant ontologique. Elle identifie et résout les superpositions sémantiques entre instances de ce composant.

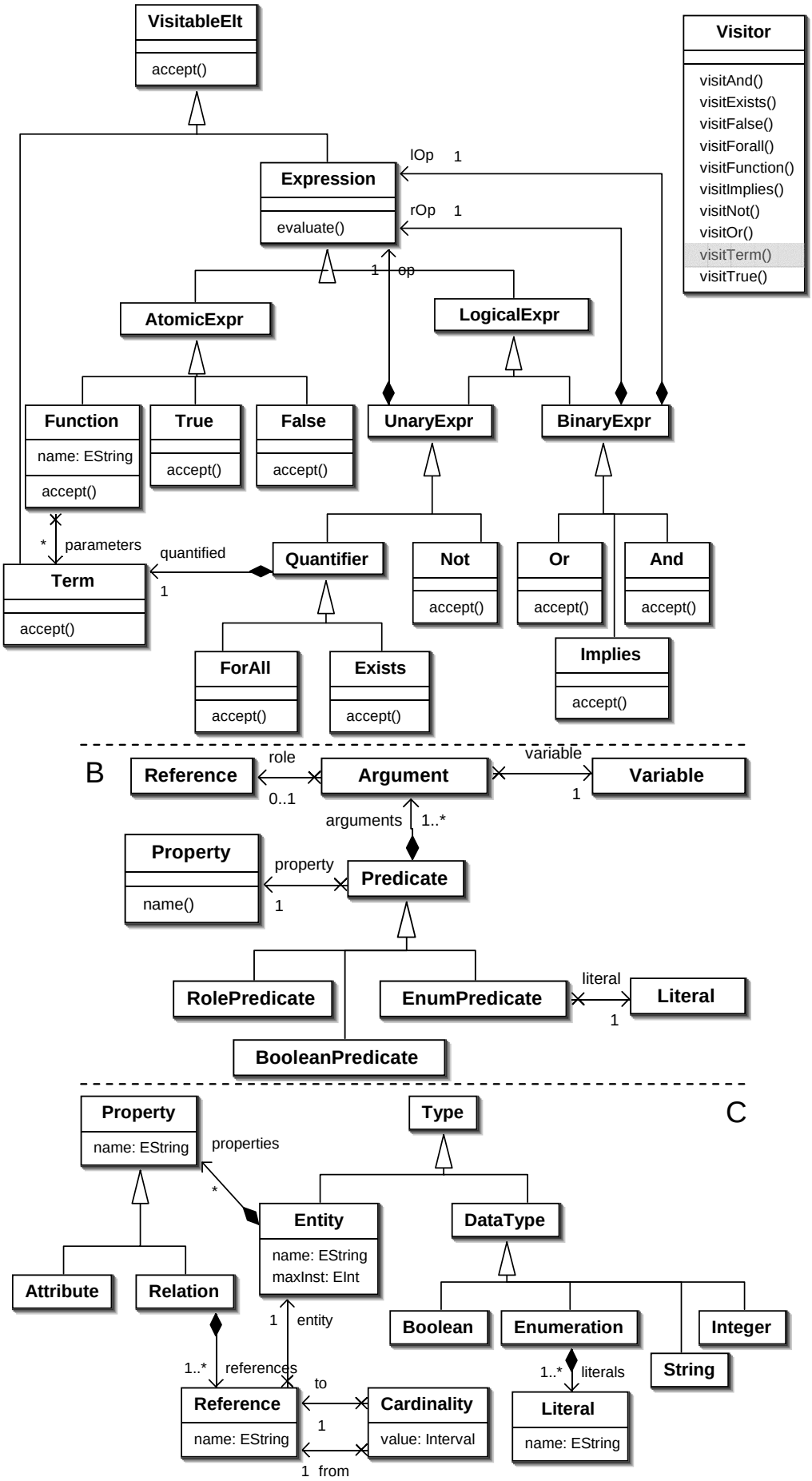
Définition – *Sémantique opérationnelle* : Une sémantique opérationnelle est associée à un composant fonctionnel ou un composant ontologique. Elle regroupe un ensemble d'actions définies au sein des métaclasses des fragments de métamodèles du composant. Ces actions implémentent tout ou partie d'une ou plusieurs fonctionnalités.

La Figure 77 présente un ensemble de composants utilisés lors de la conception de la plate-forme R2A. Ils servent dans la suite pour illustrer le processus de composition double. Ces composants décrivent diverses préoccupations lors du développement de la plate-forme R2A. Le composant D est fonctionnel, les autres sont ontologiques. Les éléments grisés sont déclarés comme abstraits, c'est-à-dire que leur implémentation n'est pas de la responsabilité du composant.

Le composant A représente la logique du premier ordre. Il est muni d'une sémantique opérationnelle, réalisée par une instance du patron de conception « visiteur » (métaclasses VISITOR et VISITABLEELT). Cette sémantique implémente l'évaluation d'une expression booléenne, instance de ce patron. Cette sémantique n'est pas complète : la méthode *visitTerm* est abstraite. Le composant B est spécifique à la plate-forme R2A. Il décrit un raffinement du composant A afin que ce dernier puisse être proprement déployé dans le cadre de la plate-forme. Les métaclasses Predicate et Variable de B joue le rôle des métaclasses Function et Term de A (voir plus loin). Les métaclasses ROLEPREDICATE, ENUMPREDICATE et BOOLEANPREDICATE sont spécifiques au formalisme RM. Le composant C représente une représentation objet d'une connaissance (diagramme Entité-Relation). C'est un composant essentiel à tout langage orienté objet (MOF, UML, KAOS, RML …), tout comme le composant A. On peut remarquer qu'il représente les entiers et les chaînes de caractères (INTEGER, STRING), inutiles pour le RM. Il est muni d'une sémantique compositionnelle.

Le composant D représente la simulation d'un système de transitions à un haut niveau d'abstraction. Ce composant comporte donc plusieurs éléments abstraits. Par exemple, les notions de transition et d'état ne sont pas décrites plus en détails ; l'implémentation des opérations *isRunnable* et *run* de TRANSITION est à la charge d'autres composants, ou d'une implémentation manuelle, après composition des composants. Le composant E représente la notion d'action. Il peut servir à décrire des contrats (au sens de Meyer [24])ou encore une description symbolique d'un système de transition (c'est le cas pour le RM). Il est muni d'une sémantique compositionnelle. Le composant F est un extrait du métamodèle du dépositaire de modèles EMF-AR (voir l'architecture de la plate-forme en Annexe D). Il représente les notions propres à la métamodélisation. Il embarque une sémantique opérationnelle utile pour la manipulation, la création et la sauvegarde de modèles dans l'environnement de développement Eclipse [170]. Le composant G représente la notion de spécification partielle. Il est muni d'une sémantique opérationnelle implémentant l'affichage des modèles d'entrée au sein d'une notation (rôle des opérations *getPresentation*[100] et de l'opération *promptInView*). C'est un composant d'IHM spécifique à la plate-forme R2A.

[100] Nous avons utilisé le patron de conception IHM PAC-Amodeus.


VisitableElt
accept()
Visitor
visitAnd()
visitExists()
visitFalse()
visitForall()
visitFunction()
visitImplies()
visitNot()
visitOr()
visitTerm()
visitTrue()
Expression
evaluate()
lOp 1
rOp 1
1 op
AtomicExpr
LogicalExpr
Function
name: EString
accept()
True
accept()
False
accept()
UnaryExpr
BinaryExpr
* parameters
quantified
Term
accept()
1
Quantifier
Not
accept()
Or
accept()
And
accept()
ForAll
accept()
Exists
accept()
Implies
accept()
B
Reference
role
0..1
Argument
variable
1
Variable
arguments 1..*
Property
name()
property
1
Predicate
RolePredicate
EnumPredicate
literal
1
Literal
BooleanPredicate
C
Property
name: EString
properties
*
Type
Entity
name: EString
maxInst: EInt
DataType
Attribute
Relation
1 entity
Boolean
Enumeration
Integer
String
1..* references
1..* literals
to
Reference
name: EString
1
Cardinality
value: Interval
Literal
name: EString
1 from

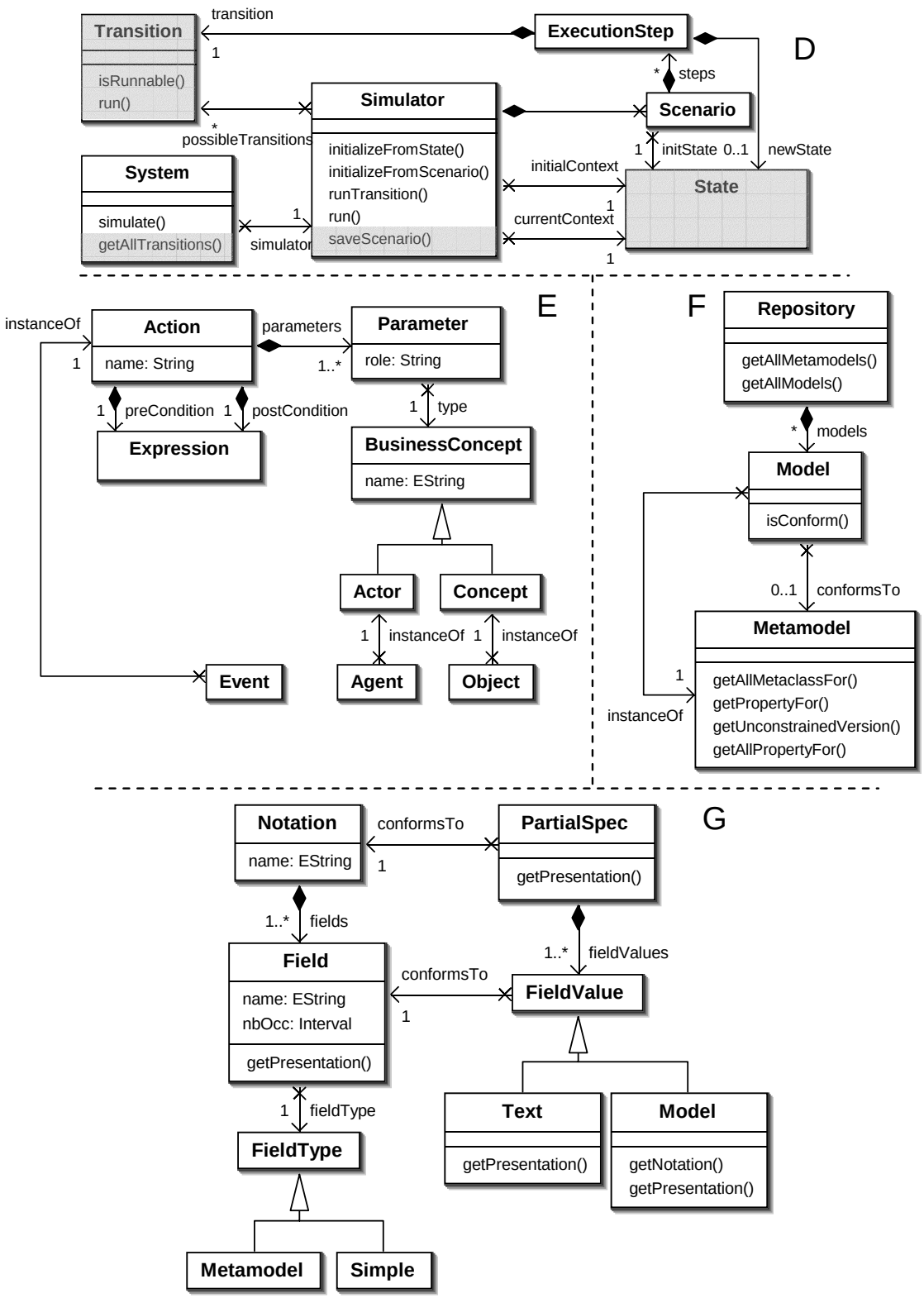


Figure 77 – Sept composants de conception pour la plate-forme R2A.

Bien que décrits séparemment, les composants de la Figure 77 sont sémantiquement liés. Ces liens doivent être décrits pour pouvoir produire la plate-forme et composer les exigences au niveau instance. La section suivante décrit le processus de composition double, défini dans ce but.

3 Processus de composition double

Cette section présente dans son ensemble le processus de composition double. La section 3.1 donne une vue générale du processus. Un des paramètres du processus est une sémantique compositionnelle définie sur le MOF. La section 3.2 décrit brièvement la version de cette sémantique que nous avons implémentée dans la plate-forme R2A. La section 3.3 présente la notion de directive de comosition et présente brièvement celles actuellement supportées. Elle décrit aussi le lien entre l'application de ces directives et les sémantiques embarquées par les composants de conception. Enfin, la section 3.4 porte sur l'analyse du métamodèle résultat du processus de composition double (détection d'incohérences au niveau M2).

3.1 Vue générale du processus

La Figure 78 donne une vue générale du processus de composition double dans le cadre de la plate-forme R2A. Ce processus travaille à deux niveaux de modélisation. Au niveau méta (niveau conception), le métamodèle coeur de la plate-forme est décrit par une collection de composants de conception. Ces composants spécifient les types d'information et les fonctionnalités de la plate-forme. Ces types d'informations et fonctionnalités dépendent d'un contexte industriel donné [101]. Les composants fonctionnels comportent une sémantique opérationnelle. Certains composants ontologiques embarquent une sémantique compositionnelle (ψ_1, …, ψ_n dans la figure).

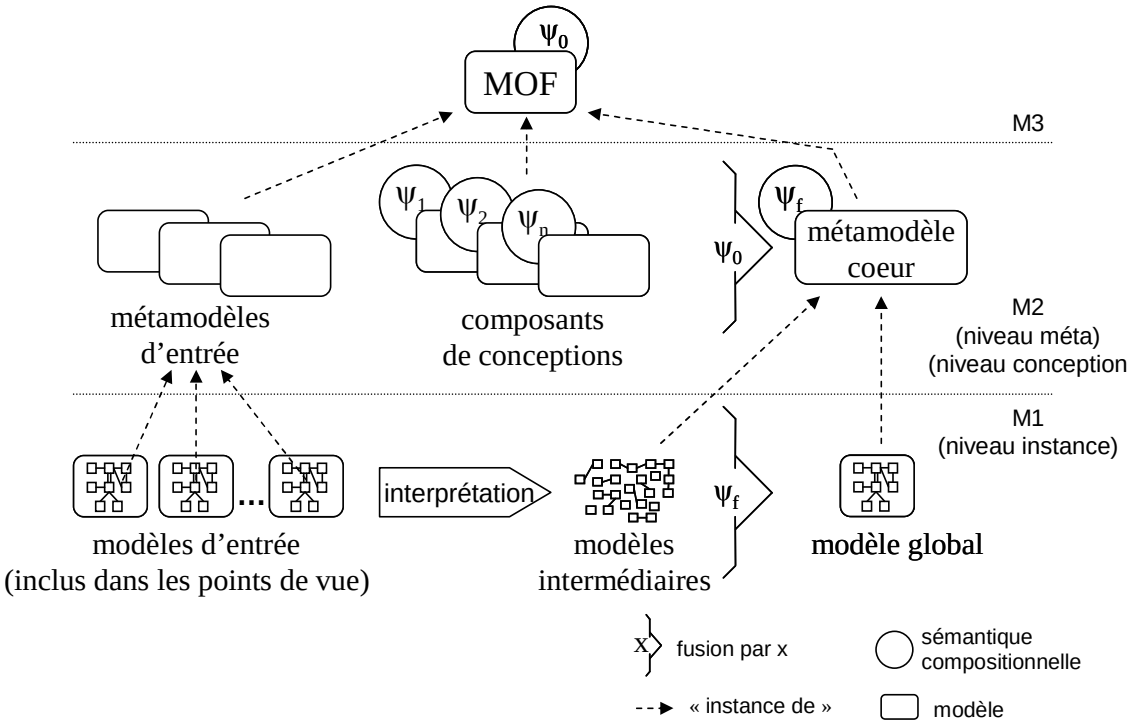


Figure 78 - Processus de composition double.

Nous appliquons le processus de composition simple au niveau méta pour composer les composants. L'étape d'interprétation n'est pas utilisée à ce niveau puisque les modèles à composer et le modèle résultant sont tous instances du MOF. L'étape de fusion est exécutée avec une sémantique compositionnelle ψ_0 définie sur le MOF. Le résultat de cette composition au niveau méta est le métamodèle cœur et sa sémantique compositionnelle ψ_f. Cette dernière est executée au niveau instance pour produire le modèle global des exigences (cf. Chapitre VI).

[101] Le métamodèle cœur peut être vu comme une instance d'une ligne de produit.

Les règles de fusion initialement embarquée dans un composant porte sur les éléments des métamodèles de ce dernier. Les règles des sémantiques compositionnelles ψ_1, …, ψ_n sont modifiées lorsque les composants sont composés, puisque la composition au niveau méta combine les éléments des métamodèles de ces composants. En outre, ces éléments deviennent des éléments du métamodèle cœur après composition. Les règles de fusion de ψ_1, …, ψ_n deviennent donc des règles de fusion de ψ_f. En d'autres termes, la composition des sémantiques compositionnelles des composants est une conséquence de la composition des composants.

Au niveau instance, les modèles inclus dans les points de vue sont instances des langages d'entrée. La sémantique de ces langages est décrite par des fonctions sémantiques, produisant des fragments de modèles regroupés dans les modèles intermédiaires. Au niveau méta, ces fonctions sémantiques sont aussi impactées par la composition de composants (ce sont des spécifications IRL dans le contexte de la plate-forme R2A). Lorsque les composants ont été composés, les fonctions sémantiques des langages d'entrée produisent des modèles intermédiaires instances du métamodèles cœur. La sémantique compositionnelle ψ_f est alors exécutée sur ces fragments pour obtenir le modèle global.

Pour résumé, le processus de composition à deux niveaux de modélisation consiste en deux étapes séquentielles, à savoir :

- Au niveau méta, l'application de l'étape de fusion du processus de composition simple sur les métamodèles des composants de conception. Dans ce cas, la sémantique compositionnelle exécutée est ψ_0, définie au niveau M3. Le métamodèle cœur et sa sémantique compositionnelle ψ_f sont produits. Cette dernière regroupe des versions modifiées des règles de fusion de ψ_1, …, ψ_n. Les fonctions sémantiques et les autres sémantiques sont aussi modifiées. Cette étape est effectuée à chaque mise à jour des composants de conception.
- Au niveau instance, l'application du processus de composition simple sur les modèles d'entrée. L'étape d'interprétation produit des modèles intermédiaires instances du métamodèle cœur. L'étape de fusion est exécutée et le modèle global est produit. Dans ce cas, la sémanique compositionnelle exécutée est ψ_f.

Les sections suivantes portent sur la première étape du processus de composition double. Nous présentons la version actuelle de ψ_{MOF}, ainsi qu'un ensemble de directives de composition. Ces dernières sont instanciées manuellement par les ingénieurs processus pour résoudre les collisions entre composants. Chacune encapsule une solution à un type de collision (modification de la structure des composants et modification des sémantiques en conséquence).

3.2 Exemple de sémantique de composition définie sur le MOF

Cette section présente brièvement la sémantique compositionnelle ψ_{MOF} actuellement embarquée dans la plate-forme R2A (cette sémantique est décrite sur Ecore dans la plate-forme, voir Annexe M). La Figure 79 présente deux règles de fusions de cette sémantique. Une version complète de cette sémantique est fournie en Annexe M. La règle *ER metaClass* déclare que deux métaclasses (ECLASS) sont équivalentes si leurs noms et paquetages sont équivalents. La fusion de deux métaclasses identifiées comme équivalentes est effectuée par la politique de résolution par défaut.

La règle *ER relation* est plus complexe. Son constructeur et ses directives de résolution font appel à une fonction fcs (pour *First Common Super-type*). Cette fonction accepte en entrée un ensemble de métaclasses et retourne le plus grand super-type commun à ces métaclasses (elle retourne void sinon). Cette règle spécifie donc que deux relations sont équivalentes si elles ont le même nom et si leurs métaclasses sources et cibles (association *eContainingClass* et *eType*) appartiennent respectivement à un même arbre d'héritage. La source et la cible de la relation résultat de la fusion sont définies par deux directives de résolution. La première spécifie que la source de la relation résultante correspond au plus grand super-type des métaclasses sources des relations équivalentes. La deuxième spécifie que la cible de la relation résultante correspond au plus grand super-type des métaclasses cibles des relations équivalentes.

```
ER metaClass (c1, c2 : EClass) : EClass {
  ?: c1.name == c2.name and c1.ePackage == c2.ePackage;
   …
}

ER relation (r1, r2 : EReference) : EReference {
  ?: r1.name  == r2.name and fcs(union(r1.eContainingClass, r2.eContainingClass)) ≠ void
                          and fcs(union(r1.eType, r2.eType)) ≠ void;
  eContainingClass := fcs(equRanges.collect(r | r.eContaingClass));
  eType   := fcs(equRanges.collect(r | r.eType));
   …
}
```

Figure 79 - Deux règles de fusions appartenant à une sémantique compositionnelle définie sur Ecore.

La Figure 80 illustre l'application de cette sémantique compositionnelle. Elle présente le résultat de la fusion par ψ_{MOF} de deux fragments de métamodèles *a* et *b*. Les relations *r1* définies dans les fragments *a* et *b* sont identifiées comme équivalentes par la règle *ER relation* car elles vérifient les trois conditions du constructeur de classes d'équivalence. Elles partagent d'abord le même nom. D'autre part, leurs sources (les métaclasses B et A des fragments *a* et *b*) font parties du même arbre d'héritage puisque B hérite de la métaclasse A de *a* identifiée équivalente à A de *b* par la règle *ER metaClass*. Enfin, leurs cibles (les métaclasses C et E) font aussi partie d'un même arbre d'héritage puisqu'elles héritent toutes deux des métaclasses D de *a* et *b* identifiés équivalentes par *ER metaClass*.

Après application de ψ_{MOF} sur les fragments *a* et *b*, on obtient donc une nouvelle relation *r1* dont la métaclasse A est la source (car fcs({A, B} = A)) et la métaclasse D est la cible (car fcs({C, E}) = D). On peut aussi remarquer que les relations *r2* des fragments *a* et *b* n'ont pas été fusionnées. En effet, leurs sources (les métaclasses A et F) ne font pas partie d'un même arbre d'héritage.

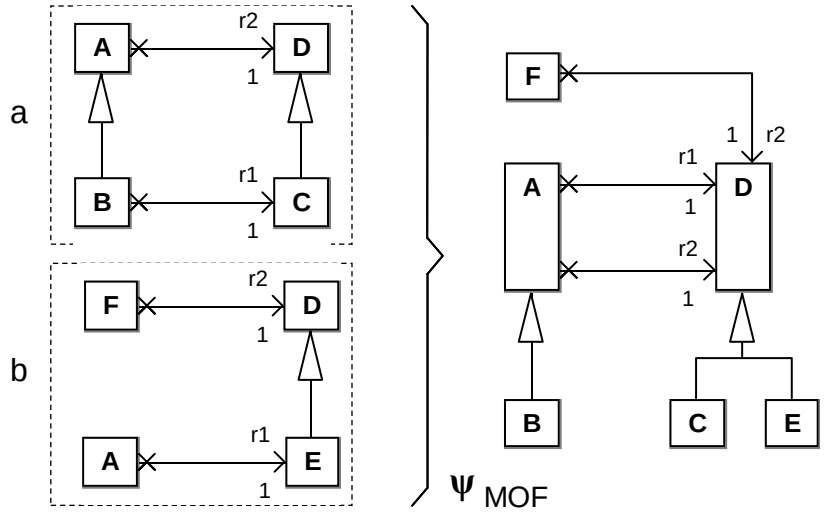


Figure 80 - Exemple simple de fusion de deux fragments de métamodèles avec une sémantique compositionnelle définie sur le MOF.

3.3 Directives de composition

La sémantique compositionnelle ψ_{MOF} est exécutée pour produire le métamodèle cœur. Par exemple, la règle *ER metaClass* présentée plus haut identifie et résout une superposition sémantique entre les métaclasses EREFERENCE des fragments B et C. C'est un cas évident d'équivalence entre concepts. Les attributs *name* des métaclasses PROPERTY des fragments B et C se superposent aussi. Lors de l'exécution de ψ_{MOF}, les règles *ER metaClass* et *ER attribute* vont identifier et résoudre automatiquement cette superposition.

L'étape de fusion proposée dans le Chapitre VI est basée sur des comparaisons syntaxiques des fragments de modèles à composer. Elle est donc très sensible aux problèmes de collision. Les collisions sont fortement probables car les fragments de métamodèles sont spécifiés par différents ingénieurs et un même concept peut être représenté par des métaclasses nommées différemment suivant le rôle qu'il joue. A titre d'exemple, les deux métaclasses ENTITY et BUSINESSCONCEPT des fragments C et E ne peuvent être composées automatiquement par ψ_{MOF}. C'est un exemple de collision dénotationnelle. Pour donner cette information nous proposons un ensemble de directives appelées *directives de composition*. Une directive agit sur un ensemble donné d'éléments des composants à composer. Elle permet de préciser manuellement une superposition (ou non superposition) entre ces éléments. Elle modifie ces éléments de sorte que ψ_{MOF} compose correctement les composants. Le Tableau 1 donne une liste des directives de composition que nous avons considérées utiles.

Définition – *Directive de composition* : Une directive de composition est une transformation de modèles modifiant la structure d'un ensemble de composants de conception afin de résoudre une collision structurelle entre ces derniers. Une directive de composition peut avoir un effet de bord sur les sémantiques embarquées par les composants de conception manipulés afin de préserver le comportement de ces derniers après composition.

a **equivalentTo** *b (*a, b : EClass)	Déclare que la métaclasse *a* est une autre représentation du concept représenté par la métaclasse *b*. Le nom et le paquetage de *a* sont remplacés par ceux de *b*.
a **equivalentTo** *b (*a, b : EAttribute)	Déclare que les attributs *a* et *b* sont équivalents. Le nom de *a* est remplacé par le nom de *b*. Si a et b n'ont pas été combinés après fusion, recherche d'un chemin d'association entre les métaclasses de *a* et de *b* tel que les cardinalités de ces associations soient inclus dans [0..1], puis déplacement de *b* dans la métaclasse de a et modification des sémantiques en conséquence (tout code naviguant *b*)
a **equivalentTo** *b (*a, b : EReference)	Déclare les associations *a* et *b* équivalentes. Le nom de *a* est remplacé par le nom de *b*. Si *a* et *b* n'ont pas été combinées après fusion, les métaclasses sources et/ou cibles de *a* et *b* n'ont pas de super-type commun. Crée ce super-type (demande d'un nom de métaclasse à l'utilisateur) et relance la fusion.
a **differentFrom** *b (*a, b : EClass)	*a* et *b* ont le même nom et font partie du même paquetage. Demande à l'utilisateur un nouveau nom pour *b* et modifie ce dernier.
a **differentFrom** *b (*a, b : EAttribute)	*a* et *b* ont le même nom. Demande à l'utilisateur un nouveau nom pour *b* et modifie ce dernier.
a **differentFrom** *b (*a, b : EReference)	*a* et *b* ont le même nom. Demande à l'utilisateur un nouveau nom pour *b* et modifie ce dernier.
a **overrides** *b* (a, b : EOperation)	Déclare que l'opération *b* est remplacée par l'opération *a*. Si la méthode b a un corps, le supprime. Le nom de *b* est remplacé par le nom de *a*.
a **subClassOf** *b* (a, b : EClass)	Déclare que la métaclasse *a* est un sous-type de la métaclasse *b*. La relation de type est crée.
a **reifies** *b (a : EClass, b : EReference)*	Déclare qu'une métaclasse *a* représente une référence *b*. Modifie en conséquence tout code naviguant *b*.

Tableau 12 – Liste des directives de composition.

La Figure 81 donne les directives de composition utilisées pour composer les composants présentés en section 2. La première directive statue par exemple que le concept représenté par la

métaclasse BUSINESSCONCEPT du composant E est équivalent à celui représenté par la métaclasse ENTITY du fragment C. L'exécution de la troisième directive demande à l'utilisateur le nom d'une métaclasse super-type des métaclasses PARAMETER et PROPERTY (pour l'exemple, l'utilisateur donne « TypedElement »). Le métamodèle de la Figure 76 est un extrait du résultat de la composition des composants de la Figure 77.

C.Entity **equivalentTo** E.BusinessConcept
F.Model.conformsTo **differentOf** G.FieldValue.conformsTo
C.Property.type **equivalentTo** E.Parameter.type
A.Function **requivalentTo** B.Predicate
A.Function.name **equivalentTo** B.Predicate.name
E.Parameter **subClassOf** B.Variable
E.Event **equivalentTo** D.Transition
A.Term **equivalentTo** B.Variable
B.Argument **reifies** A.Function.parameters

Figure 81 - Directives de composition utilisées pour composer les composants de la Figure 77.

3.4 Cohérence du métamodèle cœur et de sa sémantique de fusion

Le résultat de la composition double au niveau méta est une instance du MOF et des langages décrivant les sémantiques liées aux composants de conception. La sémantique compositionnelle ψ_{MOF} et les directives de composition actuelle ne produisent pas de métamodèles cœurs structurellement incohérents (violations des contraintes structurelles du MOF). Cependant, nous avons identifé trois règles de cohérences de niveau processus :

- (R1) Les métaclasses feuilles d'arbres d'héritages du métamodèle cœur doivent être concrètes.
- (R2) Toutes les opérations définies sur les métaclasses du métamodèle cœur doivent avoir une implémentation concrète.
- (R3) Il n'existe pas de conflits entre les règles de fusions définies sur le métamodèle cœur.

Les deux premières règles sont triviales et concerne l'aspect statique du métamodèle cœur produit. La troisième concerne la sémantique de composition. Il est possible d'obtenir par composition de composants des conflits entre règles de fusion. A titre d'exemple, si les composants E et C comportent tout les deux des règles de fusion pour les métaclasses BusinessConcept et Entity, il risque d'y avoir un conflit entre ces deux règles à terme puisque ces métaclasses sont combinées par le processus de composition double. Autrement dit, il est impossible de déterminer automatiquement quelle règle choisir. Ces conflits sont détectables sur la base du motif des règles. Ils doivent être traités manuellement (choix d'une règle ou création d'une nouvelle).

Conclusion et perspectives

France Télécom (FT) cherche à mieux maîtriser la qualité des logiciels produits au sein du groupe (logiciels internes issus de la R&D, services télécoms développés pour ses clients). Pour ce faire, FT s'intéresse à la mise en place d'un processus d'ingénierie des exigences modernes. Ce choix est d'abord pragmatique : (i) une majorité des causes d'échec des projets logiciels sont identifiables au niveau des exigences et (ii) une erreur est d'autant plus coûteuse qu'elle est introduite tôt dans le processus de développement. Il est aussi spécifique à la stratégie du groupe : l'externalisation des activités de développement chez des prestataires renforce le rôle des exigences en tant que document contractuel mais aussi en tant qu'artefact de développement logiciel productif pour la mesure de la qualité du logiciel fourni (la qualité de la prestation aussi). Cette thèse propose un environnement de fiabilisation des exigences favorisant l'application des techniques de vérification et de validation issues de l'ingénierie des exigences. Cet environnement s'inscrit dans une approche de développement dirigée par les modèles, c'est-à-dire visant l'automatisation progressive des processus de développement logiciel. Cet environnement est réalisé de manière concrète par la plate-forme R2A, spécialisée pour les besoins de France Télécom (FT). L'environnement est aussi adaptable à des contextes industriels variés ; le développement de la plate-forme R2A à l'aide de technologie IDM favorise la capitalisation des technologies issues de l'ingénierie des exigences.

Conclusion

La nature d'une spécification d'exigences est symptomatique de la nature d'une spécification logicielle. C'est une collection de spécifications partielles, hétérogènes et potentiellement incohérentes. L'obtention d'une vue globale des exigences est cruciale pour son inspection et son analyse. C'est un problème fondamental pour l'ingénierie des exigences. Nous avons proposé dans ce manuscrit un processus de composition, coeur d'une plate-forme de développement dirigée par les modèles et centrée sur les exigences. Ce processus accepte des paramètres à deux niveaux de modélisation. Au niveau instance (processus de composition simple), il accepte en entrée un ensemble de spécifications partielles, hétérogènes et potentiellement incohérentes. Ces spécifications s'intègrent au sein de notations diverses et sont classées par points de vue, ce qui facilite la gestion de spécifications d'exigences de grande taille. Le processus produit en sortie un modèle global des exigences et un modèle de traçabilité. Au niveau méta (processus de composition double), le processus accepte un ensemble de langages d'entrée et un ensemble de composants de conception, décrivant le formalisme de la plate-forme. Le résultat de l'application du processus au niveau méta est le métamodèle coeur de la plate-forme.

Au niveau instance, ce processus de composition a été construit afin de répondre à un ensemble de problématiques de l'ingénierie logicielle responsables selon nous du manque de transfert industriel des technologies de l'ingénierie des exigences :

- Le processus de composition est multi-langages et les langages d'entrée supportés pour décrire les exigences sont adaptables, ce qui n'est pas le cas des approches de composition existantes. Evidemment, l'ajout d'un langage ou sa modification suppose une modification du processus mais cette dernière est localisée au niveau des métamodèles d'entrée (spécifiés avec MOF et OCL) et au niveau de leur fonction sémantique (spécifiées avec IRL et FRL). Le langage IRL est à base de règles, ce qui permet de décrire une fonction sémantique comme un ensemble de transformations élémentaires entre deux types d'informations sémantiquement équivalents. La description d'une fonction sémantique est réalisée par deux spécifications distinctes, l'une spécifiant un homéomorphisme entre deux langages (code IRL), l'autre spécifiant une sémantique de composition au niveau du formalisme de description des exigences. Outre la séparation des préoccupations de traduction et de composition, ce choix conceptuel permet de ne décrire qu'une seule fois la sémantique de composition du formalisme, ce qui limite la charge de travail lors de l'ajout d'un langage d'entrée (réutilisation).

- Ce processus de composition supporte en entrée des langages de description des exigences syntaxiquement plus accessibles que ceux supportés par les approches de composition existantes. Ce choix favorise l'implication des parties prenantes dans la rédaction des exigences, mais nécessite de maîtriser les problèmes d'ambiguïté. Nous avons montré par l'exemple comment traiter ces problèmes, en particulier celui des variations sémantiques liées à une décomposition de l'information non conforme aux mécanismes de composition supportés par un langage. Nous avons proposé un langage d'entrée (le RDL) et un formalisme (le RM) adaptés au contexte de France Télécom. Le premier est un langage naturel contraint permettant de décrire un grand nombre de propriétés fonctionnelles et non-fonctionnelles d'un domaine (exigences et assertions environnementales). C'est un langage à base de patrons de spécification, moins expressif qu'un formalisme, mais très accessible pour les parties prenantes. Le formalisme proposé résulte d'une synthèse de différentes préoccupations considérées dans les travaux en IE s'intéressant à une description formelle des exigences : préoccupations de niveau produit (types d'informations pouvant être capturées), préoccupations de niveau instance (description du comportement des logiciels satisfaisant la spécification d'exigences) et préoccupations de niveau processus (classement de l'information, gestion des activités d'analyse, paramétrage du processus de composition).
- Le processus de composition favorise l'application de techniques de vérification formelle par définition puisqu'il résout les superpositions sémantiques entre spécifications partielles (liens sémantiques horizontaux). La plupart de ces techniques acceptent en effet un modèle global des exigences statiquement cohérent. Ce processus accepte en entrée des spécifications partielles potentiellement incohérentes, ce qui n'est pas le cas des approches de composition existantes. En outre, ce processus constitue une approche de détection d'incohérences statiques supérieure aux approches de comparaison structurelle car il autorise la détection de manque d'informations. Il transforme le problème de la cohérence statique entre modèles en un problème de cohérence intra-modèle. De ce fait, les règles de cohérence peuvent être définies sur un unique langage (un formalisme), ce qui rend leurs spécifications indépendantes des langages d'entrée supportés. Cette description unique limite drastiquement le nombre des règles devant être gérées et favorise leur capitalisation pour des contextes industriels différents.
- Le processus de composition produit automatiquement un modèle de traçabilité décrivant précisément les liens sémantiques entre les éléments du modèle global et les éléments des spécifications composées. De cette manière, une incohérence détectée au niveau du modèle global peut être analysée au niveau des spécifications partielles incriminées. Cette faculté est utile pour identifier les causes des incohérences détectées (conflits entre parties prenantes) et favoriser l'identification de nouvelles exigences.

Au niveau méta, le processus de composition double favorise la maîtrise de la complexité engendrée par la séparation des préoccupations et la décomposition des problèmes au niveau de l'expression des exigences et au niveau de la construction d'un atelier d'ingénierie des exigences. Le formalisme de la plate-forme définit le type d'informations nécessaires à l'application des techniques d'analyse retenues par les analystes au sein d'un contexte industriel donné (il définit un filtre sur l'information à extraire des spécifications partielles). Ce formalisme devient vite complexe vu le nombre de préoccupations considérées lors d'un processus d'ingénierie des exigences. Pour faciliter la maîtrise de cette complexité, le processus autorise la description du formalisme à l'aide de composants de conception. Ces composants encapsulent soit un type d'information, soit une fonctionnalité. Le modèle global obtenu par composition simple est instance du formalisme. En plus d'être sémantiquement équivalent aux informations extraites au sein des spécifications partielles, le modèle global est donc outillé pour l'exécution des fonctionnalités représentées par les composants de conception. Nous avons aussi souligné le problème de la composition des sémantiques embarquées au sein des composants de conception et le problème des collisions structurelles entre ces derniers.

Enfin, le processus est défini dans le cadre conceptuel traditionnellement adopté en IDM et est entièrement construit suivant une approche IDM. Ce cadre conceptuel nous a permis de définir un processus de composition générique (les langages d'entrée et le formalisme coeur sont des paramètres). Concernant l'ingénierie des exigences, ce processus favorise l'adaptabilité du processus d'ingénierie des exigences et offre une vue globale des spécifications d'exigences tout au long de leur

identification et raffinement. En outre, c'est un élément essentiel à l'élaboration d'un processus de développement impliquant les exigences en tant qu'artefacts productifs. Il favorise l'application des approches type MDA dès les premières phases en produisant une spécification globale d'exigences opérationnelles.

Perspectives

Application du processus à grande échelle. Des études de plus grande envergure sont nécessaires pour comparer le temps perdu à produire une spécification opérationnelle des exigences, le temps gagné à détecter des incohérences au niveau exigence, et le gain concernant la qualité du logiciel produit. En outre, la spécification de fonction sémantique est une tâche coûteuse mais capitalisable. La capacité des langages de transformation à faciliter la maintenance de ces spécifications est un point important. L'étude de Göknil [74] va dans ce sens.

Aide à la détection de superpositions sémantiques entre composants de conception. L'utilisation d'heuristique probabiliste pour l'aide à la détection de liens sémantiques entre des fragments de métamodèles (composition niveau M2 dans ce manuscrit) est une piste intéressante. L'utilisation de réseaux bayésiens pourrait être une option possible, avec un modèle de causalité basé sur les distances entre concepts et les liens sémantiques déjà identifiés. Johnson [171] propose une approche de ce type pour mesurer la qualité de modèles d'architecture d'entreprise.

Composition du MOF avec l'ensemble des langages de description des sémantiques des composants de conception. Les relations entre les langages utilisés pour implémenter la plate-forme (MOF, OCL, Kermeta, IRL, FRL, IDL et autres si nécessaires) doivent être spécifiées de sorte que la spécification d'une directive de composition ne comporte pas du code spécifique à un langage en particulier. Ces relations doivent être spécifiées pour qu'une directive puisse impacter l'ensemble des composants de conception lors de son application. En effet, la modification de la structure d'un composant de conception (un métamodèle) est susceptible de modifier toutes les expressions des langages manipulant cette structure. La description de ces relations entre MOF et chacun des langages utilisés paraît a priori suffisante car les directives de compositions portent uniquement sur la structure des composants de conception. Dans ce cas, ces relations sont déjà spécifiées par construction pour Kermeta et OCL (ce sont des extensions de MOF).

Mise en perspectives des travaux sur l'alignement des ontologies et les travaux en ingénierie dirigée par les modèles. Il pourrait être intéressant d'étudier les travaux portant sur l'alignement d'ontologie en ingénierie des exigences (voir par exemple la thèse de Anne Etien [172]) et celui de Pamela Zave [173] portant sur la composition de formalisme.

Analyse logique statique de la spécification opérationnelle des exigences, instance du formalisme. Concernant la vérification, le processus de composition simple ne permet pas de détecter des incohérences logiques au niveau des pré- et post-conditions des actions. Par exemple, il n'est pas possible d'identifier comme incohérente une précondition de la forme $e_1 \wedge e_2$ où $e_1 = \neg e_2$. En outre, une analyse logique pourrait aussi être utile pour la vérification de propriétés non-fonctionnelles par un outil de type model-checker. En effet, la vérification d'une propriété de ce type nécessite l'observation d'un sous-ensemble des états du système. Être en mesure de ne simuler que les actions susceptibles de modifier ces états est un moyen de limiter l'explosion combinatoire. Concernant la validation, une analyse statique des dépendances entre actions pourrait fournir des heuristiques utiles pour la génération de cas de test système (validation) basée sur la couverture des pré- et post-conditions des actions (voir les critères de test proposés par Clémentine Nebut [55]). L'activation d'une action dépend de l'activation d'un sous-ensemble d'autres actions. Autrement dit, l'activation de certaines actions est inutile si l'on cherche à atteindre l'activation d'une action particulière.

Mise en place d'une stratégie de résolution des ambiguïtés par simulation. Nous avons déjà évoqué cette possibilité dans le Chapitre VI. Cette approche pourrait être appliquée pour interpréter un modèle d'entrée instance d'un langage comportant des variations sémantiques. Un même scénario pourrait disqualifier un grand nombre d'interprétations. En outre, le recoupement des informations fournies par une partie prenante face à un scénario (scénario possible, activation d'une action impossible dans la séquence) paraît être une piste intéressante.

Amélioration de la plate-forme R2A. La plate-forme est encore un prototype. Son utilisation au niveau industriel suppose encore un effort important de développement, d'autant plus que certaines parties ont été implémentées de manière ad-hoc car les technologies IDM ne sont pas toutes matures.

Troisième partie : Annexes

Préambule

Cette partie rassemble l'ensemble des annexes de ce manuscrit. Le lecteur y trouvera un index, des informations complémentaires pour comprendre le contexte scientifique de cette thèse, une terminologie pour l'ingénierie des exigences, des spécifications de certains composants de la plate-forme R2A, ainsi qu'une version plus complète de l'exemple proposé dans cette thèse pour illustrer le processus de composition.

Organisation

L'Annexe A regroupe les notions clés employées dans ce manuscrit. Ces notions sont listées ci-dessous. Chacune est associée à la ou les page(s) la décrivant et/ou comportant sa définition.

L'Annexe B et l'Annexe C fournissent des informations complémentaires sur l'ingénierie des exigences. La première porte nature d'un processus d'ingénierie des exigences. La deuxième donne un cadre conceptuel pour l'ingénierie des exigences. Il fixe une terminologie.

Les trois annexes suivantes portent sur la plate-forme R2A. L'Annexe D décrit l'architecture générale de la plate-forme R2A et présente brièvement l'environnement technologique dans lequel la plate-forme a été développée. L'Annexe E présente les vues du métamodèle RM n'ayant pas été décrites dans le Chapitre IV. L'Annexe F donne des informations complémentaires sur le langage RDL (extraits de la grammaire, métamodèle RDM et exemple d'instance RDM).

Les deux annexes suivantes donnent les grammaires des deux langages proposés pour décrire une sémantique compositionnelle. L'Annexe G donne la spécification EBNF de la grammaire du langage IRL ; l'Annexe H celle de la grammaire du langage FRL.

Les quatre annexes suivantes complètes les exemples proposés pour illustrer les processus de composition simple et double dans les Chapitre VI et Chapitre VII. L'Annexe I fournit un extrait de la spécification IRL de la fonction sémantique f : RDL $\rightarrow$ RM. L'Annexe J donne la spécification IDL d'un ensemble de règles de détection d'incohérences pour le métamodèle RM. L'Annexe K fournit la spécification FRL du métamodèle RM. L'Annexe L présente l'ensemble des modèles intermédiaires produits durant la composition des modèles d'entrée de la Figure 43. Enfin, l'Annexe M donne la sémantique compositionnelle du MOF utilisée pour produire le métamodèle RM. Cette sémantique est spécifiée sur Ecore, une implémentation de EMOF (version simplifiée de MOF).

Annexe A Index.

Agent (d'un système) : page 162.
Ambiguïté : pages 36, 92.
Analyse du domaine (étape d') : page 162.
Architecture MDA : page 40.
Aspect : page 22.
Assertion environnementale : page 162.
Association (métamodèle) : page 38.
Attribut (métamodèle) : page 38.
Besoin : pages 19, 163.
Cahier des charges : page 19.
Champ d'application : page 25.
Classement de l'information : page 21.
Collision : page 29.
Composant de conception : page 141.
Composant fonctionnel : page 141.
Composant ontologique : page 141.
Concept métier : page 67.
Décomposition des problèmes : page 21.
Directive de composition : page 148.
Domaine (d'un langage) : page 35.
Domaine (d'un système) : page 164.
DSL : pages 38, 166.
Entité (d'un système) : pages 67, 162.
Environnement (d'un système) : page 162.
Exigence : page 163.
Formalisme : pages 36, 45.
Fragment de modèle : page 37.
FSA : page 23.
Homéomorphisme : page 42.
Identification des objectifs (étape d') : page 159.
Ingénierie des exigences (IE) : page 18.
Ingénierie dirigée par les modèles (IDM) : page 18.
Langage de modélisation : page 35.
Langage d'entrée (processus de composition) : page 45.
Langage naturel contraint : page 24.

Lien (modèle) : page 37.
Mécanisme d'abstraction : page 21.
Métaclasse : page 38.
Méta-métamodèle : page 40.
Métamodèle : page 37.
Métamodélisation : page 36.
Modèle : page 37.
Modèle de traçabilité : pages 168, 20.
MOF : pages 38, 40.
Motif (d'un patron de spécifications) : page 78.
Motif (d'une règle de transformation) : pages 42, 106.
Niveau instance (M0) : page 40.
Niveau processus (M2) : page 40.
Niveau produit (M1) : page 40.
Objet : page 37.
OCL : page 39.
Parties prenantes : page 18.
Patron de spécifications : page 24.
Phénomène (d'un système) : page 162.
Point de vue : pages 22, 170, 86.
Préoccupation : page 21.
Processus d'ingénierie des exigences (Processus IE) : pages 19, 158.
Production (d'une règle de transformation) : pages 42, 108.
Production d'une spécification d'exigences (étape de) : page 159.
Propriété (d'une métaclasse) : page 38.
Propriété indicative (d'un système) : page 162.
Propriété optative (d'un système) : page 163.
Propriété QoS (d'un système) : page 163.
Relation (métamodèle) : page 38.
Relation de conformité : page 39.
Relation d'instanciation : page 39.
Sémantique (d'un langage) : page 35.
Sémantique compositionnelle : pages 120, 142.
Sémantique de déploiement : page 142.
Sémantique opérationnelle : pages 43, 142.
Séparation des préoccupations : page 21.
Spécification amont : page 160.

Spécification aval : page 160.
Spécification des besoins : page 18.
Spécification des exigences : page 18.
Spécification opérationnelle des exigences : page 18.
Spécification partielle : page 19.
Superposition sémantique : pages 29, 46, 97.
Syntaxe : page 35.
Transformation de modèles : page 41.
Type d'informations : page 37.
Validation (activité de) : page 159.
Vérification (activité de) : pages 159, 28.

Annexe B Processus d'ingénierie des exigences.

L'ingénierie des exigences est une discipline foncièrement pluridisciplinaire [4], empruntant à de nombreux autres domaines de recherches des réflexions et techniques facilitant la collecte, la rédaction, la compréhension et l'évaluation des spécifications amonts. L'ingénierie des exigences fait partie de l'ingénierie logicielle et s'appuie de facto sur les travaux et résultats des sciences de l'ingénieur, de l'ingénierie système et des méthodes formelles. Elle est aussi influencée par les sciences humaines et cognitives, puisque l'information manipulée durant les étapes initiales d'un projet est obtenue par collaboration d'un grand nombre d'individus. Enfin, elle partage avec les domaines de représentation des connaissances (bases de données, intelligence artificielle …) les problématiques de modélisation et de gestion de l'information, puisque une spécification d'exigences est une collection de spécifications partielles décrites dans des langages variés [16].

L'ingénierie des exigences fournit un ensemble de techniques et méthodes pour la mise en place d'un processus d'ingénierie des exigences (IE). La mise en place d'un processus IE a pour but l'application de techniques de validation et de vérification pour la mesure de la qualité d'une spécification d'exigences et de la qualité du logiciel produit. Cette annexe porte sur la nature de ce dernier. Elle décrit la nature d'un processus d'ingénierie des exigences. Elle établit ensuite le distinguo entre activités de vérification et validation des exigences. Enfin, elle définit la notion de qualité des exigences et fait la liste des facteurs susceptibles de limiter la qualité des exigences.

1. Nature d'un processus d'ingénierie des exigences

Un processus IE est un processus de raffinement itératif visant la production d'une spécification d'exigences de qualité. Ce processus est composé d'un ensemble d'activités interdépendantes. Sommerville en distingue quatre, à savoir l'identification, l'analyse, la validation et la gestion des exigences [174]. Nuseibeh et Easterbrook [4] y ajoutent la modélisation, la vérification des exigences et identifient en outre deux activités transversales, à savoir la communication et l'évolution.

L'identification des exigences[102] vise à déterminer avec les parties prenantes les besoins, à décrire l'environnement du futur logiciel et à formuler un ensemble d'exigences satisfaisantes pour réaliser ces besoins dans cet environnement. La modélisation a pour but de produire un ensemble de spécifications reflétant ces informations. Les activités d'identification et de modélisation des exigences sont particulièrement source d'erreurs pour trois raisons principales [4] :

- Les parties prenantes expriment des exigences informelles, donc ambiguës, et pouvant ne pas correspondre au final à leurs besoins réels. Le raffinement de spécifications informelles est une tâche ardue.
- Les parties prenantes regroupent un ensemble hétéroclite de personnes (futurs utilisateurs du système, experts de domaines métiers, chefs de projets, analystes, architectes, ingénieurs, commerciaux …), qui diffèrent de par leurs compétences, connaissances, expertises, les préoccupations qu'elles doivent décrire, ainsi que par les intérêts qu'elles défendent au sein du projet.
- La taille des spécifications produites, la diversité des langages utilisés et la diversité des préoccupations considérées.
- Enfin, une erreur au niveau d'une spécification d'exigences a des risques d'être propagée par raffinement aux artefacts avals de développement (et la rectification d'une erreur est d'autant plus coûteuse qu'elle est identifiée tard).

De nombreuses méthodes et techniques associées ont été proposées pour faciliter, guider et systématiser les étapes d'identification et de modélisation. Les moins coûteuses consistent en des méthodes informelles comme les études individuelles (questionnaires, enquêtes, entretiens) et les études collectives (remue-méninges, ateliers). Les plus coûteuses comprennent les techniques de

[102] On parle aussi d'élucidation (« elicitation » en anglais), ce qui souligne la difficulté de cette tâche.

prototypage (maquettes d'interfaces homme machine, simulations fonctionnelles) et les méthodes guidant et structurant le raffinement des besoins (appelés objectifs) en exigences.

Parmi ces méthodes, on trouve les méthodologies i* [129] et KAOS [97], adaptées respectivement aux étapes d'analyse du domaine et d'identification des objectifs. La méthodologie i* permet de (i) représenter et raisonner sur les relations stratégiques entre les acteurs de l'environnement du système et (ii) de représenter les intérêts et les préoccupations des acteurs. Cette méthodologie facilite la compréhension des processus métiers de l'environnement du futur système et d'évaluer les différents périmètres possibles du système[103]. La méthodologie KAOS vise à décrire informellement les besoins à l'aide d'arbres logiques (et-ou) de raffinement. Cette méthodologie offre des méthodes pour la détection de conflits entre besoins (e.g. [98]) et d'obstacles à leurs réalisations [99].

Les langages de modélisation proposés dans la littérature pour spécifier les exigences peuvent être classés suivant quatre critères : (i) les étapes du processus IE supportées, (ii) le ou les type(s) d'informations représentés, (iii) leur précision et (iv) l'accessibilité de leur syntaxe. Ces critères ne sont pas indépendants. Chaque étape d'un processus IE nécessite la représentation d'un ou plusieurs types d'informations particuliers. Par exemple, l'étape d'analyse du domaine nécessite une description des interactions entre agents dans le but de définir un périmètre (ontologie organisationnelle, voir langage i* [129]). En outre, un grand nombre d'étapes supportées nécessite un plus grand nombre de types d'information représentés. Par exemple, le langage généraliste KAOS [97] supporte les étapes d'identification des besoins (ontologie intentionnelle) et de production d'une spécification opérationnelle (ontologie statique et dynamique). De plus, les étapes supportées influent sur le niveau de précision : seule l'étape de production d'une spécification opérationnelle nécessite un langage formel (i* et une partie de KAOS sont semi-formels). Enfin, le niveau de précision et l'accessibilité sont fortement liés : un langage formel est difficilement accessible à l'ensemble des parties prenantes car souvent muni d'une syntaxe hermétique.

2. Différence entre validation et vérification des exigences

Les activités de *validation* et de *vérification* (V&V) visent le même but, à savoir l'évaluation de la qualité des spécifications produites tout au long du développement logiciel (logiciel compris). La distinction entre validation et vérification n'est pas toujours respectée dans la littérature. Nous adoptons ici les définitions de référence proposées par Boehm [175] :

> **Définition** – ***validation*** : activité consistant à évaluer si le logiciel produit répond bien aux besoins réels des parties prenantes (répond-on au problème posé ?).
>
> **Définition** – ***vérification*** : activité consistant à évaluer si le logiciel est correctement développé (réalise-t-on correctement la solution ?).

La Figure 82 illustre cette différence. La validation est une confrontation entre une spécification et les besoins réels. Une activité de validation implique donc directement les parties prenantes. Les spécifications traditionnellement validées sont le logiciel (car concret pour l'utilisateur et exécutable) et les spécifications amonts (car ne comportant pas de détails techniques). En IE, les principales approches sont les méthodes de relecture des spécifications avec les parties prenantes (e.g. [176]), l'évaluation de prototypes (e.g. [177]), la simulation des exigences fonctionnelles et la génération de tests système [55, 57, 178-181], l'analyse des résultats des phases de test système [182]. La vérification est une confrontation entre deux spécifications. Les techniques proposées sont issues des méthodes formelles et sont chacune adaptées à la vérification d'un type particulier d'incohérences. Elles se basent toutes sur la confrontation de modèles formels. Les activités de vérification sont rarement exhaustives dans la pratique (l'explosion combinatoire est un obstacle à la preuve).

[103] Le choix d'un périmètre influe fortement sur les besoins formulés par les parties prenantes (il existe autant de systèmes possibles qu'il existe de périmètres envisageables pour un projet donné).

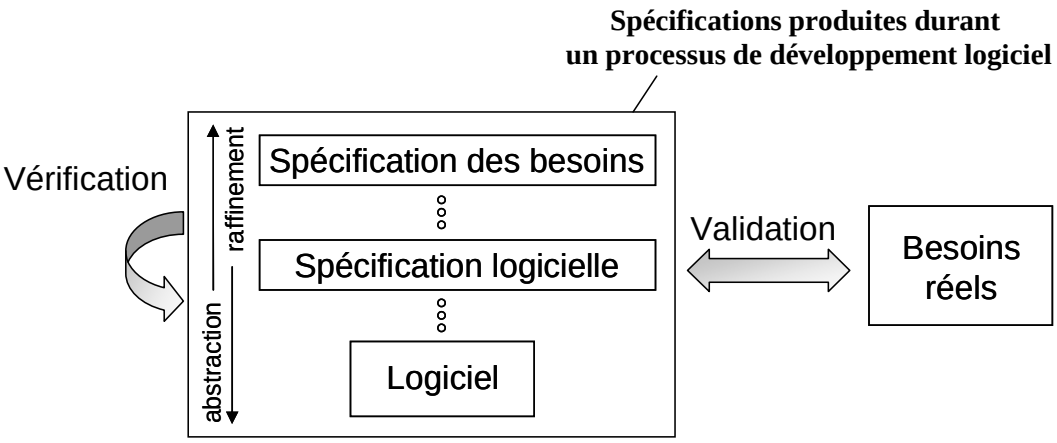


Figure 82 – Distinction entre activités de vérification et de validation.

3. Notion de qualité d'une spécification d'exigences

Il est admis de longue date que les échecs de projets logiciels sont en majorité dus à la piètre qualité des premières spécifications produites [32-36]. De récentes études statistiques de grande envergure (rapport CHAOS [48], rapport du MetaGroup [49], projet ESPITI [183]) viennent confirmer ce constat, concluant que (i) 52% des causes d'échecs sont directement imputables à des problèmes liés aux spécifications amont, et que (ii) 60% des causes d'échecs ont pour origine une mauvaise compréhension des besoins[104]. Ces études soulignent la nécessité de distinguer les erreurs produites durant les activités amont et aval de développement : (i) dans le premier cas, les spécifications amonts (spécifications des besoins, spécifications des exigences) ne reflètent pas les besoins réels et/ou les exigences ne peuvent être satisfaites car incohérentes ; (ii) dans le deuxième cas, les erreurs concernent les spécifications aval (artefacts produit durant l'implémentation) et le logiciel réalisé ne satisfait pas in fine les exigences fournies.

Le but d'un processus IE est de minimiser ou maîtriser les facteurs susceptibles de compromettre un projet durant ses phases amont, afin d'obtenir une spécification d'exigences de qualité. L'IEEE (pour *Institute of Electrical and Electronics Engineers*) fournit aux industriels une norme définissant la notion de qualité d'une spécification d'exigences (SRS). Cette norme (IEEE std 830-1993 [184]) stipule que les exigences capturées doivent satisfaire (idéalement) les propriétés suivantes :

- ***adéquation*** : chaque exigence reflète un besoin réel.
- ***non-ambiguïté*** : chaque exigence ne peut être interprétée que d'une seule manière.
- ***complétude*** : les exigences reflètent l'ensemble des besoins. Cette propriété est par nature difficile à évaluer. Elle nécessite au minimum pour être évalué une spécification globale de l'ensemble des exigences.
- ***cohérence*** : les exigences doivent (i) être cohérentes entre elles et (ii) doivent satisfaire les spécifications plus abstraites (par exemple, les objectifs).
- ***prioritisation*** : les exigences sont classées suivant leur importance et/ou leur stabilité.
- ***vérifiabilité*** : pour chaque exigence, il existe un procédé permettant de vérifier si le logiciel produit la satisfait. Ce procédé doit être raisonnable en temps et coût d'application.
- ***modificabilité*** : les exigences doivent être facilement modifiables. Cela nécessite une organisation cohérente et intuitive, un référencement précis, l'absence de redondance et une bonne modularité.
- ***traçabilité*** : chaque exigence est traçable en amont et en aval : son origine est clairement localisée dans les spécifications plus abstraites (traçabilité amont) et les artefacts logiciels

[104] Un besoin incorrect est raffiné en exigences incorrectes (ces pourcentages ne forment pas une partition).

concourant à sa satisfaction sont clairement identifiés dans les spécifications plus détaillées (traçabilité aval).

Outre la qualité d'une spécification d'exigences, un processus d'ingénierie des exigences doit limiter les facteurs de risques d'échecs d'un projet logiciel. La liste suivante fournit une synthèse de ces facteurs (voir études statistiques [48-49, 183]) :

- Manque d'implication de l'utilisateur final (risque accru de production d'exigences ne reflétant pas les besoins réels).
- Incomplétude des besoins.
- Evolution des besoins durant les activités aval de développement. Ce phénomène rend difficile la validation du logiciel car les exigences sont rarement mises à jour.
- Incompréhension des spécifications amont par des parties prenantes.
- Exigences incohérentes et/ou besoins conflictuels.
- Maintenance et manipulation d'un grand nombre de *points de vue* incohérents et hétérogènes (capture de préoccupations diverses avec des langages divers).
- Absence de traçabilité de l'information (historique d'évolution, éléments de décisions).
- Négligence de l'environnement dans lequel le logiciel est finalement déployé (particulièrement vrai pour les systèmes embarqués).

Annexe C Cadre conceptuel en ingénierie des exigences.

Les notions clés de l'IE ne portent pas toujours le même nom dans la littérature, ce qui apporte son lot de confusions et rend difficile la comparaison des travaux[105]. L'ambiguïté des termes employés est due à la diversité des communautés publiant dans le domaine de l'IE (chacune ayant une terminologie propre), à la difficulté de formaliser des notions généralistes, et dans certain cas à l'usage intempestif de termes « à la mode ». Dans cette annexe, nous fixons la terminologie utilisée dans ce manuscrit et présentons la relation de satisfaction entre exigences et logiciel. Nous prenons comme références les travaux théoriques de Zave et Jackson [85, 159, 185-186], proposant à notre connaissance le cadre conceptuel pour l'IE le plus abouti (à la fois généraliste et suffisamment formel pour ne pas laisser place à la confusion).

1. Terminologie

Système : Un système [106] englobe le *logiciel* [107] résultat d'un projet de développement et un *environnement*. Un système consiste en un ensemble d'*entités* et de *phénomènes*[108]. Une entité est un concept du domaine métier modélisé ou un agent (voir définition suivante). Un phénomène est une situation particulière observable, comme l'action d'un agent ou un intervalle de temps pour lequel une condition sur les états du système est vraie. L'étude des interactions entre l'environnement et le logiciel aboutit à la définition du *périmètre*[109] du système, c'est-à-dire l'ensemble des entités de l'environnement en interaction (directe ou pas) avec le logiciel.

Agent : Un agent (au sens de Yu [17]) est une entité de l'environnement interagissant avec les phénomènes du système. Ce peut être un *acteur* humain ou un logiciel existant. Un agent est responsable de ses actes, c'est-à-dire qu'il se comporte suivant un ensemble de règles prédéfinies.

Environnement : L'environnement est une partie de la réalité avec laquelle le logiciel interagit une fois développé et déployé. La description d'un environnement modélise les phénomènes propres à cette réalité comme les *concepts métiers* et les *agents* mis en jeu et leurs relations. Un environnement est décrit par un ensemble de propriétés appelées *assertions environnentales*. Ces propriétés sont dîtes *indicatives* [110] car vrai par définition. Nous proposons ci-dessous trois exemples d'assertions environnementales :

(a) L'eau bout à 100°C.

(b) Une porte est fermée si elle n'est pas ouverte.

(c) Un utilisateur ne peut présenter son badge à une porte que s'il se trouve dans une pièce « contenant » cette porte.

Définition – *Assertion environnementale* : une assertion environnementale est une propriété indicative sur les phénomènes de l'environnement. Elle est toujours vraie, que le système soit ou non déployé dans l'environnement.

N.B. : Dans la suite, le terme « logiciel » désigne implicitement le logiciel résultat du projet de développement.

La Figure 83 présente schématiquement ces notions. Le périmètre du logiciel est tracé en pointillés. Les interactions entre logiciels (sujets du projet de développement ou pas) sont représentées par des

[105] C'est un problème de collision. En somme, l'IE souffre des problèmes qu'elle cherche à maîtriser, comme la communication entre un grand nombre d'individus (les chercheurs sont après tout parties prenantes d'un projet visant un progrès scientifique).

[106] Nous disqualifions ici une autre dénotation possible, à savoir le logiciel uniquement.

[107] Un logiciel peut être la composition d'un ensemble de logiciels. Jackson utilise le terme de « machine ».

[108] Les définitions de ces deux notions nous sont propres.

[109] « boundaries » dans la littérature anglophone.

[110] Lamsweerde utilise le terme « descriptive ».

intersections ensemblistes, celles mettant en jeu un acteur sont représentées par un trait. Cette figure illustre le cas d'interactions indirectes (par exemple un logiciel agit sur un acteur agissant à son tour sur le logiciel en construction). Il est important de noter que ces interactions forment des chaînes de causalités complexes reliant le logiciel en construction aux éléments de l'environnement. Le logiciel en construction peut aussi modifier l'environnement (cas des systèmes réactifs). Cette figure souligne aussi la distinction entre « environnement » au sens large et environnement comme défini plus haut : les agents et logiciels n'entrant pas en interaction avec le logiciel ne sont pas considérés dans l'environnement (ils pourraient l'être en considérant un autre périmètre).

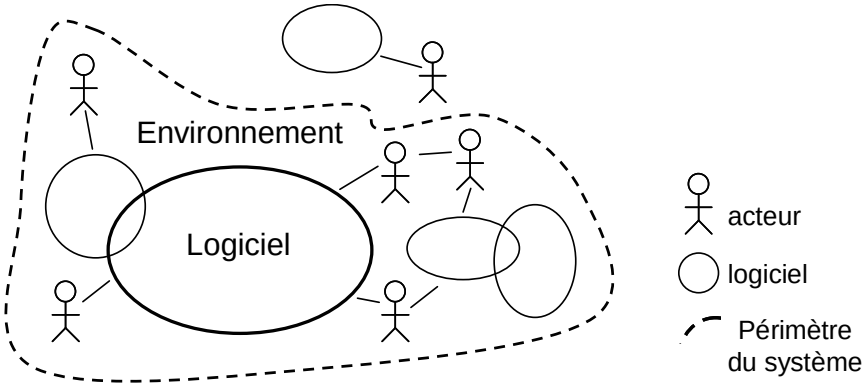


Figure 83 - Le logiciel est déployé dans et interagit avec un environnement au sein d'un système.

Besoin : Le but d'un projet logiciel est d'altérer un système existant (sans le logiciel sujet du projet de développement) de sorte qu'il satisfasse les besoins réels des parties prenantes. Une spécification des besoins est un ensemble de *besoins* (les objectifs de Lamsweerde [97]), décrivant à un haut *niveau d'abstraction* les modifications à apporter au système initial.

Exigence : Une exigence est une propriété participant à la réalisation d'au moins un besoin. C'est une propriété dite *optative* [111], c'est-à-dire souhaitée. Une spécification d'exigences contraint l'environnement de sorte que les besoins soient satisfaits. On distingue différents types d'exigences dans la littérature : exigences *fonctionnelles*, *non-fonctionnelles*, ou encore *QoS* (de *qualité de service*). Ces distinctions sont sources de confusion car rarement définies. Jackson remarque ainsi [159] que le terme « non-fonctionnel » est généralement utilisé pour qualifier une exigence ne pouvant être formalisée[112]. On notera ici que ce classement dépend donc du ou des langages utilisés pour modéliser les exigences.

> **Définition – *Exigence*** : une exigence exprime une propriété optative sur les phénomènes de l'environnement, vraie après déploiement du logiciel dans l'environnement. Une exigence peut être formalisable ou non, suivant le formalisme utilisé pour les exprimer.

Nous proposons ici notre propre définition. Une exigence est fonctionnelle si elle décrit (au moins partiellement) les liens de causalité relatifs à une action du système. Les autres exigences sont dites non-fonctionnelles. Ce sont alors des contraintes : liens de causalité entre états du système, et/ou entre activation d'une ou plusieurs actions. Cette définition fournit un premier classement des exigences (une partition). Une exigence est dite de qualité de service lorsqu'elle porte sur une préoccupation de qualité (performance, sûreté de fonctionnement, sécurité …). Cette définition fournit un deuxième classement, orthogonal au premier. Notons enfin que les qualificatifs « fonctionnel » et « non-fonctionnel » sont aussi applicables aux assertions environnementales[113] (ce qui n'est pas souligné dans la littérature), d'où la définition proposée ci-dessous :

[111] Lamsweerde utilise le terme « prescriptive ».

[112] Jackson considère comme fonctionnelles un grand nombre d'exigences désignées comme non-fonctionnelles dans la littérature. En effet, le qualificatif « non-fonctionnelle » est souvent attribué à des contraintes faisant référence à des durées mais contraignant néanmoins l'activation de fonctions du système (dans ce cas, l'exigence (e) présentée plus bas est considérée comme non-fonctionnelle).

[113] Par exemple, les assertions environnementales (a) et (b) présentées plus haut sont non-fonctionnelles.

Définition – *propriétés fonctionnelles et non-fonctionnelles* : une propriété (exigence ou assertion environnementale) est fonctionnelle si elle décrit des liens de causalité relatifs à une action d'un agent du système. Elle est dite non-fonctionnelle sinon. Une propriété non-fonctionnelle est aussi appelée *contrainte*.

Nous fournissons ci-dessous cinq exemples d'exigences. D'après les définitions données précédemment, les exigences (d, e, h) sont fonctionnelles, l'exigence (f) est non-fonctionnelle et l'exigence (g) est a priori non formalisable. En outre, (e) est une exigence de performance, (f) est une exigence de sécurité et (g) est une exigence de sûreté de fonctionnement (disons que les détecteurs X37 déterminent si un individu est dans l'embrasure d'une porte et que la norme X32-55/3 certifie leur fiabilité).

(d) Une porte s'ouvre si un utilisateur présente son badge et que ce dernier est valide.

(e) La porte doit s'ouvrir moins de 5 secondes après que l'utilisateur s'est identifié.

(f) Aucune porte n'est ouverte lorsque l'alarme est activée et qu'aucun individu n'est dans le bâtiment.

(g) Les détecteurs X37 doivent respecter la norme X32-55/3.

(h) Une porte se ferme automatiquement après le passage d'un individu si un incendie a été détecté.

Domaine : La description des exigences est indissociable d'une description de l'environnement (puisque les exigences contraignent l'environnement). Jackson appelle domaine l'union de ces deux descriptions. Le terme « domaine » fait référence au niveau M1 de l'architecture MDA (cf. section 2.3) : une description du domaine est donc un modèle de niveau M1 (c'est une base de connaissance).

2. Relation de satisfaction entre logiciel et exigences

Les exigences portent sur les phénomènes de l'environnement. Certains de ces phénomènes sont partagés par l'environnement et le logiciel ; les descriptions de l'environnement et du logiciel modélisent toutes deux ces phénomènes puisqu'ils sont observés et/ou modifiés à la fois par l'environnement et le logiciel (le diagramme d'analyse d'un logiciel représente entre autres les concepts métiers partagés). Pour reprendre les exemples d'exigences précédents, le logiciel a une représentation interne des portes d'un bâtiment et connaît leur état. Néanmoins, la plupart des exigences portent sur des phénomènes externes au logiciel [159].

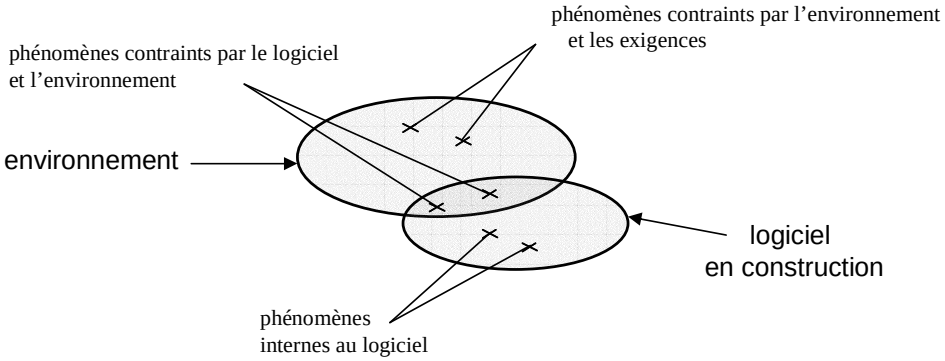


Figure 84 – Schématisation de la relation entre environnement, exigences et logiciel en construction.

Dans ce contexte, la satisfaction des exigences par un logiciel est le résultat d'une chaîne de causes à effets reliant les phénomènes internes du système aux phénomènes de l'environnement. La Figure 84 schématise ce point de vue et permet d'introduire la *relation de satisfaction* entre spécification logicielle et spécification d'exigences. Une spécification logicielle définit un ensemble de contraintes sur l'environnement de sorte que les exigences soient satisfaites une fois le logiciel déployé. Cette relation est de même nature que celle liant les besoins aux exigences ou le code d'un logiciel à la

spécification logicielle (par exemple, les besoins réels sont satisfaits ssi les assertions environnementales au sein du périmètre décrivent fidèlement la réalité et la spécification des besoins est satisfaite.).

Définition – *Relation de satisfaction* [159] : Une spécification système S satisfait un ensemble d'exigences R définies sur un environnement décrit par un ensemble d'assertions environnementales E si et seulement si on a E, S ├ R.

Cette relation est intéressante car elle permet de faire les remarques suivantes :

- Les phénomènes partagés par les spécifications d'exigences, de l'environnement et du logiciel doivent être représentés de la même manière si l'on désire vérifier la relation de satisfaction.
- Le rôle d'une spécification d'exigences lors d'une activité de vérification dépend du logiciel considéré en construction (nous avons vu qu'un système peut comporter plusieurs logiciels). Une exigence est optative si elle fait partie du logiciel vérifié, elle est indicative sinon (considérée comme une assertion environnementale). Prenons par exemple une exigence d'un service Internet proposé par une banque à un client via son site : « une transaction bancaire doit s'effectuer en moins d'une seconde ». Cette exigence sera testée si la vérification porte sur l'implémentation de ce service. En revanche, elle exprime une réalité si le logiciel vérifié est un système de gestion de portefeuille boursier utilisant ce service entre autres.
- Cette relation signifie que les exigences peuvent être déduites de la seule connaissance de E et de S. Elle signifie aussi qu'une spécification logicielle peut être dérivée de R connaissant E [186]. Jackson et Nusebeih donnent un exemple de dérivation formelle (raffinement) d'une spécification système à partir d'exigences de sécurité et de gestion de l'énergie dans [187]. Le système est un contrôleur d'appareils domotiques et les exigences de ces appareils sont dans ce cas considérées comme des assertions environnementales (les appareils sont déjà développés et font donc partie de l'environnement).

Annexe D Architecture de la plate-forme R2A.

La plate-forme R2A a entièrement été développée avec des technologies IDM. Cette section présente les technologies IDM utilisées et l'architecture générale de la plate-forme. Cette annexe présente brièvement l'environnement de métamodélisation Kermeta utilisé pour développer la plate-forme, puis présente l'architecture de la plate-forme au sein de Kermeta et l'environnement de développement Eclipse.

1. L'environnement de métamodélisation Kermeta

Nous avons vu en section 3.2.2 du Chapitre II une description du langage d'action Kermeta. Kermeta est aussi une plate-forme de métamodélisation libre, développée au sein de l'équipe Triskell [146] et intégrée à l'environnement de développement Eclipse [170] d'IBM. Cette plate-forme de métamodélisation fédère un ensemble de plug-ins Eclipse réalisant des fonctionnalités essentielles à un environnement de métamodélisation. Parmi ces plug-ins, on peut citer EMF [154] (*Eclipse Modeling Framework*), le plug-in Eclipse supportant une manipulation homogène des modèles et métamodèles au format XMI [188] et TopCased [166], un plug-in de visualisation graphique des métamodèles EMOF .

Kermeta est aussi livré avec Syntaks [160], un outil générique pour la spécification de syntaxe textuelle pour les modèles. Syntaks permet de produire une représentation sous forme de modèle d'un texte (*parsing*) mais aussi l'inverse, à savoir la représentation textuelle d'un modèle (*pretty printing*). Syntaks a été construit suivant une approche orientée modèle : la relation entre une syntaxe abstraite (un métamodèle) et une syntaxe concrète (une grammaire) est spécifiée par un modèle (modèle Syntaks). Celui-ci décrit les relations entre les classes d'un métamodèle et la représentation de leurs instances dans un texte. Nous avons utilisé Syntaks pour implémenter le langage RDL présenté en Chapitre IV.

Ces spécificités font de Kermeta un environnement homogène adapté pour la manipulation des modèles, la spécification de sémantiques opérationnelles et plus généralement la création de langages spécialisés (DSL). La section suivante présente l'architecture de la plate-forme R2A et son intégration à Kermeta et Eclipse.

2. Intégration dans les environnements Kermeta et Eclipse

Cette section ne fournit pas une description technique des composants de la plate-forme R2A, mais plutôt une vision schématique de leurs interactions avec les environnements Kermeta et Eclipse. La Figure 85 présente les dépendances entre les environnements Eclipse, Kermeta et la plate-forme R2A. Les rectangles en trait plein représentent des composants et les rectangles en pointillés représentent un composite (une plate-forme ou un environnement). La Figure 85 n'est pas exhaustive : seuls les composants les plus significatifs d'Eclipse et de Kermeta ayant un rôle dans le fonctionnement de la plate-forme R2A sont représentés. Les dépendances entre composants doivent être déduites de leurs positions relatives selon l'axe vertical : un composant dépend de ceux placés au-dessus. Par exemple, le métamodèle cœur RM dépend de l'environnement *Kermeta* et du composant *EMF* de l'environnement *Eclipse*. Tous les composants de la plate-forme R2A sont intégrés à l'environnement Eclipse sous la forme de plug-ins.

Les composants « Processus de composition », « RM » et « RDL » sont détaillés dans de ce manuscrit. Pour plus de détails sur les autres composants, le lecteur pourra consulter le manuel de la plate-forme (non fourni avec ce manuscrit). La suite de cette section donne une description succincte des autres composants représentés dans la Figure 85 (exceptés les composants KermetaLanguage et Syntaks déjà introduits) :

- ***EMF*** [154] est le dépositaire de modèles d'Eclipse. Il permet la manipulation en java des modèles et métamodèles. Il fournit le méta-métamodèle Ecore, une implémentation de EMOF, une API et un éditeur de modèles arborescent.

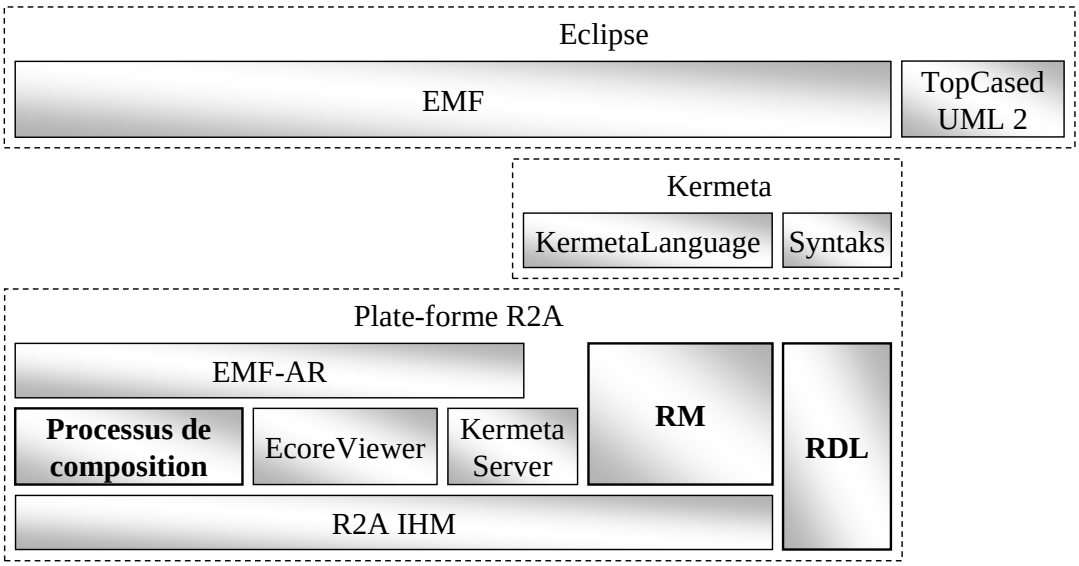


Figure 85 - Diagramme de dépendances entre la plate-forme R2A, Eclipse et Kermeta.

- ***TopCased - UML 2*** [166] est un atelier de développement pour systèmes embarqués et/ou critiques basé sur l'ingénierie dirigée par les modèles. Il fournit une implémentation du langage UML version 2 ainsi qu'un éditeur graphique dédié. TopCased offre de plus une API extensible facilitant l'implémentation d'éditeurs graphiques pour DSLs.
- ***EMF-AR*** est un dépositaire de modèles construit sur EMF. Il offre des mécanismes de manipulation de modèles plus abstraits et définit explicitement les notions de modèles et métamodèles (ce sont uniquement des ressources XMI dans EMF). Il supporte de plus la notion de type de métamodèle et offre un mécanisme de détection de violations de conformité (cf. section 4.3.1 du Chapitre VI).
- ***EcoreViewer*** est un éditeur graphique de modèles (visualisation des modèles sous forme de graphes d'objets, et des métamodèles sous forme de diagrammes de classes UML). Ce composant est très utile aussi bien dans le cadre d'un développement dirigé par les modèles que durant l'analyse d'exigences. Une représentation graphique d'un modèle est en effet plus accessible qu'une représentation textuelle. EcoreViewer est utile lors de l'implémentation d'une transformation mais également lors de son test (la validation manuelle d'une transformation implique l'inspection des modèles produits) et sert d'outil de communication entre parties prenantes durant l'analyse des modèles d'exigences. EcoreViewer et TopCased sont complémentaires. Le premier génère une représentation graphique pour tous les modèles mais ne permet pas de les éditer. Le deuxième permet la création de modèles graphiquement mais ne peut générer une vue d'un modèle qui n'a pas été créé manuellement. EcoreViewer est extensible comme TopCased (création d'un éditeur graphique dédié à un DSL). Tous les graphes d'objets dans ce manuscrit ont été générés avec EcoreViewer.
- ***KermetaServer*** permet l'interopérabilité entre des codes Java et des codes Kermeta. Dans la plate-forme R2A, il permet la synchronisation entre les actions de l'utilisateur et l'exécution des fonctionnalités codées en Kermeta et embarquées par le RM.
- ***R2A IHM*** est l'interface graphique de la plate-forme R2A. Elle permet le contrôle par l'utilisateur des fonctionnalités de la plate-forme, introduites en section suivante.

Annexe E Description des vues du formalisme RM.

1. Vue métamodélisation

La vue métamodélisation représente le cadre conceptuel IDM. Elle capture les notions de langage, métamodèle, modèle, sémantique statique et dynamique. La sémantique statique d'un langage est un métamodèle. La sémantique dynamique peut être le même métamodèle muni d'une sémantique opérationnelle définie à l'aide de Kermeta (la métaclasse KMETAMODEL), ou un homéomorphisme. Un homéomorphisme associe un domaine au langage et est spécifié à l'aide de deux transformations : une *interprétation* et une *fusion*. La vue métamodélisation est utile pour représenter les langages des spécifications partielles composées (voir la vue classification). Elle autorise de plus la sauvegarde sous la forme d'un modèle de l'état d'un processus IE.

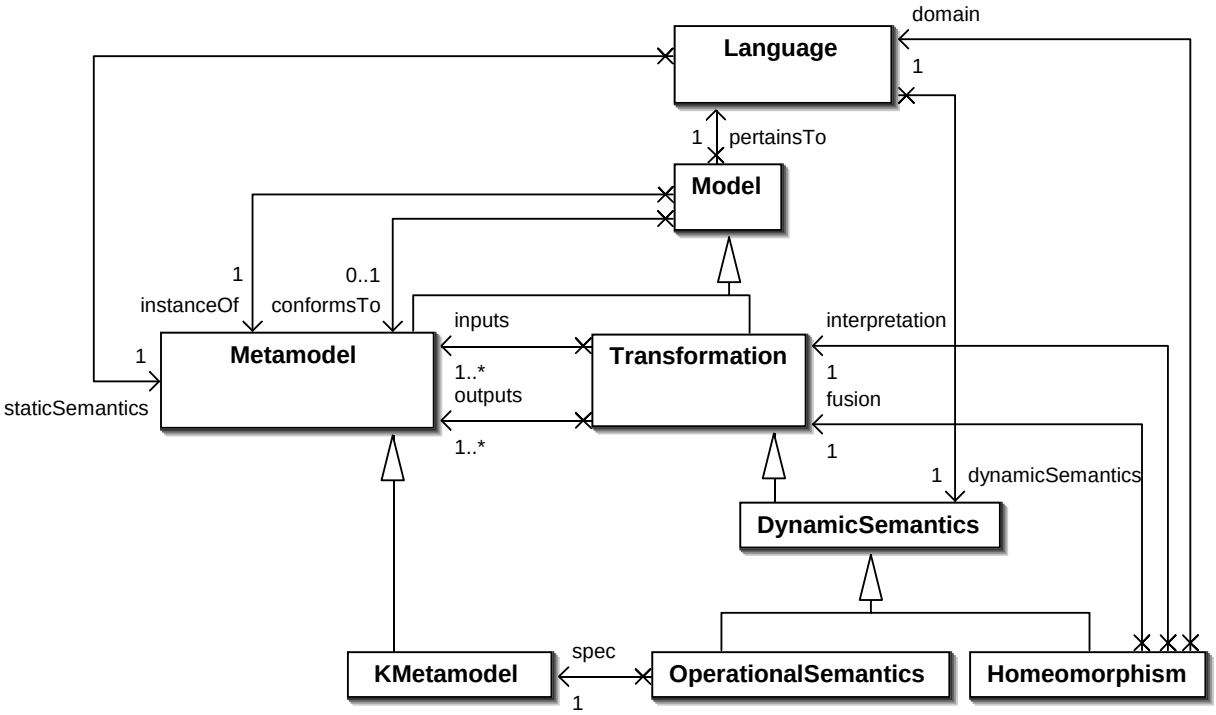


Figure 86 - Vue métamodélisation du métamodèle RM.

2. Vue traçabilité

Les liens de traçabilité peuvent être scindés en deux groupes distincts [189] : ceux concernant l'origine de l'information avant composition des spécifications partielles (liens amont) et ceux résultant de l'intégration de ces spécifications dans la plate-forme R2A via le processus de composition (liens aval) [114]. La vue traçabilité capture les liens de traçabilité aval. Les liens amonts sont spécifiés au sein des points de vue (voir la vue classification). Les liens de traçabilité produit par le processus de composition sont très précis (à l'échelle de l'objet), ce qui permet de déterminer automatiquement la correspondance sémantique entre éléments de la spécification globale et éléments des spécifications partielles composées.

Modèle de traçabilité (TRACEMODEL). Le modèle de traçabilité regroupe l'ensemble des liens de traçabilité aval (ATOMICTRACE). Un lien de traçabilité aval relie deux ensembles d'objets (EOBJECT)

[114] Respectivement *pre-requirements specification* et *post-requirements specification* dans la littérature.

appartenant aux spécifications partielles ou à la spécification globale, suivant la règle l'ayant produit (RULE). La nature et la production de ces liens sont décrites dans la section 4.2 du Chapitre VI.

Règle (RULE). Une règle est un élément de spécification d'une fonction sémantique. Ces règles sont décrites dans le Chapitre VI.

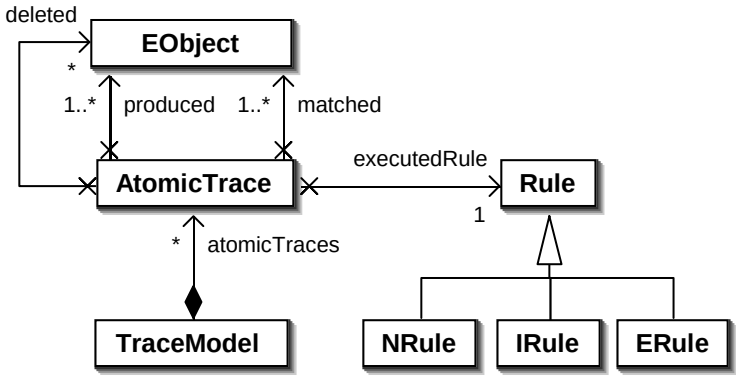


Figure 87 - vue traçabilité du métamodèle RM.

3. Vue classification

La vue classification représente les notions utilisées en IE pour classer une grande quantité d'informations. Cette vue fournit une représentation possible des points de vue de Finkelstein, auxquelles nous ajoutons la notion de spécification partielle et de notation. En capturant ces informations dans le métamodèle RM, il est possible de proposer des fonctionnalités d'extraction sélective de l'information se basant sur des critères de classification (création d'un point de vue suivant un ensemble de critères). Nous donnons ici une description sommaire des notions capturées par la vue classification. Ces notions sont illustrées dans les Chapitre V et Chapitre VI.

Notation (NOTATION). Une notation est un type de spécification partielle. Une notation comporte un ensemble de champs jouant le rôle de paramètres.

Champ (FIELD). Un champ décrit un type de donnée au sein d'une notation. On distingue les champs typés (*fieldType*) par un métamodèle et les champs simples (SIMPLE) comme un texte informel ou une énumération. A titre d'exemple, les pré-conditions, les post-conditions et les scénarios sont des champs de la notation des cas d'utilisation de Cockburn (cf. section 1 du Chapitre V pour une illustration).

Spécification partielle (PARTIALSPEC). Une spécification partielle est une instance d'une notation : c'est une notation où les champs ont été instanciés (*fieldValues*) avec une valeur conforme à leur type. Un cas d'utilisation de Cockburn est une spécification partielle par exemple.

Point de Vue (VIEWPOINT). Les points de vue offrent une structure grossière pour classer les spécifications partielles suivant des *critères de classification* (Qualitative). Certains de ces critères jouent le rôle de liens de traçabilité amont (REDACTOR par exemple). Nous ajoutons à chaque point de vue un modèle de traçabilité capturant les liens de traçabilité aval générés durant la composition.

Spécification (SPECIFICATION). La spécification est la donnée d'un ensemble de points de vue, d'un ensemble de spécifications partielles (PARTIALSPEC) et d'une spécification globale obtenue par composition de ces dernières (*globalModel*).

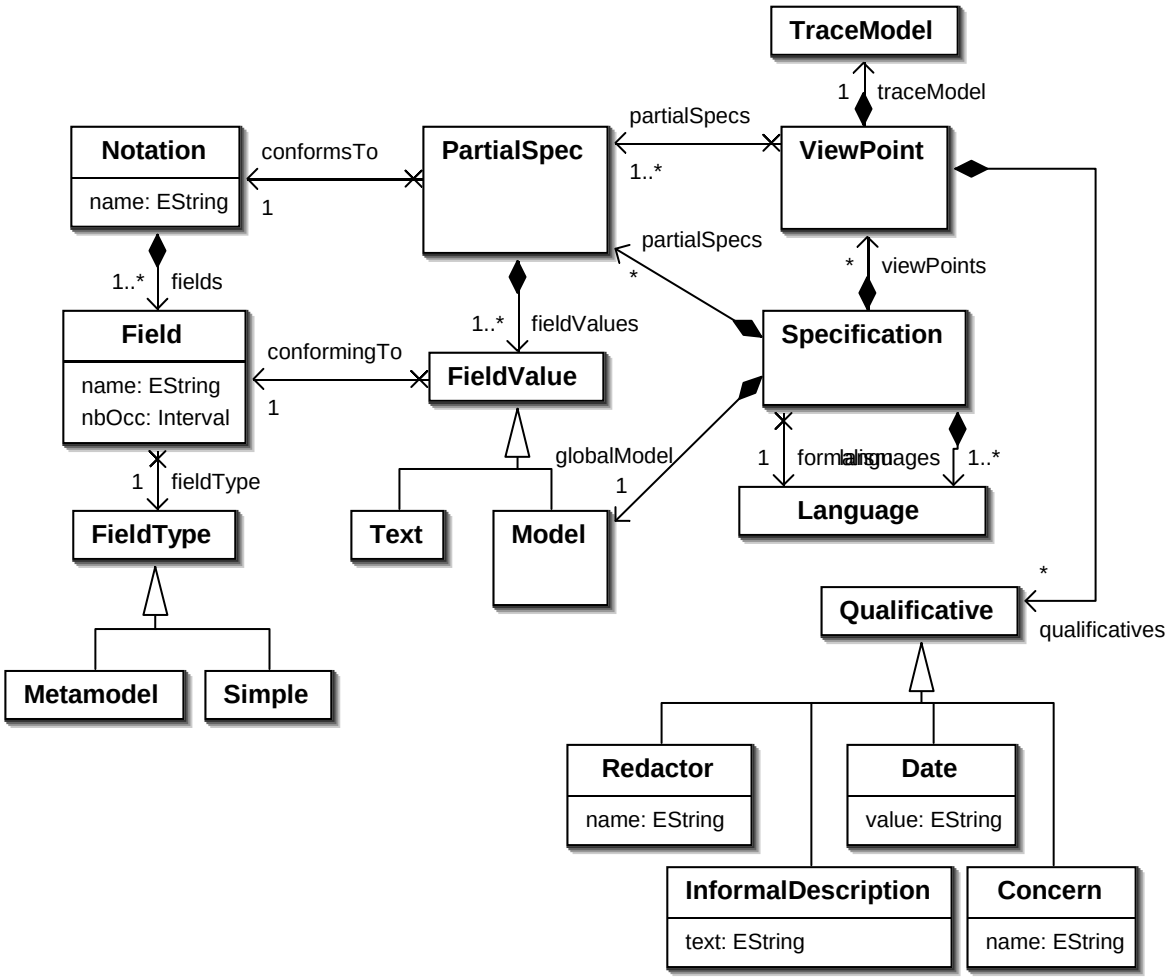


Figure 88 – vue classification du métamodèle RM.

Annexe F Langage RDL.

1. Grammaire du langage RDL pour les patrons ActionEffect et ActionCondition :

```
rdl_specification
    :       ( sentence '.' )* ;
sentence
    :       mainClause subordinateClause?
    |       subordinateClause ',' mainClause ;
mainClause
    :       clause ;
subordinateClause
    :       ( 'when' | 'When' ) clause
    |       ( 'if' | 'If' ) clause
    |       'before' clause
    |       'after' clause ;
clause
    :       subject verb complement1? complement2* ;
subject
    :       subjectPronoun
    |       nominalGroupExpr ;
verb
    :       ( 'does' | 'do' | 'did' ) 'not'? IDENT
    |       'can' 'not'? IDENT
    |       'must' 'not'? IDENT
    |       ( 'become' | 'becomes' )
    |       ( 'is' | 'are' ) ;
complement1
    :       nominalGroupExpr ;
complement2
    :       preposition nominalGroupExpr ;
nominalGroupExpr
    :       orExpr ;
orExpr
    :       andExpr ( 'or' andExpr )* ;
andExpr
    :       atomicElement ( 'and' v2 = atomicElement )* ;
atomicElement
    :       nominalGroup
    |       disjunctivePronoun ;
nominalGroup
    :       ( determinant | 'not‘ )? IDENT+ ( 'of' nominalGroup)? ;
identifier
    :       IDENT ;
determinant
    :       'this'
    |       'the' | 'The'
    |       'a' | 'A' | 'an' | 'An'
    |       'one' | 'One'
    |       'all' | 'All'
    |       'its'
    |       'his'
    |       'her'
    |       'their'
    |       'every' | 'Every' ;
preposition
    :       'to'
    |       'from'
```

```
        |       'in'
        |       'by'
        |       'with'
        |       'on'
        |       'of' ;
subjectPronoun
        :       'he'
        |       'they' ;
disjunctivePronoun
        :       'it'
        |       'them' ;
```

2. Vue patrons du métamodèle RDM

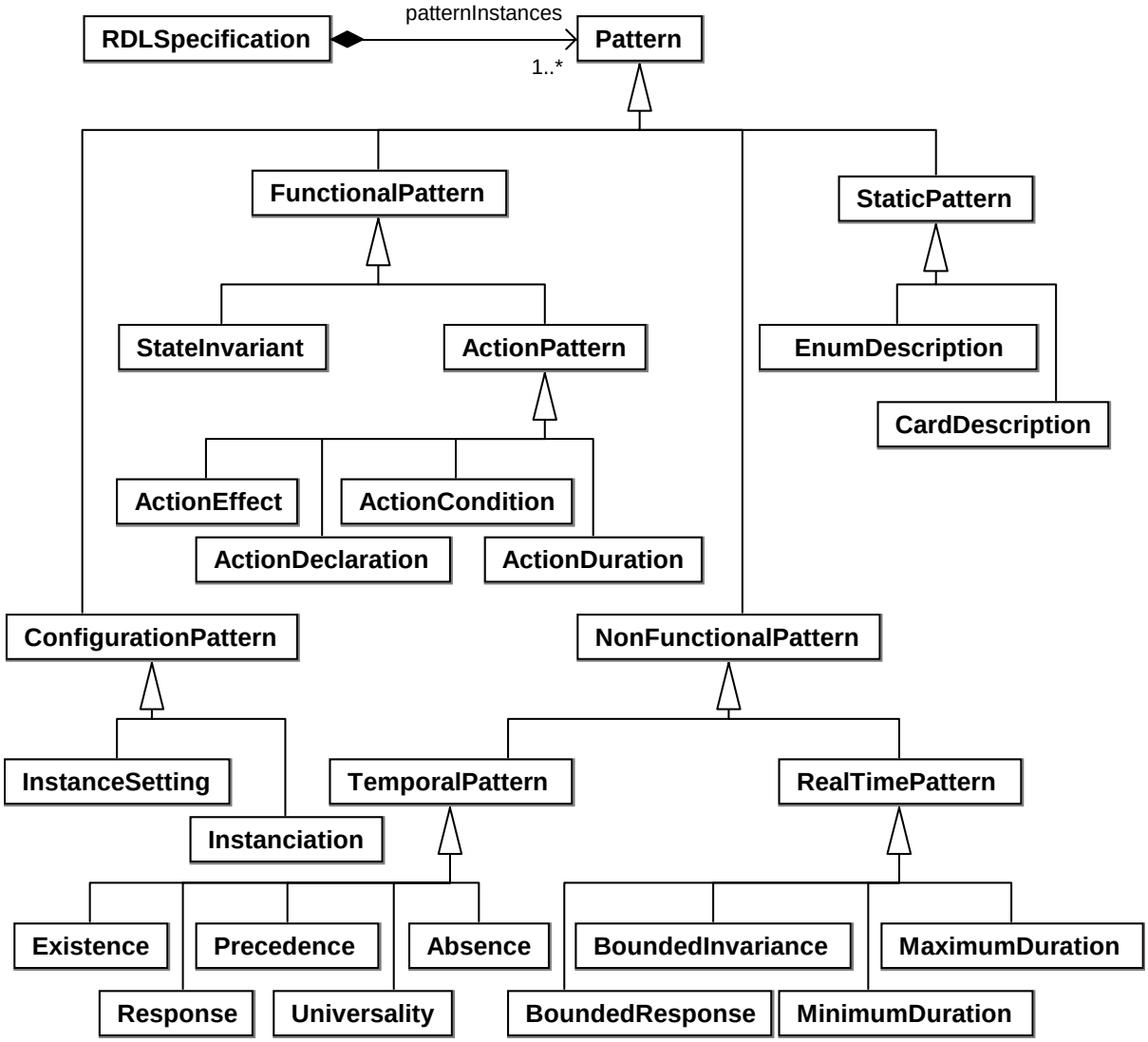


Figure 89 - Représentation des patrons de spécification RDL (extrait du métamodèle RDM).

3. Exemple d'une instance RDM complète

When the "type" of a "meeting" is "private", a "participant" must be "identified" before he can "enter" in this "meeting".

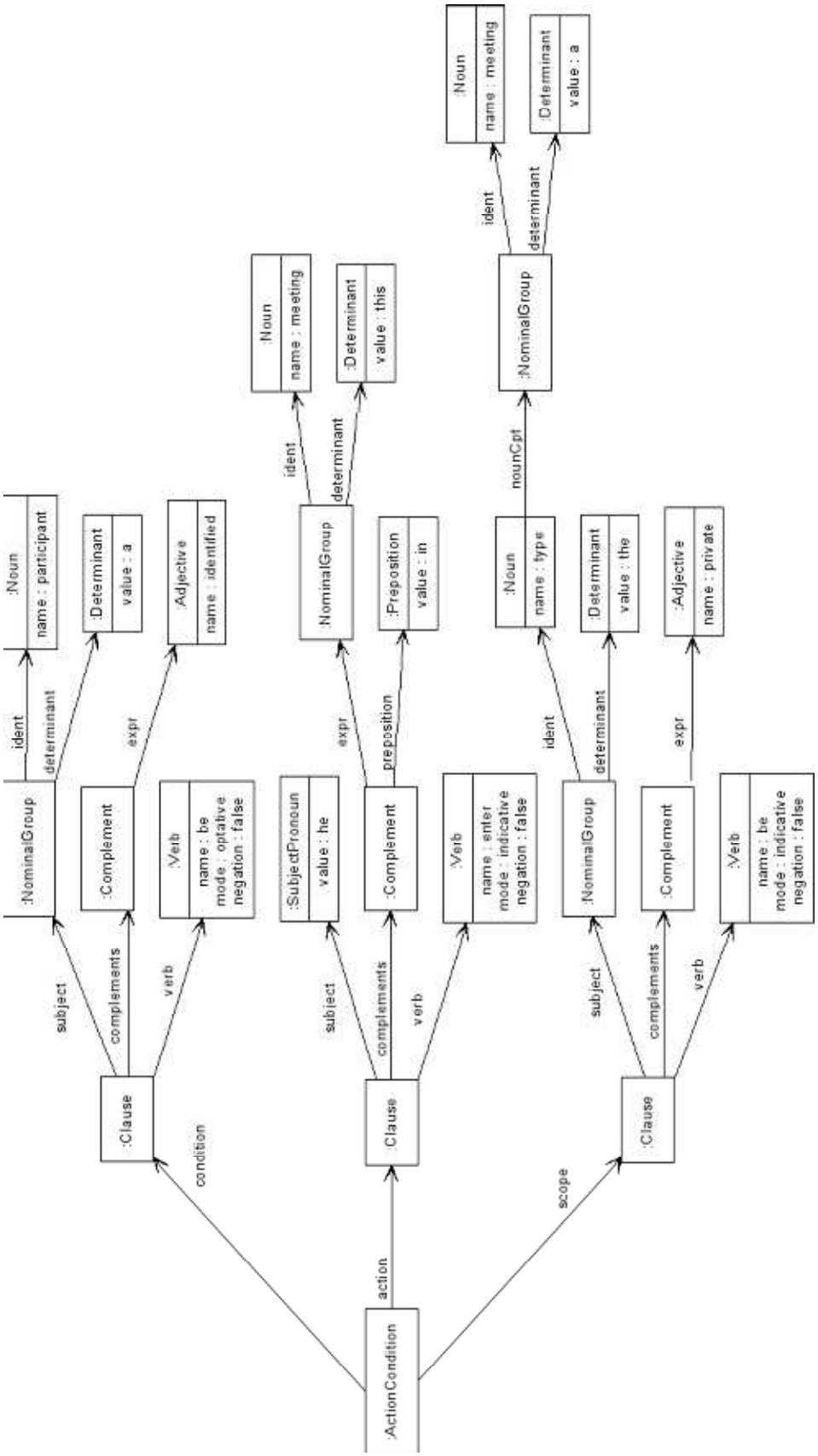


Figure 90 – Une phrase RDL décrivant une condition sur l'activation de l'action *enter in a meeting* et sa représentation comme instance du métamodèle RDM.

Annexe G Grammaire EBNF du langage IRL.

```
////////////////////
// PARSER

start           : interpretationSemantics EOF ;
interpretationSemantics
      :         importDecl*
                'IS' '(' inputParameters ')' ':' '(' inputParameters ')'
                interpretationSemanticsElt*;
interpretationSemanticsElt
      :         interpretationRule
      |         localFunction;
inputParameters
      : inputParameterDefPart (',' inputParameterDefPart)*;
inputParameterDefPart
      :         simpleParameterDef
      |         multiParameterDef;
simpleParameterDef
      :         IDENT ':' xClass;
multiParameterDef
      :         IDENT (',' IDENT)+ ':' xClass;
importDecl
      :         'import' (javaClassImport | metamodelImport) ';';
javaClassImport
      :         classPath ('alias' IDENT)?;
classPath
      :         IDENT ('.' IDENT)* '.*'?;
metamodelImport
      :         metamodelImport2;
metamodelImport2
      :         IDENT ':' metamodelType;
metamodelType
      :         METAMODEL;
interpretationRule
      :         'IR' IDENT '(' iRuleParameters ')' irGuard block;
localFunction
      :         basicFunction
      |         query
      |         pattern;
basicFunction
      :         'function' IDENT '(' inputParameters1? ')' (':' type1)?  block;
query
      :         'query' IDENT '(' inputParameters1 ')' ':' type1 instr;
pattern
      :         'pattern' IDENT '(' inputParameters1 ')' instr;
instr
      :         block
      |         '=' expression ';';
inputParameters1
      : inputParameterDefPart1 (',' inputParameterDefPart1)*;
inputParameterDefPart1
      :         simpleParameterDef1
      |         multiParameterDef1;
simpleParameterDef1
      :         IDENT ':' type1;
```

```
multiParameterDef1
        :       IDENT (',' IDENT)+ ':' type1;
iRuleParameters
        : iRuleParameterDefPart (',' iRuleParameterDefPart)*;
iRuleParameterDefPart
        :       simpleIRuleParameterDef
        |       multiIRuleParameterDef;
simpleIRuleParameterDef
        :       IDENT ':' xClass ('from' IDENT)? ;
multiIRuleParameterDef
        :       IDENT (',' IDENT )+ ':' xClass ('from' IDENT)?;
irGuard
        :       '?' ':' blockOrExpression;
blockOrExpression
        :       block
        |       expression ';';
expression
        :       orExpr;
orExpr
        :       andExpr ('or' andExpr)*;
andExpr
        :       impliesExpr ('and' impliesExpr)*;
impliesExpr
        :       equalityExpr ('=>' equalityExpr)?;
equalityExpr
        :       relExpr (('!=' | '==' ) relExpr)*;
relExpr
        :       addExpr (('<' | '>' | '<=' | '>=' | 'instanceof' ) addExpr)*;
addExpr
        :       multExpr (('+' | '-') multExpr)*;
multExpr
        :       unaryExpr (('*' | '/' | '%' ) unaryExpr)*;
unaryExpr
        :       ('-' | 'not') unaryExpr
        |       '(' expression ')'
        |       terminal;
terminal
        :       'void'
        |       primitiveInstance
        |       compositeExpression;
compositeExpression
        :       compositeElement1 compositeElement2*;
compositeElement1
        : primitiveFunction
        | mathOps
        | traceabilityQueryCall
        | metamodelReference
        | compositeElement;
compositeElement2
        :       '.' v1 = compositeElement
        |       v2 = listExtractor
        |       '#' xClass  '#'?;
traceabilityQueryCall
        :       xClass '@' ('[' stringList ']')? '(' functionParameters ')';
stringList
        :       IDENT (',' IDENT)*;
listExtractor
        :       '[' expression ']';
metamodelReference
        :       '[' '/'? (IDENT '/')* IDENT '.' IDENT ']';
```

```
primitiveFunction
        :       'inc' '(' expression ')'
        |       'dec' '(' expression ')';
mathOps
        :       'union' '(' expression ',' expression ')'
        |       'inter' '(' expression ',' expression ')'
        |       'diff'  '(' expression ',' expression ')'
        |       'card'  '(' expression ')';
compositeElement
        :       'asSet'
        |       'flatten'
        |       functionCall
        |       selectOperation
        |       collectOperation
        |       forallOperation
        |       existOperation
        |       foreachOperation
        |       elementReference;
functionCall
        :       IDENT '(' functionParameters? ')';
functionParameters
        :       expression (',' expression)*;
elementReference
        :       IDENT (':' IDENT)+
        |       IDENT '<' IDENT '>'
        |       IDENT '~' IDENT?
        |       IDENT;
selectOperation
        :       'select'  '(' IDENT (':' type)? ('|' expression)? ')';
collectOperation
        :       'collect' '(' IDENT (':' type)? '|' expression ')';
forallOperation
        :       'forall' '(' IDENT (':' type)? '|' expression ')' ;
existOperation
        :       'exist' '(' IDENT (':' type)? '|' expression ')';
foreachOperation
        :       'foreach' '(' IDENT (':' type)? ')' instruction;
type
        : modelType
        | type1;
type1
        : xClass
        | primitive;
modelType
        :       '[' '/'? (IDENT '/')* IDENT '.' IDENT ']';
xClass
        :       IDENT ( ('.' IDENT)* | (':' IDENT)* ) ('<' type (',' type)* '>')?;
instruction
        :       block
        |       conditional
        |       while_
        |       variableDeclaration
        |       expression (':=' v6 = expression)? ';'
        |       'return' expression ';';
primitive
        : 'int'
        | 'float';
block
        :       '{'  instruction* '}';
conditional
```

```
        :       'if' '(' expression ')' instruction
                ('elseif' '(' expression ')' instruction)*
                ('else' instruction)?;
while_
        :       'while' '(' expression ')' instruction;
variableDeclaration
        :       'var' IDENT ':' type (';' | 'init' blockOrExpression);
primitiveInstance
        :       'true'
        |       'false'
        |       STRING
        |       number;
number
        :       INT
        |       FLOAT;

//////////////////
// LEXER

INT
        :       ('0'..'9')+;
FLOAT
        :       ('0'..'9')+ '.' ('0'..'9')+;
IDENT
        :       ( ('a'..'z') | ('0'..'9') | ('A'..'Z') | ('_') )+ ;
STRING
        :       '"' (~('"'))* '"';
METAMODEL
        :       '[' (~(']'))* ']';
```

Annexe H Grammaire EBNF du langage FRL.

Les grammaires des langages IRL et FRL partagent un grand nombre de règles. Nous ne donnons ici que les règles spécifiques à FRL, c'est-à-dire celle n'existant pas dans la grammaire IRL ou celles comportant des différences. Les autres règles sont disponibles en Annexe G.

```
start           : fusionRuleSet EOF ;

fusionRuleSet
      :         importDecl*
                'FS' '(' fusionSemanticModelDef ')' fusionRule*;

fusionSemanticModelDef
      :         IDENT ':' type;

fusionRule
      :         equivalenceRule
      |         normalizationRule;

normalizationRule
      :         'NR' IDENT '(' nrInputParameters ')' '{' normalizationRuleBody '}';

nrInputParameters
      : inputParameterDefPart (',' inputParameterDefPart)*;

normalizationRuleBody
      :         '?' ':' block '!' ':' block;

equivalenceRule
      :         'ER' IDENT '(' erInputParameters ')' ':' xClass equivalenceRuleBody?;

erInputParameters
      :         simpleParameterDef ',' simpleParameterDef
      |         IDENT ',' IDENT ':' xClass;

equivalenceRuleBody
      :         '{' variableDeclaration* equivalenceRuleGuard
                (equivalenceRuleSetter | variableDeclaration | block)* '}';

equivalenceRuleGuard
      :         '?' ':' v1 = expression {e = b.createERGuard(v1);} ';';

equivalenceRuleSetter
      :         IDENT ':=' expression ';'
      |         IDENT ':=' block;

compositeElement1
      : primitiveFunction
      | mathOps
      | metamodelReference
      | compositeElement;

compositeElement
      :         'equRange'
      |         'asSet'
      |         'flatten'
      |         functionCall
      |         selectOperation
```

```
|       collectOperation
|       forallOperation
|       existOperation
|       foreachOperation
|       elementReference;
```

Annexe I Spécification IRL de la fonction sémantique du langage RDL définie sur RM (f : RDL → RM).

La fonction sémantique f : RDL → RM est plus complexe que les fonctions sémantiques implémentées pour le diagramme d'activité ou le diagramme de classe. En effet, le RDL est un langage à base de patrons de spécifications et c'est un langage naturel contraint. Le métamodèle fourni à la fin du code correspond à l'aspect Kermeta ajouté au métamodèle RDM et dont les méthodes sont utilisés dans le code de f.

1. spécification de f : RDL → RM.

```
import java.lang.String;
import java.lang.Boolean;
import java.lang.Integer;
import java.util.ArrayList;
import java.lang.Object;
import java.util.List;
import java.lang.System;
import javaLibs.Utils;
import nativeUtils.Tracer;
import analysis.Resolver;
import org.eclipse.emf.ecore.EObject;
import debug.ILDebuger;

import RDM : [[http://rdm]];
import RM : [[http://rm]];

IS(m1 : RDM) : (m2 : RM)

query getCptsFromSubjectAttribute(a : Adjective) : List<<Complement>> {
        var o : Complement init getComplementFor(a);
    return a.clause().complements.select(c | c != o);
}
pattern subject(e : EntityRef) = e.subject~ != void;
pattern complement(e : EntityRef) = e.expr~ != void;
pattern property(a : Adjective) = a.cpt() != void and and a.clause().verb.isStateVerb()
        and a.clause().subject instanceof NominalGroup
        => not designateProperty(a.clause().subject#NominalGroup);
pattern designateProperty (e : NominalGroup) = e.noun.nounCpt != void and ((subject(e)
        => e.clause().subjectAttributeCpt() != void) or (complement(e) => e.clause().verb.isSetVerb()))
        and Dico.isProperty(e);
pattern designateContainment(ng : NominalGroup) = ng.noun.nounCpt != void and not
designateProperty(ng)
        and Dico.isThing(ng);
pattern designateRole(ng : NominalGroup) = ng.noun.nounCpt != void and not designateProperty(ng)
        and not designateContainment(ng);

//************ STATIC ************
//------------ ENTITY -------------
IR conceptOrActor(e : EntityRef)
? : e instanceof NominalGroup
=> (not designateProperty(e#NominalGroup) and not designateRole(e#NominalGroup));
{
    var entity : Entity;
    if (subject(e) and e.subject~.verb.isActionVerb()) //actor
            entity := Actor.new;
    else
            entity := Concept.new;
        if (e instanceof NominalGroup)
```

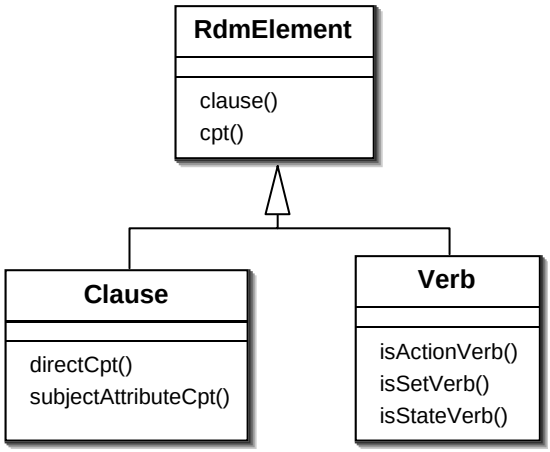
```
                c.name := e#NominalGroup#.noun.name;
        else { //instanceof Pronoun
                var l : List<<NominalGroup>> init Resolver.getReferencedNominalGroups(p);
                if (l.size() >= 1)
                entity.name  := l.get(0).noun.name;
        }
}
//------------ ACTION PARAMETERS AND VARIABLES-------------
pattern variableOrParameter(e : EntityRef) = e instanceof NominalGroup
            => not designateProperty(e#NominalGroup);
IR actionParameter(e : EntityRef)
? : variableOrParameter(e) and e.clause().verb.isActionVerb() and (subject(e) or complement(e));
{
            var p : Parameter init Parameter.new;
            if (subject(e))
                p.role := "actor";
        else { //complement
                var prep : Preposition init e.expr~.preposition;
                if (prep != void)
                        p.role := prep.name;
                else
                        p.role := "direct";
        }
        Action@(v).parameters.add(p);
        p.type := Entity@(e);
}
IR variable(e : EntityRef)
? : variableOrParameter(e) and (e.clause().verb.isActionVerb() => (not subject(e) and not complement(e)));
{
        var v : Variable init Variable.new;
        p.type := Entity@(e);
}
IR variableEquivalence(e : EntityRef)
? : variableOrParameter(e);
{
        var l : List<<NominalGroup>> init Resolver.getReferencedNominalGroups(e);
        if (l.size() >= 1)
                Tracer.declareAsEquivalent(Variable@(e), Variable@(l.get(0)), "variable");
}
//------------ ACTION -------------
IR action(v : Verb) : (a : Action)
? : actionVerb(v);
{
        a.name := v.name;
}
//------------ ROLE -------------
IR role(ng : NominalGroup)
? : designateRole(ng);
{
        var r : Role init Role.new;
        r.name := ng.noun.name;
        Entity@(ng.clause().subject).possibleRoles.add(r);
        r.type := m2.Boolean.new;
        var ref : Reference init Reference.new;
        r.references.add(ref);
        r.actor := ref;
        ref.entity := Entity@(ng);
        ref := Reference.new;
        r.references.add(ref);
            ref.entity := Entity@(ng.noun.nounCpt);
```

```
}
//------------ RELATION -------------
pattern relation(a : Adjective) = property(a) and getCptsFromSubjectAttribute(a).size() > 0;
IR relation(a : Adjective)
? : relation(a);
{
	var r : Relation init Relation.new;
	r.name := a.name;
	Entity@(a.clause()..subject).properties.add(r);
	var ref : Reference init Reference.new;
	r.references.add(ref);
	ref.entity := Entity@(a.clause().subject);
	getCptsFromSubjectAttribute(a).forall(c : Complement) {
		ref := Reference.new ;
		r.references.add(ref);
		ref.entity := Entity@(c.expr);
		ref.name := c.preposition.name;
	}
	var b : Boolean init Boolean.new;
	r.type := b;
}
//------------ CONTAINMENT -------------
IR containment(ng : NominalGroup)
? : designateContainment(ng);
{
	var c : Containment init Containment.new;
	    var containerEntity : Entity init Entity@(ng.noun.nounCpt);
	containerEntity.properties.add(c);
	var ref : Reference init Reference.new;
	c.references.add(ref);
	ref.entity := Entity@(ng);
	c.contained := ref;
	ref := Reference.new;
	    c.references.add(ref);
	ref.entity := containerEntity;
	c.container := ref;
}
//------------ ATTRIBUTE -------------
pattern attribute(a : Adjective) = property(a) and getCptsFromSubjectAttribute(a).size() == 0;
IR attribute(a : Adjective)
? : attribute(a);
{
	var att : Attribute init Attribute.new;
	att.name := a.name;
	Entity@(a.clause().subject).properties.add(att);
	var b : Boolean init Boolean.new;
	att.type := b;
}
IR attribute(ng : NominalGroup)
? : designateProperty(ng);
{
	var att : Attribute init Attribute.new;
	var n : Noun init ng.noun;
	att.name := n.name;
	Entity@(n.nounCpt).properties.add(att);
	var e : Enumeration init Enumeration.new;
	att.type := e;
	var l : Literal init Literal.new;
	l.name := ng.clause().directCpt.expr#Adjective#.name;
	e.literals.add(l);
```

```
}
//************ DYNAMIC ************
//------------ ARGUMENT -------------
IR argumentsForRelation(a : Adjective)
? : relation(a);
{
	var refs : List<Reference> init Reference@[relation](a);
	createArgument(Variable@(a.clause().subject), refs[0]);
	refs.remove(0);
		var l : List<Complement> init getCptsFromSubjectAttribute(a) ;
		var i : int init 0;
		refs.forall(r) {
			createArgument(Variable@(l[i]), r);
			inc(i);
	}
}
IR argumentsForAttribute(a : Adjective)
? : attribute(a);
{
	createArgument(Variable@(a.clause().subject), void);
}
IR argumentsForAttribute(ng : NominalGroup)
? : designateProperty(ng);
{
	createArgument(Variable@(ng.noun.nounCpt), void);
}
IR argumentsForRole (ng : NominalGroup)
? : designateRole(ng);
{
	var refs : List<Reference> init Reference@[role](ng);
	createArgument(Variable@(ng), refs[0]);
		createArgument(Variable@( ng.noun.nounCpt), refs[1]);
}
IR argumentsForContainment(ng : NominalGroup)
? : designateContainment(ng);
{
	var refs : List<Reference> init Reference@[containment](ng);
	createArgument(Variable@(ng), refs[0]);
		createArgument(Variable@( ng.noun.nounCpt), refs[1]);
}
function createArgument(v : Variable, r : Reference)
{
	var arg : Argument init Argument.new;
	arg.variable := v;
	arg.role := r;
}
//------------ PREDICATE -------------
IR booleanPredicate(a : Adjective)
? : relation(a) or attribute(a);
{
	var b : BooleanPredicate init BooleanPredicate.new;
	setPredicate(b, Property@(a), Property@(a));
}
IR enumPredicate(ng : NominalGroup)
? : designateProperty(ng);
{
	var e : EnumPredicate init EnumPredicate.new;
	setPredicate(e, Attribute@(ng), Argument@(ng));
	e.literal := Literal@(ng);
}
```

```
function setPredicate(pre : Predicate, pro : Property, args : Object)
{
	pre.property := pro;
	if (args instanceof List)
		pre.arguments.addAll(args#List);
	else
		pre.arguments.add (args#Argument);
}
IR rolePredicate (ng : NominalGroup)
? : designateRole(ng);
{
	var r : RolePredicate init RolePredicate.new;
	setPredicate(r, Role@(ng), Argument@(ng));
}
//------------ LOGICAL EXPR -------------
IR negation(n : rdm:Not)
{
	var not_ : rm:Not init rm:Not.new;
	not_.op := Expression@(n.operand);
}
```

2. Code Kermeta ajouté au métamodèle RDM.



RdmElement.clause() : retourne la clause contenante.
Verb.isStateVerb() : retourne vrai si le verb est un verbe d'état.
Verb.isActionVerb() : retourne vrai si le verb est un verbe d'action.
Verb.isSetVerb() : retourne vrai si le verb est un verbe de paramétrage.
RdmElement.cpt() : retourne le complément contenant.
Clause.subjectAttributeCpt() : retourne l'attribut du sujet de la clause.
Clause.directCpt() : retourne le complément d'objet direct.

Annexe J Spécification IDL de règles de détection d'incohérences pour le métamodèle RM.

IDR enumWithOneValue (e : Enumeration) : Lack
? : card(e.literals) == 1;
"the value of the property {e.type~[0].name} can be only {e.literals[0].name}."

IDR collisionBetweenPropertyAndLiteral (l : Literal, a : Attribute) : Lack, SCollision, Optional
var p : Property init l.literals~.type~[0];
? : l.name == a.name and Algo.linkedBySuperType(p.properties~[0], a.properties~[0]);
"It seems that the property {a.name} designate the same domain element as the literal {l.name} of the property {p.name}."

IDR noReferenceToLiteral (a : Attribute, l : Literal) : Lack
? : a.type#Enumeration#.literals.contains(l)
and not a.property~.exists(e : EnumPredicate | e.literal == l);
"The literal {l.name} is never referenced by a contract of an action."

IDR severalTypeForParameter (p : Parameter) : Contradiction, DCollision
? : card(r.type) > 1;
"One parameter of the action {p.parameters~.name} has more than one type.
It could be a conceptual disagrement between stakeholders. Perhaps these types {r.entity} are linked by a super-type relation. In that case, add this information in requirements. It could also be a terminology clash. "

IDR cardinalityConflict (c : Cardinality) : Contradiction
? : card(c.value) > 1;
"The cardinality bewteen {c.from.type} and {c.to.type} has several values : {c.value}."

IDR severalEntityForReference (r : Reference) : Contradiction, DCollision
? : card(r.entity) > 1 and r.references~[0] instanceof Relation;
"One reference of the relation {r.references~[0].name} has more than one type.
It could be a conceptual disagrement between stakeholders. Perhaps these types {r.entity} are linked by a super-type relation. In that case, add this information in requirements. It could also be a terminology clash."

IDR actionOnlyDefinedInInstanceLevel(a : Action) : Lack, Processus
? : entryModels(a).forall(m) {not m.notation.isAtProductLevel()};
"The action {a.name} is only defined at the instance level."

Annexe K Spécification FRL de la sémantique compositionnelle du métamodèle RM (φ_{RM}).

```
import helper.Algo;
import helper.IHM;
import java.lang.String;
import RM : [[http://rm]];
import java.util.ArrayList;
import java.lang.System;
import debug.FLDebuger;

FS(model : RM)

//Initialisation
{ entityDesignateRole }
//Fusion
{ boolean, actorAndConcept, actor, concept, actorAndConceptVoid, conceptVoid, actorVoid,
attribute, relation, attributeAndRelation, action, enumeration,  literal, reference, role,
parameter, variable, superTypeAndEntityRelations, superTypeAndTypeRelations,
entityWithNoName }*
//Completion
{ variable, deleteExists, addExists }
{ multiPreCond, multiPostCond }
{ normalizeNot, normalizeBinaryExpr }*
{ preCondVoid, postCondVoid}

var treatedParametersInRole : List<Parameter> init ArrayList<Parameter>.new;
NR entityDesignateRole(role : Role, e : entity)
var p : Parameter init e.type~[0]#Parameter;
var a : Action init p.parameters~[0];
? : role.name == e.name;
! : {
	e.name := void;
	p.type.add(role.references.select(r | r.name == “actor”)[0].entity);
	if (not treatedParametersInRole.contains(p)) {
		var v : Variable init Variable.new;
		v.role := role.references.select(r | r.name == “of”)[0].entity;
		var exp : Expression init a.preCond;
		if (expr == void)
			a.preCond := createExists(v, createRolePredicate(role, e.type~[0], v));
		else
			a.preCond := createExists(v,createImplies(createRolePredicate(role, e.type~[0], v), expr));
		exp := a.postCond;
		if (expr != void)
			a.preCond := createImplies(createRolePredicate(role, e.type~[0], v), expr);
		treatedParametersInRole.add(p);
	}
}
function createExists(quantified : Variable, op : Expression) : Exists
{
	var ex : Exists init Exists.new;
	ex.quantified := quantified;
	ex.op := op;
	return ex;
}
function createRolePredicate(r : Role, p : Parameter, v : Variable) : RolePredicate {
	var rp : RolePredicate init RolePredicate.new;
	rp.property := r;
	var arg : Argument init Argument.new;
```

```
        rp.arguments.add(arg);
        arg.role := r.references.select(r | r.name == “actor”)[0];
        arg.variable := p;
        arg := Argument init Argument.new;
        rp.arguments.add(arg);
        arg.role := r.references.select(r | r.name == “of”)[0];
        arg.variable := v
        return rp;
}
function createImplies(lOp : Expression, rOp : Expression) : Implies
{
        var im : Implies init Implies.new;
        im.lOp := lOp;
        im.rOp := rOp;
        return im;
}
ER boolean (b1 : Boolean, b2 : Boolean) : Boolean
? : e1.type~ == e2.type~;

ER actorAndConcept (a : Actor, c : Concept) : Actor
? : a.name != void and a.name == c.name;

ER actor (a1 : Actor, a2 : Actor) : Actor
? : a1.name != void and a1.name == a2.name;

ER concept (c1 : Concept, c2 : Concept) : Concept
? : c1.name != void and c1.name == c2.name;

ER actorAndConceptVoid (a : Actor, c : Concept) : Actor
? : a.name == void and
        (inter(a.entity~, c.entity~).size() > 0 or inter(a.type~, c.type~).size() > 0);

ER actorVoid (a1 : Actor, a2 : Actor) : Actor
? : a1.name == void and
        (inter(a1.entity~, a2.entity~).size() > 0 or inter(a1.type~, a2.type~).size() > 0);

ER conceptVoid (c1 : Concept, c2 : Concept) : Concept
? : c1.name == void and
        (inter(c1.entity~, c2.entity~).size() > 0 or inter(c1.type~, c2.type~).size() > 0);

ER attribute(a1 : Attribute, a2 : Attribute) : Attribute
var e1 : Entity init a1.properties~[0];
var e2 : Entity init a2.properties~[0];
? : a1.name == a2.name
        and (e1 == e2 or Algo.linkedBySuperType(a1.properties~[0],  a2.properties~[0]));
properties~ := Algo.fcs(equRange.collect(a | a.properties~[0]).flatten);

ER relation(r1 : Relation, r2 : Relation) : Relation
var e1 : Entity init r1.properties~[0];
var e2 : Entity init r2.properties~[0];
? : r1.name == r2.name
        and (e1 == e2 or Algo.linkedBySuperType(e1,  e2));
properties~ := Algo.fcs(equRange.collect(a | a.properties~[0]).flatten);

ER attributeAndRelation(r : Relation, a : Attribute) : Relation
var e1 : Entity init r.properties~[0];
var e2 : Entity init a.properties~[0];
? : r.name == a.name
        and (e1 == e2 or Algo.linkedBySuperType(e1,  e2));
properties~ := Algo.fcs(equRange.collect(a | a.properties~[0]).flatten);
```

```
ER cardinality(c1 : Cardinality, c2 : Cardinality) : Cardinality
	? : c1.from == c2.from and c1.to == c2.to;

ER action (uc1 : Action, uc2 : Action) : Action
? : uc1.name == uc2.name;

ER enumeration (e1 : Enumeration, e2 : Enumeration) : Enumeration
	? : e1.type~ == e2.type~;

ER literal (e1 : Literal, e2 : Literal) : Literal
	? : e1.literal~ == e2.literal~
		and e1.name == e2.name;

ER reference(r1 : Reference, r2 : Reference) : Reference
	? : e1.entity~[0] == e2.entity~[0] and e1.name == e2.name;

ER role(r1 : Role, r2 : Role) : Role
	? : r1.name == r2.name;

ER parameter(p1 : Parameter, p2 : Parameter) : Parameter
	? : p1.parameters~ == p2.parameters~
				and p1.role != void and p1.role == p2.role;

NR superTypeAndEntityRelations(e1 : Entity, e2 : Entity, r : Reference)
? : r.entity!.contains(e1) and r.entity!.contains(e2) and Algo.linkedBySuperType(e1, e2);
! : {
	if (Algo.isSuperType(e1,e2))
		r.entity!.remove(e1);
	else
		 r.entity!.remove(e2);
}
NR superTypeAndTypeRelations(e1 : Entity, e2 : Entity, v : Variable)
? : v.type!.contains(e1) and v.type!.contains(e2) and Algo.linkedBySuperType(e1, e2);
! : {
	if (Algo.isSuperType(e1,e2))
		v.type!.remove(e1);
	else
		 v.type!.remove(e2);
}

NR preCondVoid(a : Action)
? : a.preCond == void;
! : {a.postCond := True.new;}

NR postCondVoid(a : Action)
? : a.postCond == void;
! : {a.postCond := True.new;}

NR multiPreCond(a : Action)
? : card(a.preCond) > 1;
! : {a.postCond := IHM.defineCombination(a.preCond!);}

NR multiPostCond(a : Action)
? : card(a.postCond) > 1;
! : {a.postCond := IHM.defineCombination(a.postCond!);}

ER variable (v1 : Variable, v2 : Variable) : Variable
var e1 : Entity init v1.type;
var e2 : Entity init v2.type;
```

```
? : e1 == e2 or linkedBySuperType(e1, e2) and IHM.areEquivalent(v1, v2);
```

Annexe L Modèles intermédiaires produits durant la composition des modèles de la Figure 43.

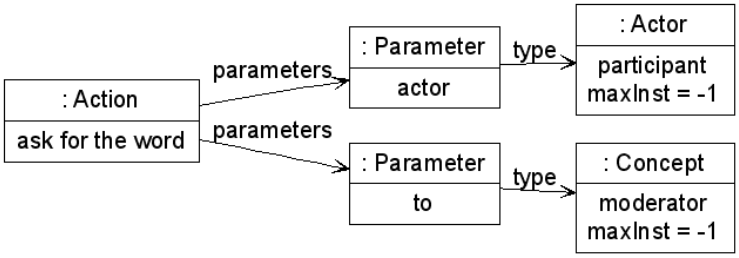


Figure 91 – modèle intermédiaire résultant de l'interprétation du modèle (a).

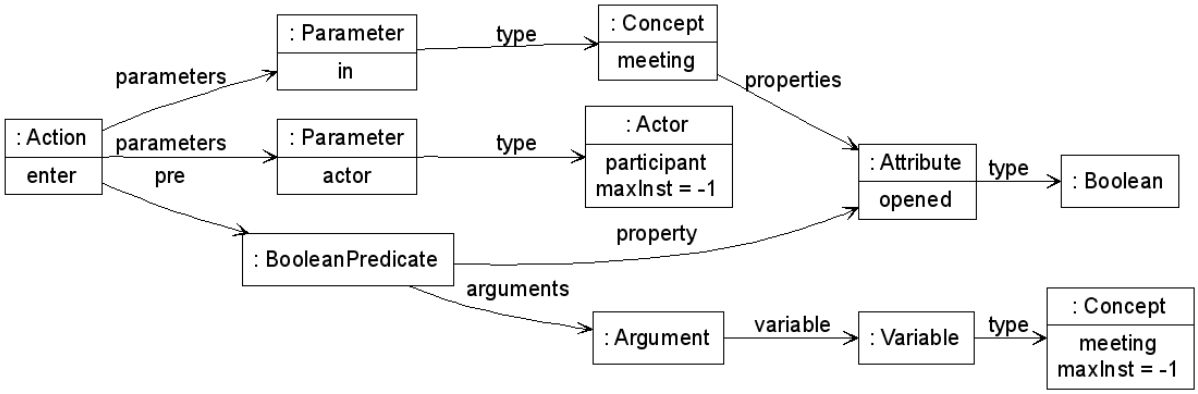


Figure 92 – modèle intermédiaire résultant de l'interprétation du modèle (g).

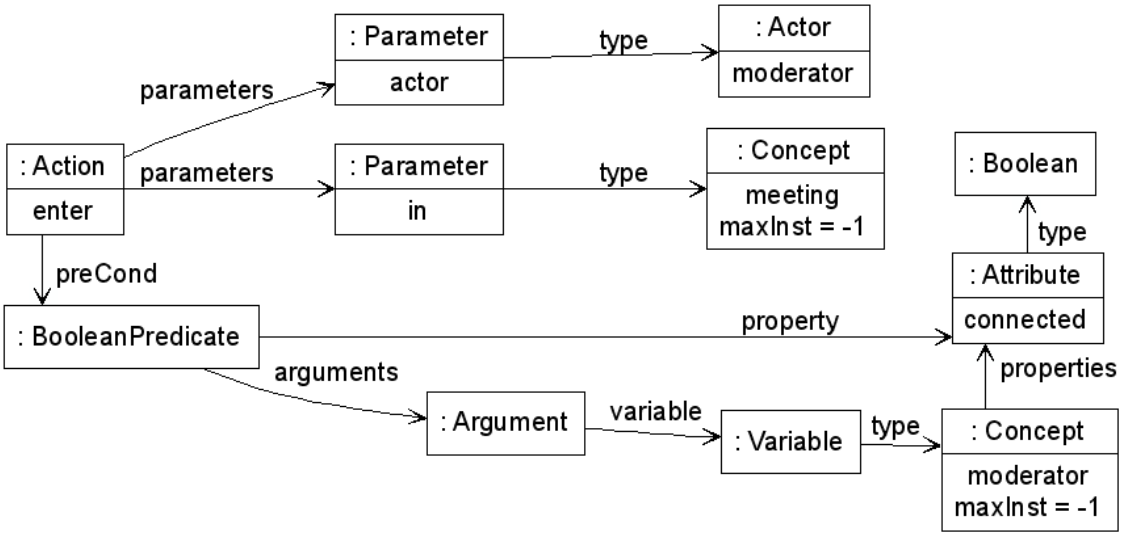


Figure 93 – modèle intermédiaire résultant de l'interprétation du modèle (f).

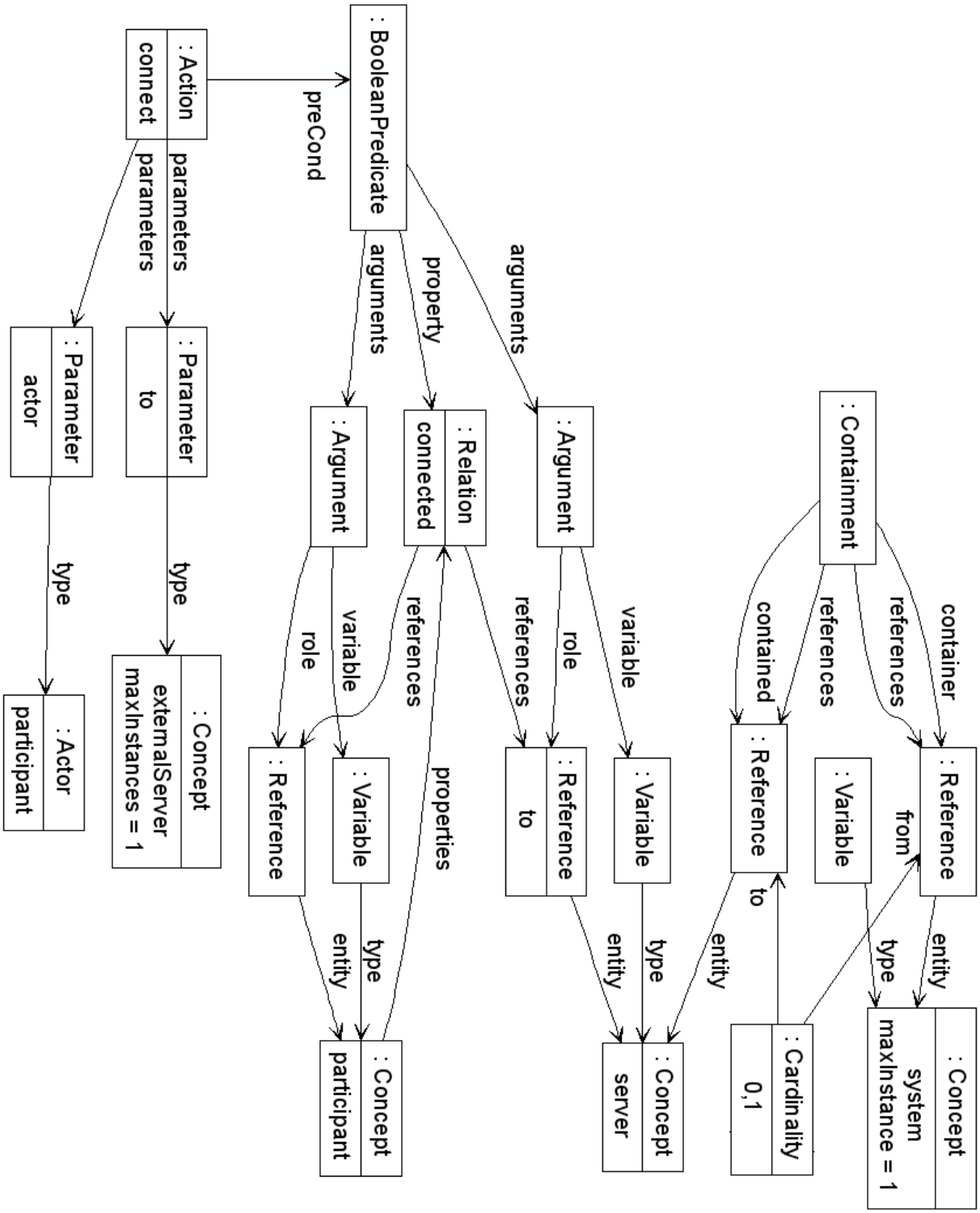


Figure 94 – modèle intermédiaire résultant de l'interprétation du modèle (h).

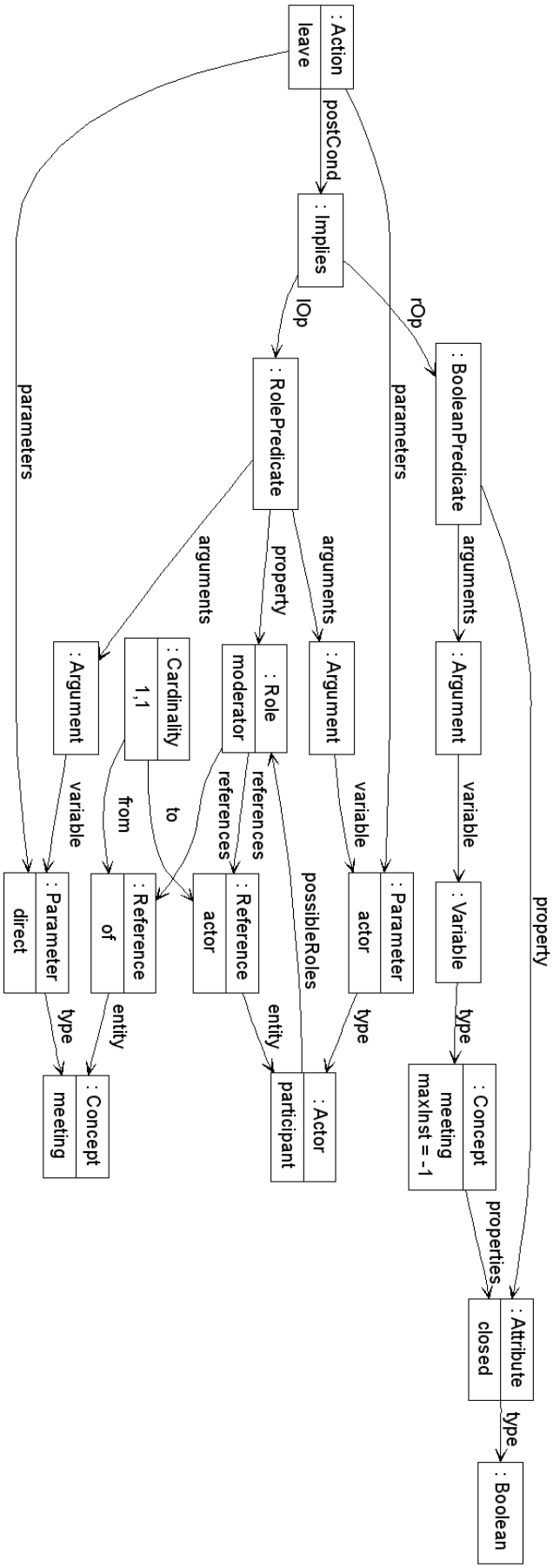


Figure 95 – modèle intermédiaire résultant de l'interprétation du modèle (b).

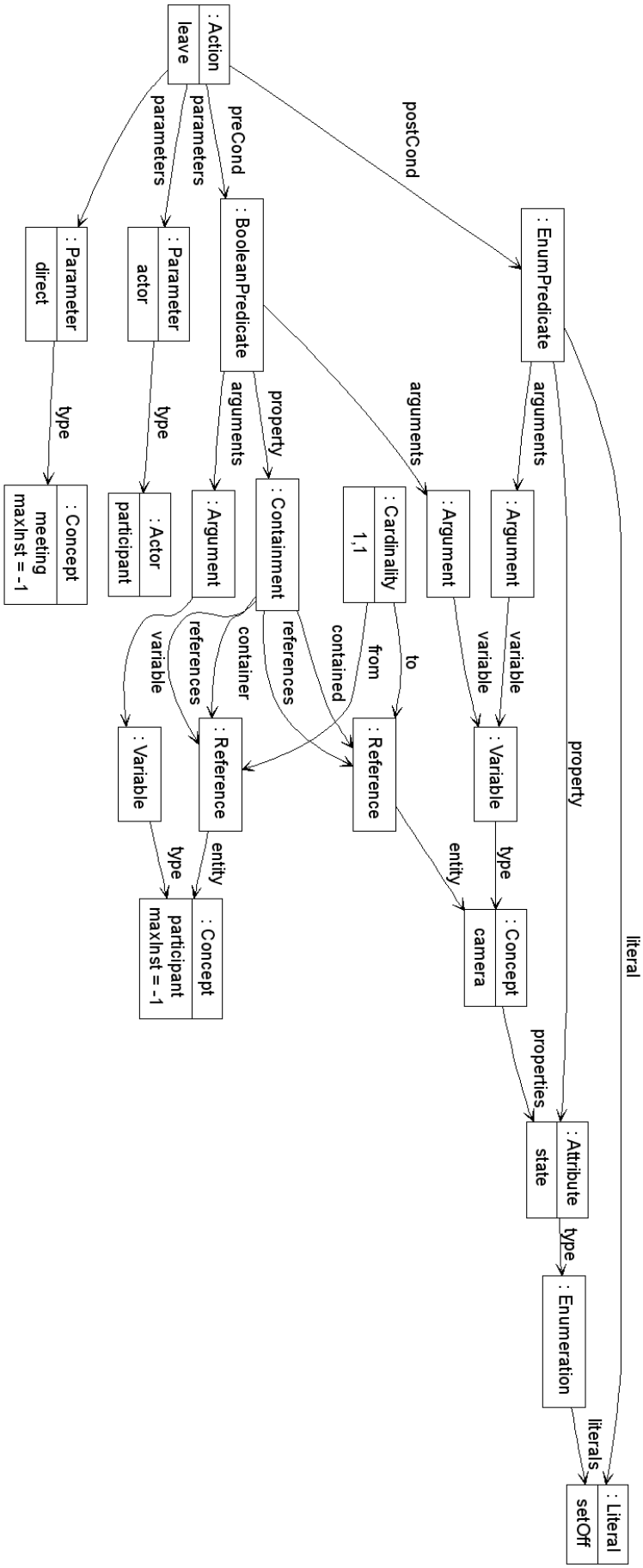


Figure 96 – modèle intermédiaire résultant de l'interprétation du modèle (c).

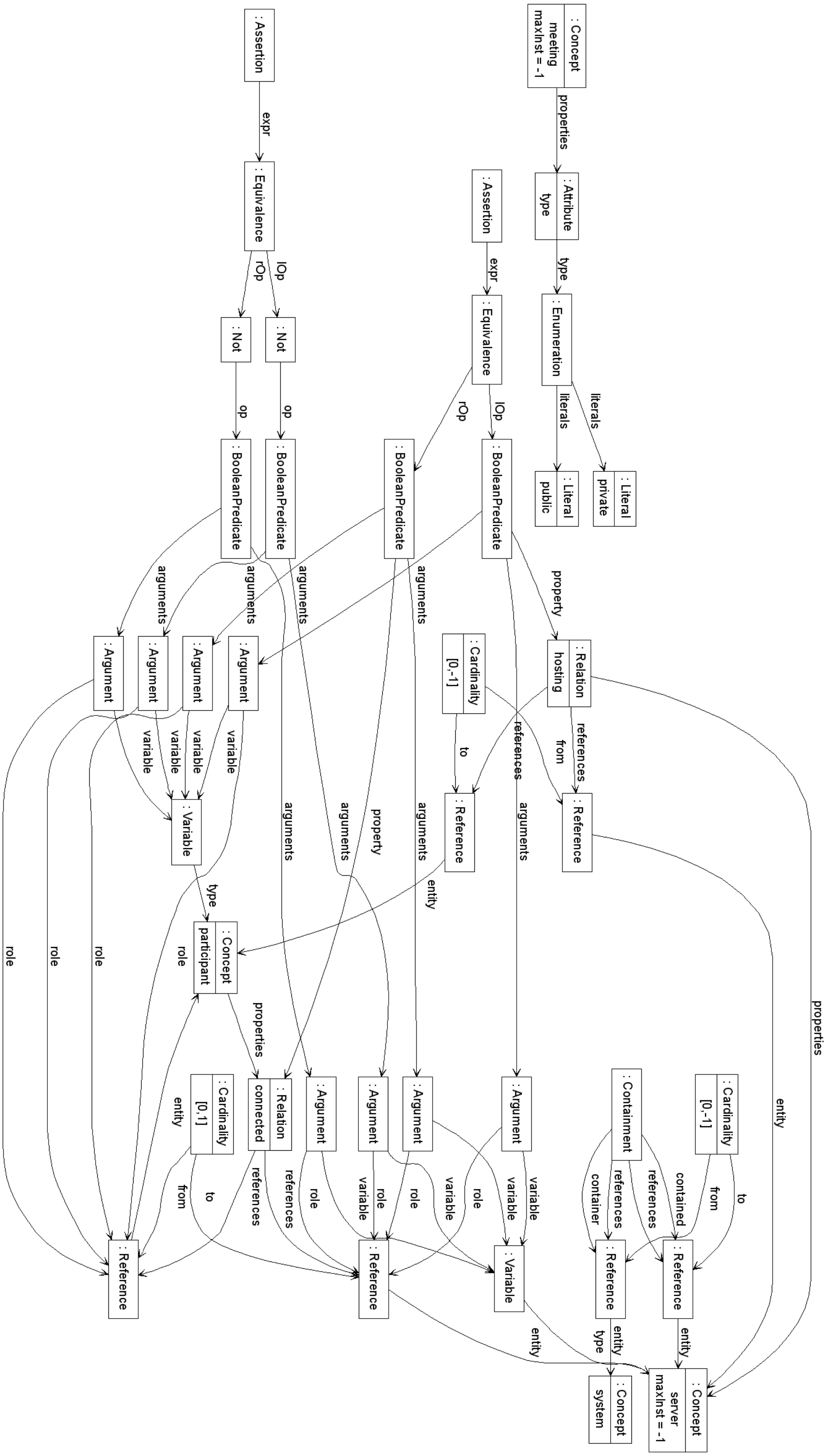


Figure 97 – modèle intermédiaire résultant de l'interprétation du modèle (e).

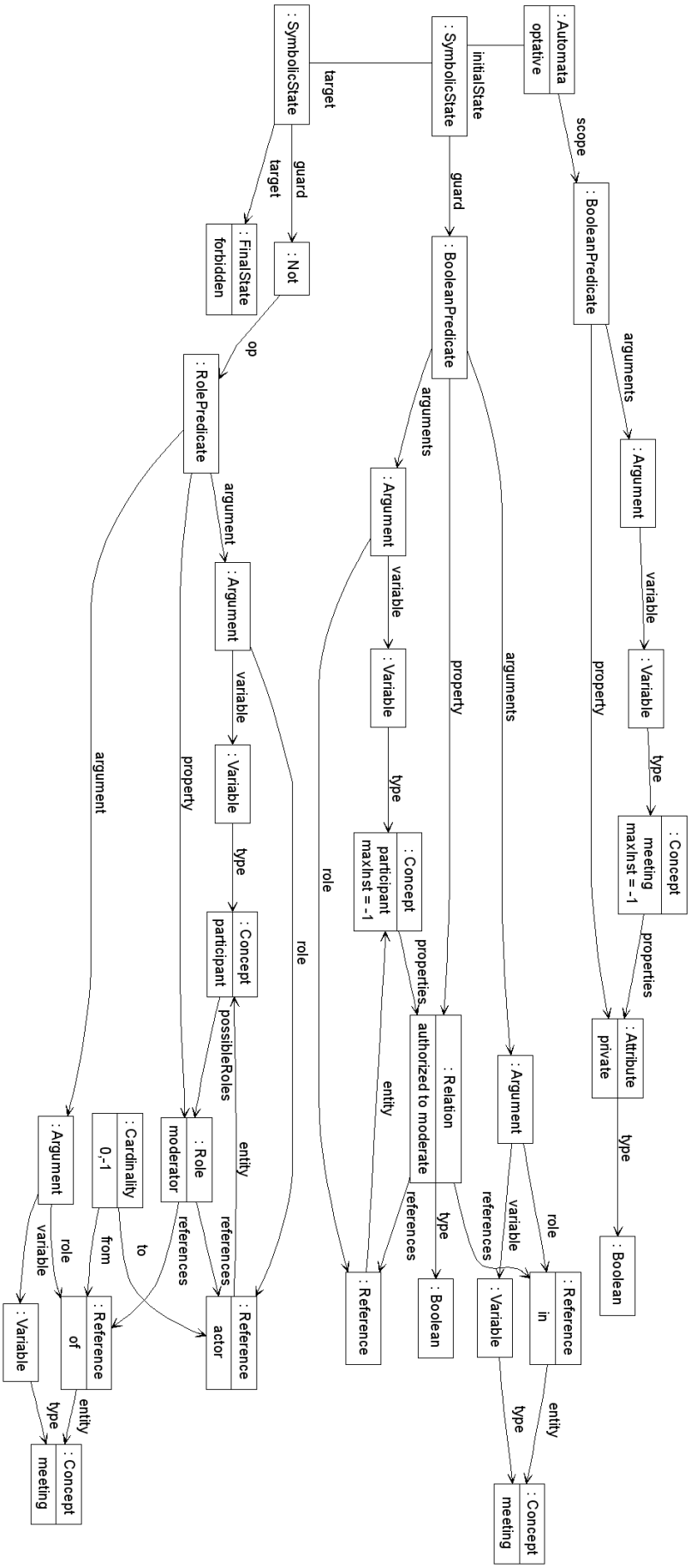


Figure 98 – modèle intermédiaire résultant de l'interprétation du modèle (i)

Annexe M Spécification FRL de la sémantique compositionnelle du méta-métamodèle MOF (ψ_{MOF})

```
import java.util.ArrayList;
import Ecore : [http://www.eclipse.org/emf/2002/Ecore];
import javaLibs.Algo;
import java.lang.System;

FS(model : Ecore)

ER metaClass (c1, c2 : EClass) : EClass
{
	? : c1.name == c2.name;
	abstract := {return equRange.forall(c : EClass | c.abstract);}
	interface := {
	  if (equRange.collect(c : EClass | c.interface).contains(true))
	  {
	    if (card(equRange.collect(c:EClass | c.eAllStructuralFeatures).flatten) == 0)
	      return true;
	  }
	  return false;
	}
}
ER relation (r1, r2 : EReference) : EReference
{
	? : r1.name == r2.name and r1.eContainingClass != void and r2.eContainingClass != void
				and Algo.fcs(union(r1.eContainingClass.asSet, r2.eContainingClass.asSet)) != void;
	eContainingClass := Algo.fcs(equRange.collect(r : EReference | r.eContainingClass))#EClass;
	eType := Algo.fcs(equRange.collect(r : EReference | r.eType))#EClass;
	eGenericType := {
		var c : EClass init Algo.fcs(equRange.collect(r : EReference | r.eType))#EClass;
		var gt : EGenericType init EGenericType.new;
		gt.eClassifier := c;
		return gt;
	}
}
ER attribute (a1, a2 : EAttribute) : EAttribute
{
	? : a1.name == a2.name
				and Algo.fcs(union(a1.eContainingClass.asSet, a2.eContainingClass.asSet)) != void;
	eContainingClass := Algo.fcs(equRange.collect(a : EAttribute | a.eContainingClass))#EClass;
}
ER package (p1, p2 : EPackage) : EPackage
{
	? : p1.name == p2.name and p1.eSuperPackage == p2.eSuperPackage;
	eFactoryInstance := {return equRange[0]#EPackage#.eFactoryInstance;}
}
ER dataType (dt1, dt2 : EDataType) : EDataType
{
	? : dt1#EObject#.eClass().getName() == "EDataType" and dt2#EObject#.eClass().getName() ==
"EDataType"
		and dt1.instanceClassName == dt2.instanceClassName
		and dt1.ePackage == dt2.ePackage;
	name := {return equRange.collect(dt : EDataType | dt.name)[0];}
	defaultValue := {return equRange.collect(dt : EDataType | dt.defaultValue)[0];}
}
ER genericType1 (gt1, gt2 : EGenericType) : EGenericType
```

```
{
	? : gt1.eClassifier == gt2.eClassifier
		and gt1.eGenericSuperTypes~ != void and gt1.eGenericSuperTypes~ ==
gt2.eGenericSuperTypes~;
}
ER genericType2 (gt1, gt2 : EGenericType) : EGenericType
{
	? : gt1.eClassifier == gt2.eClassifier
		and gt1.eGenericType~ != void and gt1.eGenericType~ == gt2.eGenericType~;
}
NR genericSuperType (gt : EGenericType)
{
	? : {return gt.eGenericType~ == void and gt.eGenericSuperTypes~ == void;}
	! : {
				model.elicit(gt);
		}
}
```

Publications

Conférences nationales

1. Jean-Marie Mottu, Benoit Baudry, Yves Le Traon, and Erwan Brottier. **Génération automatique de test pour les transformations de modèles**. 1ère Journées sur l'Ingénierie Dirigée par les Modèles, Paris, June 2005.

Conférences internationales

2. Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry, Yves Le Traon. **Metamodel-based test generation for model transformations: an algorithm and a tool**. Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE'06), Raleigh, USA, 2006.

3. Erwan Brottier, Benoit Baudry, Yves Le Traon, David Touzet, and Bertrand Nicolas. **Producing a global requirement model from multiple requirement specifications**. Proceedings of the 11th International Conference of Entreprise Distributed Object Computing Conference (EDOC'07), Annapolis, USA, 2007.

4. Gilles Perrouin, Erwan Brottier, Benoit Baudry and Yves Le Traon. **Composing Models for Detecting Inconsistencies: A Requirements Engineering Perspective**. Proceedings of the 15th International Working Conference on Requirements Engineering: Fondation for Software Quality (RefsQ'09), Amsterdam, Netherlands, 2009.

Bibliographie

1. Schmidt, D.C., *Model-Driven Engineering.* IEEE Computer, 2006. **39**(2): p. 25 - 31.
2. France, R. and B. Rumpe, *Model-driven Development of Complex Software: A Research Roadmap*, in *FOSE*. 2007.
3. Kent, S. *Model Driven Engineering*. in *IFM'02 (Integrated Formal Methods)*. 2002. Turku, Finland.
4. Nuseibeh, B. and S. Easterbrook. *Requirements Engineering: A Roadmap*. in *The Future of Software Engineering (joint event of ICSE'00)*. 2000. Limerick, Ireland.
5. Lamsweerde, A.v. *Requirements Engineering in the Year 00: A Research Perspective*. in *International Conference on Software Engineering*. 2000.
6. Zave, P., *Classification of research efforts in requirements engineering.* ACM Computing Surveys, 1997. **29**(4): p. 315-321.
7. Rosenblum, D., *Formal methods and testing: why the state-of-the art is not the state-of-the practice.* ACM SIGSOFT Software Engineering Notes, 1996. **21**(4): p. 64.
8. Dwyer, M.B., G.S. Avrunin, and J.C. Corbett. *Patterns in property specifications for finite-state verification*. in *International Conference on Software Engineering*. 1999.
9. Heitmeyer, C. and N.R.L.W. DC, *On the need for practical formal methods*. 1998, Springer.
10. Konrad, S. and B.H.C. Cheng. *Facilitating the Construction of Specification Pattern-based Properties*. in *RE'05 (Requirements Engineering)*. 2005. Paris, France.
11. Lamsweerde, A.v. *Formal Specification: a Roadmap*. in *Proceedings of the conference on The future of Software Engineering*. 2000.
12. Brooks, F.P., *No Silver Bullet: Essence and Accidents of Sotfware Engineering.* IEEE Computer, 1987. **20**(4): p. 10-19.
13. Consel, C. and R. Marlet, *Architecturing software using a methodology for language development.* Proceedings of the 10 thInternational Symposium on Programming Language Implementation and Logic Programming: p. 170-194.
14. van Deursen, A., P. Klint, and J. Visser, *Domain-specific languages: an annotated bibliography.* ACM SIGPLAN Notices, 2000. **35**(6): p. 26-36.
15. Czarnecki, K., *Overview of Generative Software Development.* LECTURE NOTES IN COMPUTER SCIENCE, 2005. **3566**: p. 326.
16. Finkelstein, A., et al., *Viewpoints: A Framework for Integrating Multiple Perspectives in System Development.* International Journal of Software Engineering and Knowledge Engineering, 1992. **2**(1): p. 31-58.
17. Yu, E., *Agent-Oriented Modelling: Software Versus the World.* Agent-Oriented Software Engineering AOSE-2001 Workshop Proceedings. LNCS. **2222**: p. 206–225.
18. Parnas, D.L., *On the criteria to be used in decomposing systems into modules.* Communications of the ACM, 1972. **15**(12): p. 1053-1058.
19. Jackson, M., *Problem frames: analyzing and structuring software development problems*. 2000: Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.
20. E. Gamma, R.H., R. Johnson, J. Vlissides, *Design Patterns: Elements of ReusableObject-Oriented Software*. 1994: Addison-Wesley.
21. Chitchyan, R., et al., *Survey of Aspect-Oriented Analysis and Design Approaches.* AOSD-Europe-ULANC-9, AOSD-EUROPE network of excellence. May, 2005.
22. Heineman, G. and W. Councill, *Component-based software engineering: putting the pieces together*. 2001: Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.
23. Clements, P. and L. Northrop, *Software product lines*. 2002: Addison-Wesley Boston.
24. Meyer, B., *Applying Design by Contract.* IEEE Computer, 1992. **25**(10): p. 40 - 51.
25. Clarke, E.M., E.A. Emerson, and A.P. Sistla, *Automatic verification of finite-state concurrent systems using temporal logic specifications.* ACM Trans. Program. Lang. Syst., 1986. **8**(2): p. 244-263.

26. Ramesh, B. and M. Jarke, *Toward Reference Models for Requirements Traceability*. 2001.
27. Almeida, J.P., P. van Eck, and M.-E. Iacob, *Requirements traceability in model-driven development: Applying model and transformation conformance*. Information Systems Frontiers, 2007. **9**(4): p. 327-342.
28. Aizenbud-Reshef, N., et al., *Model Traceability*. IBM System Journal, 2006. **45**(3).
29. Galvao, I. and A. Goknil, *Survey of Traceability Approaches in Model-Driven Engineering*. Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)-Volume 00, 2007.
30. Sommerville, I., *Integrated Requirements Engineering: A Tutorial*. IEEE Software, 2005. **22**(1): p. 16-23.
31. Spanoudakis, G., et al., *Rule-based generation of requirements tracability relations*. Journal of Systems and Software, 2004. **72**(2): p. 105-127.
32. Spanoudakis, G. and A. Finkelstein. *Overlaps among requirements specifications*. in *Proceedings of the ICSE Workshop on Living with Inconsistency*. 1997.
33. Finkelstein, A.C.W., et al., *Inconsistency handling in multiperspective specifications*. Software Engineering, IEEE Transactions on, 1994. **20**(8): p. 569-578.
34. Easterbrook, S. and B. Nuseibeh, *Using Viewpoints for inconsistency management*. Software Engineering Journal, 1996. **11**(1): p. 31-43.
35. Nentwich, C., W. Emmerich, and A. Finkelstein. *Consistency management with repair actions*. in *International Conference on Software Engineering*. 2003.
36. Mens, T., G. Taentzer, and O. Runge, *Analysing refactoring dependencies using graph transformation*. Software and Systems Modeling, 2007. **6**(3): p. 269-285.
37. Nuseibeh, B., J. Kramer, and A. Finkelstein, *A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification*. Transaction on Software Engineering, 1994. **20**(10): p. 760-773.
38. Nentwich, C., W. Emmerich, and A. Finkelstein, *Flexible Consistency Checking*. ACM Transactions on Software Engineering and Methodology, 2001.
39. Nuseibeh, B., S. Easterbrook, and A. Russo, *Making inconsistency respectable in software development*. Journal of Systems and Software, 2001. **58**(2): p. 171-180.
40. Sabetzadeh, M. and S. Easterbrook, *An Algebraic Framework for Merging Incomplete and Inconsistent Views*, in *International Conference on Requirements Engineering*. 2005.
41. Zave, P. and M. Jackson, *Conjunction as Composition*. Transaction on Software Engineering and Methodology, 1993. **2**(4): p. 379-411.
42. Ainsworth, M., et al., *Viewpoint Specification and Z*. Information and Software Technology, 1994. **36**(1): p. 43-51.
43. Day, N.A. and J.J. Joyce, *A Framework for Multi-Notation Requirements Specification and Analysis*, in *IEEE International Conference on Software Engineering*. 2000.
44. Baniassad, E. and S. Clarke. *Theme: An Approach for Aspect-Oriented Analysis and Design*. in *ICSE'04 (Int. Conference in Software Engineering)*. 2004. Edinburgh, Scotland.
45. Moreira, A., A. Rashid, and J. Araújo. *Multi-Dimensional Separation of Concerns in Requirements Engineering*. in *RE'05 (Requirements Engineering)*. 2005. Paris, France.
46. Rashid, A., A. Moreira, and J. Araújo, *Modularisation and composition of aspectual requirements*. Proceedings of the 2nd international conference on Aspect-oriented software development, 2003: p. 11-20.
47. Konrad, S. and B.H.C. Cheng. *Automated Analysis of Natural Language Properties for UML Models*. in *MoDeVa 05 workshop in conjunction with MoDELS'05*. 2005. Montego Bay, Jamaica.
48. The_Standish_Group, *Chaos Standish Group Internal Report*. 1995, The Standish Group.
49. META_Group, *Research on Requirements Realization and Relevance*. 2003.
50. OMG. *UML specifications*. Available from: http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML.
51. Dwyer, M.B., G.S. Avrunin, and J.C. Corbett. *Property specification patterns for finite-state verification*. 1998: ACM New York, NY, USA.
52. Konrad, S. and B.H.C. Cheng. *Real-time Specification Patterns*. in *International Conference on Software Engineering*. 2005.

53. Smith, R.L., et al., *PROPEL: an approach supporting property elucidation.* Proceedings of the 24th International Conference on Software Engineering, 2002: p. 11-21.
54. Nebut, C., et al. *Automated Requirements-based Generation of Test Cases for Product Families*. in *ASE'03 (Automated Software Engineering)*. 2003.
55. Nebut, C., et al., *Automatic Test Generation: A Use Case Driven Approach.* IEEE Transactions on Software Engineering, 2006.
56. Nebut, C., et al. *A Requirement-based Approach to Test Product Families.* in *PFE'03 (Product Families Engineering)*. 2003. Sienna, Italy.
57. Nebut, C., et al. *Requirements by contracts allow automated system testing*. in *ISSRE'03 (Int. Symposium on Software Reliability Engineering)*. 2003. Denver, CO, USA.
58. Nebut, C., et al. *Reusable Test Requirements for UML-Modeled Product Lines*. in *REPL (workshop on Requirements Engineering for Product Lines)*. 2002. Essen, Germany.
59. Schoman, D.T.R.a.K.E., *Structured Analysis for Requirements definition.* IEEE Transaction on Software Engineering, 1977. **3**(1): p. 6-15.
60. Boehm, B.W., *A Spiral Model of Software Development and Enhancement.* Computer, 1988. **21**: p. 61-72.
61. Kotonya, G. and I. Sommerville, *Requirements engineering with viewpoints.* Software Engineering Journal, 1996.
62. Mylopoulos, J., *Information Modeling in the Time of the Revolution.* Information Systems, 1998. **23**(3-4): p. 127-155.
63. Engels, G., et al., *A Combined Reference Model-and View-Based Approach to System Specification.* Int. Journal of Software and Knowledge Engeneering, 1997. **7**(4): p. 457-477.
64. Sommerville, I., P. Sawyer, and S. Viller, *Viewpoints for requirements elicitation: a practical approach*, in *IEEE International Conference on Requirements Engineering*. 1998.
65. Sommerville, I. and P. Sawyer, *Viewpoints: principles, problems and practical approach to requirements engineering.* Annals of Software Engineering, 1997(3): p. 101-130.
66. Lamsweerde, A.v., E. Letier, and C. Ponsard. *Leaving Inconsistency*. in *Proceedings of the ICSE'97 workshop on "Living with Inconsistency"*. 1997.
67. Spanoudakis, G. and A. Zisman, *Inconsistency management in software engineering: Survey and open research issues.* Handbook of Software Engineering and Knowledge Engineering, 2001. **1**: p. 24-29.
68. Bohner, S.A., *Impact analysis in the software change process: A year 2000 perspective.* Proceedings International Conference on Software Maintenance ICSM. **96**: p. 42-51.
69. Gotel, O.C.Z. and C.W. Finkelstein, *An analysis of the requirements traceability problem.* Requirements Engineering, 1994., Proceedings of the First International Conference on, 1994: p. 94-101.
70. Wiegers, K.E., *Automating Requirements Management.* Software Development, 1999. **7**(7): p. 1-5.
71. Vessey, I. and S.A. Conger, *Requirements Specification: Learning Object, Process, and Data Methodologies.* Communications of the ACM, 1994. **37**(5).
72. Dijkstra, E.W., *A Discipline of Programming.* 1976.
73. Harel, D., et al., *STATEMATE: a working environment for the development of complexreactive systems.* Software Engineering, IEEE Transactions on, 1990. **16**(4): p. 403-414.
74. Göknil, A., N. Topaloglu, and K. van den Berg, *Operation Composition in Model Transformations with Complex Source Patterns*. 2008, Technical Report - University of Twente.
75. Tarr, P., et al. *N degrees of separation: multi-dimensional separation of concerns*. 1999: IEEE Computer Society Press Los Alamitos, CA, USA.
76. Kiczales, G., et al. *Aspect-Oriented Programming.* in *ECOOP'97 (European Conference for Object-Oriented Programming)*. 1997.
77. Tzilla, E., et al., *Discussing aspects of AOP.* Communications of the ACM, 2001. **44**(10): p. 33-38.
78. France, R., et al. *Providing support for Model Composition in Metamodels*. in *IEEE EDOC*. 2007.

79. Rashid, A., et al. *Early Aspects: A Model for Aspect-Oriented Requirements Engineering*. in *RE'02 (Requirements Engineering)*. 2002. Essen, Germany.
80. Sampaio, A., A. Rashid, and P. Rayson. *Early-AIM: An Approach for Identifying Aspects in Requirements*. 2005.
81. Yu, Y., J. Leite, and J. Mylopoulos, *From goals to aspects: discovering aspects from requirements goal models.* Proc. RE, 2004: p. 38-47.
82. Sousa, G., et al., *Separation of Crosscutting Concerns from Requirements to Design: Adapting an Use Case Driven Approach*, in *Early Aspects Workshop: Aspect-Oriented Requirements Engineering and Architecture Design in conjunction with AOSD'04*. 2004.
83. Hausmann, J.H., R. Heckel, and G. Taentzer. *Detection of conflicting functional requirements in a use case-driven approach: a static analysis technique based on graph transformation*. 2002: ACM New York, NY, USA.
84. DeMillo, R., R. Lipton, and F. Sayward, *Hints on Test Data Selection : Help For The Practicing Programmer*. IEEE Computer, 1978. **11**(4): p. 34 - 41.
85. Jackson, M. and P. Zave. *Domain Descriptions*. in *Second IEEE International Symposium on Requirements Engineering*. 1993: IEEE Computer Society Press.
86. Ellsberger, J., D. Hogrefe, and A. Sarma, *SDL: formal object-oriented language for communicating systems*. 1997: Prentice Hall.
87. E. Clarke, E.E., A. Sistla. *Automatic verification of finite-state concurrent systems using temporal logic specifications*. in *Transactions on Programming Languages and Systems*. 1986.
88. Pnueli, Z.M.a.A., *The temporal logic of reactive and concurrent systems*. 1992: Springer-Verlag New York.
89. Olender, K.M. and L.J. Osterweil, *Cecil: a sequencing constraint language for automatic static analysis generation.* IEEE Transactions on Software Engineering, 1990. **16**(3): p. 268-280.
90. Bozga, M., S. Graf, and L. Mounier. *IF-2.0: A validation environment for component-based real-time systems*. in *Conference on Computer Aided Verification*. 2002.
91. Alur, R., *Techniques for automatic verification of real-time systems*. 1991, Stanford University.
92. Emerson, E., et al., *Quantitative temporal reasoning.* Real-Time Systems, 1992. **4**(4): p. 331-352.
93. Borgida, A., S. Greenspan, and J. Mylopoulos, *Knowledge Representation as the Basis for Requirements Specifications.* IEEE Computer, 1985.
94. Greenspan, S.J., J. Mylopoulos, and A. Borgida. *Capturing more knowledge in the requirements specification*. in *International Conference on Software Engineering*. 1982.
95. J. Mylopoulos, A.B., M. Jarke and M. Koubarakis, *Telos: Representing knowledge about information systems.* ACM Transactions on Information Systems, 1990.
96. Du Bois, P., E. Dubois, and J.M. Zeippen, *On the Use of a Formal RE Language.* Proceedings ofthe Third IEEE International Symposium on Requirements Engineering, January, 1997. **1**: p. 28-1.
97. Dardenne, A., A. van Lamsweerde, and S. Fickas, *Goal-Directed Requirements Acquisition.* Science of Computer Programming, 1993. **20**(1-2): p. 3-50.
98. Lamsweerde, A., R. Darimont, and E. Letier, *Managing Conflicts in Goal-Driven Requirements Engineering.* IEEE Transactions on Software Engineering, 1998. **24**(11).
99. Van Lamsweerde, A. and E. Letier. *Integrating Obstacles in Goal-Driven Requirements Engineering*. 1998: IEEE COMPUTER SOCIETY.
100. Gamma, E., et al., *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing. 1995: Addison-Wesley.
101. Dillon, L.K., et al., *A graphical interval logic for specifying concurrent systems.* ACM Transactions on Software Engineering and Methodology (TOSEM), 1994. **3**(2): p. 131-165.
102. J. C. Corbett, G.S.A. *Using integer programming to verify general safety and liveness properties*. in *Formal Methods in System Design*. 1995.
103. Koymans, R., *Specifying real-time properties with metric temporal logic.* Real-Time Systems, 1990. **2**(4): p. 255-299.

104. L. E. Moser, Y.S.R., G. Kutty, P. M. Melliar-Smith, L. K. Dillon, *A graphical environment for the design of concurrent real-time systems.* Transactions on Software Engineering and Methodology, 1997. **6**: p. 31-79.
105. van Lamsweerde, A., R. Darimont, and E. Letier, *Managing Conflicts in Goal-Driven Requirements Engineering.* IEEE Transactions on Software Engineering, 1998. **24**(11).
106. Elaasar, m. and L. Briand, *An overview of uml consistency management*. 2004.
107. OMG. *UML 2.0 Object Constraint Language (OCL) Final Adopted specification.* 2003 [cited 2005; Available from: http://www.omg.org/cgi-bin/doc?ptc/2003-10-14.
108. Bozga, M., et al., *Kronos: A Model-Checking Tool for Real-Time Systems.* LECTURE NOTES IN COMPUTER SCIENCE, 1998: p. 298-302.
109. Holzmann, G.J., *The Spin Model Checker: Primer and Reference Manual*. 2004: Addison-Wesley Professional.
110. Cimatti, A., et al., *NuSMV: A New Symbolic Model Verifier.* LECTURE NOTES IN COMPUTER SCIENCE, 1999: p. 495-499.
111. Pnueli, A. and E. Shahar, *A platform for combining deductive with algorithmic verification.* LECTURE NOTES IN COMPUTER SCIENCE, 1996: p. 184-195.
112. Pettersson, P. and K.G. Larsen, *Uppaal2k.* Bulletin of the European Association for Theoretical Computer Science, 2000. **70**(40-44): p. 2.
113. Eshuis, R. and R. Wieringa. *Verification support for workflow design with UML activity graphs*. 2002: ACM New York, NY, USA.
114. Schäfer, T., A. Knapp, and S. Merz. *Model Checking UML State Machines and Collaborations*. in *Workshop on Software Model Checking in connection with CAV '01*. 2001.
115. Dhaussy, P., J. Roger, and F. Boniol, *« Mise en œuvre d'unités de preuve pour la vérification formelle de modèles»*. Ingénierie Dirigée par les Modèles (IDM'07), Toulouse, France, 2007.
116. McUmber, W.E. and B.H.C. Cheng. *A General Framework for Formalizing UML with Formal Languages*. 2001.
117. Dehne, F., R. Wieringa, and H. van de Zandschulp, *Toolkit for Conceptual Modeling (TCM): User's Guide and Reference*. 1997: Vrije Universiteit, Faculteit der Wiskunde en Informatica.
118. Wang, E.Y., H.A. Richter, and B.H.C. Cheng. *Formalizing and Integrating the Dynamic Model within OMT*. 1997.
119. Evans, A. and S. Kent, *Core Meta-Modelling Semantics of UML: The pUML Approach.* LECTURE NOTES IN COMPUTER SCIENCE, 1999: p. 140-155.
120. Estublier, J., et al., *Action Spécifique CNRS sur l'Ingénierie Dirigée par les Modèles*. 2005, CNRS.
121. Selic, B., *The pragmatics of Model-Driven Development.* IEEE Software, 2003. **20**(5): p. 19-25.
122. Sztipanovits, J. and G. Karsai, *Model-Integrated Computing.* Computer, 1997. **30**(4): p. 110-111.
123. Sztipanovits, J., et al., *MULTIGRAPH: An Architecture for Model-Integrated Computing.* Proceedings of the IEEE ICECCS'95, 1995: p. 361-368.
124. Greenfield, J. and K. Short, *Software factories: assembling applications with patterns, models, frameworks and tools.* 2003: ACM Press New York, NY, USA.
125. OMG. *Object Management Group*. 1997 [cited 2009; Available from: http://www.omg.org;.
126. OMG. *Model Driven Architecture*. 2003 [cited 2008; Available from: http://www.omg.org/mda/.
127. Harel, D. and B. Rumpe, *Modeling languages: Syntax, semantics and all that stuff-part I: The basic stuff.* in the Belfer Institute of Mathematics and Computer Science, 2000.
128. Spivey, J., *The Z notation: a reference manual*. 1992.
129. Yu, E.S.K. *Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering*. in *International Symposium on Requirements Engineering*. 1997.
130. Brottier, E., et al. *Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool*. in *ISSRE'06*. 2006. Raleigh, NC, USA.
131. Charlet, J., B. Bachimont, and R. Troncy, *Ontologies pour le web sémantique*. Web sémantique : Action spécifique 32 CNRS/STIC, 2003.
132. OMG, *Meta-Object Facility (MOF), version 1.4*. 2002.

133. Brown, A., *Model driven architecture: Principles and practice*. Software and Systems Modeling, 2004. **3**(4): p. 314-327.
134. Gabbay, D. and A. Hunter, *Making Inconsistency Respectable: Part 2-Meta-level handling of inconsistency*. LECTURE NOTES IN COMPUTER SCIENCE, 1993: p. 129-129.
135. Nissen, H.W., et al., *Managing Multiple Requirements Perspectives with Metamodels*. IEEE Software, 1996. **13**(2): p. 37-48.
136. Harel, D. and B. Rumpe, *Modeling Languages: Syntax, Semantics and All That Stuff, Part I: The Basic Stuff*. 2000.
137. Baudry, B., et al., *Barriers to Systematic Model Transformation Testing*. Communications of the ACM, 2009.
138. Czarnecki, K. and S. Helsen, *Feature-based survey of model transformation approaches*. IBM Systems Journal, 2006. **45**(3): p. 621-646.
139. Czarnecki, K. and S. Helsen, *Classification of Model Transformation Approaches*. Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture, 2003.
140. Lawley, M. and J. Steel, *Practical declarative model transformation with Tefkat*. LECTURE NOTES IN COMPUTER SCIENCE, 2006. **3844**: p. 139.
141. Jouault, F. and I. Kurtev, *Transforming models with ATL*. LECTURE NOTES IN COMPUTER SCIENCE, 2006. **3844**: p. 128.
142. OMG. *MOF 2.0 Q/V/T OMG Revised submission*. 2005 [cited 2005; Available from: http://www.omg.org/cgi-bin/doc?ad/05-03-02.
143. France_Télécom. *SmartQVT - A QVT Implementation*. 2006; Available from: http://smartqvt.elibel.tm.fr/.
144. Kolovos, D., R. Paige, and F. Polack, *Merging models with the epsilon merging language (EML)*. LECTURE NOTES IN COMPUTER SCIENCE, 2006. **4199**: p. 215.
145. Muller, P.A., F. Fleurey, and J.M. Jézéquel, *Weaving executability into object-oriented meta-languages*. Proceedings of MODELS/UML, 2005. **3713**: p. 264-278.
146. Kermeta. *The Kermeta Project Home Page*. 2005 [cited 2008; Available from: http://www.kermeta.org.
147. France, R., et al., *The UML as a formal modeling notation*. Computer Standards and Interfaces, 1997. **19**(7): p. 325-334.
148. Gordon, M. and T. Melham, *Introduction to HOL*. 1993: Cambridge University Press.
149. Matthew B. Dwyer, G.S.A., James C. Corbett. *A System of Specification Patterns*. 1997; Available from: http://patterns.projects.cis.ksu.edu/.
150. McUmber, W. and B. Cheng. *A general framework for formalizing UML with formal languages*. 2001.
151. Sampaio, A., A. Rashid, and P. Rayson. *Early-AIM: an approach for identifying aspects in requirements*.
152. Lugato, D., et al. *Automated Functionnal Test Case Synthesis from THALES industrial Requirements*. in *RTAS'04*. 2004. Toronto, Canada.
153. Saeed, W.A., et al. *Model Driven Requirements Engineering*. 2006; Available from: http://www.irisa.fr/triskell/Softwares/protos/mdre/.
154. Budinsky, F., S. Brodsky, and E. Merks, *Eclipse modeling framework*. 2003: Pearson Education.
155. Baudry, B., C. Nebut, and Y. Le Traon, *Model-driven Engineering for Requirements Analysis*, in *EDOC (Entreprise Distributed Object Computing Conference)*. 2007: Annapolis.
156. Maiden, N.A.M., *CREWS-SAVRE: Scenarios for Acquiring and Validating Requirements*. Automated Software Engineering, 1998: p. 419-446.
157. Maiden, N.A.M., et al., *CREWS-SAVRE: Systematic Scenario Generation and Use*, in *International Conference on Requirements Engineering*. 1998.
158. OMG. *Unified Modeling Language: Superstructure Version 2.0*. 2003 [cited 2009; Available from: http://www.uml.org/.
159. Jackson, M., *The meaning of requirements*. Annals of Software Engineering, 1997. **3**: p. 5-21.
160. Muller, P.-A., et al. *Model-Driven Analysis and Synthesis of Concrete Syntax*. in *MoDELS'06*. 2006. Genova, Italy.

161. Cockburn, A., *Writing effective use cases*. 2001: Addison-Wesley Boston.
162. Cheng, B.H.C. and J.M. Atlee. *Research directions in requirements engineering*. in *FOSE '07 : Future of software engineering*. 2007. Washington DC: IEEE Computer Society.
163. Chauvel, F. and J. Jezequel, *Code Generation from UML Models with Semantic Variation Points*. LECTURE NOTES IN COMPUTER SCIENCE, 2005. **3713**: p. 54.
164. Parr, T. *ANTLR*. 1997 [cited 2006 February]; Available from: http://www.antlr.org/.
165. Mottu, J.-M., et al. *Génération automatique de test pour les transformations de modèles*. in *IDM'05 (Ingénierie Dirigée par les Modèles)*. 2005. Paris, France.
166. TOPCASED. *The TOPCASED project*. Available from: http://www.topcased.org/.
167. Clarke, S. and R.J. Walker. *Composition patterns: An approach to designing reusable aspects*. in *ICSE'01 (Int'l Conf. on Software Engineering)*. 2001.
168. Clarke, S. and R.J. Walker. *Towards a Standard Design Language for AOSD*. in *AOSD'02 (Aspect-Oriented Software Development)*. 2002. Enschede, The Netherlands.
169. Tarr, P., et al. *N Degrees of separation: Multi-Dimensional Separation of Concerns*. in *IEEE International Conference of Software Engineering*. 1999. Los Angeles.
170. IBM. *Eclipse platform*. Available from: http://www.eclipse.org/.
171. Narman, P., P. Johnson, and L. Nordstrom. *Enterprise Architecture: A Framework Supporting System Quality Analysis*. in *IEEE International Enterprise Distributed Object Computing Conference (EDOC)*. 2007.
172. Etien, A., *L'ingénierie de l'alignement : Concepts, Modèles et Processus. La méthode ACEM pour la correction et l'évolution d'un système d'information aux processus d'entreprise*, in *Centre de recherche en informatique (CRI)*. 2006, Université de Paris I: Paris.
173. Zave, P., *A Compositional Approach to Multiparadigm Programming.* IEEE Software, 1989.
174. Sommerville, I. and G. Kotonya, *Requirements Engineering: Processes and Techniques*. 1998: John Wiley & Sons.
175. Boehm, B.W., *Software Engineering Economics*. 1981: Prentice-Hall.
176. Some, S.S., *Use Cases based Requirements Validation with Scenarios.* Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE'05)-Volume 00, 2005: p. 465-466.
177. Peter, K., *Requirements Elicitation and Validation with Real World Scenes*. 1998.
178. Kösters, G., H.-W. Six, and M. Winter, *Coupling Use Cases and Class Models as a Means for Validation and Verification of Requirements Specifications.* Requirements Engineering, 2001. **6**(3): p. 17.
179. Somé, S.S., *Supporting use case based requirements engineering.* Information and Software Technology, 2006. **48**(1): p. 43-58.
180. Maiden, N.A.M., et al. *CREWS-SAVRE: systematic scenario generation and use*. 1998.
181. Some, S. *An environment for use cases based requirements engineering*. 2004.
182. J. Ryser, S.B.a.M.G., *On the state of the art in requirements-based validation and test of software*. 1998, University of Zurich.
183. *European User Survey Analysis*, in *Report USV_EUR 2.1*. 1996, ESPITI Project.
184. IEEE_computer_society, *IEEE Recommended Practice for Software Requirements Specifications. IEEE Std 830-1993*. 1993.
185. Gunter, C., et al., *A reference model for requirements and specifications.* IEEE Software, 2000. **17**(3): p. 37-43.
186. Jackson, M. and P. Zave, *Deriving specifications from requirements: an example.* Proceedings of the 17th international conference on Software engineering, 1995: p. 15-24.
187. Laney, R., et al. *Composing Features by Managing Inconsistent Requirements*. in *Ninth International Conference on Feature Interactions in Software and Communication Systems (ICFI)*. 2007.
188. OMG, *XML Metadata Interchange (XMI) Specification.* Version. **1.2**: p. OMG document 02-01-01.
189. Gotel, O. and C. Finkelstein. *An analysis of the requirements traceability problem*. 1994.

VU :

Le Directeur de thèse

Yves Le Traon

VU :

Le Responsable de l'Ecole Doctorale

VU pour autorisation de soutenance

Rennes, le

Le Président de l'Université de Rennes 1

Guy CATHELINEAU

Vu apès soutenance pour autorisation de publication :

Le président de Jury,

Résumé

Dans cette thèse, nous nous intéressons à la définition d'une plate-forme industrielle favorisant une meilleure intégration des techniques de vérification et de validation des exigences au sein des processus de développement. Cette plate-forme, appelée R2A (pour *Requirements To Analysis*) est entièrement construite à l'aide de technologies issues de l'ingénierie dirigée par les modèles. Le cœur de la plate-forme est un processus de composition de modèles à deux niveaux de modélisation. Au niveau instance, il produit une spécification globale des exigences à partir d'une collection de spécifications d'exigences partielles, hétérogènes et potentiellement incohérentes. Au niveau de la conception de la plate-forme (niveau meta), il produit le formalisme interne de la plate-forme (types d'information pouvant être capturée et fonctionnalités supportées) à partir de composants de conception embarquant des sémantiques opérationnelles, de composition et de déploiement. Ce processus favorise l'adaptabilité de la plate-forme à des contextes industriels variés.

L'obtention d'une spécification globale des exigences (i) autorise l'application des techniques modernes de vérification et de validation pour la détection d'incohérences et (ii) favorise une approche de développement dirigée par les modèles (MDD) dès les premières étapes du développement logiciel (synchronisation exigences et artefacts de développement aval). Dans sa version actuelle, la plate-forme est spécialisée pour le contexte industriel de France Télécom. Elle supporte quatre langages de description des exigences : les diagrammes d'activité et de classes UML, un langage naturel contraint (le RDL, pour *Requirements Description Language*) et son formalisme interne (le RM, pour *Requirements Metamodel*). Les fonctionnalités principales sont (i) la génération d'objectifs de test système, (ii) la production d'une première spécification d'analyse et (iii) la vérification de la cohérence des exigences.

Abstract

We focus in this PhD on the definition of an industrial platform promoting a better integration of techniques for requirements verification and validation within software development processes. This platform called R2A (which stands for *Requirements for Analysis*) is entirely built using technologies from model-driven engineering. The core of the platform is a model composition process working at two modeling levels. At the instance-level, it produces a global specification of requirements from a collection of partial, heterogeneous and potentially inconsistent specifications. At the design level (so-called meta-level), it produces the internal formalism of the platform (types of information that can be captured and supported features) from design components which embedd operational, compositional and deployment semantics. As such, this process promotes the adaptability of the platform to various industrial contexts.

Obtaining a global specification of requirements (i) allows the application of modern techniques of verification and validation in order to detect inconsistencies and (ii) promotes a model-driven development approach (MDD) from the early stages of software development (synchronization between requirements and downstream development artifacts). The current version of the R2A platform is specialized for the industrial context of France Télécom. It supports four requirements description languages: UML activity diagrams, UML class diagrams, a constrained natural language (the RDL, which stands for *Requirements Description Language*) and the internal formalism (the RM, which stands for *Requirements Metamodel*). The main features are (i) the generation of system test objectives, (ii) the production of a first analysis model and (iii) requirements consistency checking.