UNIVERSITÉ CÔTE D'AZUR

UNIVERSITÉ CÔTE D'AZUR
ÉCOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

pour l'obtention du grade de

Docteur en Sciences

de l'Université Côte d'Azur

Discipline : Informatique

par

Ivan Logre

Préserver la Séparation des Préoccupations durant l'Intégration de Domaines Hétérogènes dans les Systèmes Logiciels

Dirigée par *Michel Riveill*

Soutenue le 01 septembre 2017

Devant le jury composé de :

M.	Jean-Michel	Bruel	Professeur des Universités, Toulouse	Rapporteur
M.	Franck	Fleurey	Senior Scientist, SINTEF IKT	Rapporteur
M.	Jörg	Kienzle	Professeur des Universités, McGill	Rapporteur
M.	Philippe	Lahire	Professeur des Universités, Côte d'Azur	Examinateur
M.	Sébastien	Mosser	Maître de Conférence, Côte d'Azur	Encadrant
M.	Michel	Riveill	Professeur des Universités, Côte d'Azur	Directeur

TABLE DES MATIÈRES

Table des figures **iv**

1 Introduction **1**
1.1 Contexte . . . 1
1.2 Propriétés et problématique . . . 2
1.3 Cas d'étude . . . 3
1.4 Structure du document . . . 4

2 État de l'art **5**
2.1 Approches Architecturales de Séparation de Préoccupations . . . 6
2.1.1 Contexte : Vues et Points de vues . . . 6
2.1.2 Frameworks architecturaux . . . 6
2.1.3 Approches de Spécification Multi-Vues . . . 7
2.1.4 Langages de Description d'Architectures . . . 7
2.1.5 Discussion sur les approches architecturales . . . 9
2.2 Critères de comparaison et familles de composition . . . 10
2.2.1 Critères de comparaison . . . 10
2.2.2 Familles de composition . . . 11
2.3 Approches par composition de modèles . . . 13
2.3.1 Fusion . . . 13
2.3.2 Tissage . . . 15
2.3.3 Embarqué . . . 16
2.3.4 Intégration . . . 17
2.3.5 Hybride . . . 18
2.4 Approches par composition de langages . . . 20
2.4.1 Fusion . . . 20
2.4.2 Embarqué . . . 21
2.4.3 Intégration . . . 21
2.4.4 Hybride . . . 23
2.5 Discussion . . . 26
2.5.1 Espace couvert par l'état de l'art . . . 26
2.5.2 Enjeux de la contribution . . . 26

3 Intégration de Méta-Modèles **29**
3.1 Problématiques d'isolation . . . 29
3.2 Intégration de services . . . 31
3.2.1 Architecture Orientée Services (SOA) . . . 31
3.2.2 Propriétés des services et de l'intégration . . . 33
3.3 Modélisation des domaines en tant que services . . . 35

3.3.1 Modélisation en services 35
3.3.1.1 Méta-concepts de l'approche d'intégration 35
3.3.1.2 Sémantique d'exécution des messages 35
3.3.2 Communication par messages 36
3.3.2.1 Définition d'un message 36
3.3.2.2 Effet d'un message sur un service de domaine 37
3.3.3 Interfaces de Domaines 39
3.3.3.1 Encapsulation du domaine en boite noire 39
3.3.3.2 Illustration sur l'exemple fil rouge 40
3.3.3.3 Code fonctionnel du Service 41
3.3.4 Coût de l'approche 42
3.4 Implémentation 42
3.5 Conclusions 43

4 Règles Métier et Cohérence de l'Architecture 45
4.1 Problématique de Collaboration de Domaines et Cohérence 45
4.2 Grammaire des Règles Métier 48
4.2.1 Définition d'une Règle de Routage 49
4.2.2 Définition d'une Règle d'Intégration 51
4.3 Cohérence Inter-Domaines 52
4.3.1 Propagation de Déclenchement de Règles 52
4.3.2 Expressivité des Règles d'Intégration 53
4.3.3 Écriture et Composition de règles 54
4.4 Implémentation 55
4.5 Conclusions 56

5 Concrétisation et Variabilité Technologique 58
5.1 Problématique de variabilité dans la concrétisation du système 58
5.1.1 Influence de la variabilité dans la concrétisation du système 58
5.1.2 État de l'art en variabilité 60
5.1.3 Modélisation de la variabilité par les Feature Models 61
5.2 Création d'un catalogue 63
5.3 Sélection d'un produit 67
5.4 Implémentation 71
5.5 Conclusions 74

6 Application à la visualisation de données 75
6.1 Visualisation de données en provenance de capteurs 75
6.1.1 Présentation des domaines 75
6.1.2 Pertinence du cas d'application 78
6.1.3 Implémentation du prototype 79
6.2 Apport des contributions sur le cas d'application 80
6.2.1 Intégration des méta-modèles 80
6.2.2 Cohérence de l'architecture 83
6.2.3 Variabilité et concrétisation du système 85

7 Validation **89**
7.1 Conditions d'expérimentation . . . 89
7.1.1 Objectifs . . . 90
7.1.2 Catégorisation des données . . . 90
7.2 Exp #1 : Comparaison des solutions . . . 91
7.2.1 Protocole expérimental . . . 91
7.2.2 Observations . . . 92
7.2.3 Discussion et perspectives . . . 93
7.3 Surcoût de l'encaspulation en service . . . 94
7.3.1 Hypothèses et données . . . 94
7.3.2 Observations . . . 95
7.3.3 Conclusions . . . 96
7.4 Gestion des interactions par des Règles Métier . . . 96
7.4.1 Hypothèses et données . . . 97
7.4.2 Observations . . . 98
7.4.3 Conclusions . . . 99
7.5 Automatisation de l'approche . . . 100
7.5.1 Hypothèses . . . 100
7.5.2 Observations . . . 101
7.5.3 Conclusions . . . 101
7.6 Exp #2 : Caractérisation et concrétisation . . . 101
7.6.1 Présentation de l'expérience . . . 101
7.6.2 Observations . . . 104
7.6.3 Conclusions et discussions . . . 108
7.7 Conclusions . . . 109

8 Conclusions et Perspectives **111**
8.1 Conclusions . . . 111
8.2 Perspectives . . . 113
8.2.1 Généralisation de la validation . . . 113
8.2.2 Analyse de Satisfaction des Spécifications par le Système Produit 114

Annexes **116**

A Modèles et code du cas d'application **117**
A.1 Extraits de code des services . . . 117
A.2 Syntaxes concrètes . . . 120
A.3 Syntaxes Abstraites . . . 121

B Expérience : Caractérisation et Concrétisation **129**
B.1 Protocole Expérimental - Version A . . . 130
B.2 Protocole Expérimental - Version B . . . 131
B.3 Spécification du dashboard de suivi d'une équipe de projet étudiant . . . 132
B.4 Taxonomie et définition des termes . . . 132
B.5 Résultats bruts . . . 133

Bibliographie **138**

TABLE DES FIGURES

1.1 Diagramme de Venn des domaines impliqués dans la conception de *dashboards*. . . . 3

2.1 Schémas des approches de composition regroupées en cinq familles 12
2.2 Points d'extension de ThingML [Harrand 16]. . . . 21
2.3 Vision d'ensemble des contributions de BCOoL [Vara Larsen 16]. . . . 22
2.4 Vision d'ensemble des contributions de Melange [Degueule 15]. . . . 23

3.1 Schéma de collaboration des domaines isolés. . . . 31
3.2 Architecture des services de domaines et instanciation. . . . 34
3.3 Diagramme de classes simplifié des concepts de l'intégration. . . . 35
3.4 Diagramme de séquence de l'interprétation d'un message. . . . 36
3.5 Diagrammes de classe décrivant la structure des messages 37
3.6 Impact de l'invocation d'une opération par message sur le modèle. . . . 38
3.7 Comparaison des approches sur l'exemple du VersionControlSystem (VCS). 39
3.8 Extrait du méta-modèle de gestionnaire de versions encapsulé dans un service. . . . 40
3.9 Extrait du méta-modèle de gestion de tickets encapsulé dans un service. . 41

4.1 Schéma de acteurs de la gestion de la cohérence pour domaines isolés. . . 46
4.2 Schéma des domaines impliqués. . . . 47
4.3 Diagramme de classes simplifié des concepts de la gestion de la cohérence. 48
4.4 Diagramme de séquences des interactions entre l'expert du domaine et le service. . . . 50

5.1 Extrait d'un *Feature Model* pour la Gestion de Versions. . . . 62
5.2 Schéma des acteurs de la gestion de la variabilité. . . . 63
5.3 Schéma de la gestion de la variabilité. . . . 64
5.4 Processus de fusion utilisé pour construire le modèle de variabilité. . . . 66
5.5 *Feature Model* du domaine de la Gestion de Versions. . . . 68

6.1 Diagramme d'architecture du cas d'application. . . . 76
6.2 Extrait de syntaxe abstraite du domaine RE, adaptée de Gherkin. . . . 77
6.3 Syntaxe abstraite du domaine DM, réutilisée du framework DEPOSIT. . . 78
6.4 Extrait de syntaxe abstraite du domaine VD, basée sur la taxonomie du *data journalism*. . . . 79
6.5 Diagramme de classes des Messages échangés avec le service `DataManagement`. 82
6.6 Diagramme de dépendance des projets Maven. . . . 83
6.7 Extrait de la matrice de caractérisation d'AmCharts Caractéristiques de la taxonomie satisfaites par artefact (*widget*). . . . 86

6.8 *Feature Model* fusionné des 73 *widgets* de visualisation. 87

7.1 Exemple simplifié d'énoncé et de modèle de référence. 92
7.2 Exemple d'une paire de réponse. 92
7.3 Proportion de code généré par rapport au code spécifique à l'intégration à implémenter manuellement, et détail du code manuel. 95
7.4 Répartition (en %) des domaines ciblés par les interactions par domaine. 99
7.5 Volume de code a fournir manuellement par l'intégrateur par rapport au code généré ou fourni par l'approche car générique. 100
7.6 Mesures par caractérisation en quartiles (min, Q1, médiane, Q3, max). . 105
7.7 Mesures d'utilisation de la méthode par caractérisation 105
7.8 Comparaison des temps moyens (en s) de traitement des visualisations et gain (en %) entre les méthodes. 107
7.9 Influence du nombre de visualisations après itération sur la satisfaction. . 110

8.1 Architecture de conception d'un système à trois domaines hétérogènes. . 112

A.1 Matrice de caractérisation d'AmCharts 119
A.2 Feature Model fusionné des 73 widgets de visualisation 120
A.3 Exemple d'implémentation EMF de la syntaxe concrète du domaine RE, réutilisée de Gherkin. 121
A.4 Exemple d'implémentation EMF de la syntaxe concrète du domaine VD, réutilisant la taxonomie du data journalisme. 122
A.5 Exemple d'implémentation MPS de la syntaxe concrète du domaine SD, réutilisée du framework DEPOSIT. 123
A.6 Exemple de syntaxe abstraite du domaine RE, adaptée de Gherkin. . . . 124
A.7 Exemple de syntaxe abstraite du domaine VD, réutilisant la taxonomie du data journalisme. 125
A.8 Exemple de syntaxe abstraite du domaine SD, réutilisée du framework DEPOSIT. 126
A.9 Méta-Modèles en entrée de *Match and Merge* 127
A.10 Méta-modèle fusionné résultat de *Match and Merge*. 128

B.1 Étapes 1 et 2 des participants au protocole A 133
B.2 Étapes 3 des participants au protocole A 134
B.3 Étapes 1 et 2 des participants au protocole B 135
B.4 Étapes 3 des participants au protocole B 136

REMERCIEMENTS

Écrire une thèse c'est un peu comme bâtir un maison pour y exposer ses travaux. On a beau avoir vu des maisons, avoir passé du temps à admirer leur contenu et leur finalité, c'est en construisant que l'on devient maçon. Après un plan dans un langage qu'on est le seul à comprendre, on commence par un bout, sans trop savoir si conceptuellement il s'agit des fondations où de la cheminée. Brique après brique, en évitant de s'enfermer dans son propre labyrinthe. De temps en temps, un collègue vient questionner la solidité d'un mur, donner un coup pinceau, de truelle ou de masse. La thèse c'est comme bâtir une maison, c'est une aventure à la fois personnelle et emplie d'échanges. Vous en lisez le résultat, et je souhaite remercier ici ceux qui y ont apporté leur pierre. Attention à la marche.

"*Essayez d'apprendre quelque chose sur tout et tout sur quelque chose.*"
Thomas Henry Huxley

Ceux qui ont du ciment sur les doigts

En premier lieu, je souhaite remercier ceux qui lu ce manuscrit en entier et ont participé à sa réalisation. Jean-Michel Bruel, Frank Fleurey, Jörg Kienzle, un grand merci pour voir accepté de rapporter cette thèse, pour vos remarques pertinentes qui m'ont aidé à l'améliorer et à pousser mes raisonnements. Merci également à Philippe Lahire d'avoir accepté de faire partie de mon jury malgré ses responsabilités et à Philippe Collet pour sa pertinence scientifique et comique qui m'ont inspiré dans les heures les plus sombres de mes fusions de produits.

Michel mérite un remerciement considérable pour son soutien incontestable depuis mon stage dans l'équipe, le recul qu'il m'a apporté au long de ces années et qui m'a permis de mettre tant de choses en perspective.

Anne-Marie, d'abord enseignante, puis chef, puis mentor et enfin amie, un soutient considérable, inestimable et la personne la plus humaine que je connaisse, je te remercie de m'avoir toujours poussé vers le mieux.

Sébastien, je ne saurais te remercier assez d'avoir rendu cette aventure possible, de m'avoir retenu de retourner prématurément dans mes montagnes, à coup de (méta-)discussions passionnées, merci d'avoir été l'Umbre de mon Fitz.

Ceux qui ont cohabité dans la poussière

Une thèse c'est également une aventure humaine, faite d'échanges et de moment de vie. Ici je prends le temps de remercier ceux qui ont fait mon quotidien au laboratoire, les "co-bureaux" et assimilés.

Christian fut le témoin loyal de mes débuts et m'a continuellement encouragé. Ton dévouement sans faille aux étudiants est un exemple que je garderai en tête aussi longtemps que j'aurais la chance d'enseigner.

Cyril a été mon filleul et mon ami avant d'être mon presque frère de thèse. Tes *in-doc* n'ont fait que mettre en valeur l'importance de ta présence et de nos interminables discussions dans les bons moments comme dans les pires.

Mes voisins ne sont pas oubliés. Benjamin, le petit dernier, étonnant dans sa capacité à alterner entre les arguments scientifiques et les prévisions de la prochaine "animation d'équipe".

Simon et son estomac réglé comme une horloge, lui qui m'a donné une bonne vision de l'épreuve de la rédaction à coup de grognements et qui m'a sauvé la vie plus d'une fois avec FAMILIAR , merci d'avoir refait le monde quelques soirs autour d'une table et d'une bière.

Plus généralement, à tous ceux qui ont participé à maintenir une ambiance de travail agréable au sein de Sparks, avec une mention honorable pour Stéphanie, Lucas, Mélanie, Clément, Yoann, Katy, Frank et Sami qui ont vécu, ou vivent encore, la thèse avec moi, merci.

Les inoubliables d'I3S

Sabine, pour ta minutie dans l'organisation des événements qui ont ponctué ces années, conférences, journées d'équipe, école d'été et soutenance, merci.

Françoise, tu as été la toute première personne de l'université à qui j'ai parlé en arrivant à l'université il y a sept ans et tu as gardé ce sourire et cette bonne humeur durant toutes ces années, bravo !

Je remercie aussi ceux qui m'ont observé d'un peu plus loin. Johan pour ton intérêt, ta curiosité et tes questions pertinentes lors de chacune de mes présentations, ainsi que pour ton sourire discret mais rassurant après mon audition pour le financement de l'ED. Alain qui a su apporter un regard neuf sur des hypothèses qui me semblaient naturelles. Philippe R. pour m'avoir poussé dans les retranchements de mon raisonnement et obligé à regarder le problème sous différents angles. Audrey, pour toutes ces discussions entre modèles et IHM qui ont ouvert la voie à beaucoup de réflexions, toujours en cours !

Ceux qui entretiennent la flamme

L'école d'ingé nous a appris à apprendre, là où le doctorat m'a poussé à dépasser ce que j'avais appris. Quelle meilleure façon de remettre en cause ce qui est établi que de tenter de l'expliquer à d'autres ? En cela, l'enseignement me semble indissociable de la recherche et je remercie ici ceux qui m'ont permis de m'y épanouir.

Mireille, pour ton efficacité sur tous les fronts et ta persévérance à toute épreuve, je te remercie pour tes conseils et ta confiance.

Stéphane, ta rigueur et ta disponibilité m'ont apporté beaucoup pour la première matière que j'ai eu à enseigner à partir de rien, un mois *avant* le début de ma thèse.

Hélène, merci pour ton ouverture à la réflexion et l'attention que tu portes à la réussite des étudiants lors de leur première expérience de programmation.

Je me dois de remercier à nouveau ici Anne-Marie et Sébastien qui furent mes deux premiers modèles, démontrant chacun à leur manière une passion pour ce métier sans cesser d'adapter leur pédagogie.

Ceux qui observent de loin

Bien entendu je dois me tourner vers ma mère Nathalie en premier lieu, je ne pourrais jamais remercier assez pour tout ce qu'elle m'a apporté et permis d'accomplir. Merci également au reste de ma famille qui m'a supporté de là-bas pendant tout ce temps, en m'offrant des périodes de décompression occasionnelles.

Gaëtan, tu me supportes depuis l'IUT et c'est pas fini, tu as été un soutient moral quotidien et inconditionnel.

Olivier, merci pour ta bonne humeur que tu me partages depuis le collège.

Isabella, nous nous suivons depuis la primaire, jusqu'à vivre nos thèses en parallèle, et tu ne faillis pas à ta mission !

Etienne et Tanguy, merci de m'avoir permis de sortir la tête de l'eau occasionnellement avec ces balades sur Pandora, respectivement accompagnées de sushis et d'hypocras.

CHAPITRE 1

INTRODUCTION

Table des matières

1.1 Contexte 1
1.2 Propriétés et problématique 2
1.3 Cas d'étude 3
1.4 Structure du document 4

1.1 Contexte

Depuis les années soixante-dix, l'évolution des langages de programmation préconise l'introduction d'une Séparation des Préoccupations (SoC) [Dijkstra 76, Parnas 72], c'est-à-dire la décomposition des systèmes en plusieurs unités, fonctionnelles ou comportementales, plus maîtrisables et réutilisables. Plusieurs approches ont été proposées depuis afin de découper le code, en commençant par la programmation orientée objets (OO), puis orientée composants [Nierstrasz 92] et la programmation orientée aspects [Kiczales 97]. Les systèmes logiciels actuels sont des systèmes composites par essence nommés Systèmes de Systèmes (SoS) [Keating 03, Boardman 06] dans lesquels nous retrouvons par exemple le cas des Systèmes Cyber-Physique et de l'Internet des Objets. Ces systèmes nécessitent la prise en compte de différentes préoccupations, qu'elles soient transverses (comme le passage à l'échelle, la sécurité et la persistance), ou bien orientées métier (*e.g.,* la modélisation des capteurs dans les CPS).

Dans les approches d'Ingénierie Dirigée par les Modèles (*Model-Driven Engineering* ou MDE), chaque préoccupation est résolue par l'intervention d'un expert domaine en concevant des modèles. Une des approches classiques de manipulation de modèles est l'utilisation d'un Langage Spécifique au Domaine (*Domain Specific Language* ou DSL) Un DLS comprend : (*i*) une syntaxe abstraite définissant l'ensemble des concepts du domaine et les relations et contraintes qui les régissent et (*ii*) une syntaxe concrète permettant d'instancier facilement des modèles en ne décrivant que ce qui est nécessaire [Fowler 10]. Il est également possible de définir un méta-modèle et d'instancier et manipuler des modèles qui lui sont conformes.

> "*The task of creating a metamodel is the task of creating a language that is capable to describe the relevant aspects of a subject under consideration that are of interest for the future users of the created models.*" [Höfferer 07]

Après avoir décomposé le SoS en préoccupations et permis pour chacune la manipulation de modèles pour chacune, il reste à les intégrer, *i.e.,* recomposer les parties pour retrouver une vision globale du système, à des fins de validation, de simulation, et de

génération de code par exemple. Cette intégration est la clef de la coopération d'experts de différents domaines contribuant à un seul et même système.

> "*Having divided to conquer, we must then reunite to rule.*" [Jackson 90]

1.2 Propriétés et problématique

Dans le cadre de cette thèse, nous partons de l'hypothèse que le système global est déjà décomposé et que les experts réutilisent des DSLs pré-existants capturant leur préoccupation de façon pertinente.

Nous cherchons à atteindre les propriétés suivantes :

— Les modèles qui coexistent pour représenter les différents aspects du SoS sont **hétérogènes**. En effet les DSLs ont été conçus en amont, indépendamment les uns des autres, nous ne pouvons faire aucune hypothèse sur leur structure, leur comportement ou les concepts qu'ils embarquent.

— Préserver l'**isolation** des domaines permet à chaque expert de contribuer à la conception du système global en utilisant les concepts pertinents, sans subir la complexité des autres domaines. Un DSL est sujet à des changements à la fois dans la manière dont il capture son domaine, *i.e.,* les concepts qui le composent, et dans la façon dont il est utilisé, *i.e.,* sa syntaxe concrète, son outillage et les contraintes sémantiques. L'isolation des domaines facilite également la gestion des versions d'un DSL [Bryant 15], sans avoir à reconstruire l'intégralité du SoS.

— Les DSLs contiennent classiquement une fonction de validation de conformité, permettant de s'assurer qu'un modèle est conforme à la syntaxe abstraite considérée, ce qui assure la cohérence locale de chaque domaine. Pour maintenir la **cohérence** globale du système, l'intégration doit gérer les intersections entre ces espaces conceptuels, *i.e.,* l'ensemble des cas où il existe une interaction entre des domaines.

— Enfin, la préservation de l'**intégrité** du méta-modèle d'un domaine permet de s'assurer l'interopérabilité des modèles produits avec l'outillage du DSL utilisé.

> "*By definition, DSLs evolve as the concepts in a domain and the expert understanding of the domain evolve.*" [Cheng 15]

Notre problématique porte sur la préservation de la séparation des préoccupations, *i.e.,* l'isolation des représentation abstraites des domaines impliqués, durant leur intégration. Cette intégration a pour but de permettre aux experts de ces domaines de contribuer indépendamment au système global en vérifiant la cohérence du système global. L'intégration des domaines a pour objectif la production de modèles hétérogènes et d'assurer la possibilité d'une concrétisation du système de systèmes. La variabilité des domaines impliqués dans la conception de cette classe de systèmes implique :

(i) un effort de composition des méta-modèles hétérogènes représentant ces domaines,

(ii) une gestion de la cohérence inter-domaine des modèles produits en isolation et

(iii) une gestion de la multiplicité des cibles atteignables dans l'espace des solution de chacun des domaines.

1.3 Cas d'étude

Le Gartner Group annonce une progression de 31% du nombre d'objets connectés entre 2016 et 2017, atteignant 8, 4 milliards d'objets [1]. Actuellement, seule une minorité des données produites par ces objets est exploitée, *e.g.,* 1% des données issues des 30.000 capteurs d'une plateforme pétrolière [Manyika 15]. De cet accroissement du volume de données produites émerge un besoin de visualisation pour permettre l'analyse et la prise de décision par les utilisateurs.

Dans le domaine des systèmes cyber-physiques, les systèmes logiciels dédiés à la visualisation de données en provenance de capteurs sont appelés tableaux de bord ou *dashboards*. Un *dashboard* est défini comme un ensemble de visualisations cohérentes présentant l'information nécessaire à son utilisateur pour aider à la prise de décisions [Few 06], par exemple le tableau de bord d'une voiture. Chaque *dashboard* est conçu pour remplir un ensemble d'objectifs précis ❶ (*e.g.,* connaitre sa vitesse à tout instant), en utilisant des visualisations adaptées ❷ (*e.g.,* pourquoi une jauge angulaire et pas une courbe ?), en fonction des données de capteurs en entrée ❸ (*e.g.,* leur format, leur fréquence).

La production d'un tel système exécutable nécessite la collaboration de plusieurs préoccupations orientées domaine, comme illustré par la FIGURE 1.1, *i.e.,* :

❶ l'ingénierie des besoins, décrivant les spécifications du *dashboard*.

❷ la visualisations de données, modélisant les *widgets* et leur organisation.

❸ la gestion d'un réseau de capteurs, définissant les ressources à dispositions et leurs caractéristiques.

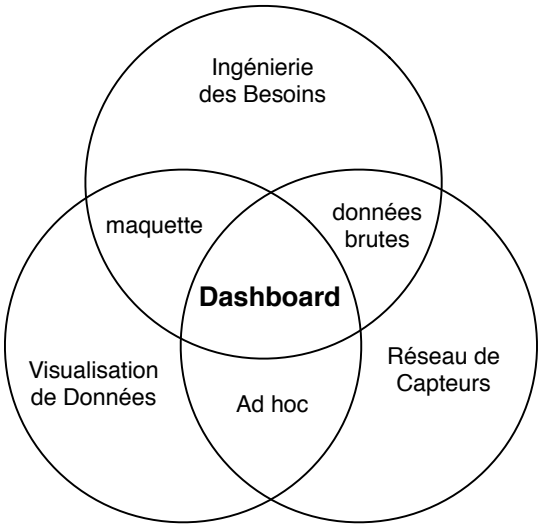


FIGURE 1.1 – Diagramme de Venn des domaines impliqués dans la conception de *dashboards*.

Plusieurs méta-modèles existent déjà pour chacun de ces domaines, *e.g.,* DOORS [2], Gherkin [Wynne 12] ou CTT [Paternò 97] en ingénierie des besoins. Chaque expert domaine travaille en isolation et sans ordre, par exemple la capture des besoins de l'utilisateur est un processus itératif qui doit pouvoir être raffiné sans avoir défini au préalable

1. http ://www.gartner.com/newsroom/id/3598917
2. http ://www-03.ibm.com/software/products/en/ratidoor

la liste exhaustive des capteurs à disposition. C'est en étudiant les interactions entre ces domaines que des incohérences émergent, ce qui peut mener dans le cas précédent à une ré-évaluation d'un objectif ❶ ou au développement d'un capteur virtuel par composition qui expose la donnée requise ❸. Dans ce cas d'étude, les données sont représentées de façon abstraites différemment dans les trois domaines : elle servent à exprimer sur quoi s'applique un besoin, ce que consomme une visualisation et ce que produit un capteur, ce qui en fait une interaction n-aire dont la cohérence doit être gérée.

La conception de *dashboard* présente donc un cas d'application de SoS pertinent vis-à-vis de la problématique annoncée en matière d'intégration de DSLs hétérogènes pré-existants et d'interactions inter-domaines. Au sein des chapitres de contribution, nous proposons l'utilisation d'un exemple fil-rouge simpliste pour illustrer les contributions par des exemples minimaux.

1.4 Structure du document

La suite du document se découpe comme suit :

Le Chapitre 2 étudie en premier lieu les approches architecturales qui proposent un cadre complet de séparation de préoccupations dans les systèmes logiciels, puis détaille plus spécifiquement l'angle de la composition sur les modèles et les DSLs.

Le Chapitre 3 détaille l'intégration des modèles du domaine encapsulés dans des services inspirés de SOA.

Le Chapitre 4 expose la résolution de la détection d'incohérences et le fonctionnement du moteur de règles métier associé.

Le Chapitre 5 décrit la gestion de la variabilité technologique par l'utilisation de Lignes de Produits Logiciels (SPLs).

Le Chapitre 6 déroule les trois contributions sur le cas d'application de la visualisation de données en provenance de capteurs.

Le Chapitre 7 présente la validation de l'approche à travers un ensemble d'expérimentations.

Le Chapitre 8 apporte une conclusion au document et propose des perspectives d'approfondissement.

CHAPITRE 2

ÉTAT DE L'ART

"A complex system is much more effectively described by a set of interrelated views [...] than by a single overloaded model." [Rozanski 11]

Table des matières

2.1 Approches Architecturales de Séparation de Préoccupations **6**
2.1.1 Contexte : Vues et Points de vues 6
2.1.2 Frameworks architecturaux 6
2.1.3 Approches de Spécification Multi-Vues 7
2.1.4 Langages de Description d'Architectures 7
2.1.5 Discussion sur les approches architecturales 9
2.2 Critères de comparaison et familles de composition **10**
2.2.1 Critères de comparaison 10
2.2.2 Familles de composition 11
2.3 Approches par composition de modèles **13**
2.3.1 Fusion 13
2.3.2 Tissage 15
2.3.3 Embarqué 16
2.3.4 Intégration 17
2.3.5 Hybride 18
2.4 Approches par composition de langages **20**
2.4.1 Fusion 20
2.4.2 Embarqué 21
2.4.3 Intégration 21
2.4.4 Hybride 23
2.5 Discussion **26**
2.5.1 Espace couvert par l'état de l'art 26
2.5.2 Enjeux de la contribution 26

L'ingénierie dirigée par les modèles (*Model-Driven Engineering* ou MDE) a pour principal objectif de permettre le développement d'un système logiciel indépendamment de sa mise en œuvre concrète. Pour cela, le MDE propose de baser le développement du système visé sur une représentation abstraite de celui-ci en support à différents raisonnements. Gérer la complexité croissante des systèmes étudiés nécessite de les décomposer en plusieurs modèles plus petits et maitrisables. Cette décomposition peut être effectuée par domaine impliqué, par préoccupation du système, par notation utilisée, etc. La conformité de chacun de ces modèles peut-être vérifiée par les approches classiques de MDE.

Toutefois, afin d'obtenir une vision complète du système, d'en vérifier la cohérence globale et d'obtenir un modèle exécutable, *e.g.,* du code source, il est nécessaire de recomposer ces modèles.

Ce chapitre traite deux angles : (*i*) les approches architecturales classiques de séparation de préoccupations (Framework Architecturaux, Spécification Multi-Vues, et Langages de Description d'Architectures), puis (*ii*) les approches de composition sur lesquelles repose la séparation des préoccupations (composition de modèles, composition de langages). Nous concluons ce chapitre en mettant en évidence les problèmes non couverts par les solutions actuelles et que nous avons choisi de traiter dans le cadre de cette thèse.

2.1 Approches Architecturales de Séparation de Préoccupations

La construction d'un logiciel de grande taille repose sur sa décomposition en sous-systèmes de plus petites tailles, plus homogènes dans leur préoccupation et donc plus maitrisables. Dans cette section, nous étudions les approches classiques présentant un système décomposé et ayant pour objectif de gérer la cohérence des sous-systèmes. Nous cherchons une approche qui ne nécessite pas la création d'une représentation abstraite unique et dont les interactions entre sous-systèmes sont gérées de façon explicite et intentionnelle, *i.e.,* faisant l'objet d'un concept dédié au niveau méta. Cette section commence par introduire les notions de *vues* et *point de vue* communes aux approches architecturales dédiées à la gestion de différents aspects d'un système. Nous présentons en particulier les *framework* architecturaux (AFs), les approches de spécifications multi-vues (MVS) et les langages de description d'architecture (ADLs), puis discutons de leur applicabilité dans le contexte de cette thèse.

2.1.1 Contexte : Vues et Points de vues

Selon Rozanski et Woods [Rozanski 11], les approches à base de vues proposent un cadre de modélisation centré sur la séparation des préoccupations. Le système est divisé en plusieurs *points de vues*, un par domaine ou métier impliqué. Ces points de vues permettent la modélisation de *vues* par les utilisateurs, *e.g.,* les développeurs. Chaque vue est conçue indépendamment et doit être conforme au point de vue dont elle dépend.

L'approche à base de vues a pour avantages de permettre la séparation des préoccupations pour gérer la complexité du système et de concentrer l'effort du développeur sur un seul aspect du système à la fois. Elle nécessite toutefois de sélectionner un ensemble de points de vues pertinents et de gérer les incohérences entre les vues produites en isolation, ce qui nécessite l'intervention d'un architecte. Ces définitions sont communes aux *frameworks* architecturaux et aux approches de spécifications multi-vues détaillées dans les sections suivantes.

2.1.2 Frameworks architecturaux

Un *framework* architectural (*Architectural Framework* ou AF) établit un ensemble de bonnes pratiques pour créer, analyser et utiliser des descriptions d'architectures, à destination d'un domaine d'application spécifique. Les AFs regroupent notamment le *Federal Enterprise Architecture Framework* [FEAF 99], le *Department of Defense Architectural*

Framework [DoDAF 03], le Zachman Framework for Enterprise Architecture [Zachman 03] et *The Open Group Architectural Framework* [TOGAF 09]. Une liste plus exhaustive est maintenue en ligne [1].

Les AFs proposent des approches structurées et systématiques pour la conception de systèmes [Tang 04]. Pour cela, ils définissent un ensemble pertinent de points de vues sur le système, *e.g.,* du point de vue fonctionnel, du développement ou du déploiement. Ces points de vues permettent d'instancier des vues représentant différentes perspectives. Les types de vues utilisables, et parfois l'ordre d'utilisation, sont définis statiquement par chaque *framework*. L'implémentation de ces *frameworks* repose généralement sur l'utilisation d'un système de Spécification Multi-Vues (MVS) et font l'objet d'extensions dans ce domaine sur les méthodes de gestion de la cohérence inter-vues, comme RM-ODP [Romero 09] et ArchiMate [Iacob 12] détaillés dans la section suivante.

2.1.3 Approches de Spécification Multi-Vues

Les approches *Multi-View Specification* ou MVS tendent à rendre les modélisations d'architectures proposées par les AFs plus flexibles et génériques. Du fait du nombre croissant d'approches MVS proposées, il est nécessaire de caractériser ces contributions pour mieux comprendre les avantages et inconvénients de chacune. Nous ne considérons ici que les contributions outillées. Nous réutilisons les critères suivants [Atkinson 15] pour caractériser les principales approches MVS au sein du tableau 2.1 :

— Les approches *Synthétiques* proposent à un architecte d'intégrer un ensemble de vues isolées au sein d'un modèle commun du système ; au contraire des approches *Projectives* qui proposent de partir d'une modélisation de l'intégralité du système en un modèle pour en dériver un ensemble de points de vues. Dans les approches Projectives, ce modèle sous-jacent est appelé *Single Underlying Model* (SUM).

— Les correspondances entre éléments de différentes vues peuvent faire l'objet d'une représentation *Explicite* si l'approche dédie un élément de langage pour les représenter, ou être *Implicite* si ces correspondances ne reposent par exemple que sur une convention de nommage.

— Les correspondances entre éléments de différentes vues peuvent être exprimées au seul niveau des instances par une représentation *Exhaustive* ou bien faire l'objet d'une déclaration dite *Intentionnelle* au niveau des points de vues, par exemple sous la forme de contraintes.

— Le SUM des approches projectives peut être minimaliste, sans redondance interne auquel cas il est dit *Essentiel*, ou bien être un agrégat de sous-modèles dont dérivent les points de vues et qualifié de *Pragmatique*.

2.1.4 Langages de Description d'Architectures

ABC/ADL [Mei 02] est un langage de description d'architecture (*Architecture Description Language* ou ADL) permettant la composition de composants logiciels. Il repose sur la définition d'architectures à base de composants et permet l'automatisation de la génération d'une application exécutable à partir de règles de configuration travaillant sur les connecteurs entre composants sur étagère, développés par exemple en JEE ou

1. http ://www.iso-architecture.org/42010/afs/frameworks-table.html

Table 2.1 – Caractérisation des approches de Spécification Multi-Vues (MVS)

Approches MVS	**Synthétique - Projective**	**Implicite - Explicite**	**Intentionnelle - Exhaustive**	**Essentiel - Pragmatique**
RM-ODP [Romero 09]	Synthétique	Explicite	Intentionnelle	NaN
Große-Rhode [Große-Rhode 13]	Synthétique	Explicite	Exhaustive	NaN
VOSE-Finkelstein [Finkelstein 92]	Synthétique	Implicite	Exhaustive	NaN
OSM [Atkinson 08]	Projective	Implicite	Intentionnelle	Essentielle
Vitruvius [Kramer 13]	Projective	Implicite	Intentionnelle	Pragmatique
Archimate [Iacob 12])	Projective	Implicite	Intentionnelle	Essentielle

CORBA. Un composant définit une interface externe et une interface interne. L'interface externe expose les services fournis par le composant et les dépendances requises. L'interface interne spécifie les contraintes inhérentes à la structure interne du composant. Le modèle de composants définit l'architecture des composants utilisés, montrant la cohérence du système global en résolvant chaque dépendance requise des composants utilisés par un service fourni par un autre. Ces liaisons de dépendances passent par un connecteur, élément liant les composants deux à deux.

Fractal est un modèle de composants [Bruneton 04] pour les systèmes distribués. Il propose la définition de composants réflexifs, dotés de capacités de contrôles permettant de personnaliser un composant lors de sa réutilisation, grâce à ses capacités de réflexivité. Ainsi l'architecte peut par exemple choisir un compromis entre performances et degré de configuration du système à l'exécution. Les interactions entre composants sont limitées à une relation de dépendance exprimée par chaque composant en terme de fournis et de requis au sein d'une interface. Tous les composants sont des entités exécutables et doivent exister lors de l'exécution, cela permet la gestion dynamique des dépendances. La configuration des composants et de leurs dépendances peut être réalisée par programmation ou en utilisant un ADL compatible.

MontiArc [2] est un langage de description d'architectures développé en utilisant le *framework* de conception de DSLs MontiCore [Krahn 10]. Il permet à un architecte de modéliser des architectures logicielles de systèmes cyber-physiques (CPS) en définissant les composants et les connecteurs composant celle-ci [Haber 14]. La simulation d'un système distribué passe par une conception évènementielle des interactions et un typage des ports, des connecteurs et des flux de données entre les composants. La définition de la variabilité des composants d'architecture [Haber 11] basée sur l'héritage facilite la réutilisabilité des composants.

Des enquêtes dédiées aux langages de descriptions d'architectures, à leur adoption par l'industrie et aux défis qui en découlent sont régulièrement menées. Di Ruscio conclut que les ADLs apportent des solutions pertinentes au découpage d'une architecture comme

2. http ://www.monticore.de/languages/montiarc/

un ensemble de composants, de connecteurs et d'interfaces de communication, mais échouent à relever le défi de la séparations des préoccupations métiers des experts domaines [Di Ruscio 10]. Il propose en réponse le *framework* BYADL permettant à un architecte de construire son propre ADL personnalisé à partir d'un ADL existant, en utilisant des opérations de *match and merge*, d'héritage, de référence et d'extension sur les concepts de l'ADL source. Suite à une enquête menée auprès des industriels utilisant des ADLs [Malavolta 13], Malavolta conclut sur (*i*) un besoin d'extensibilité des *frameworks* pour aller vers des ADLs modulaires, (*ii*) un manque de support à la collaboration de domaines, (*iii*) une forte rigidité des ADLs existants, pénalisant l'utilisation des formalismes et des syntaxes utilisées par les architectes.

2.1.5 Discussion sur les approches architecturales

Les approches MVS *synthétiques* visent l'existence d'un méta-modèle global, représentant l'intégralité des points de vue du système, cette hypothèse est contraire à notre contexte où des experts de domaines différents ont besoin de collaborer à la frontière de leurs domaines respectifs. De même, le besoin identifié de gestion de la cohérence inter-domaines nécessite la définition explicite des correspondances entres modèles, ce qui est contraire aux approches *implicites*. Enfin, ces correspondances doivent être exprimées au niveau des méta-modèles pour pouvoir raisonner sur ces liens et maintenir la cohérence, ce qui est incompatible avec les approches *exhaustives*. L'analyse des MVS selon ces critères proposés par [Atkinson 15] montre qu'il n'existe, à notre connaissance, pas d'approche à l'intersection des critères "*projective*, *explicite* et *intentionnelle*" répondant aux exigences motivées dans le Chapitre 1.

Les ADLs offrent la possibilité de modéliser un système comme un ensemble de composants en interaction. Les compositions définies sur ces composants sont les suivantes :

- la dépendance entre composants, sous la forme de services fournis et requis,
- les mécanismes d'héritage, basés sur la réflexivité, pour définir des composants réutilisables au sein de composites,
- l'extension de composant permet la personnalisation du comportement à l'exécution sur des aspects transverses,
- les opérateurs d'appariement permettent de détecter des composants à fusionner pour la définition d'un nouvel ADL à partir d'un existant.

Ces approches permettent, par construction, la génération d'un système exécutable, chaque composant embarquant son code. Toutefois le développement des connecteurs entre composants peut être nécessaire si l'architecte souhaite apporter des composants métiers dans son formalisme habituel. Ces approches définissent les interactions au niveau des instances et sont peu adaptées à la capture de plusieurs domaines pour la collaboration d'experts.

Les AFs, les approches MVS et ADLs permettent le découpage d'un système en sous-systèmes et reposent tous sur des mécanismes de composition des sous-systèmes pour gérer les interactions entre ces derniers. Nous nous proposons donc d'élargir cet état de l'art à l'analyse des mécanismes inhérents à cette problématique de collaboration en prenant la composition comme angle d'analyse.

2.2 Critères de comparaison et familles de composition

2.2.1 Critères de comparaison

Cette section expose des approches différentes, et diverses méthodes qui les implémentent, pour résoudre les difficultés liées à la collaboration de plusieurs domaines dans le cadre de la conception d'un système. Afin de définir un socle commun de comparaison entre les approches étudiées dans cet état de l'art, nous proposons sept critères de comparaison que nous définissons dans cette section.

Niveau d'abstraction : Les approches considérées impliquent l'utilisation d'au moins deux niveaux d'abstraction : le *Type* et l'*Instance*. Certaines considèrent également le *Code* du système final produit. Ce critère permet de caractériser le niveau d'abstraction de la composition, *i.e.,* à quel niveau se situent les éléments manipulés par l'approche pour assurer la collaboration de plusieurs domaines. Une approche préconisant une composition au niveau type, mais qui requière une intervention manuelle au niveau instance, est qualifiée d'*Hybride*.

Ce critère prend valeur dans $\mathbb{N}iveau = \{Type, Instance, Code, Hybride\}$.

Hétérogénéité : La séparation des préoccupations au cœur de ces approches a pour objectif la collaboration entre plusieurs domaines ou aspects d'un même système. Certaines approches dépendent de la conformité de ces sous-systèmes à une même notation, *e.g.,* des méta-modèles conformes au même méta-méta-modèle, ces sous-systèmes sont dits *Homogènes*. Au contraire, si aucune hypothèse n'est faite sur la conformité des sous-systèmes à une même notation, ceux-ci sont *Hétérogènes*. D'autres approches définissent un sous-ensemble statique de notations auxquelles les sous-domaines peuvent être conformes, les sous-systèmes ne sont donc ni homogènes ni libres de toute notation, ils sont donc *Partiellement* hétérogènes. Nous souhaitons caractériser ici l'hypothèse faite sur cette conformité.

Ce critère prend valeur dans $\mathbb{H}étérogénéité = \{Hétérogène, Homogène, Partielle\}$.

Symétrie : L'ensemble des sous-systèmes collaborants doit être composé pour obtenir un système final. Cette composition peut être *Asymétrique*, *i.e.,* basée sur une hypothèse de rôles fixés entre les différents sous-systèmes, à la manière des approches maître-esclaves. Au contraire, elle peut être libre de toute hypothèse sur les relations entre sous-systèmes et dite *Symétrique*. Pour les approches utilisant un opérateur de composition, cela revient à évaluer la commutativité de cet opérateur.

Ce critère prend valeur dans $\mathbb{S}ymétrie = \{Symétrique, Asymétrique\}$.

Arité : Il est possible d'exprimer la composition comme :

- une opération *Binaire* composant les sous-systèmes deux à deux,
- une opération *N-aire* considérant d'un seul coup l'ensemble des sous-systèmes,
- une organisation *Ad Hoc* définissant statiquement les interactions d'un ensemble de sous-systèmes.

Ce critère prend valeur dans $\mathbb{A}rité = \{Binaire, N\text{-}aire, Ad\ Hoc\}$.

Variabilité : Ce critère évalue la capacité de gestion de la multiplicité des cibles atteignables dans l'espace de solution. Le lien entre le niveau des instances et celui du code peut être effectué par :

— un lien définissant une cible de génération unique pour une instance donnée de façon *Directe*,

— un système d'*Appariement* exploitant les propriétés requises par l'instance et fournies par les éléments de code.

Ce critère prend valeur dans $\mathbb{V}ariabilité = \{Directe, Appariement\}$.

Automatisation : Le degré d'automatisation d'une approche est complexe à évaluer. Nous cherchons à estimer l'effort automatisé par l'approche par rapport à l'effort requis par l'utilisateur, pour ce qui concerne la collaboration des sous-systèmes. Nous utilisons le principe de Pareto pour caractériser deux classes d'automatisation. Les approches offrant une forte automatisation, *i.e.,* où 80% de la composition est produit par 20% d'effort, sont dites *80-20*. Au contraire les approches dites *20-80* nécessitent une forte intervention de la part d'un architecte.

Ce critère prend valeur dans $\mathbb{A}utomatisation = \{80\text{-}20, 20\text{-}80\}$.

Cohérence : Dans un domaine, la conformité des modèles à leur méta-modèle est assurée par les méthodes classiques du MDE. Nous cherchons ici à caractériser les apports en matière de cohérence inter-domaine. Cette cohérence peut faire l'hypothèse de l'existence d'un méta-modèle unifié regroupant tous les concepts du système, assurant la *Conformité* des modèles à cette représentation du système par les méthodes classiques. Certaines approches souhaitent assurer une *Synchronisation* d'instances conformes à des types différents, *i.e.,* propager une modification d'un sous-système à un autre si deux éléments de ces sous-systèmes sont définis comme équivalents. Cela convient aussi dans le cas où les aspects structurels et comportementaux des domaines sont chacun capturés au sein de modèles dédiés. Enfin, il est possible de définir des instances génériques sur étagère, réutilisables par le concepteur pour modéliser son sous-système et dont la cohérence dépend de la vérification de la chaîne de *Dépendance*.

Ce critère prend valeur dans $\mathbb{C}ohérence = \{Conformité, Synchronisation, Dépendance\}$.

2.2.2 Familles de composition

Les approches de séparation de préoccupations détaillées dans la Section 2.1 reposent toutes sur la composition pour gérer les interactions entre les sous-systèmes. Les deux sections suivantes détaillent les méthodes existantes pour composer des (méta-)modèles et des langages. Afin de faciliter la lecture, nous proposons une répartition des contributions dans ces domaines en cinq familles, illustrées par la Figure 2.1 :

la Fusion vise la production d'un méta-modèle unique, unifiant les concepts des différents domaines au sein d'un seul paradigme ;

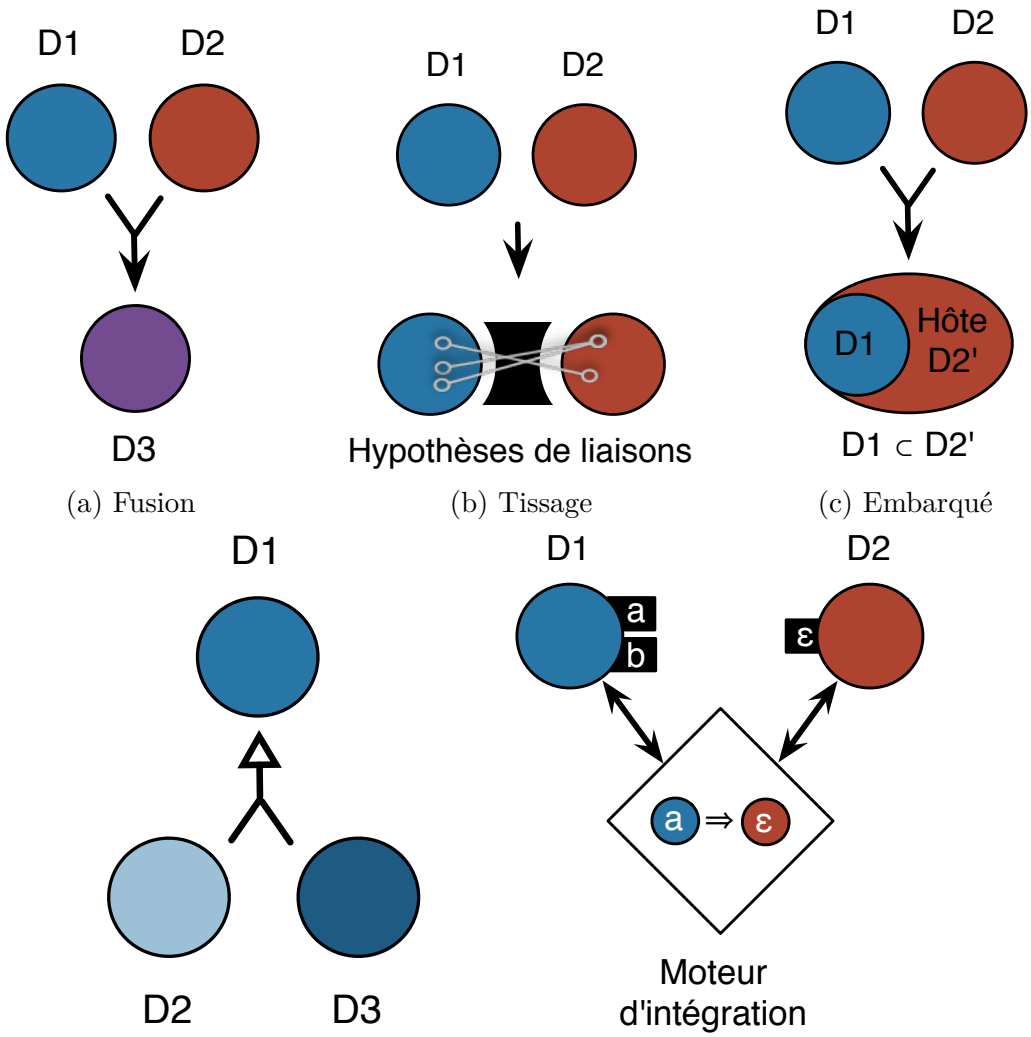
le Tissage permet la conception de préoccupations transverses réutilisables et leur composition au sein de modèles métiers ;

l'Embarqué est l'adaptation d'un méta-modèle à un autre, l'opération est intrusive pour le modèle hôte mais pas pour le modèle embarqué ;

l'Héritage est un mécanisme de réutilisation de méta-modèles ou de fragments basé sur le typage ;

l'Intégration repose sur l'arbitrage des intégrations entre méta-modèles par une entité externe.

Les méthodes utilisant plusieurs mécanismes de composition sont caractérisées comme hybrides.



(a) Fusion (b) Tissage (c) Embarqué

(d) Héritage (e) Intégration

FIGURE 2.1 – Schémas des approches de composition regroupées en cinq familles

2.3 Approches par composition de modèles

Nous nous intéressons dans cette section aux approches de composition de modèles. Les contributions de chaque famille seront présentées et caractérisées en fonction des critères exposés dans la Section 2.2.1.

2.3.1 Fusion

Nous présentons dans cette section les approches de composition basées sur la fusion, c'est-à-dire basée sur la production d'un modèle unique à partir de modèles existants. Les approches regroupées dans cette famille peuvent décrire un processus de fusion ou bien reposer sur l'existence d'un modèle déjà fusionné. Cette fusion peut se concevoir au niveau méta, au niveau des modèles ou sur le code.

Kompose propose une composition de fusion de méta-modèles homogènes [Fleurey 07, France 07]. L'opérateur est indépendant du langage de modélisation utilisé pour les méta-modèles en entrée. Cette opération se décompose en deux étapes :

1. Apparier les éléments. Deux éléments de chaque méta-modèle peuvent être appairés d'après leur signature. La signature d'un élément définit les éléments structuraux à utiliser pour le comparer aux autres éléments.
2. Fusionner les méta-modèles. Deux éléments appariés mènent à la création d'un seul élément fusionné regroupant les caractéristiques des deux. Les éléments non-appariés sont ajoutés tels quels au méta-modèle résultant.

Il est possible d'appliquer des directives [Reddy 06] préliminaires sur les méta-modèles. Elles spécifient des modifications simples pour forcer ou interdire l'appariement en renommant, supprimant ou ajoutant des éléments. De même, des directives postérieures peuvent être appliquées sur le méta-modèle fusionné pour le finaliser. La gestion des erreurs permet de vérifier la cohérence d'un élément fusionné vis à vis de ses éléments appariés. L'opérateur de Kompose peut être spécialisé pour fonctionner avec tout méta-modèle conforme à l'Essential Meta-Object Facility (EMOF) de l'OMG. Il s'agit d'une composition binaire et symétrique, définie au niveau des types, pour des modèles homogènes (EMOF).

Kurpjuweit propose de séparer les concepts nécessaires à la conception d'un système en plusieurs domaines isolés, possédant chacun son sous-ensemble de spécifications et un fragment de méta-modèle pour y répondre. Tous les fragments de méta-modèles sont ensuite fusionnés en un seul méta-modèle [Kurpjuweit 07], pour fournir une solution complète au système considéré. L'approche exposée est voulue indépendante de la technologie choisie, l'implémentation de l'opérateur n'est donc pas détaillée. Il s'agit d'une composition n-aire et symétrique, définie au niveau des types qui repose sur l'hypothèse que tous les fragments sont homogènes car conformes au même méta-modèle.

Ring est un méta-modèle unifié au sein du projet Pharo [Black 10] [3]. Il vise à diminuer l'effort de maintenance des différents méta-modèles co-existants pour la gestion des versions de code [Gómez 12], *e.g.,* Monticello, Store et l'API réflexive de Smalltalk. En

3. http ://www.pharo-project.org

proposant un unique méta-modèle unifié, auquel les autres modèles actuellement utilisés doivent se conformer, Ring facilite le passage d'un outil à l'autre et leur interopérabilité. L'approche unifie six modèles de code source et permet une abstraction de leurs opérations, assurant ainsi de pouvoir synchroniser la gestion des versions d'une instance dans les autres modèles de code. Il s'agit d'une composition n-aire et symétrique, définie au niveau des types, pour des modèles conformes à un méta-modèle fusionné au préalable.

Boucké expose le problème de la redondance de spécifications dans les modèles décrivant les architectures [Boucké 10]. Il propose d'intégrer les relations de composition au sein même de l'ADL pour permettre à l'architecte de concevoir plus efficacement ses modèles. Un opérateur de sous-typage permet d'étendre un élément de modèle via un ensemble de concepts dans un autre modèle. Une composition par fusion permet de spécifier la possibilité d'unifier deux éléments de modèles appartenant à des modèles différents. L'identification des relations est à la charge de l'architecte, mais permet de diminuer la taille globale du système modélisé et de vérifier la conformité statique d'éléments de modèles. Il s'agit d'une composition n-aire et symétrique, définie au niveau des instances, sur des éléments de modèles conformes aux notations prévues par l'ADL.

EML pour *Epsilon Merging Language*, est un langage défini dans la plateforme Epsilon et dédié à la fusion de modèles [Kolovos 06].. L'opérateur y est décomposé en quatre phases :

1. la phase de comparaison qui identifie les correspondances entre les éléments de modèles équivalents,
2. la phase de vérification de la conformité identifiant parmi les correspondances les possibles conflits de fusion,
3. la phase de fusion, prenant en entrée une correspondance et retournant un unique élément de modèle,
4. la phase de réconciliation, pour finaliser le modèle fusionné en supprimant les hypothétiques inconsistances induites par la fusion.

EML repose sur la définition de différents types de règles : les règles de *match*, les règles de fusion et les règles de transformation. Les premières comparent deux modèles conformes à deux méta-modèles connus, et retournent vrai si les modèles sont compatibles et conformes l'un à l'autre, formant une correspondance. Ces règles contiennent donc la sémantique de cohérence inter-domaine. Les règles de fusion sont utilisées sur les correspondances et fusionne les éléments équivalents. Les règles de transformation traitent les éléments restant pour les traduire dans la notation du modèle résultant. L'originalité d'EML est de permettre la fusion de modèles hétérogènes. Cependant, le modèle fusionné résultant n'est conforme qu'à l'un ou l'autre des méta-modèles d'entrée. Il s'agit d'une composition binaire et asymétrique, définie au niveau des types sur des modèles hétérogènes.

ModelBus définit un opérateur de fusion dédié aux modifications concurrentes dans une base de modèles[Sriplakich 08]. L'approche attaque le problème de passage à l'échelle dans la modélisation collaborative. Dans ce contexte, chaque contributeur copie un sous-ensemble des modèles du dossier commun, les analyse puis les modifie, créant une version locale qui est ensuite fusionnée dans le dossier commun. L'opérateur proposé repose sur le concept de delta, représentant les différences entre deux versions d'un modèle, *i.e.,* le

modèle de base et le modèle local. L'opérateur prend en entrée un ensemble de modèles hétérogènes, mais la fusion est définie sur une paire de modèles hétérogènes. Pour chaque modèle de l'ensemble, ModelBus calcule les deltas entre le modèle de base et le modèle local, puis les intègre dans le modèle de base du dossier commun. Puisque celui-ci a pu être modifié depuis par un autre contributeur, ModelBus définit une opération de recouvrement pour retrouver un élément spécifique du modèle local dans le nouveau modèle de base. La cohérence au sein de l'ensemble des modèles passé en entrée est assurée par la préservation de liens inter-modèles. L'opérateur vérifie pour chaque lien l'existence des deux extrémités au sein du modèle de base, et utilise l'opérateur de recouvrement le cas échéant. Bien que le but de cette composition diffère de notre contexte, elle adresse des problématiques complémentaire dont notre approche pourrait bénéficier. Il s'agit d'une composition n-aire et asymétrique, définie au niveau des types sur des modèles hétérogènes.

2.3.2 Tissage

Nous présentons dans cette section les approches de modélisation orientées aspect (*Aspect Oriented Modeling* ou AOM), c'est-à-dire basée sur le tissage de modèles, ou fragments de modèles, transverses au sein d'un modèle domaine. Les techniques de développement et modélisation orientées aspect proposent de découper un système étudié en différentes préoccupations, *e.g.,* séparer la gestion de la persistance, la sécurité et l'interface homme-machine. Les solutions à ces préoccupations, appelées aspects, sont conçues pour être génériques d'un point de vue domaine et réutilisables sur différents systèmes. Le système final est obtenu par un mécanisme de *tissage*. Ces approches détectent des *points de jonction* au sein du *modèle de base* où effectuer les transformations. Les transformations prennent la forme de modèles dits *advice* et sont tissés sur les *points de jonction* sélectionnés par la coupe. Les approches AOM sont donc par définition asymétriques.

CORE[4] (anciennement RAM, Modèles à Aspects Réutilisables [Kienzle 09]) propose d'intégrer différentes préoccupations d'un système modélisé par des modèles hétérogènes [Schöttle 15], *e.g.,* des diagrammes de classes, des diagrammes de séquences, des diagrammes d'états, des diagrammes d'impact et des modèles de variabilité. L'approche permet de représenter à la fois la structure et le comportement de chaque préoccupation du système. Le système de dépendance entre aspects permet de réutiliser des aspects plus simples pour définir de nouveaux aspects réutilisables plus complexes. Le moteur de tissage est chargé de gérer la cohérence des dépendances et de l'instanciation des aspects pour satisfaire les préoccupations définies dans le système étudié. Un effort de gestion de la cohérence inter-vues est fourni, permettant par exemple de vérifier que la séquence d'échange définie dans un diagramme de séquence est cohérente avec les transitions d'états décrites dans le diagramme de machine à états. Enfin, un systèmes de variantes permet de spécifier plusieurs variants fonctionnels et leurs impacts respectifs pour chaque préoccupation modélisée à travers une interface de variation [Schöttle 16]. Le choix d'un variant pertinent est outillé par l'utilisation de *feature models* représentant la hiérarchie des fonctionnalités et les contraintes entre elles. Il s'agit d'une composition n-aire et asymétrique, utilisant les niveaux des types et des instances.

4. http ://touchcore.cs.mcgill.ca/

GeKo propose d'appliquer les techniques de composition orientée aspects aux lignes de produits logicielles (SPLs) [Morin 08], puis a été étendu pour être utilisé sur tout méta-modèle conforme à EMOF [Klein 12]. Le système d'identification de la partie du modèle de base concernée, appelée point de jonction, est complètement automatisé par l'utilisation d'un moteur d'intégration à base de règles [5]. L'appariement entre le point de jonction ainsi détecté et le modèle *advice* à appliquer peut nécessiter l'intervention de l'utilisateur. Ces appariements sont conçus au niveau des instances et nécessitent que le modèle de base, le modèle de points de jonction et le modèle *advice* soient conformes au même méta-modèle. L'approche n'est toutefois pas spécifique à un seul méta-modèle. L'application sur les SPLs permet une gestion fine de la variabilité des artefacts de code à générer. Il s'agit d'une composition n-aire et asymétrique, définie au niveau des instances sur des modèles homogènes conformes à EMOF.

SmartAdapters est une approche à base d'aspects dédiée à l'adaptation [Perrouin 12]. Elle a défini des mécanismes de tissage au niveau du code, puis a été étendue au niveau des modèles. Un aspect est composé de trois éléments : un modèle cible définissant où l'aspect doit être tissé (où), une définition structurelle de la préoccupation de l'aspect (quoi) et une description du protocole de tissage de l'aspect (comment). A la différence d'un modèle de points de jonction, les modèles cibles utilisés sont des fragments de modèle qui n'ont pas de référence directe vers des éléments de modèles. Ils définissent un schéma, abstrait des contraintes du méta-modèle considéré. L'appariement est automatisé par l'utilisation d'un moteur de *pattern matching*, mais l'utilisateur doit sélectionner manuellement la coupe, *i.e.,* le sous-ensemble des points de jonction sur lesquels il souhaite tisser l'aspect. Les modèles de base, de points de jonction et d'*advice* doivent être conformes au même méta-modèle, mais l'approche est générique. Il s'agit d'une composition n-aire et asymétrique, définie au niveau des types sur des modèles homogènes.

MATA propose une approche de tissage basée sur les mécanismes de transformation de graphes pour les modèles UML [Whittle 09]. Un modèle MATA contient à la fois la partie gauche et la partie droite des règles de transformation. Ces règles sont définies sur les modèles UML et utilisent les stéréotypes `Create` et `Delete` pour manipuler les éléments de modèles. Les éléments peuvent être définis comme composables ou non selon l'utilisation du stéréotype `Context`. Le modèle de base et le modèle de transformation MATA sont transformés en graphes. L'approche tire profit de la théorie des graphes pour analyser les interactions entre les aspects et les fonctionnalités des modèles durant le tissage. Le graphe résultant est ensuite retransformé en UML. Il s'agit d'une composition binaire et asymétrique, définie au niveau des instances sur des modèles homogènes conformes à UML.

2.3.3 Embarqué

Nous présentons dans cette section les méthodes de spécialisation d'un modèle hôte à partir d'un modèle embarqué. Dans ces approches embarquées, ou d'extension, le second modèle demeure intact tandis que le premier est adapté pour la cohabitation. Elles sont par définition asymétriques.

5. http ://www.drools.org/

Schottle définit un opérateur de composition embarquée qui permet d'étendre son utilisation à des méta-modèles hétérogènes [Schottle 13]. Il préserve l'intégrité d'un des deux méta-modèles en entrée. Le second méta-modèle est adapté, créé par-dessus le premier en unifiant les concepts des deux domaines. La facilité d'intégration est dépendante de l'alignement des méta-modèles et du niveau de détail relatif entre le méta-modèle embarqué et l'hôte. L'approche permet toutefois une meilleure réutilisabilité non-intrusive des méta-modèles existants. Il s'agit d'une composition binaire et asymétrique, définie au niveau des types sur des modèles hétérogènes.

Brunelière propose un DSL dédié à la définition d'extension de méta-modèles pour faciliter la réutilisation de ces derniers [Bruneliere 15]. Partant d'un méta-modèle existant dit originel, le concepteur définit des extensions de méta-modèles en ajoutant, modifiant ou filtrant des éléments du méta-modèle originel. De l'application d'une ou plusieurs extensions résulte un méta-modèle dit étendu, qui peut à son tour servir de base pour une autre extension. Cette composition n'est pas intrusive pour le méta-modèle originel et ne nécessite aucune migration ni transformation sur les modèles conformes aux méta-modèles étendus. Les modèles développés au préalable sont conservés et l'utilisateur du méta-modèle étendu travaille sur une projection rendue conforme par application des extensions à l'exécution. Il s'agit d'une composition binaire et asymétrique, définie au niveau des types sur des modèles homogènes conformes au méta-modèle originel.

I-MMD est une approche de développement modulaire de méta-modèles à base d'interfaces [Živković 15]. Elle permet la réutilisation en boite noire de méta-modèles ou fragments de méta-modèles existants. Les nouveaux méta-modèles composites sont construits autour, en utilisant les capacités du fragment réutilisé qu'à travers les fonctionnalités exposées par l'interface. Le mécanisme de sous-typage d'interface permet une gestion de la conformité des modèles composites vis-à-vis des spécifications des fragments réutilisés. L'approche est générique, indépendante du langage de modélisation utilisé, mais le composite doit être conforme au même méta-modèle que le fragment qu'il étend. Il s'agit d'une composition n-aire et asymétrique, définie au niveau des types sur des modèles homogènes.

2.3.4 Intégration

Nous présentons dans cette section les méthodes basées sur la définition d'un concept dédié régissant les interactions entre méta-modèles. Cet élément de lien inter-domaine n'est pas conforme au même méta-méta-modèle que les représentations de domaines elles-mêmes et ne donne pas lieu à l'existence d'un méta-modèle unique.

Openflexo propose une approche de synchronisation de modèles hétérogènes en séparant l'espace de modélisation en deux [Golra 16a]. L'espace technologique est spécifique à chaque méta-modèle et permet son utilisation et son outillage pour produire des modèles conformes par l'expert. L'espace conceptuel est constitué d'un ensemble de modèles fédérés pour développer un *modèle virtuel*. Le *modèle virtuel* regroupe les concepts communs aux différents domaines. Le contrat de communication qui permet la synchronisation entre ces deux espaces prend la forme de connecteurs bidirectionnels. Les experts travaillent sur leurs espaces technologiques respectifs et l'intégration se fait dans l'espace conceptuel, qui permet de maintenir un *modèle virtuel* se situant à l'intersection des domaines considérés,

grâce à la synchronisation. L'utilité de la fédération de modèles est exemplifiée sur le cas de l'ingénierie des besoins [Golra 16b]. Il s'agit d'une composition n-aire et asymétrique, définie au niveau des types sur des modèles hétérogènes.

2.3.5 Hybride

KerTheme est une extension de Theme [Baniassad 04], l'approche orientée aspect d'UML. Elle permet la définition de deux types de préoccupations : les préoccupations de base et les préoccupations d'aspect [Barais 08]. Les premières représentent les préoccupations domaines, tandis que les secondes sont transverses par définition. Deux mécanismes de composition différents sont utilisés. Le premier sert à composer deux préoccupations de base par fusion, le second permet de tisser une préoccupation d'aspect sur une basique. Chaque préoccupation étant constituée d'un modèle structurel (*i.e.,* un diagramme de classes) et d'un comportemental (*i.e.,* un scénario), deux opérateurs sont proposés pour chaque type composition. Il s'agit de compositions binaires, définies au niveau des types. La fusion est symétrique et prend en entrée des modèles homogènes. Le tissage est asymétrique et considère des modèles hétérogènes.

MEGAF propose d'utiliser les techniques de méta-modélisation pour permettre aux architectes de concevoir leur propre framework d'architecture [Hilliard 12]. Cela a pour but d'augmenter la réutilisabilité des vues architecturales communes à plusieurs AFs. Cette contribution se découpe en deux phases : *(i)* la définition d'un nouvel AF en créant de nouveaux points de vues et en définissant leurs correspondances par intégration, *(ii)* la personnalisation d'un AF pré-existant pour un domaine d'application spécifique par embarquement. Les correspondances entre points de vues sont augmentées par des règles spécifiques à chaque nouvel AF que l'architecte doit écrire pour améliorer la gestion de la cohérence inter-vues. De même, les mécanismes de composition entre les points de vues ajoutés et ceux existant dans MEGAF sont à la charge de l'architecte. Il s'agit de compositions n-aires et asymétriques, définies au niveau des types sur des modèles hétérogènes.

TABLE 2.2 – Matrice de caractérisation des approches de composition de modèles

Familles	**Méthodes**	**Critères de comparaison**						
		Abstraction	**Hétéro-généité**	**Symétrie**	**Arité**	**Variabilité**	**Automa-tisation**	**Cohérence**
Fusion	Kompose [Fleurey 07]	Type	Homogène	Symétrique	Binaire	N/A	80-20	Conformité
	Ring [Gómez 12]	Type	Partielle	Symétrique	N-aire	Directe	80-20	Synchronisation
	Boucké [Boucké 10]	Instance	Partielle	Symétrique	N-aire	N/A	80-20	Conformité
	Kurpjuweit [Kurpjuweit 07]	Type	Homogène	Symétrique	N-aire	N/A	20-80	Conformité
	EML [Kolovos 06]	Type	Hétérogène	Asymétrique	Binaire	N/A	20-80	Conformité
	ModelBus [Sriplakich 08]	Type	Hétérogène	Asymétrique	N-aire	N/A	80-20	Synchronisation
	KerTheme [Barais 08]	Type	Partielle	Asymétrique	Binaire	N/A	80-20	Synchronisation
Tissage	CORE [Kienzle 09]	Type Instance	Partielle	Asymétrique	N-aire	Appariement	80-20	Dépendance Synchronisation
	SmartAdapters [Perrouin 12]	Type	Homogène	Asymétrique	N-aire	Appariement	20-80	Conformité
	MATA [Whittle 09]	Instance	Homogène	Asymétrique	Binaire	N/A	80-20	Conformité
	Geko [Morin 08] [Klein 12]	Instance	Homogène	Asymétrique	N-aire	Appariement	80-20	Conformité
Embarqué	Brunelière [Bruneliere 15]	Type	Homogène	Asymétrique	Binaire	N/A	20-80	Conformité Synchronisation
	I-MMD [Živković 15]	Type	Homogène	Asymétrique	N-aire	N/A	80-20	Conformité
	Schottle [Schottle 13]	Type	Hétérogène	Asymétrique	Binaire	N/A	80-20	Synchronisation
Intégration	MEGAF [Hilliard 12]	Type	Hétérogène	Asymétrique	N-aire	N/A	20-80	Synchronisation
	Openflexo [Golra 16a]	Type	Hétérogène	Asymétrique	N-aire	N/A	20-80	Synchronisation

2.4 Approches par composition de langages

Nous nous intéressons ici aux mécanismes définis dans les approches de composition de langages de modélisation spécifiques au domaine (DSMLs). Ces langages reposent sur une syntaxe abstraite définissant les concepts abstraits et les relations entre eux. La composition de DSMLs repose sur des mécanismes proches de la composition de modèles et nous proposons de guider la lecture en utilisant le même regroupement en familles exposées dans la Section 2.2.2. [6] Cette section présente les contributions de chaque famille et les caractérise chacune en fonction des critères exposés dans la Section 2.2.1.

2.4.1 Fusion

Nous présentons dans cette section l'équivalent des approches de fusion de la Section 2.3.1 du point de vue des langages. Ces méthodes sont basées sur l'existence ou la production d'un DSML unique regroupant les concepts des différents domaines.

Di Natale propose une approche pour composer un DSML décrivant les aspects fonctionnels d'un système avec un DSML dédié au déploiement de l'application [Di Natale 14]. L'approche repose sur la production d'un langage unique, intégrant les deux DSMLs en entrée et appariant des éléments syntaxiques du langage fonctionnel avec des éléments syntaxiques du langage de déploiement, afin de lier des fonctionnalités à des ressources spécifiques de la plateforme sur laquelle sera déployé le système. Cette composition inter-domaine repose sur des connecteurs spécifiques au langage d'appariement. Ceux-ci embarquent également la connaissance nécessaire à la génération de code conforme aux deux DSMLs, à la manière des ADLs. Il s'agit d'une composition symétrique, définie au niveau des instances conformes à un ensemble ad hoc de DSLs.

ThingML est un langage de modélisation spécifique à la conception d'applications par dessus l'internet des objets (IoT) [Harrand 16, Morin 17]. Il définit la relation entre les *Things*, contenues dans les *Configurations* qui représentent les applications. Des points d'extensions sont définis sur plusieurs aspects de ces deux concepts. Ils utilisent deux langages, respectivement dédiés aux *Things* et aux *Configurations*, permettant d'assurer la génération de code, comme illustré par les points 1 à 8 de la Figure 2.2. Le modèle d'architecture permet à un concepteur de définir son application à partir de composants, de ports, de connecteurs et de messages asynchrones. La logique métier des composants est ensuite définie à l'aide d'un langage d'actions spécifique à ThingML et de diagrammes à états gérant le parallélisme et les états composites. Le langage d'actions définit les éléments comportementaux du composant (*e.g.,* les variables et les fonctions) et permet à la machine à états d'envoyer et de recevoir des messages via les ports du composant. La composition de ces différents langages de modélisation utilisés repose sur l'existence du méta-modèle unifié ThingML. Il s'agit d'une composition asymétrique, définie au niveau des types conformes à un ensemble ad hoc de DSLs.

6. Nous utilisons le terme *intégration* comme défini par la communauté des architectures orientées services (SOA) durant cette thèse.

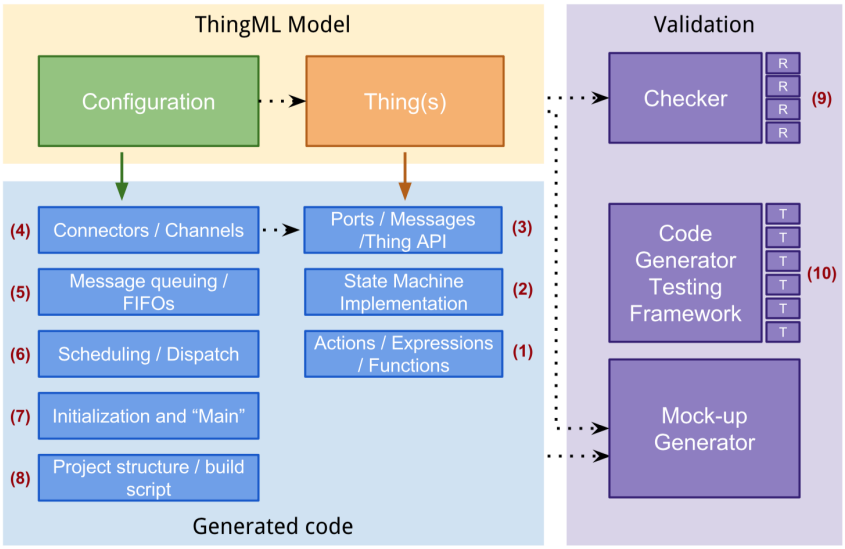


FIGURE 2.2 – Points d'extension de ThingML [Harrand 16].

2.4.2 Embarqué

Nous présentons dans cette section l'équivalent des approches d'extension embarquée de la SECTION 2.3.3 du point de vue des langages. Ces méthodes visent la spécialisation d'un langage hôte à un langage embarqué et sont par définition asymétriques.

Ptolemy et ModHel'X proposent deux approches similaires de coordination d'acteurs exécutables à base de langages comportementaux [Buck 94, Boulanger 08]. Ils permettent la définition d'acteurs composites, composés d'acteurs embarqués et synchronisés par un modèle comportemental appelé directeur. Un acteur composite dirigé par une machine à états finis (FSM) peut donc être composé avec d'autres acteurs au sein d'un nouvel acteur composite hôte par un directeur en flux de données synchronisés (SDF). Ces approches permettent donc d'organiser l'exécution d'instances de langages comportementaux hétérogènes de façon hiérarchique. Il s'agit de compositions n-aires asymétriques, définies au niveau des instances sur des modèles hétérogènes.

2.4.3 Intégration

Nous présentons dans cette section l'équivalent des approches de la SECTION 2.3.4 du point de vue des langages. Ces méthodes sont basées sur la définition d'éléments de liaisons externes aux DSLs à composer et dédiés aux interactions entre ces DSLs.

BCOoL est un langage de composition de DSMLs proposé dans le cadre de l'initiative GEMOC. A travers la définition d'interfaces comportementales pour chaque DSML composé, BCOoL permet de gérer l'intégration de leur exécution [Larsen 15]. Le langage d'interfaces comportementales est évènementiel, spécifique au domaine du DSML associé et expose un sous-ensemble de la syntaxe abstraite de ce DSLM comme contexte. Comme illustré au niveau langage de la FIGURE 2.3, les spécifications de coordinations entre ces interfaces sont exprimées dans le langage BCOoL, pour chaque paire de DSMLs comportementaux hétérogènes et utilisent comme élément commun les évènements et les états de

ces modèles. Les règles de coordination qui sont ensuite générées permettent de coordonner leur exécution, par exemple en définissant des points de rendez-vous où la simulation d'exécution des modèles produits doit se synchroniser. L'intégration des DSMLs conçue avec BCOoL peut être simulée directement au sein de l'environnement GEMOC Studio, comme illustré au niveau *runtime* de la FIGURE 2.3, intégré pleinement au *framework* de modélisation d'Eclispe (EMF) [Steinberg 09]. Il s'agit d'une composition binaire et symétrique, définie au niveau des types sur des DSLs hétérogènes.

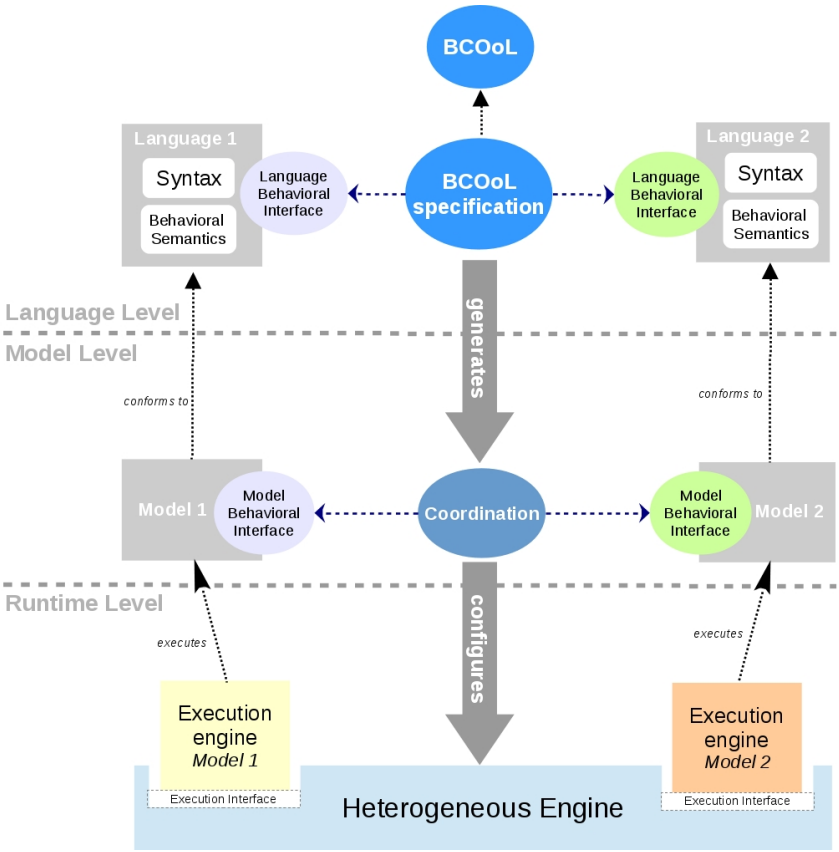


FIGURE 2.3 – Vision d'ensemble des contributions de BCOoL [Vara Larsen 16].

Kermeta propose un environnement de développement et d'exécution dédié aux DSLs qui utilise un méta-modèle par aspect considéré [Muller 05, Jézéquel 15]. Trois aspects sont proposés : la syntaxe abstraite sous Ecore, la sémantique statique avec OCL et la sémantique comportementale avec Kermeta. Les modèles de trois aspects d'un DSL sont transformés en code et la composition s'effectue à ce niveau. Les auteurs présentent les conditions à remplir pour le langage de programmation cible et défendent l'utilisation de Scala pour sa proximité avec OCL et Kermeta, les capacités de composition offertes par le langage et l'efficacité du programme généré. Kermeta permet donc à un concepteur de gérer les aspects non-fonctionnels de son DSL en dehors du méta-modèle tout en s'assurant un résultat exécutable et performant. Il s'agit d'une composition asymétrique, fonctionnant au niveau du code sur trois DSLs pré-définis.

2.4.4 Hybride

MPS est un environnement de développement pour la conception et l'exécution de DSMLs [Voelter 12]. Il permet de définir la structure d'un langage, le comportement ainsi que la syntaxe concrète de chacun de ses éléments, via un éditeur projectionel plutôt qu'un éditeur de texte. MPS [7] propose deux types de compositions. La première, basée sur les approches d'extension embarquée, permet la réutilisation d'un ou plusieurs DSMLs existants au sein d'un nouveau DSML hôte dont la structure et le comportement peuvent exprimer des dépendances vers les langages embarqués. La seconde composition proposée est basée sur l'héritage et permet de définir des familles de DSMLs en spécialisant les éléments de langage du DSML père au nouveau contexte du DSML fils, permettant de définir des DSML modulaires. Il s'agit de compositions asymétriques définies au niveau des types sur des DSLs hétérogènes.

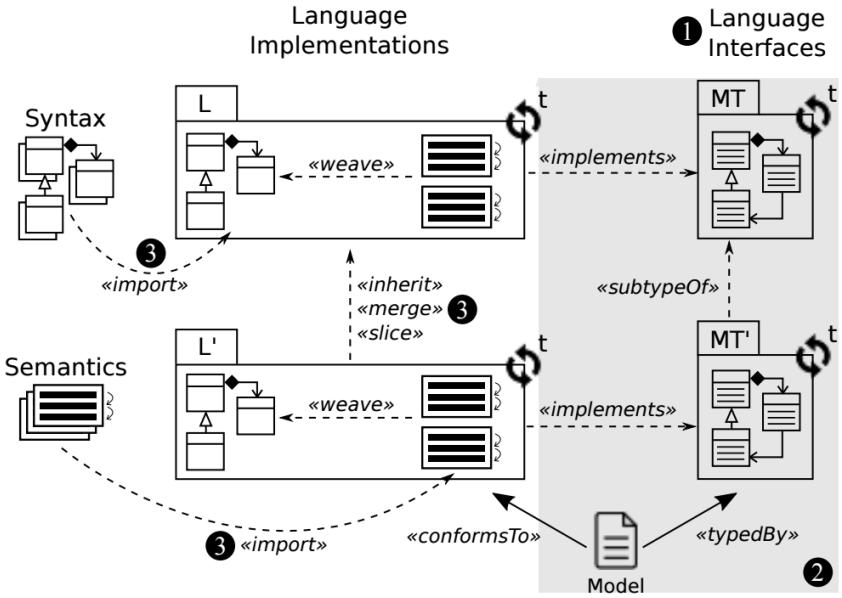


FIGURE 2.4 – Vision d'ensemble des contributions de Melange [Degueule 15].

Melange est une approche de composition facilitant la réutilisation de DSMLs existants [Degueule 15]. L'approche a pour but d'améliorer la modularité des DSMLs, de faciliter leur évolution en minimisant son impact négatif sur leur utilisation et l'interopérabilité avec les outils externes. Elle repose sur la définition d'interfaces de langages qui permettent d'exposer un aspect du langage dans un formalisme indépendant de l'implémentation du DSML, comme illustré sur la FIGURE 2.4 (❶). L'utilisation du DSML se base sur son interface, cela permet de faire évoluer son implémentation sans incidence sur son interopérabilité tant que le langage implémente l'interface, à la manière d'une interface en Java. L'approche permet également la définition de familles de DSLMs, définissant une hiérarchie de typages entre interfaces de langages qui partagent des communalités en se reposant sur les notions de polymorphisme de famille et de typage structurel au niveau des DSMLs. Un DSML peut implémenter une interface de langage, et une interface de langage peut être un sous-type d'une autre interface. Un modèle est donc conforme à une

7. https ://www.jetbrains.com/mps/

implémentation de langage et typé par une interface de langage (❷). Enfin, des opérations de composition entre implémentations de DSMLs sont proposées pour faciliter la réutilisation d'implémentation, notamment l'héritage, la fusion et la sélection d'un sous-ensemble d'éléments du langage (❸). Le concepteur peut donc créer un nouveau DSML adapté à son domaine en réutilisant l'implémentation des DSMLs existants qui implémentent les interfaces souhaitées, via des opérateurs d'héritage, de fusion et de sélection. Il s'agit de compositions n-aires asymétriques définies au niveau des types sur des DSLs hétérogènes.

TABLE 2.3 – Matrice de caractérisation des approches de composition de DSMLs

Familles	**Méthodes**	**Critères de comparaison**						
		Abstraction	**Hétéro-généité**	**Symétrie**	**Arité**	**Variabilité**	**Automa-tisation**	**Cohérence**
Fusion	ThingML [Harrand 16] [Morin 17]	Type	Partielle	Asymétrique	Ad Hoc	Appariement	80-20	Dépendance Conformité
	Di Natale [Di Natale 14]	Instance	Partielle	Symétrique	Ad Hoc	Directe	80-20	Conformité
Héritage	Melange [Degueule 15]	Type	Hétérogène	Asymétrique	N-Aire	N/A	80-20	Conformité
	MPS [Voelter 12]	Type	Hétérogène	Asymétrique	N-Aire	N/A	80-20	Dépendance Conformité
Embarqué	Ptolemy [Buck 94]	Instance	Hétérogène	Asymétrique	N-Aire	Directe	20-80	Synchronisation
	ModHel'X [Boulanger 08]	Instance	Hétérogène	Asymétrique	N-Aire	Directe	20-80	Synchronisation
Intégration	BCOoL [Larsen 15]	Type	Hétérogène	Symétrique	Binaire	Directe	20-80	Synchronisation
	Kermeta [Jézéquel 15]	Code	Partielle	Asymétrique	Ad Hoc	Directe	80-20	Conformité

2.5 Discussion

Les sections précédentes détaillent les contributions en matière de composition de (méta-)modèles et de langages spécifiques aux domaines. Nous discutons ici de l'espace couvert par l'état de l'art et identifions un manque à combler quant à la composition et la concrétisation de méta-modèles isolés. Ensuite, nous en déduisons les enjeux auxquels cette thèse doit répondre.

2.5.1 Espace couvert par l'état de l'art

L'analyse de l'état de l'art en fonction des critères exposés dans la Section 2.2.1 est résumée au sein de la Table 2.4. Nous avons identifié dans le Chapitre 1 la nécessité de pouvoir composer des domaines en préservant leur isolation. Afin de permettre la collaboration de différents domaines tout en assurant la génération d'un système exécutable, il est nécessaire de raisonner sur la composition à un haut niveau d'abstraction et d'automatiser la gestion de la variabilité des artefacts concrets. Aucune d'hypothèse n'est faite sur la nature des représentations de domaine (*i.e.,* comportementales, structurelles, ni à quel méta-modèle chacune se conforme). Leur composition doit donc pouvoir gérer les interactions de plusieurs domaines simultanément, sans rôle prédéfini dans les opérandes et sans hypothèse d'un méta-modèle commun auquel elles se conforment toutes.

Nos critères d'acceptation correspondent donc aux caractéristiques suivantes : *Type*, *Hétérogène*, *Symétrique*, *N-aire* et *Appariement*. Les méthodes de composition asymétriques, homogènes, définies sur les instances, le code, ou un sous-ensemble de domaines Ad Hoc ou qui ne gèrent pas la multiplicité des cibles au niveau du code ne sont pas adaptées pour répondre à ce besoin. Par souci de lisibilité, ces caractéristiques sont grisées dans la Table 2.4, où nous pouvons voir qu'aucune approche ne remplit tous les critères posés dans le cadre de cette étude.

2.5.2 Enjeux de la contribution

Nous identifions dans la Table 2.4 un manque quant à une composition au niveau des types, hétérogène, symétrique, n-aire, et adressant la variabilité du code. Cela nous permet de définir les enjeux de cette thèse, détaillés en exigences.

❶ Proposer une composition symétrique de méta-modèles hétérogènes :

- (a) préserver l'intégrité de la représentation du domaine,
- (b) exposer une interface du domaine avec l'expressivité nécessaire,
- (c) vérifier la conformité des modèles produits vis-à-vis de la représentation du domaine.

❷ gérer les interactions n-aire entre les domaines au niveau d'abstraction des types pour assurer la cohérence inter-domaine du système :

- (a) assurer l'exécution du comportement minimum attendu lors d'une action sur un domaine,
- (b) permettre la continuité de la conception même en cas d'incohérence dans un domaine,
- (c) faire remonter l'information pertinente au domaine correspondant en cas de détection d'incohérence.

TABLE 2.4 – Matrice récapitulative de caractérisation des approches de composition

Approches	Critères de comparaison						
	Abstraction	Hétéro-généité	Symétrie	Arité	Variabilité	Automa-tisation	Cohérence
Kompose [Fleurey 07]	Type	Homogène	Symétrique	Binaire	N/A	80-20	Conformité
Ring [Gómez 12]	Type	Partielle	Symétrique	N-aire	Directe	80-20	Synchronisation
Boucké [Boucké 10]	Instance	Partielle	Symétrique	N-aire	N/A	80-20	Conformité
EML [Kolovos 06]	Type	Hétérogène	Asymétrique	Binaire	N/A	20-80	Conformité
ModelBus [Sriplakich 08]	Type	Hétérogène	Asymétrique	N-aire	N/A	80-20	Conformité Synchronisation
Kurpjuweit [Kurpjuweit 07]	Type	Homogène	Symétrique	N-aire	N/A	20-80	Conformité
KerTheme [Barais 08]	Type	Partielle	Asymétrique	Binaire	N/A	80-20	Synchronisation
CORE [Kienzle 09]	Type Instance	Partielle	Asymétrique	N-aire	Appariement	80-20	Dépendance Synchronisation
SmartAdapters [Perrouin 12]	Type	Homogène	Asymétrique	N-aire	Appariement	20-80	Conformité
MATA [Whittle 09]	Instance	Homogène	Asymétrique	Binaire	N/A	80-20	Conformité
Geko [Morin 08] [Klein 12]	Instance	Homogène	Asymétrique	N-aire	Appariement	80-20	Conformité
Brunelière [Bruneliere 15]	Type	Homogène	Asymétrique	Binaire	N/A	20-80	Conformité Synchronisation
I-MMD [Živković 15]	Type	Homogène	Asymétrique	N-aire	N/A	80-20	Conformité
Schottle [Schottle 13]	Type	Hétérogène	Asymétrique	Binaire	N/A	80-20	Synchronisation
MEGAF [Hilliard 12]	Type	Hétérogène	Asymétrique	N-aire	N/A	20-80	Synchronisation
Openflexo [Golra 16a]	Type	Hétérogène	Asymétrique	N-aire	N/A	20-80	Synchronisation
ThingML [Harrand 16] [Morin 17]	Type	Partielle	Asymétrique	Ad Hoc	Appariement	80-20	Dépendance Conformité
Di Natale [Di Natale 14]	Instance	Partielle	Symétrique	Ad Hoc	Directe	80-20	Conformité
Melange [Degueule 15]	Type	Hétérogène	Asymétrique	N-Aire	N/A	80-20	Conformité
MPS [Voelter 12]	Type Instance	Hétérogène	Asymétrique	N-Aire	N/A	80-20	Dépendance Conformité
Ptolemy [Buck 94]	Instance	Hétérogène	Asymétrique	N-Aire	Directe	20-80	Synchronisation
ModHel'X [Boulanger 08]	Instance	Hétérogène	Asymétrique	N-Aire	Directe	20-80	Synchronisation
BCOoL [Larsen 15]	Type	Hétérogène	Symétrique	Binaire	Directe	20-80	Synchronisation
Kermeta [Jézéquel 15]	Code	Partielle	Asymétrique	Ad Hoc	Directe	80-20	Conformité

❸ Automatiser la prise en considération de la multiplicité des artefacts concrets disponibles pour chaque domaine lors de la concrétisation :

(a) prendre en compte les artefacts concrets issus de sources hétérogènes,

(b) tirer profit de la multiplicité des solutions sans coût pour le concepteur à l'exécution,

(c) automatiser l'appariement des modèles aux artefacts concrets de l'espace des solutions.

Les chapitres suivants relèvent ces enjeux et détaillent comment ces exigences sont résolues et quelles propriétés sont respectées par les contributions.

> "*A commmon temptation - one should strongly avoid - is to try to answer all of these questions by means of a single, heavily overloaded, all-encompassing model. It tends to poorly serve individual stakeholders because they struggle to understand the aspect that interest them.*" [Rozanski 11]

CHAPITRE 3

INTÉGRATION DE MÉTA-MODÈLES

Table des matières

3.1 Problématiques d'isolation **29**
3.2 Intégration de services **31**
3.2.1 Architecture Orientée Services (SOA) 31
3.2.2 Propriétés des services et de l'intégration 33
3.3 Modélisation des domaines en tant que services **35**
3.3.1 Modélisation en services 35
3.3.2 Communication par messages 36
3.3.3 Interfaces de Domaines 39
3.3.4 Coût de l'approche 42
3.4 Implémentation **42**
3.5 Conclusions **43**

3.1 Problématiques d'isolation

Ce chapitre présente notre première contribution adressant la collaboration de domaines par intégration. Le paradigme de l'Ingénierie des Modèles (*Model-Driven Engineering* ou MDE) propose de représenter ces domaines à un niveau d'abstraction pertinent pour la manipulation et le raisonnement sur ses concepts, sous la forme de *modèles du domaine* [Arango 89]. Plusieurs approches sont possibles pour implémenter ce modèle du domaine, utilisant soit : *(i)* un méta-modèle (MM), *(ii)* un *viewpoint* ou *(iii)* un Langage Spécifique au Domaine (DSL). Ces approches sont connexes et compatibles : l'implémentation des viewpoints est souvent un méta-modèle [Fischer 12, Dijkman 08], un méta-modèle peut-être outillé par un DSL pour offrir une syntaxe concrète à ses experts, et la syntaxe abstraite d'un DSL peut-être représentée sous la forme d'un méta-modèle. Nous faisons donc l'hypothèse que chaque domaine considéré possède une représentation abstraite appelée méta-modèle.

> "*The structure of a DSL is captured in its metamodel, commonly referred to as its abstract syntax. [...] A model created in terms of our DSL's abstract syntax is commonly referred to as an instance model; the DSL is then the metamodel, which makes Ecore the metametamodel.*" [Gronback 09]

Les approches de composition habituelles se proposent d'adresser la problématique de collaboration de plusieurs méta-modèles soit *(i)* par une approche structurelle, qui implique des effets de bord sur les méta-modèles, soit *(ii)* par une approche comportementale dédiée aux domaines exclusivement comportementaux, voire homogènes. Or,

le Chapitre 1 motive le besoin d'une approche découplée pour la collaboration de méta-modèles. L'objectif premier de ce chapitre est de permettre la collaboration des méta-modèles tout en préservant leur isolation, les méta-modèles n'ayant donc aucune hypothèse à faire sur les autres domaines du système pour fournir à l'expert du domaine l'outillage suffisant pour concevoir un aspect spécifique du système final.

> "*The more* ***assumptions*** *two parties make about each other [...] the less tolerant the solution is of interruptions or* ***changes*** *because the parties are tightly coupled to each other.*" [Hohpe 04]

L'originalité de l'approche exposée dans ce chapitre est d'adopter une approche d'intégration au lieu d'une composition. Plutôt que de composer les domaines en un seul méta-modèle offert à des experts aux connaissances divergentes puis de synchroniser leurs contributions sur un modèle commun, nous considérons chaque domaine comme une entité isolée et outillée et nous centralisons la gestion de la cohérence inter-domaine par des outils de l'état de l'art en matière d'intégration, comme illustré par la Figure 3.1. Des approches existantes utilisent le découplage inhérent à SOA pour faciliter la synchronisation de plusieurs occurences du même modèle, au niveau de l'instance [Clavreul 11], dans le cadre d'une modélisation collaborative par plusieurs experts qui utiliseraient le même domaine, là où nous contribuons à une collaboration au niveau méta, pour intégrer des domaines hétérogènes. Afin d'assurer la réutilisabilité des domaines et l'interopérabilité avec leur écosystème existant qui motivent cette approche d'intégration, nous définissons les exigences suivantes :

Exigence 3.1: Intégrité

La structure de la représentation du domaine n'est pas altérée par sa modélisation en service.

Exigence 3.2: Expressivité

La modélisation d'un domaine en service possède des capacités de manipulation du modèle sous-jacent suffisante pour l'usage de l'expert domaine.

Exigence 3.3: Conformité

Le service du domaine peut vérifier la conformité d'un modèle par rapport au méta-modèle encapsulé.

La Section 3.2 motive la réutilisation des principes des Architectures Orientées Services (SOA) et en présente les propriétés dans notre contexte. La Section 3.3 présente la modélisation des méta-modèles en service, tout d'abord en explicitant les procédés de communication entre l'expert et son domaine dans la Section 3.3.2 puis en définissant la notion d'interface de domaine exposant le comportement du domaine dans la Section 3.3.3. La Section 3.4 détaille l'implémentation de cette modélisation. Enfin, la Section 3.5 résume l'approche et en présente les propriétés.

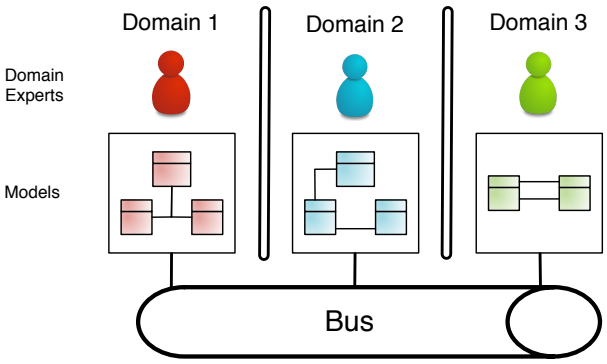


FIGURE 3.1 – Schéma de collaboration des domaines isolés.

3.2 Intégration de services

Nous nous proposons dans cette section d'étudier une solution communément utilisée en composition logicielle, l'intégration de services, et de comparer son apport aux problèmes soulevés dans ce manuscrit. Cette section explicite ensuite les propriétés d'une intégration SOA dont bénéficie notre approche. La couche de médiation et les mécanismes de gestion de la cohérence sont détaillés dans le CHAPITRE 4.

3.2.1 Architecture Orientée Services (SOA)

L'Architecture Orientée Service (SOA) [Fowler 02] est un paradigme de définition de systèmes distribués orientés métier, largement utilisé pour diminuer la complexité de conception des systèmes de systèmes [Josuttis 07, Daigneau 11]. Il repose sur l'identification de composants logiciels auto-suffisants appelés services. Le concept de service, bien que communément utilisé en entreprise, souffre d'un manque de définition précise et unanime.

> "*Services are self-describing, platform agnostic computational elements.*"
> [Papazoglou 03]

Un service est une entité logicielle, fortement découplée des autres, exposant ses fonctionnalités métier à travers une **interface**. L'interface offre à la fois une description des capacités fonctionnelles offertes par le service et des informations attendues par celui-ci pour son bon fonctionnement. L'utilisation d'un service est donc indépendante des détails de son implémentation et de sa plate-forme d'exécution. Utilisé en boite noire, il permet d'exposer des capacités haut-niveau, proches du métier adressé, et ainsi d'être utilisé en collaboration avec d'autres services à travers une couche de médiation.

> "*Services reflect a "service–oriented" approach to programming, based on the idea of* ***describing*** *available computational resources [...] as services that can be delivered through a standard and well-defined* ***interface****.*"
> [Papazoglou 08]

> "*A service is a software resource (discoverable) with an externalized service* ***description****. This service description is available for searching, binding, and invocation by a service consumer. Services should ideally be governed by* ***declarative policies*** *and thus support a dynamically re-configurable architectural style.*"
> [Arsanjani 04]

La mise en place d'une couche de médiation entre services prend généralement la forme d'un bus de messages, implémenté par exemple par un Enterprise Service Bus (ESB), qui connaît les services et organise la communication entre eux et les acteurs de l'architecture intéressés à les utiliser à travers des échanges de messages. L'organisation de cette communication mène classiquement *(i)* soit à la conception de processus métier composant les services par orchestration, *(ii)* soit à la définition d'une chorégraphie par l'analyse des flux d'échanges de messages entre services.

Du fait de notre besoin de conserver les domaines isolés, notre problématique de collaboration de méta-modèles est connexe à la réalisation d'une orchestration de service. Orchestrer des services en concevant un processus métier revient à spécifier pour un scénario précis une séquence d'actions cohérente utilisant des conditions, des exceptions et des invocations aux services collaborant. Cela peut se faire à l'aide d'un langage spécifique, *e.g.,* BPEL ou BPMN, pour exposer le processus composite ou en utilisant des règles métiers pour agir en réaction à l'invocation d'opération sur les services. Cette solution centralise la connaissance de l'interaction entre les services au sein d'un moteur de composition, qu'il s'agisse de l'environnement d'exécution du processus métier ou d'un moteur de règles. A l'inverse, une approche basée sur une chorégraphie préconise une approche décentralisée de la gestion des interactions, sa gestion émergeant de l'analyse des flux de messages échangés entre services. Une chorégraphie suppose une connaissance préalable des dépendances entre services suffisante pour la réifier en une composition. Cette condition n'est pas compatible avec notre hypothèse de collaboration de domaines apportés par des experts isolés, l'utilisation d'une orchestration est donc plus naturelle dans notre cas d'étude.

> "*Orchestration always represents control from one party's perspective. This differs from choreography, which is more collaborative and allows each involved party to describe its part in the interaction. Choreography tracks the message sequences among multiple parties and sources [...] rather than a specific business process.*" [Peltz 03]

Ainsi dans les deux cas, le paradigme SOA propose une solution d'intégration de services isolés. Cette intégration se réalise à l'extérieur des services et n'implique pas de modifications sur leur interface ou leur implémentation. L'idée clef du chapitre est de bénéficier des capacités d'intégration de SOA en l'adaptant pour gérer la collaboration de méta-modèles. La collaboration prend donc la perspective spécifique du système conçu. Ce point de vue fait émerger un rôle d'intégrateur, responsable de la cohérence des domaines concernés. Notre approche offre à l'expert du domaine la possibilité de venir avec sa représentation du domaine, qu'il s'agisse d'un langage, d'un méta-modèle ou d'un *viewpoint*. L'intégrateur, en tant que responsable de l'architecture, a pour rôle de travailler avec l'expert du domaine à la capture de l'interface nécessaire afin de modéliser le domaine comme un service.

> "*A pro forma list of responsabilities for an architect would include [...] to capture and interpret input from technical and domain specialists (and represent this accurately to stakeholders as needed).*" [Rozanski 05]

Exemple 3.1: **Scénario d'intégration**

Dans le cadre de la gestion de projet, Nathalie, développeuse logiciel, contribue sur le gestionnaire de versions en produisant du code, tandis que Pascal, le *product owner*, gère les tickets pour le sprint en cours. Pascal utilise une gestion agile sous JIRA [a], dont Nathalie ne connait pas les détails. A l'inverse, Pascal ne connaît pas la différence entre SVN [b] et Git [c]. Dans notre approche, chacun adresse un service dédié à son domaine, présentant les fonctionnalités métier nécessaires pour contribuer. Par exemple, Pascal signale un bug, Nathalie récupère alors la dernière version du code de la branche concernée, code, puis enregistre une version réparant le bug, enfin Pascal le marque comme résolu. L'approche par intégration de domaines permet de proposer des fonctionnalités à ce niveau d'abstraction, indépendant de l'outillage interne, puis d'orchestrer leur contributions, pour gérer les interactions entre les domaines en maintenant la cohérence du projet.

a. https ://www.atlassian.com/software/jira
b. https ://tortoisesvn.net/
c. https ://git-scm.com/

Chaque opération interagit en interne avec les éléments structuraux sous-jacents du méta-modèle, conformément à l'approche classique SOA [Hohpe 04] et comme illustré par la FIGURE 3.2. Le respect des bonnes pratiques SOA par les services de domaines de notre architecture et leurs opérations respectives assure l'isolation des services et évite les effets de bord.

3.2.2 Propriétés des services et de l'intégration

Un service au sens de SOA doit respecter les propriétés suivantes :

Propriété 3.1: **Sans État**

"*When people refer to statelessness they mean an object that doesn't retain state between requests.*" *[Fowler 02]*

Propriété 3.2: **Idempotent**

"*Idempotence means that no matter how many times a procedure is invoked with the same data, the same results should occur.*" *[Daigneau 11]*

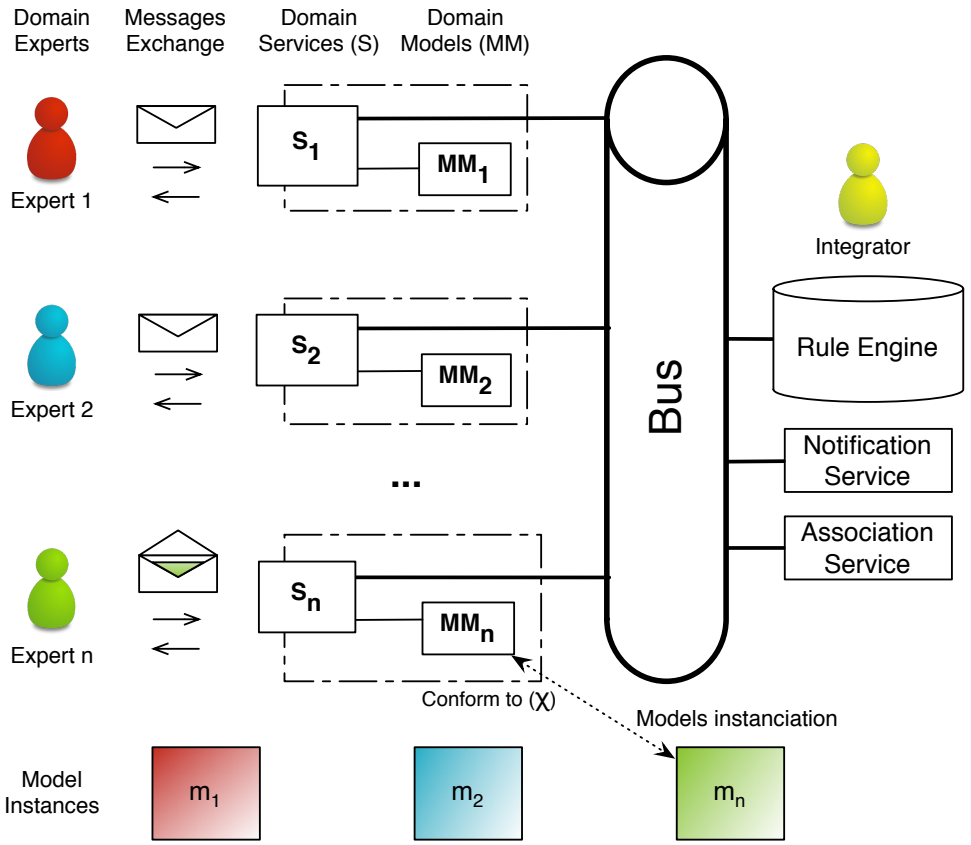


FIGURE 3.2 – Architecture des services de domaines et instanciation.

Propriété 3.3: Découplage

*"The core principle behind **loose coupling** is to **reduce the assumptions** two parties (components, applications, services, programes, users) make about each other." [Hohpe 04]*

Ainsi, l'exécution d'une opération sur un service ne repose ni sur l'état d'autres services ni sur l'historique d'actions exécutées au préalable sur ce service. Cela implique que le message envoyé à un service contient la totalité de l'information nécessaire à la réalisation de l'opération. L'implémentation d'un service garantit également que de multiples invocations successives d'une même opération de service avec des messages identiques ne contrarieront pas la cohérence du système produit. L'opération ciblée ignore les invocations répétées superflues, au lieu de s'exécuter à nouveau. Cette propriété est nécessaire à la sémantique d'exécution de la couche de médiation détaillée dans le CHAPITRE 4. Enfin, l'implémentation d'une opération ne peut pas invoquer une autre opération, qu'elle soit exposée par le même service ou non. Chaque opération exposée est alors un accès indépendant à l'exécution d'une tâche orientée métier. Nous définissons donc un service de domaine comme suit :

DEF: SERVICE

Entité logicielle orientée métier, découplée des autres et sans état, exposant au travers d'une interface un ensemble d'opérations haut-niveau, idempotentes et non-réentrantes.

3.3 Modélisation des domaines en tant que services

Dans cette section, nous exposons comment adapter les concepts clefs de SOA pour encapsuler des méta-modèles et ainsi profiter de la propriété d'isolation définie dans le CHAPITRE 1 et motivée en début de ce chapitre. En détaillant l'adaptation des notions de service, de message et d'interface au contexte des modèles, nous en détaillons le gain d'un point de vue composition.

3.3.1 Modélisation en services

3.3.1.1 Méta-concepts de l'approche d'intégration

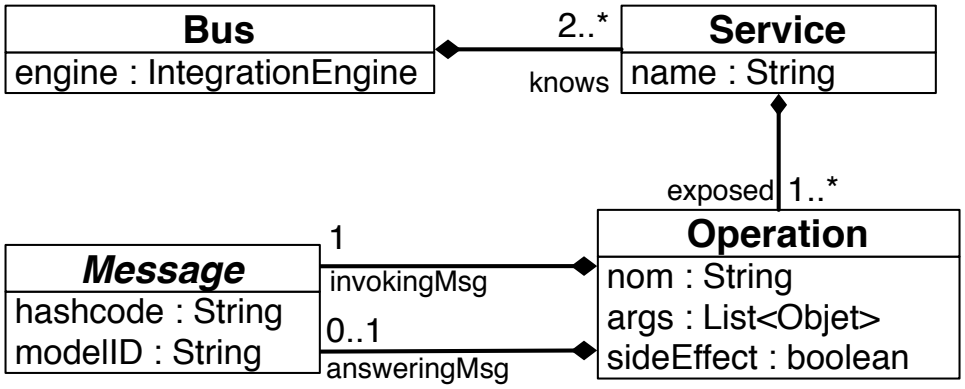


FIGURE 3.3 – Diagramme de classes simplifié des concepts de l'intégration.

L'architecture d'intégration proposée dans ce chapitre manipule les méta-concepts illustrés dans la FIGURE 3.3. Le bus connaît les services à intégrer. Chaque service expose une interface composée d'opérations et de la signature de chaque message attendu et éventuellement retourné par celles-ci. Une opération définit si son invocation provoque des effets de bord sur le système et l'ensemble des arguments qu'elle requiert. Un message contient nécessairement l'identifiant de son modèle cible du fait de la PROPRIÉTÉ 3.1. De plus, la fonction d'égalité entre messages est définie à partir des informations contenues par ceux-ci afin de permettre le respect de la PROPRIÉTÉ 3.2. Le fonctionnement de la couche de médiation, représentée ici par un moteur d'intégration appartenant au bus, est détaillé dans le CHAPITRE 4.

3.3.1.2 Sémantique d'exécution des messages

Les services comme définis dans la SECTION 3.2.1 exposent donc dans leur interface *(i)* les informations attendues par chaque opération pour fonctionner et *(ii)* les données

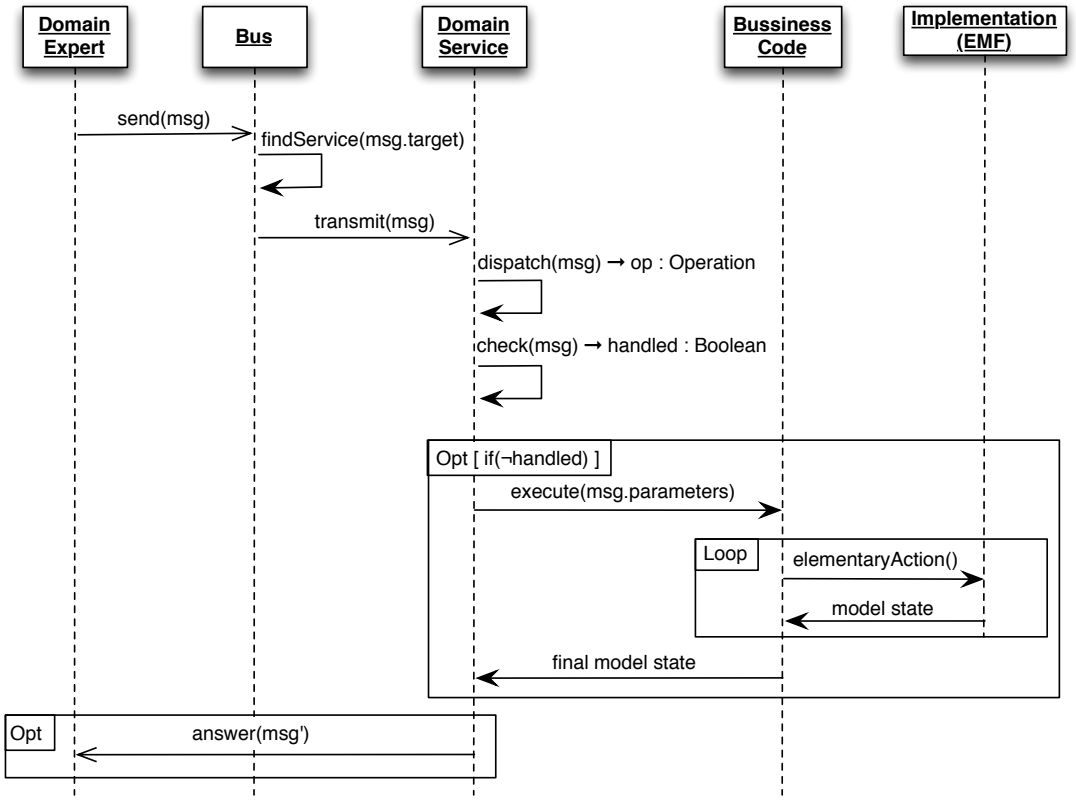


FIGURE 3.4 – Diagramme de séquence de l'interprétation d'un message.

fournies en retour sous la forme d'un message. La sémantique de prise en charge des messages est illustrée par la FIGURE 3.4. Les messages sont transmis par l'expert domaine vers la couche de médiation afin d'invoquer une opération. Le bus trouve le service pertinent à qui adresser ce message et lui transmet. Ce service a alors la responsabilité d'invoquer l'opération ciblée, mais également de vérifier qu'un message égal n'a pas déjà été exécuté. Le cas échéant, le message est ignoré. Dans le cas contraire, le code métier de l'opération est exécuté, utilisant des appels à l'implémentation du domaine sous la forme d'actions atomiques sur les éléments du méta-modèle. Si nécessaire, un message est retourné à l'expert lorsque l'opération a été accomplie pour renvoyer de l'information sur l'état du système résultant.

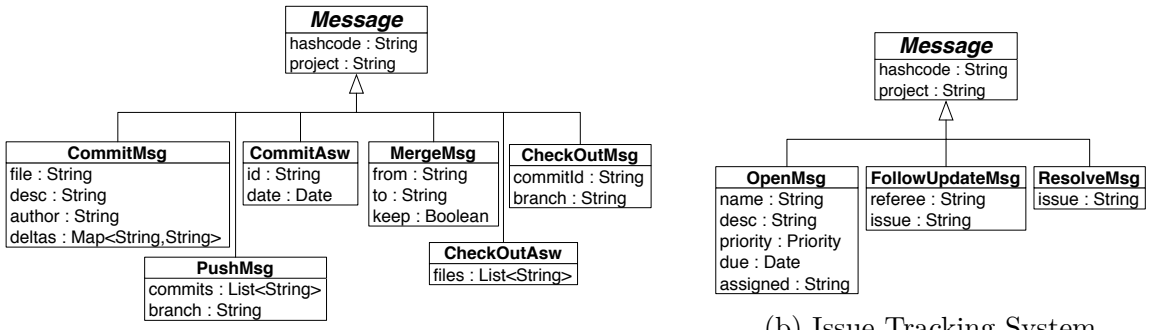
3.3.2 Communication par messages

3.3.2.1 Définition d'un message

Les interactions entre les experts domaines et la couche métier reposent sur le concept de *messages* défini comme suit.

DEF: MESSAGE

Structure de données orientée métier contenant l'information nécessaire à l'exécution d'une opération spécifique sur un service connu.



(a) Version Control System

(b) Issue Tracking System

FIGURE 3.5 – Diagrammes de classe décrivant la structure des messages

La définition des messages échangés avec le service d'un domaine découle de l'usage que l'expert a du domaine. La séparation entre la structure du message attendue par une opération et la structure interne du méta-modèle permet d'abstraire l'expert domaine des choix d'implémentation de cette représentation du domaine. Un message peut être défini à l'aide de types primitifs et d'objets métier. Les objets métier sont des descriptions structurelles de concepts forts du domaine, utiles à sa manipulation, *e.g.,* la notion de priorité d'une tâche.

Cette divergence est illustrée par la structure des messages de l'exemple fil rouge dans la FIGURE 3.5a, découplée de la structure du méta-modèle présentée dans la FIGURE 3.8. Cette description orientée métier facilite la collaboration entre experts, car elle permet à un expert d'un autre domaine d'identifier aisément les informations nécessaires aux fonctionnalités haut-niveau exposées et de faire la traduction dans son propre domaine, *cf.* EXEMPLE 3.2.

Exemple 3.2: **Message de commit et résolution de tâche**

Malgré la séparation de leurs domaines de compétences, Nathalie exprime le besoin de résoudre automatiquement une tâche lors du commit correspondant. D'après l'interface du service de Gestion de Tickets (`VersionControlSystem`), il suffit de spécifier l'identifiant de la tâche. Nathalie souhaite donc, lorsque son message de commit contient le mot clef `#close` et l'identifiant d'une tâche, pouvoir envoyer un message au service de Gestion de Tickets (`IssueTracking`) pour propager la résolution apportée par son nouveau code.

3.3.2.2 Effet d'un message sur un service de domaine

La définition d'un message étant spécifique à une et une seule opération, le service du domaine peut automatiquement exécuter l'opération correspondante à la réception d'un message. L'opération est alors responsable de l'extraction des informations du message pour réaliser la fonctionnalité décrite dans le contrat du service. La FIGURE 3.6 décrit le comportement de l'architecture lorsque le service du domaine reçoit un message.

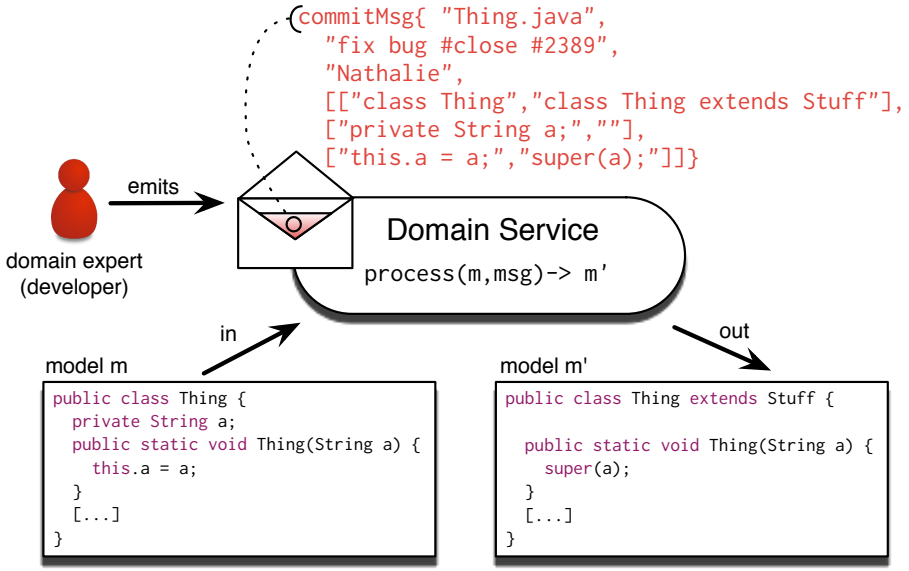


FIGURE 3.6 – Impact de l'invocation d'une opération par message sur le modèle.

Exemple 3.3: Message d'ouverture d'une tâche

L'expert domaine envoie un message de commit comprenant la description textuelle du travail réalisé et les changements effectués dans le code. L'opération `commit()` du service `VersionControlSystem` récupère le modèle correspondant au fichier "`Thing.java`" et est chargé d'appliquer les modifications pour décharger la responsabilité de l'attribut "`a`" sur la classe "`Stuff.java`". L'implémentation de l'opération, cachée à l'expert domaine, instancie la méta-classe `Commit`, lui affilie les attributs tirés du messages (*e.g.,* la description et l'auteur) et ceux calculés automatiquement (*e.g.,* l'identifiant, la date) et la lie à de nouvelles instances de la méta-classe `Delta` pour enregistrer la nouvelle version du code.

Cet exemple, illustré par la FIGURE 3.7, montre la différence d'expressivité entre une modélisation en manipulant directement les éléments structuraux du méta-modèle par de nombreuses actions atomiques, et une modélisation en service exposant un ensemble restreint d'opérations haut-niveau. Pour un seul *commit* n'affectant que 3 lignes, 17 actions élémentaires sont nécessaires sur le méta-modèle, tandis que l'envoi d'un seul message suffit.

Ainsi, l'utilisation du concept de message comme unité de communication permet une manipulation moins fine du domaine par rapport à l'utilisation des actions élémentaires sur chacun des éléments du méta-modèle. Cependant, l'objectif premier de ce chapitre étant la collaboration de domaines isolés, comme les messages permettent une séparation stricte entre les choix d'implémentation des méta-modèles et l'utilisation du domaine, ils facilitent de par cette communication orientée métier la compréhension du domaine par tous les experts, l'identification d'interactions, et donc leur collaboration.

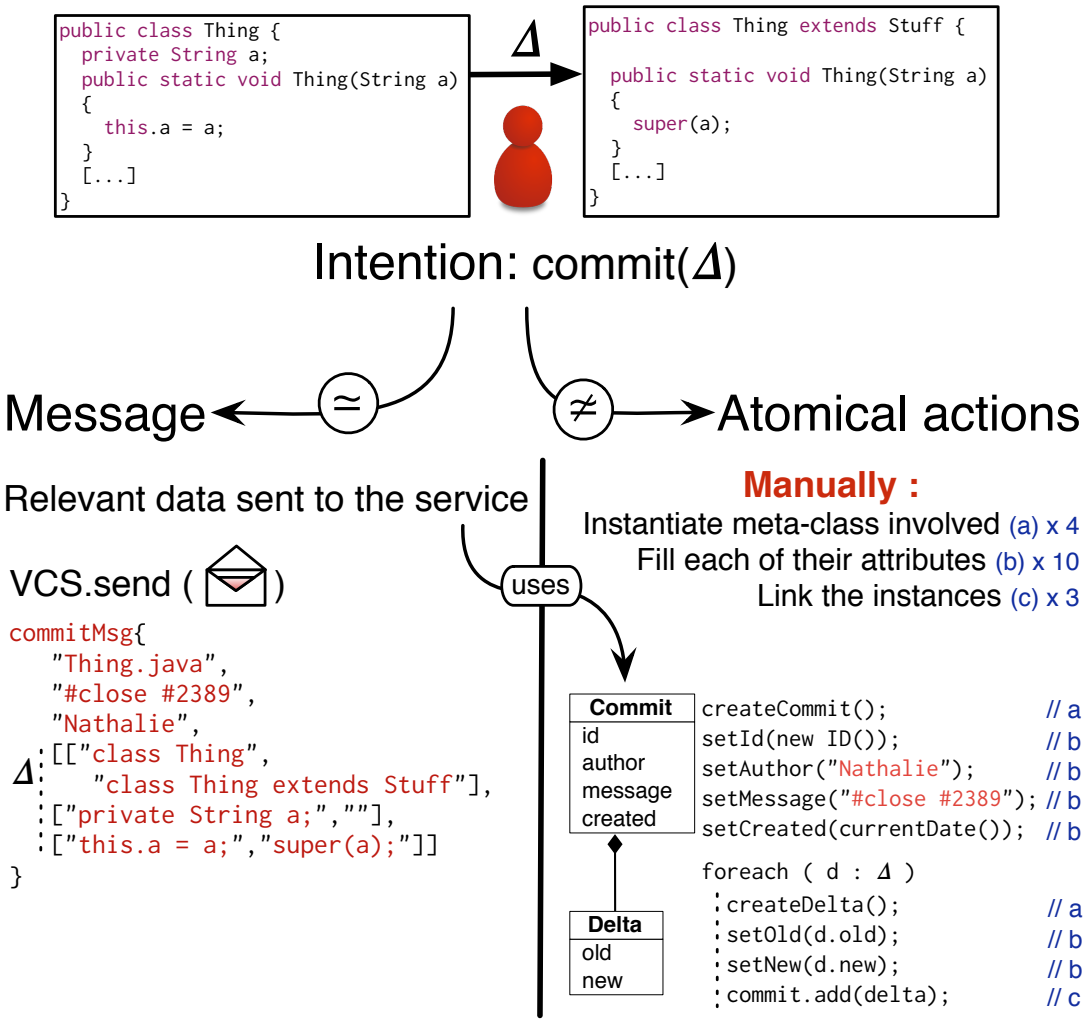


FIGURE 3.7 – Comparaison des approches sur l'exemple du VersionControlSystem (VCS).

3.3.3 Interfaces de Domaines

3.3.3.1 Encapsulation du domaine en boite noire

Pour adapter l'approche comportementale classique d'intégration SOA à la collaboration de méta-modèles, nous proposons de modéliser les domaines comme des services. Là où une représentation structurelle d'un domaine, *e.g.,* un méta-modèle outillé par un DSL, offre à l'expert du domaine des moyens pour ajouter, lire, éditer, et supprimer des éléments atomiques (*CRUD*) tels que décrits dans le méta-modèle (*i.e.,* des classes, des attributs et des relations), une interface de service expose à la place un nombre réduit d'opérations de haut-niveau, empaquetant plusieurs actions atomiques en une seule opération d'une granularité plus proche de la connaissance métier de l'expert, comme illustré par la FIGURE 3.7.

Les bonnes pratiques de SOA conseillent de n'exposer que le comportement pertinent en opérations. Cela se traduit dans notre cas par un ensemble minimal d'opérations permettant à l'expert du domaine de concevoir un modèle cohérent. Pour cela, notre architecture repose sur des services isolés encapsulant les modèles des domaines qui collaborent.

Comme illustré par la FIGURE 3.2, l'expert du domaine n'interagit pas directement avec le méta-modèle, mais consomme un service exposant les fonctionnalités nécessaires à la conception d'un modèle conforme au domaine en question. Selon le paradigme SOA, les fonctionnalités exposées par les interfaces de service sont extraites des besoins métier et de l'usage, comme un contrat entre l'expert du domaine et le service [Fowler 02].

La valeur ajoutée de l'utilisation d'opérations haut-niveau, au lieu de méthodes spécifiques à l'échelle de l'objet, provient de la proximité des fonctionnalités exposées avec le métier de l'expert du domaine. Cela lui permet de contribuer au niveau de l'espace des problèmes, en manipulant des opérations orientées métier sur un modèle, plutôt que de travailler en *CRUD* dans l'espace des solutions nécessitant des connaissances techniques sur les choix d'implémentation du méta-modèle.

3.3.3.2 Illustration sur l'exemple fil rouge

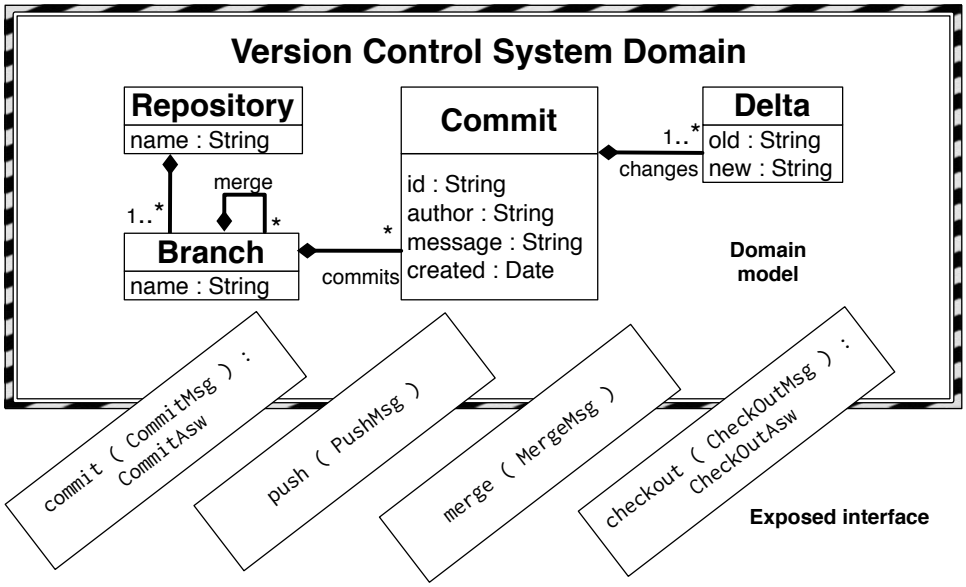


FIGURE 3.8 – Extrait du méta-modèle de gestionnaire de versions encapsulé dans un service.

Dans l'exemple fil rouge de gestion de projet, nous présentons les méta-modèles simplifiés des domaines *(i)* d'un gestionnaire de versions et *(ii)* d'un gestionnaire de tickets, à modéliser en services.

Comparativement aux techniques de composition classiques, notre objectif est de proposer une collaboration des domaines tout en maintenant leur isolation. Par conséquent, chaque domaine opère comme une boite noire, un service encapsulant les mécanismes internes de son implémentation pour assurer la séparation des préoccupations et exposant son comportement par une interface.

La caractérisation de l'usage que font les experts de leur domaine mène à la définition d'une interface suffisante. La définition de cette interface passe par l'identification de l'expressivité minimale nécessaire à un développeur et un chef de projet pour leur

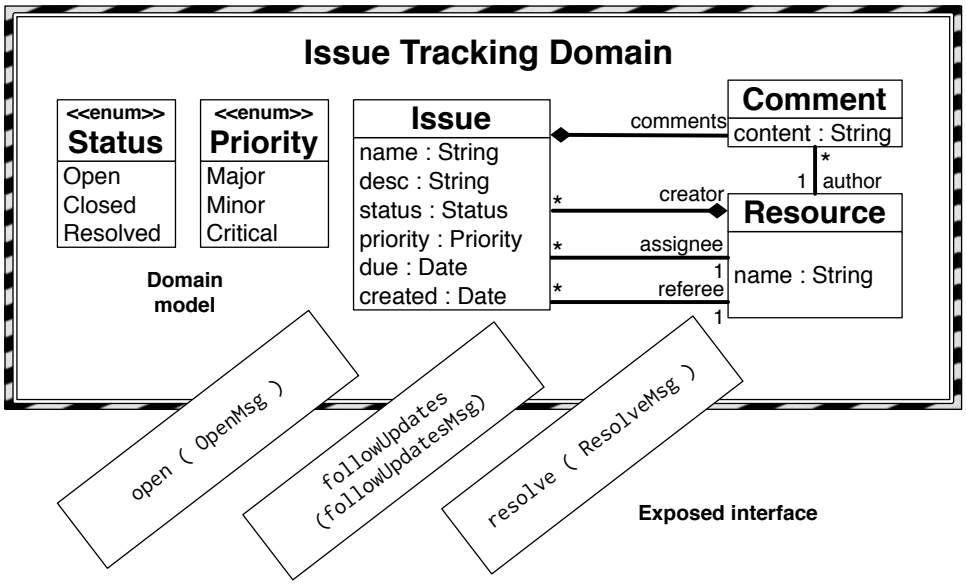


FIGURE 3.9 – Extrait du méta-modèle de gestion de tickets encapsulé dans un service.

tâche de modélisation. La manipulation directe des éléments des méta-modèles par des actions atomiques *CRUD* n'est pas nécessaire et oblige les experts à connaître les choix d'implémentation des méta-modèles. Au contraire, l'invocation d'une opération sur un service repose seulement sur le contrat exposé par ce dernier, illustré par la FIGURE 3.8 et la FIGURE 3.9.

Exemple 3.4: **Services de versions et de tickets**

La FIGURE 3.8 illustre le service du gestionnaire de versions qui expose les opérations pour *(i)* enregistrer un ensemble de modifications dans le code (`commit`), *(ii)* pousser du code (`push`), *(iii)* fusionner deux branches (`merge`) et *(iv)* récupérer l'état du projet à une version donnée (`checkout`). Le service du gestionnaire de tickets illustré par la FIGURE 3.9 expose les opérations pour *(i)* ouvrir une nouvelle tâche (`open`), *(ii)* suivre une tâche (`followUpdate`) et *(iii)* déclarer la résolution d'une tâche (`resolve`).

Cet exemple illustre la préservation des méta-modèles et l'identification d'une interface minimale adressée à l'expert du domaine et démontre que l'architecture décrite dans ce chapitre répond à la problématique d'isolation des domaines que ce chapitre adresse.

3.3.3.3 Code fonctionnel du Service

Notre approche repose sur l'hypothèse que toutes les opérations d'un service ont accès aux éléments du méta-modèle encapsulé et aux actions atomiques de manipulation des modèles. La responsabilité de l'opération est de faire le lien entre la fonctionnalité haut-niveau exposée dont découle la structure du message requis, et les choix d'implémentation

techniques du méta-modèle encapsulé. Pour cela, en premier lieu l'opération extrait les informations nécessaires du message et transforme les données ou en calcule de nouvelles au besoin. Du fait de la propriété Propriété 3.1, l'opération sans état ne contient pas le modèle cible à priori, il est donc nécessaire de le retrouver parmi les instances du méta-modèle à l'aide des informations du message. Enfin, elle exécute sur ce modèle un ensemble d'actions atomiques pour instancier les méta-éléments pertinents et leur attribuer les valeurs correctes.

3.3.4 Coût de l'approche

L'encapsulation de chaque domaine par un service et la mise en œuvre des mécanismes de communication ont un cout d'implémentation. Il est à noter que le code implémentant le méta-modèle et les actions élémentaires CRUD sur ses éléments fournis par l'*Eclipse Modeling Framework* (EMF), ainsi que la sémantique définie sur le méta-modèle, par exemple en OCL, peuvent être réutilisés en totalité. Les mécanismes de communication et d'invocation, *i.e.,* le bus, la vérification de l'unicité des messages, et les services auxiliaires (illustrés dans la Figure 3.2 et utiles à la couche de médiation du Chapitre 4) sont développés une seule fois et indépendants des domaines intégrés. Ainsi le coût d'ajout d'un service pour un nouveau domaine revient à en lister les opérations, en spécifiant les paramètres nécessaires à chacune et ce qu'elle retourne sous forme de messages, et à en écrire le code métier.

3.4 Implémentation

Cette approche est implémentée par un prototype développé en Java, pour des raisons d'interopérabilité avec l'outil utilisé dans la couche de médiation, *cf.* Chapitre 4. Un projet défini les concepts de base de l'approche sous la forme de classes abstraites, *e.g.,* `Service.Java`, `Message.Java`, `Answer.Java`. Chaque service de domaine est développé au sein d'un projet Maven n'ayant pas de dépendance vers les autres services pour assurer la propriété d'isolation, mais contenant une dépendance vers une implémentation du méta-modèle correspondant implémenté avec EMF. Chacun de ces projets ne contient qu'une seule classe publique héritant de `Service.Java`. Les méthodes publiques de cette classe sont les opérations exposées, annotées `@Operation`. L'opération `commit` du scénario introduit dans la Figure 3.7 est détaillé dans l'Exemple 3.5. Un package `message` décrit le protocole d'échange entre l'expert du domaine et le service tel qu'illustré par l'Exemple 3.6, et un package `businessobject` donne la structure des objets métiers manipulables pour ces invocations, utilisé par exemple pour définir le concept de priorité d'une tâche par une énumération. Une limite de cette implémentation est de ne pas avoir distribué ces services via un fournisseur de service de l'état de l'art ou d'offrir un *framework* réutilisable complètement indépendant du cas d'application.

Exemple 3.5: **Extrait de l'implémentation de l'opération `commit()`**

L'opération `commit()` du service `VersionControlSystem` fait le lien entre la fonctionnalité exposée et l'implémentation sur la pile logicielle *EMF* du méta-modèle associé en tirant profit des *factory* générées.

```
1 @Service
```

```
public class VersionControlSystem {

  @Operation
  public static void commit(CommitMsg msg) {

    if( ! alreadyHandled.contains(msg.getId())
      private String file = msg.getFile();
      private String desc = msg.getDescription();
      private String author = msg.getAuthor();
      private Map<String, String> deltas = msg.getDeltas();

      File preexisting = loadModel(file);
      Commit c = VersionControlSystemLanguageFactory.eINSTANCE.
          createCommit();
      c.affectID();
      c.setDescription(desc);
      c.setAuthor(author);
      c.setDate(Calendar.getInstance().getTime());
      for (String s : deltas.getKeys()) {
        Delta d = VersionControlSystemLanguageFactory.eINSTANCE.
            createDelta();
        d.setOld(s);
        d.setNew(deltas.get(s);
        c.getDeltas.add(d);
      }
      preexisting.getCommits().add(c);
      persist(preexisting);
    [...]
}
```

Exemple 3.6: **Structure du message d'ouverture d'une tâche**

```
@Message
public class CommitMsg extends Message {
  private String file;
  private String desc;
  private String author;
  private Map<String, String> deltas;
  public CommitMsg(String desc, String author, Pair<String, String>...
      delta) {
    super();
    this.description = description;
    this.author = author;
    this.delta = new HashMap<>();
    delta.foreach( (k,v) -> this.delta.put(k,v) );
  }
  [...]
}
```

3.5 Conclusions

En considérant le concept de message comme l'unité de communication entre l'expert domaine et le domaine, nous vérifions que chaque opération est idempotente, satisfaisant la Propriété 3.2. Cette modélisation en service permet de réutiliser chaque méta-modèle tel quel. Ainsi, l'outillage pré-existant du domaine est automatiquement compatible et le service du domaine résultant peut bénéficier, par exemple, de la fonction de conformité d'un méta-modèle sur ses instances, *cf.* Exigence 3.3. De plus, l'implémentation du service a accès à la totalité des actions atomiques fournies par le méta-modèle, ce

qui permet au service d'exposer les opérations nécessaires pour préserver l'expressivité nécessaire à l'expert, *cf.* Exigence 3.2. De même, l' Exemple 3.5 illustre la capacité de réutilisation de l'outillage du méta-modèle par le service, ici les *factory* générées par *EMF*. La modélisation des domaines en service respecte donc les trois propriétés énoncées dans la Section 3.2.2. De plus, les trois exigences (Exigence 3.2,Exigence 3.1 et Exigence 3.3) explicitées dans la Section 3.1 sont satisfaites par notre adaptation de l'intégration de SOA à des méta-modèles.

Notre approche d'intégration permet de maintenir une isolation entre des méta-modèles hétérogènes, structuraux ou comportementaux. nous proposons une solution où :

1. il est possible d'encapsuler un méta-modèle dans un service,
2. c'est l'expert du domaine possédant la connaissance métier qui défini l'interface pertinente du service,
3. on a alors la possibilité d'utiliser les capacités d'orchestrations connues dans le paradigme SOA.

Tout en conservant l'interopérabilité des méta-modèles avec l'outillage pré-existant, nous proposons aux experts de travailler sur les interfaces de leurs domaines, à un niveau d'abstraction plus proche de leur métier. La couche de médiation permettant l'utilisation de ces services par les experts ainsi que les mécanismes de gestion de la cohérence sont détaillés dans le Chapitre 4.

CHAPITRE 4

RÈGLES MÉTIER ET COHÉRENCE DE L'ARCHITECTURE

Table des matières

4.1 Problématique de Collaboration de Domaines et Cohérence **45**
4.2 Grammaire des Règles Métier **48**
4.2.1 Définition d'une Règle de Routage 49
4.2.2 Définition d'une Règle d'Intégration 51
4.3 Cohérence Inter-Domaines **52**
4.3.1 Propagation de Déclenchement de Règles 52
4.3.2 Expressivité des Règles d'Intégration 53
4.3.3 Écriture et Composition de règles 54
4.4 Implémentation **55**
4.5 Conclusions **56**

4.1 Problématique de Collaboration de Domaines et Cohérence

Dans un contexte de modélisation collaborative, nous avons introduit dans le Chapitre 3 une solution d'encapsulation des méta-modèles en services, permettant à chaque expert du domaine d'adresser son domaine indépendamment des autres aspects du système. La modélisation collaborative d'un système nécessite une gestion de la cohérence du produit. Cette cohérence est double [Logre 15] puisqu'il faut pouvoir vérifier *(i)* la cohérence locale à un domaine, *i.e.,* la conformité d'un modèle produit par un expert par rapport à la définition du méta-modèle du domaine, mais également *(ii)* la cohérence globale du système, *i.e.,* la compatibilité inter-domaines des modèles produits par différents experts.

La cohérence locale est classiquement gérée par une fonction de validité vérifiant si un modèle (m) donné est conforme à un méta-modèle (MM) donné $\chi : m \times MM \rightarrow \mathbb{B}$, pour une vérification structurelle, et *(ii)* sur l'utilisation d'un langage de contraintes pour la sémantique, *e.g., Object Constraint Language* (OCL). Classiquement, la cohérence inter-domaines repose sur différentes mécaniques, par *mapping*, par transformation ou par évènement en réaction à une action atomique. Ces méthodes de gestion de la cohérence reposent sur un travail au niveau des méta-éléments implémentant les domaines. Dans notre approche, les méta-modèles sont isolés et encapsulés dans des services. Ce chapitre détaille une solution d'externalisation des interactions dans un moteur dédié.

L'objectif de ce chapitre est d'adapter les mécanismes de gestion de la cohérence évènementielle à notre approche basée sur des services pour vérifier la cohérence des modèles de différents domaines, en minimisant le nombre d'hypothèses que chaque domaine fait sur les autres. Nous définissons pour cela un ensemble d'exigences sur l'architecture que le système de gestion de la cohérence doit respecter.

Exigence 4.1: Comportement attendu

L'invocation explicite d'une opération par un expert du domaine doit toujours se solder par l'exécution de l'opération par le service correspondant.

Exigence 4.2: Non bloquant

Le système ne doit jamais bloquer un expert du domaine dans sa modélisation à la suite d'une action de cet expert ou d'un autre.

Exigence 4.3: Boucle de rétro-action

Les effets de bord d'une action sur le système qui a compromis la cohérence de celui-ci doivent remonter aux experts concernés avec l'information pertinente pour la résolution du problème de cohérence.

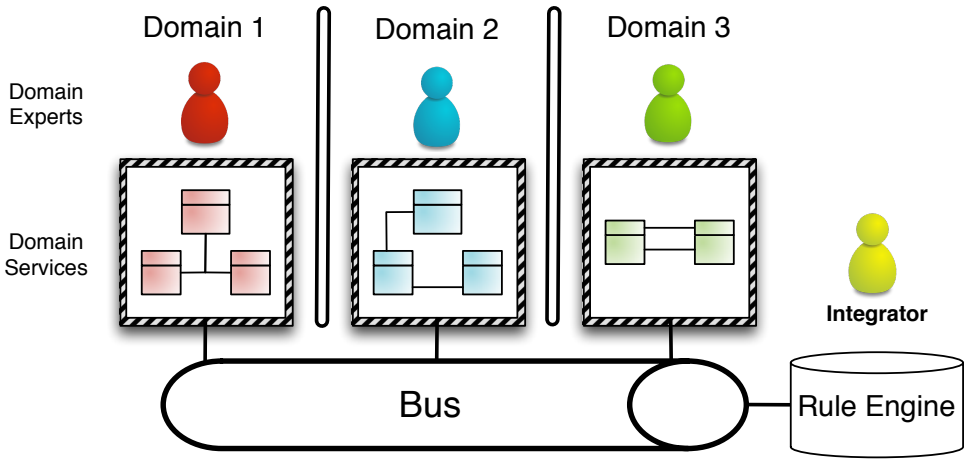


FIGURE 4.1 – Schéma de acteurs de la gestion de la cohérence pour domaines isolés.

Dans ce contexte, nous nous appuyons sur les mécanismes de composition offerts par SOA, puisque le découplage des services est une des préoccupations premières de ce paradigme. Afin d'éviter la prolifération de services composites engendré par une orchestration, nous choisissons de travailler au niveau de l'échange des messages via une chorégraphie. En réutilisant les règles métiers, nous permettons d'exprimer les interactions

de façon unitaires et atomiques, favorisant l'enrichissement de la gestion de la cohérence par l'ajout et l'édition des cas traités. Les règles métiers bénéficient en effet de propriété d'évolutivité détaillés dans la section suivante. Notre solution repose donc sur un rôle d'intégrateur illustré par la FIGURE 4.1, qui établit une médiation entre les domaines à l'aide de leurs experts respectifs pour en exprimer les interactions et formalisent ces dernières sous la forme de règles métier.

A travers ces règles, l'intégrateur définit explicitement les interactions entre les domaines menant à un système global incohérent. La responsabilité de ces règles métier est de détecter les incohérences dans les modèles de différents domaines, *i.e.,* hétérogènes, conçus indépendamment. La résolution de ces incohérence nécessite une prise de décision par un expert, l'automatisation complète n'est donc pas atteignable. Au contraire, induire une non-conformité locale à un domaine peut faciliter la résolution de l'incohérence globale du système. Le moteur d'intégration peut donc ajouter de l'incohérence, tant qu'il est possible de tracer l'invocation et le contexte qui ont mené à cette réaction.

Exemple 4.1: **Incohérence locale pour aller vers une cohérence globale**

Timothé, intégrateur du système de Gestion de Projet, travaille sur les interactions des domaines de Gestion de Tickets et de Gestion de Versions avec leurs experts respectifs, Pascal et Nathalie. A cette occasion émerge une exigence sur le système : tous les commits émis par les développeurs doivent être rattachés à un ticket. Timothé définit alors une règle métier, réagissant à l'invocation de l'opération de commit, pour retrouver le ticket associé et lier les deux éléments. Toutefois, si le commit ne contient pas de numéro de ticket dans sa description, cette recherche est impossible. L'opération de commit sera bien prise en compte et la nouvelle version du code sera sauvegardée, mais la cohérence globale du système n'est pas respectée et Timothé doit faire remonter à Nathalie cette incohérence. De même, si la description du commit contient un numéro de ticket inconnu du domaine de Gestion de Ticket, Timothé peut décider de créer automatiquement un squelette incomplet de ticket, rendant incohérent ce modèle, et de remonter à Pascal la nécessité de le compléter pour retrouver une cohérence locale.

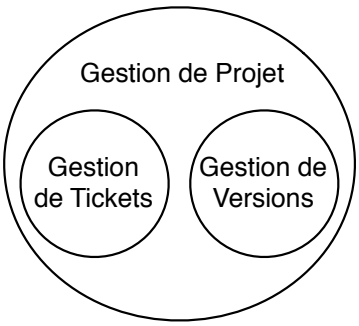


FIGURE 4.2 – Schéma des domaines impliqués.

4.2 Grammaire des Règles Métier

La couche de médiation de notre architecture s'appuie sur la définition d'un ensemble de règles métier. Les domaines utilisés au sein d'un SoS sont amenés à évoluer, à être remplacés. Or en définissant un ensemble de bonnes propriétés sur les règles nous favorisons la maintenabilité de l'intégration.

Propriété 4.1: Indépendance

Les règles sont conçues indépendamment les unes des autres, il n'y a pas d'hypothèse sur l'existence d'autres règles.

Propriété 4.2: Sans ordre

Aucun ordre n'est garanti quant au déclenchement des règles, l'ordre de déclaration n'a aucune incidence et une règle ne peut pas faire l'hypothèse qu'une autre a été déclenchée auparavant pour son contexte d'exécution.

Ces règles régissent les interactions entre les domaines en définissant la réaction de chacun à l'invocation d'une opération sur un service. Une règle métier est constituée d'une partie gauche, *e.g.,* le déclencheur, et d'une partie droite définissant une séquence d'action à exécuter en réaction à ce contexte.

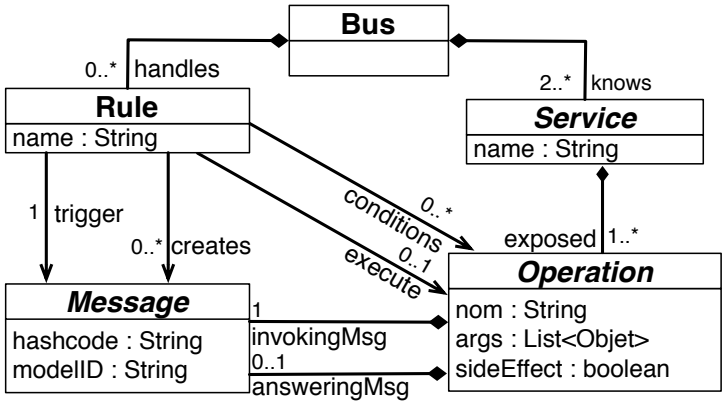


FIGURE 4.3 – Diagramme de classes simplifié des concepts de la gestion de la cohérence.

Dans notre architecture, le déclencheur est la réception d'un message d'un type donné. Il peut également poser des conditions sur l'état du système. La séquence d'actions de la réaction est constituée de créations de nouveaux messages et possiblement de l'invocation d'une opération sur un service. La description structurelle d'une règle est illustrée par la FIGURE 4.3. Ainsi, dans le contexte de l'intégration de domaines isolés, une règle métier se place au même niveau d'abstraction que les interfaces exposées par les services, ce qui facilite la médiation entre les domaines par l'intégrateur. Les règles métiers sont indépendantes des choix d'implémentation des méta-modèles des domaines, permettant

de changer un méta-modèle par un autre du même domaine sans affecter la base de règle tant qu'ils exposent la même interface métier.

DEF: RÈGLE MÉTIER = DÉCLENCHEUR → RÉACTION

$$(Message, Conditions) \rightarrow \begin{cases} \texttt{Bus.send(new Message(...))}^* \\ \texttt{Service.Operation(Message)}^* \end{cases}$$

Les règles sont contenues dans un moteur de règles qui respecte la sémantique d'exécution illustrée par la FIGURE 4.4. Lors de l'envoi par un expert d'un message à la couche de médiation, le bus déclenche la base de règles sur le message. Chaque règle correspondant à ce type de message est déclenchée, sous réserve que ses conditions sur le système soient respectées. Une règle peut alors exécuter sa réaction en créant des messages à envoyer sur le bus pour agir sur les autres domaines. Elle peut également invoquer l'opération cible du message déclencheur en s'adressant directement au service approprié. Pour que cette sémantique soit valide, la base de règles doit également respecter la propriété suivante.

Propriété 4.3: **Sans cycle**

Une règle r_i peut engendrer le déclenchement d'une règle r_j par effet de bord, mais cette propagation ne doit pas déclencher à nouveau la règle r_i.

La grammaire d'une règle métier peut être exprimée sous sa forme Backus-Naur (BNF) ainsi :
BR : := *Trigger* → *Reaction**
Trigger : := *Message* ∧ *Condition**
Message : := String
Condition : := [¬] *Service*.*Operation* [= PrimitiveValue]
Service : := String
Operation : := String
Reaction : := [*Service*.*Operation*]
Bus.send(*Message*)*

Notre architecture prend en considération deux types de règles : les règles de routage et les règles d'intégration, respectivement détaillées dans la SECTION 4.2.1 et la SECTION 4.2.2.

4.2.1 Définition d'une Règle de Routage

Les règles de routage permettent le bon fonctionnement de la couche de médiation dans le cas nominal de l'envoi d'un message par un expert qui doit être transmis au service compétent pour l'invocation de l'opération cible, sans que cela n'implique d'interaction avec d'autres domaines. Une règle de routage est définie par : *(i)* une partie gauche décrivant le déclencheur, *i.e.,* le type de message attendu, *(ii)* une partie droite précisant la réaction, *i.e.,* l'invocation de l'opération sur le service.

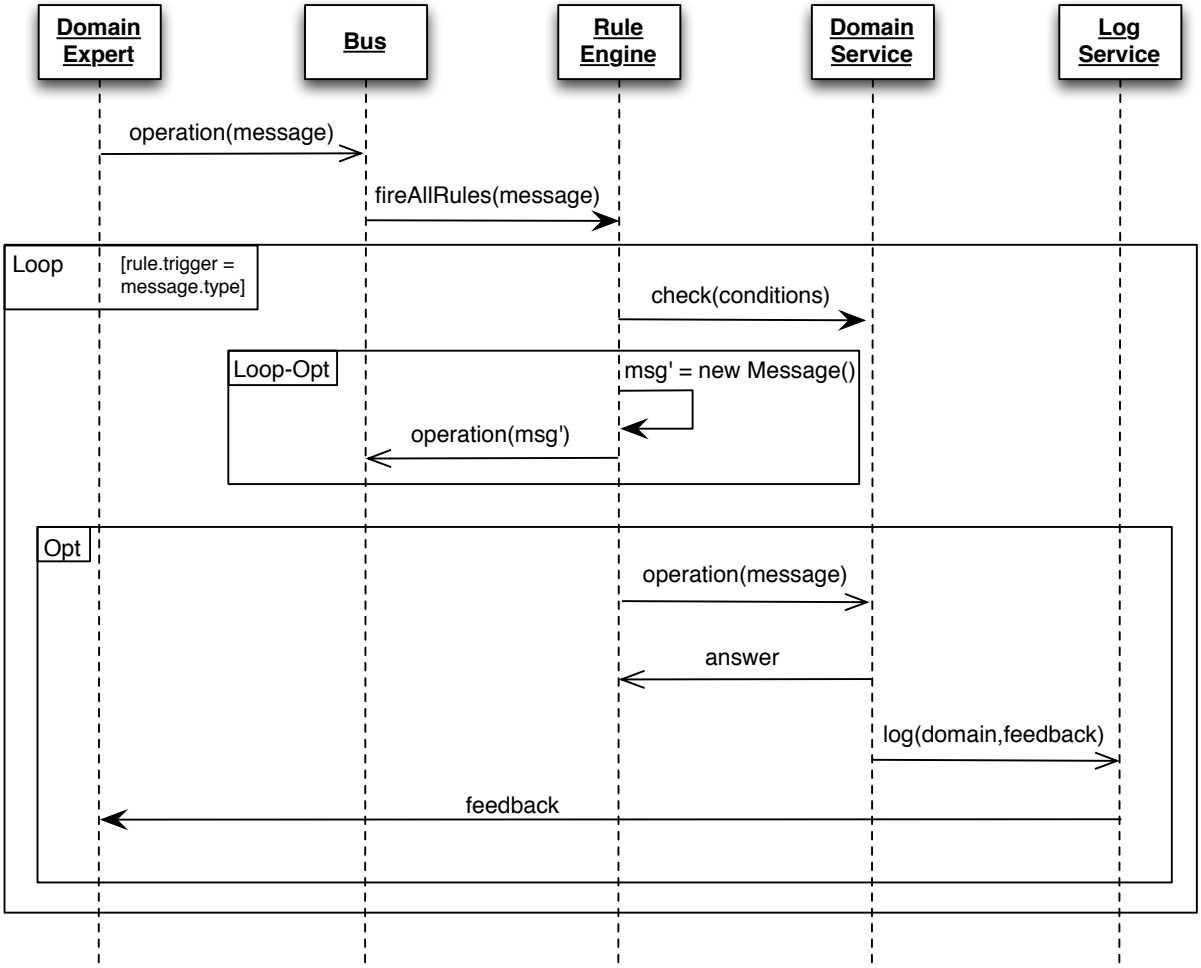


FIGURE 4.4 – Diagramme de séquences des interactions entre l'expert du domaine et le service.

DEF: RÈGLE DE ROUTAGE

$$r_{routage} = Message\ m \rightarrow Service.Operation(m)$$

Ainsi, l'expert adresse son message non pas directement au service qu'il cible, mais à un proxy qui le transmet au moteur d'intégration pour déclencher les règles pertinentes. Ces règles de routage assurent indépendamment de la gestion des interactions entre domaines, l'opération de modélisation voulue par l'expert sera exécutée.

Exemple 4.2: **Règle de routage sur `resolve()`**

$$routage_{resolve} = ResolveMsg\ msg \rightarrow IssueTrackingService.resolve(msg);$$

4.2.2 Définition d'une Règle d'Intégration

Les règles d'intégration centralisent les interactions entre domaines. Elles définissent l'impact de l'invocation d'une opération sur les autres services et ont donc par définition un effet de bord sur le système global. Une règle d'intégration est définie comme suit.

DEF: RÈGLE D'INTÉGRATION

$$r_{integration} : Message\ m \wedge Condition \rightarrow Reaction$$
$$Condition = expr_{bool}(\ \wedge\ expr_{bool}\ )^*$$
$$Reaction = Service.Operation(m)^* \mid Bus.send(newMessage(expr^*))^*$$

La condition du déclencheur permet d'envoyer des messages directement aux services pour tester l'état du système afin de déclencher ou non cette règle. Seules les opérations qui n'affectent pas l'état du système sont autorisées, *e.g.,* consulter les détails d'un ticket. La partie droite permet d'instancier de nouveaux messages à destination de n'importe quel service et de les transmettre au bus.

Du fait de la sémantique d'indépendance des règles, l'envoi d'un message m par un expert du domaine peut mener à l'exécution d'une règle de routage et d'une ou plusieurs règles d'intégration. La PROPRIÉTÉ 3.2 sur les opérations idempotentes et la sémantique d'exécution des messages détaillée dans le CHAPITRE 3 nous permet d'assurer que malgré la possibilité d'envois multiples du message m au service compétent par le moteur de règles, cette opération ne sera exécutée qu'une fois. Il n'y a donc pas de conflit entre les règles d'intégration déclenchées par un message et sa règle de routage.

Afin de satisfaire l'EXIGENCE 4.1, nous définissons la propriété d'accessibilité suivante sur la base de règles. Pour vérifier cette propriété, il est possible de définir une règle de routage pour chaque opération exposée aux experts. Si l'intégrateur décide de gérer plus finement les conditions dans lesquelles l'opération sera appelée, il peut ne pas définir de règle de routage et utiliser plusieurs règles d'intégration. Il est alors de sa responsabilité d'étudier la couverture des conditions de ces règles pour vérifier la propriété d'accessibilité. Cela peut servir par exemple à mettre le système dans un certain état par l'invocation d'une autre opération avant d'exécuter l'opération cible.

Propriété 4.4: **Accessibilité**

Parmi l'ensemble de règles ayant pour déclencheur un type de message donné, l'exécution d'au moins une de ces règles doit mener à l'invocation de l'opération cible, quel que soit le contexte du système.

Exemple 4.3: **Règle d'intégration sur `commit()`**

$integration_{commit}$ = (CommitMsg msg) $\wedge$ msg.desc.contain("#close #"+[0-9]*) $\rightarrow$

Bus.send(new ResolveMsg(msg.desc.substring("#close #", ISSUE_SIZE)))
VersioningControlSystemService.commit(msg) ;

4.3 Cohérence Inter-Domaines

4.3.1 Propagation de Déclenchement de Règles

Les règles métier ont la responsabilité de maintenir la cohérence de l'ensemble des méta-modèles intégrés. Pour maintenir la cohérence inter-domaine des modèles, une règle se déclenchant sur les messages destinés à l'opération o du service s peut instancier des messages de réaction destinés à d'autres opérations du service s ou à d'autres services. Ces messages ont eux-mêmes une influence sur le système, et doivent donc être gérés à leur tour par le moteur d'intégration. En adressant les messages créés par la partie droite de la règle aux proxys des services, ils seront eux-mêmes envoyés au moteur d'intégration par le bus. Chaque message de réaction déclenche alors les règles qui lui sont propres, selon l'état du système à ce moment. Cette propagation par réentrance dans le moteur d'intégration permet au service s de bénéficier des règles d'intégration conçues pour les autres services sans avoir à les connaître et assure qu'aucune invocation d'opération n'outrepasse le moteur d'intégration. Cela participe à la satisfaction de l'Exigence 4.3 en permettant au moteur d'intégration de connaître la chaîne de propagation des règles.

L'expressivité de la partie droite des règles métier est limitée par les interfaces des services. Cette limitation permet à l'intégrateur de travailler avec chacun des experts au bon niveau pour définir les interactions entre les domaines sans être dépendant des choix d'implémentation des méta-modèles. Cependant, la partie droite d'une règle n'a pas accès à toutes les actions élémentaires sur les modèles du système.

Dans le cas où une règle d'intégration détecte un conflit de cohérence entre deux domaines mais ne peut pas la résoudre, l'objectif est d'alerter le ou les domaines concernés en posant un blâme sur les modèles compromettant la cohérence et en retenant quelles actions ont été exécutées pour amener à cette incohérence. En pratique, nous utilisons un service auxiliaire de Notification, comme illustré dans la Figure 4.4, pour publier et consulter des informations concernant l'utilisation des services et pour signaler l'incohérence aux domaines. Chaque expert peut ainsi vérifier à tout moment l'état de cohérence des modèles de son domaine. L'intégrateur est responsable de la caractérisation des problèmes de cohérence, afin d'adresser à chaque expert du domaine un message approprié à sa connaissance métier qui aidera à la résolution du conflit. Ce service est fourni avec l'architecture et son fonctionnement est indépendant de l'implémentation des domaines. Il offre des opérations pour *(i)* ajouter un message à la boite de notification d'un domaine (`Publish()`), *(ii)* consulter les notifications d'un domaine (`Consult()`) et faire remonter une incohérence en plaçant un blâme sur un domaine (`RaiseConsistencyIssue()`). Une même incohérence peut mener à plusieurs blâmes sur des domaines différents. Les types de blâmes sont gérés par l'intégrateur comme un moyen de communiquer une inconsistance à un expert dans son langage métier.

4.3.2 Expressivité des Règles d'Intégration

Les règles d'intégration permettent donc de capturer les interactions et les conflits entre domaines pendant la modélisation du système. Il est à noter que ces règles sont n-aires puisqu'elles peuvent envoyer de nouveaux messages à plusieurs services différents. Cela permet à l'intégrateur d'exprimer des règles plus fines qu'un ensemble de règles binaires, car dépendantes de l'état de plusieurs domaines. De même, cela diminue le nombre de règles en les regroupant, et participe donc à faciliter le maintien de la base de règles. Toutefois, certains outils travaillent exclusivement sur des règles binaires. Nous étudions ici la possibilité d'exprimer une règle n-aire comme un ensemble de règles binaires pour démontrer l'expressivité supérieure des n-aires.

Démonstration 4.1: Passage d'un règle n-aire à n règles binaires

Soit {S1, S2, S3, S4} ∈ Services, {a, b, c, d} ∈ Opération,
Considérons la règle d'intégration suivante
Message(S1.a) ∧ Condition (S2.b ∧ ¬S3.c) → S4.d
Elle peut s'exprimer en logique du premier ordre sous la forme
S1.a ∧ S2.b ∧ S3.c ⇒ S4.d
≡ S1.a ⇒ (S2.b ⇒ (¬ S3.c ⇒ S4.d))) [a]

a. par transfert

La transformation d'une règle n-aire en n règles binaires détaillées dans la Démonstration 4.1 produit un ensemble de règles chaînées où la première règle déclenchée normalement par le message envoyé par l'expert doit dans sa partie droite provoquer explicitement le déclenchement d'une autre règle pour vérifier une condition, et ce autant de fois qu'il y a de conditions. Cette contrainte viole la Propriété 4.1 et la Propriété 4.2 et complexifie la maintenance de la base de règles. Cette équivalence n'est donc pas exploitable dans notre contexte.

Démonstration 4.2: Passage d'un règle n-aire à n-1 règles binaires

Soit {S1, S2, S3, S4} ∈ Services, {a, b, c, d} ∈ Opération,
Considérons la règle d'intégration suivante
Message(S1.a) ∧ Condition (S2.b ∧ ¬S3.c) → S4.d
Elle peut s'exprimer en logique du premier ordre sous la forme
S1.a ∧ S2.b ∧ S3.c ⇒ S4.d
≡ ¬ (S1.a ∧ S2.b ∧ ¬ S3.c) ∨ S4.d [a]
≡ (¬ S1.a ∨ S4.d) ∧ (¬ S2.b ∨ S4.d) ∧ (S3.c ∨ S4.d) [b]
≡ (S1.a ⇒ S4.d) ∧ (S2.b ⇒ S4.d) ∧ (¬ S3.c ⇒ S4.d) [c]

a. par définition de ⇒
b. par distributivité de ∨ sur ∧
c. par définition de ⇒

La transformation d'une règle n-aire en n-1 règles binaires détaillées dans la Démonstration 4.2 produit un ensemble de règles interdépendantes où chacune ne doit être déclenchée que si elles sont toutes déclenchées. Cette contrainte viole la Propriété 4.1 et entre en contradiction avec la sémantique d'exécution d'un moteur de règles, cette équivalence n'est donc pas exploitable dans notre contexte.

4.3.3 Écriture et Composition de règles

Les règles de routages peuvent être automatiquement générées à partir des interfaces de service. Pour chaque message entrant défini par le service, une règle est générée, appelant l'opération associée. Les règles d'intégration doivent être conçues par l'intégrateur. En recueillant les interactions entre domaines avec l'aide des experts, il doit exprimer la réaction de chaque domaine à l'invocation d'une opération. Du fait de la Propriété 3.1 des opérations de services, statuant que le service ne conserve pas l'état d'un modèle entre deux invocations, le moteur d'intégration doit lui aussi pouvoir retrouver les instances concernées pour pouvoir s'enquérir de leur état et construire les messages pertinents à l'intégration. Pour cela nous avons défini dans notre architecture un service auxiliaire d'Association permettant aux experts de lier des modèles conformes à différents métamodèles ou des représentations différentes, chacun du point de vue de son domaine, d'un même concept.

Exemple 4.4: **Récupération du ticket correspondant à un commit**

Un message de commit envoyé par Nathalie doit contenir l'information nécessaire à l'opération `commit()` du service `VersionControlSystem` pour s'exécuter. Il existe une règle d'intégration inter-domaines concernant cette opération, dans le cas où la description du commit contient un numéro de tâche. Le moteur doit donc également retrouver l'instance correspondante du domaine `IssueTracking` à partir de ces données, pour vérifier qu'il existe une tâche possédant ce numéro. Si le service d'Association donne une réponse satisfaisante, un message destiné à l'opération `resolve()` de service `IssueTracking` peut être émis. Il sera à son tour intercepté par le moteur d'intégration et déclenchera les règles pertinentes.

Pour ajouter un nouveau service de domaine, étant donnée une base de règles existantes, l'intégrateur peut choisir d'affiner la gestion de la cohérence en couvrant de nouveaux cas de détection d'incohérence. Pour cela il étudie *(i)* l'impact de ce nouveau service de domaine sur les pré-existants, *e.g.,* une règle par opération exposée par le nouveau service, et *(ii)* comment ce domaine est impacté par les services existants. Dans ce cas, l'intégrateur n'a pas besoin d'éditer les règles pré-existantes du fait de la Propriété 4.1. Si toutefois le nouveau domaine est lié à la gestion même de la cohérence, par exemple un service de gestion de la sécurité définissant les droit d'accès, l'intégrateur peut retravailler la base existante de règles pour en affiner l'expressivité.

Afin de diminuer la complexité d'une base de règles, il est possible de composer par fusion les règles automatiquement dans les cas suivants :

— Si deux règles R_1 et R_2 possèdent une partie droite strictement égales.
$R_1 : \bigvee\{c_i...c_j\} \in Condition \rightarrow \{a_i...a_j\} \in Action$

$R_2 : \bigvee\{c_n...c_m\} \in Condition \rightarrow \{a_n...a_m\} \in Action$
tels que $\{a_i...a_j\} = \{a_n...a_m\}$
$\equiv R_{12} \bigvee\{c_i...c_j, c_n...c_m\} \rightarrow \{a_i...a_j\}$

— Si deux règles R_1 et $R_{1'}$ possèdent une partie gauche strictement égales.
$R_1 : \bigvee\{c_i...c_j\} \in Condition \rightarrow \{a_i...a_j\} \in Action$
$R_{1'} : \bigvee\{c_n...c_m\} \in Condition \rightarrow \{a_n...a_m\} \in Action$
tels que $\{c_i...c_j\} = \{c_n...c_m\}$
$\equiv R_{1''} \bigvee\{c_i...c_j\} \rightarrow \{a_i...a_j, a_n...a_m\}$

4.4 Implémentation

L'implémentation du moteur de règles repose sur l'utilisation d'un système de gestion des règles métier (SGRM) de l'état de l'art : Drools [1] [Browne 09]. Une banque de règles est conçue par l'intégrateur, en accord avec les experts domaines pour ce qui est des interactions. Le moteur est branché sur le bus et reçoit les messages envoyés par les experts. Il est responsable du déclenchement des règles pertinentes pour ce message. Pour cela, il travaille sur un ensemble de faits de sa base de connaissance, dont la structure du message intercepté et les valeurs qu'il contient. D'autres faits peuvent être ajoutés à la base de connaissance, pendant l'évaluation de la partie gauche des règles, en consultant l'état du système via les opérations exposées par les services par exemple.

Le déclenchement des règles par Drools est basé sur l'algorithme de filtrage par motif de Rete [Forgy 82] : au lieu d'une passe naïve sur toutes les règles, le moteur produit un graphe à partir des parties gauches des règles, dont les nœuds autres que la racine représentent l'état d'une condition. En parcourant le graphe selon les faits de la base de connaissance, le moteur sait alors quelles règles déclencher. L'ordre de déclenchement des règles est alors établi de façon arbitraire par le moteur. Le moteur détecte automatiquement les conflits, par exemple l'existence de cycles dans la base de règles, et propose des moyens de les résoudre, en déclarant qu'une règle donnée ne peut être exécutée qu'une fois par évaluation ou en indiquant une priorité sur certaines règles métier.

Notre implémentation, utilisant le moteur de règles Drool, respecte donc la Propriété 4.2 et la Propriété 4.3. Nous adaptons ainsi l'expressivité des règles de l'outil pour convenir à des services.

Afin d'intercepter les messages émis par les experts, le bus expose des proxys de services, *i.e.,* des classes générées automatiquement à partir de l'interface de chaque service. Chaque proxy expose les mêmes opérations que le service qu'il cache et n'a pour seul responsabilité que de transmettre le message au moteur d'intégration.

Exemple 4.5: **Implémentation Drools des règles de `commit()` et `resolve()`**

```
1 /** Routing Rules **/
  //  ResolveMsg msg → resolve(msg)
3 rule "Resolve Issue"
    no-loop true
5   when
```

1. http ://www.drools.org/

```
    $msg : ResolveMsg()
  then
    IssueTrackingService.resolve($msg);
end

//  CommitMsg msg → commit(msg)
rule "Commit Code"
  no-loop true
  when
    $msg : CommitMsg() and
  then
    VersioningControlSystemService.commit($msg);
end

/** Integration Rules **/
// CommitMsg msg ∧ hasLinked() → (newResolveMsg(...), commit(msg))
rule "Commit Code"
  no-loop true
  when
    $msg : CommitMsg( $desc : desc, $file : file )
    and $asw : RetreiveLinkedAsw( $project : linked )
    from AssociationService.retreiveLinked( new
       RetreiveLinkedMsg($file) )
  then
    $issue = $desc.substring("#close #",ISSUE_SIZE);
    Bus.send( new ResolveMsg($issue) );
    VersioningControlSystemService.commit($msg);
end
```

```
public class IssueTrackingProxy  {
        public static void open(OpenMsg msg){
                Scenario.bus.handle(msg);
        }
        public static void followUpdate(FollowUpdateMsg
            msg){
                Scenario.bus.handle(msg);
        }
        public static void resolve(ResolveMsg msg){
                Scenario.bus.handle(msg);
        }
}
```

4.5 Conclusions

L'externalisation de la gestion des interactions entre domaines permet à l'intégrateur de concevoir et de maintenir une base de règles pertinentes pour les domaines considérés. L'utilisation d'un outil industriel comme Drools, basé sur des fondamentaux solides,

assure à la couche de médiation la robustesse nécessaire, et l'adaptation à notre contexte d'intégration de services permet une évolutivité dans le nombre de domaines intégrés. L'algorithme Rete est connu pour permettre un passage à l'échelle sur des bases de règles de grandes tailles, mais est consommateur d'espace mémoire du fait de la création et du maintien du graphe sous-jacent.

Cette intégration repose donc sur la conception d'une base couvrante de règles métier, ce qui est un défi connu du paradigme SOA et un problème de recherche ouvert [Nalepa 09, Bona 11]. Cette communauté apporte des bonnes pratiques durant la définition des règles métier, à la fois au sein de livres de référence [Halle 01, Taylor 11], et à travers la commercialisation de méthodes industrielles telles qu'ISSIS d'IBM [2]. De plus, des méthodes d'analyse de règles aident leur conception et la recherche de leur atomicité.

> "*Breaking complex business rules into several simpler more atomic ones, detecting redundancies, overlaps or contradictions between rules, documenting the business motivations of rules*" [Boyer 11]

L'approche à base règles métier permet à l'intégrateur de définir explicitement les interactions entre domaines provoquant une incohérence du système global. La couche de médiation présentée dans ce chapitre permet d'assurer aux experts (*i*) l'accessibilité des opérations de leur service, (*ii*) la possibilité de continuer la conception des sous-systèmes en cas d'incohérence du système global et (*iii*) la remontée d'informations sur l'incohérence lors du déclenchement d'une règle.

La sémantique d'exécution du moteur de règles permet de détecter automatiquement les cycles sur une base de règles définies indépendamment, et dont l'ordre d'exécution n'est pas prédéfini, ce qui facilite le maintient de la base par l'intégrateur.

2. http ://www-304.ibm.com/services/learning/ites.wss/zz/en ?pageType=page&c=M005457R05249Y27

CHAPITRE 5

CONCRÉTISATION ET VARIABILITÉ TECHNOLOGIQUE

5.1 Problématique de variabilité dans la concrétisation du système

5.1.1 Influence de la variabilité dans la concrétisation du système

Les chapitres précédents décrivent comment les experts des domaines peuvent instancier leur méta-modèle à travers des interfaces de services décrites dans le Chapitre 3. Les modèles ainsi produits sont cohérents par construction vis-à-vis de leur domaines respectifs et entre eux, du fait des propriétés du moteur d'intégration décrit dans le Chapitre 4. Ces modèles restent à un niveau d'abstraction au dessus du code, ce qui facilite le raisonnement et la manipulation du système. Toutefois, afin d'être utilisable, le système final doit être à un niveau d'abstraction inférieur, exécutable par une machine. L'objectif de ce chapitre est de concrétiser le système modélisé par ces modèles dans le but d'obtenir un système exécutable conforme aux choix de conception effectués par les experts domaines. Classiquement, l'obtention d'un système à partir de l'instanciation d'un modèle se fait à l'aide d'un générateur de code [Fowler 10, Kelly 08, Herrington 03]. Ce générateur peut prendre plusieurs formes, allant d'une transformation formelle *model to text* (M2T), dans un langage spécifique de type *Query - View - Transformation* (QVT) [1] comme ATL [Jouault 06] ou Acceleo [Musset 06], ou bien l'implémentation du patron de conception visiteur utilisant l'arbre de composition du méta-modèle pour concrétiser chacun de ses éléments.

Dans le cadre de cette thèse nous considérons des domaines qui apportent une multitude de codes concrets existants provenant de différentes sources. Nous souhaitons considérer cet ensemble de sources comme des artefacts de code sur étagère, disponibles pour une réutilisation. Au sein d'un même domaine, les artefacts ne répondent pas tous au même besoin et ne proposent pas les mêmes fonctionnalités. De plus, le nombre de ces artefacts, mais aussi les fonctionnalités de chacun évoluent quotidiennement dans des domaines comme l'Internet des Objets (IoT), le Cloud Computing et les applications Web couvrant de nombreux espaces de solution. Cette problématique de caractérisation et de sélection d'éléments parmi une famille d'artefacts est classiquement résolue par des méthodes de gestion de la variabilité. Les Lignes de Produits Logicielles (SPLs) sont communément utilisées pour capturer la variabilité d'un ensemble d'artefacts logiciels et fournir des outils d'assistance à la configuration pour sélectionner un produit valide

1. http ://www.omg.org/spec/QVT/1.1/

parmi cet ensemble qui convienne au contexte d'exécution. Nous définissons l'ensemble des artefacts réutilisables d'un domaine comme son *espace des solutions*.

Exemple 5.1: **Concrétisation d'un modèle de Gestion de Tickets**

Pascal cherche à concrétiser son modèle de gestion de tickets, *i.e.,* choisir l'outil concret de gestion des tickets répondant aux besoins qu'il a exprimé durant la modélisation et y acter les tickets modélisés. Il existe des outils séparés ciblant une partie spécifique de son domaine, par exemple Kanban pour le statut des tickets et le Planning Poker pour l'estimation de leur difficulté. Certains outils proposent une gestion globale, englobant tous les aspects de son domaine, mais imposent des contraintes sur leurs fonctionnalités, *e.g.,* SCRUM impose des contraintes les deux aspects précédant. Sans processus automatisé, Pascal devrait analyser les capacités et les contraintes de chacune des solutions existantes, qui évoluent chaque jour, et faire un choix en fonction des besoins de chaque projet qu'il gère.

Ce chapitre décrit une contribution permettant de concrétiser un système modélisé à un niveau d'abstraction supérieur au code. Cette contribution bénéficie de l'existence d'artefacts concrets dans le domaine considéré. Cette concrétisation doit donc gérer la multitude des artefacts à disposition, l'hétérogénéité des sources de ces artefacts et assister les experts du domaines en définissant un processus de concrétisation le plus automatisé possible. Cette étape permet de réifier les méta-éléments abstraits d'un modèle en choisissant des artefacts logiciels concrets appropriés pour les fonctionnalités voulues, à l'aide de SPLs. Nous définissons donc les exigences suivantes.

Exigence 5.1: **Hétérogénéité**

L'espace des solutions considéré par le système de gestion de la variabilité doit pouvoir prendre en considération des sources hétérogènes d'artefacts concrets, *e.g.,* différents fournisseurs, différents langages de programmation, différents paradigmes.

Exigence 5.2: **Multiplicité**

La génération du système exécutable doit pouvoir tirer profit de l'existence d'une multitude d'artefacts logiciels réutilisables sans imposer à l'expert de considérer manuellement chacune de ces solutions lors de la concrétisation de son modèle.

Exigence 5.3: **Semi-automatisation**

Le système de gestion de la variabilité doit guider les experts durant la concrétisation du système en minimisant le nombre de choix explicites demandés qui requièrent une connaissance de l'ensemble des artefacts disponibles.

5.1.2 État de l'art en variabilité

Les recherches sur les Lignes de Produits Logicielles [Clements 01] apportent depuis le début des années 2000 des techniques de gestion de la variabilité. Dans cette section, nous proposons de caractériser ces contributions en fonction des exigences de notre contexte de concrétisation d'un système de systèmes.

Le paradigme d'ingénierie des SPLs a initialement été défini comme des processus complémentaires [Pohl 05]. Ces axes présentent des défis propres et font l'objet de contribution spécifiques.

L'ingénierie du domaine : responsable de la représentation de la variabilité d'une famille logicielle, cet axe regroupe les préoccupations de modélisation des points communs et variables d'une ligne de produit, classiquement sous la forme d'un *Feature Model*. De nombreuses notations existent et présentent des différences dans la nature de leur représentation des fonctionnalités et des contraintes (textuelle ou graphique) et dans leur expressivité (*e.g.,* cardinalités, attributs, couches, versions) [Jézéquel 12, Benavides 10, Schobbens 06, Czarnecki 04].

Def: Domain Engineering

"Domain engineering is the process of software product line engineering in which the commonality and the variability of the product line are defined and realised." [Pohl 05]

L'ingénierie des applications : responsable de la production d'une application par réutilisation d'artefacts pertinents. Une fois une configuration valide et complète sélectionnée, la SPL doit permettre la dérivation du produit correspondant, *i.e.,* fournir des mécanismes de réutilisation d'artefacts de code concrets, réutilisables et configurés pour le fonctionnement du système final. La nature des artefacts considérés étant variable (*e.g.,* un composant, un document de conception, une base de données de spécifications ou des procédures de tests [Withey 96]), les approches de dérivation nécessitent un compromis entre l'automatisation, la généricité et la flexibilité du procédé [Perrouin 08].

DEF: APPLICATION ENGINEERING

"Application engineering is the process of software product line engineering in which the applications of the product line are built by reusing domain artefacts and exploiting the product line variability." [Pohl 05]

Dans le cadre de cette thèse, la concrétisation d'un système modélisé en utilisant plusieurs méta-modèles hétérogènes nécessite une gestion de la variabilité orthogonale à ces considérations. La modélisation de la variabilité et la dérivation d'un produit exécutable final doivent supporter la prise en compte d'une multitude d'artefacts hétérogènes (*cf.* EXIGENCE 5.1 et EXIGENCE 5.2), quel que soit le domaine concerné. De plus, la sélection d'un produit parmi une famille de logiciels doit être automatisée (*cf.* EXIGENCE 5.3), bénéficiant des choix de modélisation de l'expert lors de l'instanciation des modèles décrivant le système. Chacun de ces critères est satisfait par des contributions différentes en SPL. L'originalité de notre contribution repose dans l'agencement de ces approches en un processus cohérent répondant aux exigences de notre contexte.

5.1.3 Modélisation de la variabilité par les Feature Models

DEF: SOFTWARE PRODUCT LINES (SPLS)

"A set of software-intensive systems that share a common, managed set of features and that are developed from a common set of core assets in a prescribed way." [Pohl 05]

L'utilisation de SPLs pour la gestion de la variabilité d'un espace de solutions repose sur l'identification *(i)* des fonctionnalités communes à chacun des artefacts considérés, *(ii)* des points de variation dans leurs fonctionnalité et *(iii)* des relations de dépendance entre ces fonctionnalités. Ces trois aspects sont classiquement modélisés au sein d'un *Feature Model* (FM) [Kang 90]. Un FM est une représentation arborescente de toutes les fonctionnalités offertes par l'espace de solutions, appelées *feature*, qui en forment les nœuds. Les relations entre ces fonctionnalités disposent d'une représentation graphique sur les arêtes. Un exemple simplifié est présenté dans la FIGURE 5.1.

DEF: FEATURE MODEL (FM)

"A feature model represents a hierarchy of properties of domain concepts [...] used to discriminate between concept instances." [Riebisch 03]

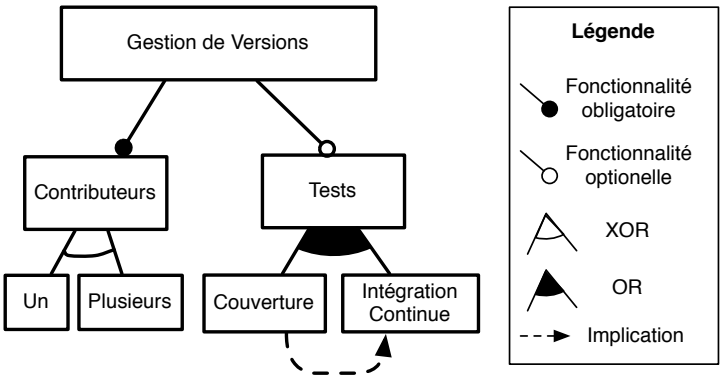


FIGURE 5.1 – Extrait d'un *Feature Model* pour la Gestion de Versions.

Exemple 5.2: **Concrétisation d'un modèle de Gestion de Versions**

La concrétisation d'un modèle de Gestion de Ticket conçu par Nathalie revient à :

- sélectionner une solution concrète pour le gestionnaire de versions parmi toutes celles qui existent (*e.g.,* Github, BitBucket, Google Code, Tortoise, Gogs), conforme aux besoins modélisés (centralisé ou décentralisé, public ou privé, etc),
- concrétiser le *repository* par une commande de création spécifique (*e.g.,* `git clone git://github.com/sample.git`),
- concrétiser chaque commit en une commande spécifique au gestionnaire de versions retenu (*e.g.,* `git add *.java; git commit -m 'version initiale du projet'`) en changeant de branche auparavant si nécessaire.

Nathalie a donc besoin de *sélectionner* un produit pertinent dans l'espace de solutions à sa disposition en terme d'outils de gestion de versions. Lors de la concrétisation, chacune des actions qu'elle a modélisé est dérivée en un bout de code concret, par exemple un appel à l'API REST dépendant de l'outil concret sélectionné pour créer le dépôt de code pour ce projet.

Concrétiser un système sur un domaine donné revient, pour chaque élément du modèle à concrétiser, *(i)* à sélectionner un artefact concret offrant des fonctionnalités pertinentes, *(ii)* ce qui implique d'avoir en amont construit un catalogue de produits dont on connaît les fonctionnalités à partir de l'espace de solutions considéré. Un FM modélise par définition les fonctionnalités d'un ensemble de produits, alignés selon un vocabulaire commun, ce qui en fait une représentation adaptée pour un catalogue (*cf. (ii)*). La construction du catalogue d'un domaine est présentée dans la SECTION 5.2. Il est également possible de transformer la représentation arborescente d'un FM en une formule logique du premier ordre, compatible avec des outils de l'état de l'art en matière de résolution, utiles pour la sélection d'un produit pertinent (*cf. (i)*), ce que nous détaillons dans la SECTION 5.3.

Afin de répondre aux exigences formulées, nous proposons une gestion de la variabilité en modélisant l'ensemble des artefacts logiciels concrets d'un domaine sous la forme d'une SPL. Le FM résultant sert à sélectionner les artefacts pertinents à partir des modèles produits par les experts. Un générateur de code produit le système final à partir de

l'ensemble des produits sélectionnés et des artefacts de codes correspondant. Une vue d'ensemble de cette contribution est illustrée par la FIGURE 5.2.

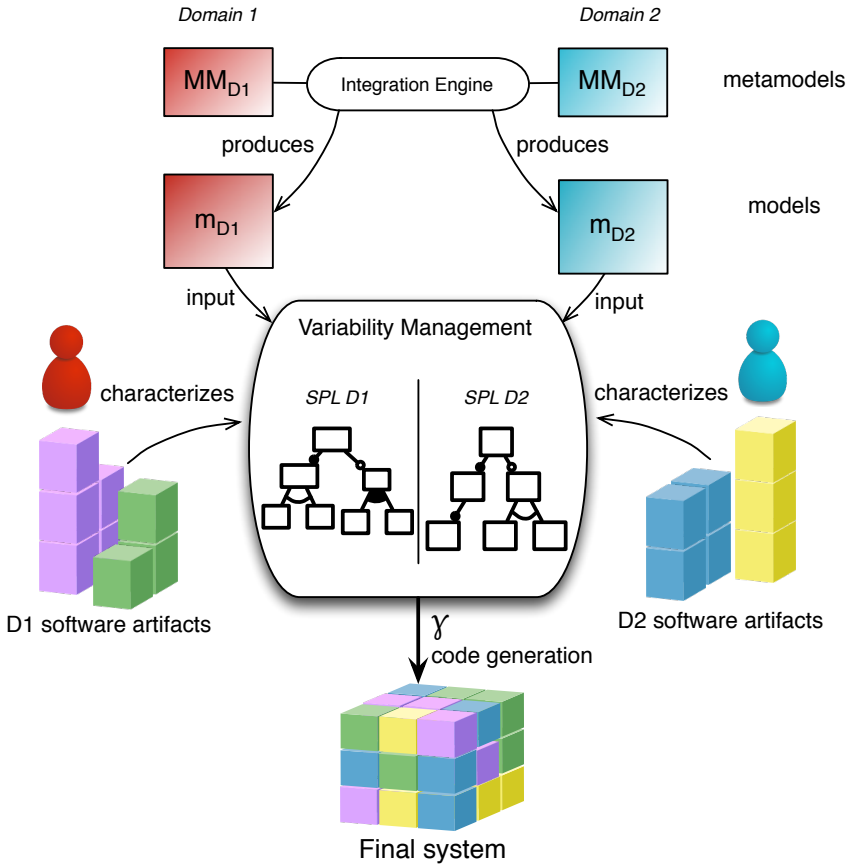


FIGURE 5.2 – Schéma des acteurs de la gestion de la variabilité.

5.2 Création d'un catalogue

La principale difficulté est maintenant de construire le modèle de variabilité. Nous souhaitons produire un modèle de variabilité permettant d'exprimer les capacités des produits vis-à-vis des points de variations du domaine en matière de fonctionnalités. Le procédé détaillé dans cette section est illustré par la FIGURE 5.3. Dans ce but, nous utilisons une méthode outillée qui repose sur la conception de FMs [Kang 98, Batory 05] pour modéliser la variabilité, ainsi que sur l'usage d'un opérateur de fusion sur ces FMs [Acher 09] (noté μ), dont le comportement est décrit dans la FIGURE 5.4.

Exemple 5.3: Variabilité dans la Gestion de Versions

Nathalie, en charge de gérer le code du projet, a le choix entre plusieurs outils de gestion de versions *e.g.,* Github, BitBucket, Google Code, Tortoise, Gogs, reposant sur des technologies différentes *e.g.,* SVN, Git, Mercurial. Dans son choix, elle souhaite considérer les critères suivants :

— L'accès au code du projet est-il public ou privé, pour un seul ou plusieurs contributeurs ?

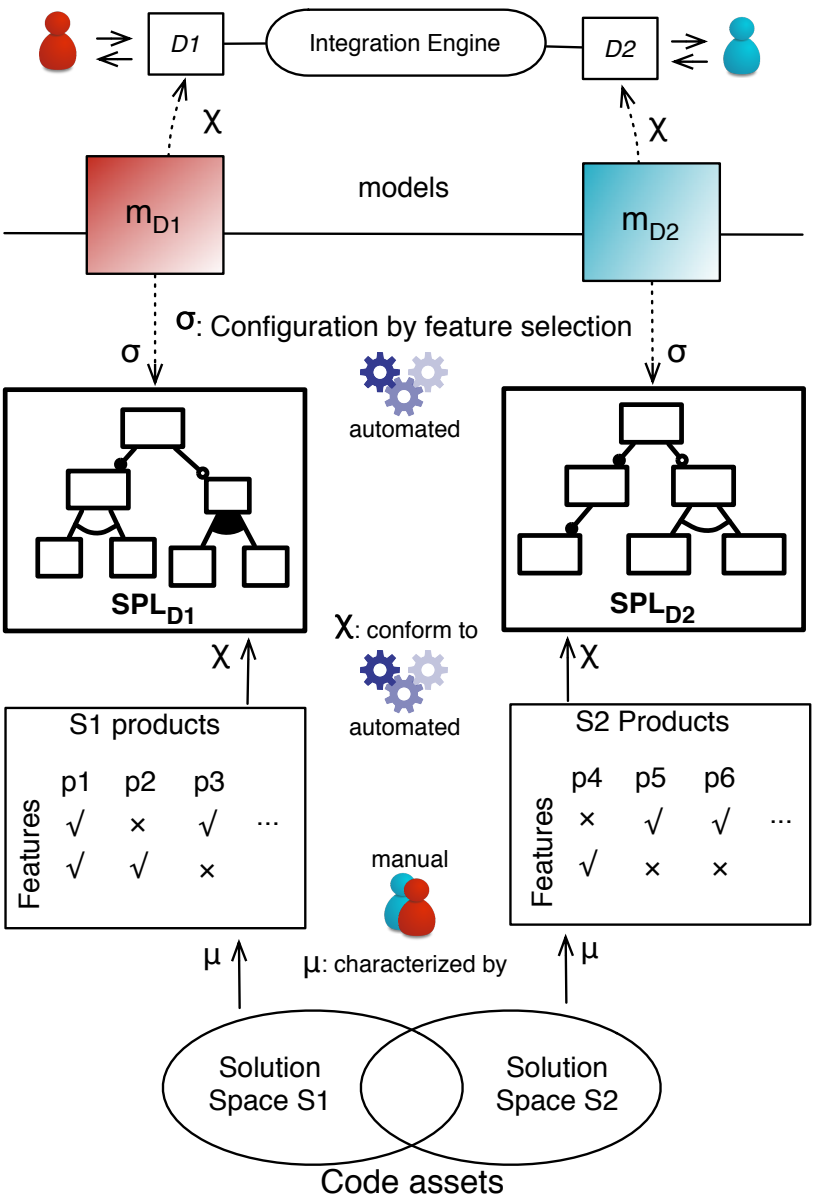


FIGURE 5.3 – Schéma de la gestion de la variabilité.

— La persistance des versions doit-elle être centralisée pour des questions de confidentialité, ou décentralisée pour assurer une sauvegarde distante ?

— Le gestionnaire doit-il autoriser l'intégration continue et l'analyse de la couverture de tests ?

— Une page HTML doit-elle être disponible pour le projet, pour le présenter en Markdown ou pour afficher la documentation par exemple ?

C'est donc selon ces caractéristiques (*i.e.,* le contrôle d'accès, le nombre de contributeurs, le mode de persistance, les tests et l'affichage) et leurs variants respectifs (public ou privé, mono-acteur ou multi-collaborateurs, centralisé ou décentralisé, ...) que Nathalie choisira un gestionnaire de version plutôt qu'un

autre et ce sont donc les critères que nous utilisons pour caractériser les produits concrets.

L'idée clef de cette méthode est de considérer en premier lieu les capacités de la multitudes des éléments de l'espace des solutions en produisant une matrice caractérisant chaque produit selon un vocabulaire commun de fonctionnalité appelée *feature*, spécifiant pour chaque *feature* si le produit la satisfait ou non. Le modèle de variabilité global est produit en fusionnant les caractérisation des produits, assurant ainsi sa validité par construction, puisqu'il représente uniquement les capacités des produits considérés. Cette technique est directement inspirée de la construction de FMs à partir de description de produits [Acher 12]. L'ensemble des FMs $\{s_1, \ldots, s_n\}$ est alors fusionné en utilisant l'opérateur μ qui implémente une "fusion avec strict union" [Acher 09]. En réutilisant l'opérateur μ, nous profitons de ses propriétés [Acher 11].

— Un produit est un ensemble valide de *features* modélisant un élément de l'espace des solutions : $p = [\![f_1, f_2, ..., f_n]\!]$

— Deux produits sont égaux s'ils sont modélisés par le même ensemble de *features* : $p_i = [\![f_1, f_2, \ldots, f_n]\!] \wedge p_j = [\![f_1, f_2, \ldots, f_n]\!] \implies p_i = p_j$.

— Un FM contient un ensemble de produits. Le nombre de produit d'un FM est défini par $|FM|$. $FM \ni [\![p_1, p_2, ..., p_n]\!], n = |FM|$.

— Une SPL est représentée par un FM fusionné, construit à partir d'un ensemble de FMs représentant les produits de la SPL : $FM_{SPL} = \mu(FM_1, FM_2, \ldots, FM_n)$.

— Le nombre de produits du FM fusionné est inférieur ou égal au nombre de FMs atomiques le constituant : $|FM_{SPL}| \leq |FM_1| + |FM_2| + \cdots + |FM_n|$.

— Chaque produit p représenté par le FM fusionné correspond à un produit d'un FM constituant de la SPL : $\forall p = [\![f_1, f_2, ..., f_n]\!] \in FM_{SPL}, \exists FM_i \ni p \wedge FM_i \subset FM_{SPL}$

— Chaque produit p représenté par un FM constituant de la SPL est un produit possible du FM fusionné de cette SPL : $\forall p \in FM_i, FM_i \subset FM_{SPL} \implies p \in FM_{SPL}$

Les propriétés de la fusion avec strict union sont utiles dans notre contexte pour vérifier par construction la validité du FM construit pour chaque domaine. Son utilisation dans notre approche répond au problème de création d'un catalogue. Il permet d'exploiter la connaissance des fonctionnalités de chacun des produits pour produire un seul modèle de variabilité représentant l'intégralité de l'espace des solutions.

Cette opération étant automatique, elle facilite l'ajout et l'édition de nouvelles descriptions de produits pour étendre l'espace des solutions considéré. Formellement, cette opérateur assure qu'étant donnés deux FMs s et s', le résultat de $\mu(s, s')$ peut être utilisé pour dériver les produits modélisés par s et ceux modélisés par s', sans ajout ni restriction. En conséquence, le résultat de $\mu(s_1, \mu(s_2, ...)) = s_S$ combiné aux descriptions et aux codes concrets des produits, implémente une SPL qui capture exactement tous les produits concrets disponibles dans l'espace des solutions S. Afin d'étendre cette ligne de produit avec un nouvel espace S', le même procédé est appliqué pour produire le FM $s_{S'}$ à partir de ses descriptions de produits, puis la ligne de produit complète est finalement obtenue par $S = \mu(s_S, s_{S'})$ comme illustré par la FIGURE 5.4. A noter, les FMs atomiques obtenus à partir de la caractérisation des produits sont dénués de variabilité,

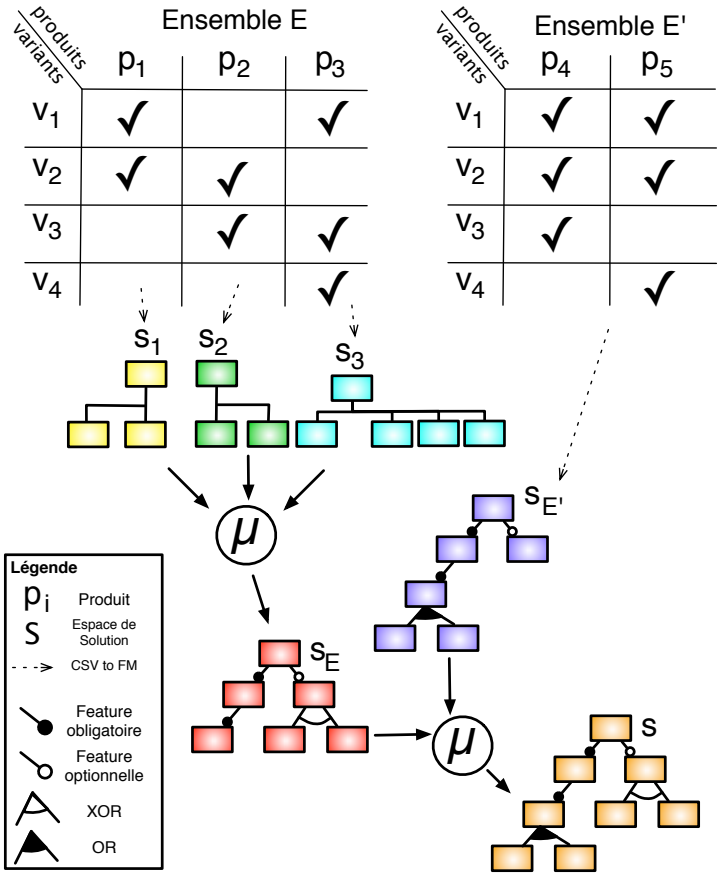


FIGURE 5.4 – Processus de fusion utilisé pour construire le modèle de variabilité.

par définition. C'est par application de l'opérateur μ que la variabilité émerge dans le FM résultant.

L'utilisation de l'opérateur μ sur les modèles de variabilité nous permet de vérifier l'hétérogénéité de l'EXIGENCE 5.1, puisqu'il permet de capturer les capacités de produits issus de différentes sources en les caractérisant selon les mêmes fonctionnalités. Il permet également vérifier la PROPRIÉTÉ 5.1 d'accessibilité de tous les artefacts considérés sous la forme de produits, *i.e.,* de configurations complètes et valides du FM fusionné, et la PROPRIÉTÉ 5.2 de non émergence de configuration valide qui ne correspondent à aucun artefact concret. Étant automatique, cette méthode de conception d'un FM participe au respect de l'EXIGENCE 5.3 de semi-automatisation.

Propriété 5.1: Accessibilité

Le modèle de variabilité global permet d'accéder à tous les artefacts concrets, conformément à leurs descriptions produits données en entrée.

Propriété 5.2: **Non-Émergence**

Seule les configuration correspondant à un produit de l'espace des solutions sont considérées valides par le modèle de variabilité global.

Exemple 5.4: **Enrichissement de l'espace des solutions**

La hiérarchie de Nathalie a décidé d'acheter des licences pour la suite logicielle "*TropBien*" d'outils de gestion de projet. Nathalie souhaite donc modéliser les deux solutions de gestionnaire de versions incluses, une légère et portable et l'autre plus complète, et d'ajouter ces produits à son modèle de variabilité existant qui ne contenait que des solutions gratuites. Pour cela, elle caractérise chacune des solutions dans une matrice conformément aux caractéristiques exprimées dans l'Exemple 5.3, puis génère un FM pour chacune et utilise l'opérateur de fusion μ pour obtenir une représentation de la variabilité des gestionnaires de versions "*TropBien*". En utilisant à nouveau l'opérateur μ sur ce nouveau FM et celui représentant les outils gratuits qu'elle utilisait auparavant, elle enrichit sa modélisation de la variabilité et augmente ses options de concrétisation.

5.3 Sélection d'un produit

Étant donné un modèle de variabilité d'un domaine, un expert peut configurer un produit à l'aide d'une SPL en sélectionnant les fonctionnalités nécessaires [Svahnberg 05]. L'idée est de considérer que les artefacts réutilisables d'un domaine forment une famille de logiciels, dont la diversité encapsule les aspects communs et variables. Cette famille est alors représentée sous une forme qui facilite la réutilisation systématique. Dans la pratique, il s'agit d'un FM, un modèle interactif mettant en relation les artefacts concrets à disposition et les fonctionnalités haut-niveaux, issues du domaine, qu'ils partagent ou sur lesquelles ils divergent. Notre approche propose d'identifier pour chaque modèle *(i)* les concepts ayant besoin d'être concrétisés, et *(ii)* les caractéristiques qui peuvent être utilisées pour réduire la sélection d'un artefact concret.

Nous outillons donc chaque domaine avec un modèle de variabilité, modélisant les fonctionnalités possibles des instances du domaine. Chaque nœud, appelé *feature*, sous la racine peut être optionnel ou obligatoire. La sélection d'une *feature* fille implique toujours la sélection de sa *feature* mère. Un ensemble de *features* sœurs peut être regroupé sous un opérateur logique *OR* ou *XOR*. Un ensemble de contraintes logiques d'implication ou d'exclusion est attaché à l'arbre pour permettre d'exprimer des relations entre les *features* de branches différentes.

Exemple 5.5: **Sémantique du *Feature Model* du domaine de la Gestion de Versions.**

Le modèle de variabilité illustré par la FIGURE 5.5 présente un extrait de la variabilité manipulée par Nathalie sous la forme d'un *Feature Model*. La racine en est le concept à concrétiser, ici le gestionnaire de versions en lui-même. On définie qu'un gestionnaire de version doit avoir une politique de contrôle d'accès, de persistance et de nombre de contributeur, chacun ayant deux *features* fille disjointes implémentées par un opérateur *XOR*. Deux *features* optionnelles, la gestion automatisée des tests et la mise à disposition d'une page web pour le projet, possèdent chacune des *features* non disjointes implémenté par un opérateur *OR*. Trois contraintes sont représentées graphiquement par souci de lisibilité, par exemple la sélection des *features* `Privé` et `Plusieurs` implique l'invalidation de `Centralisée`, exprimable en logique par la formule $Prive \wedge Plusieurs \rightarrow \neg\ Centralisee$.

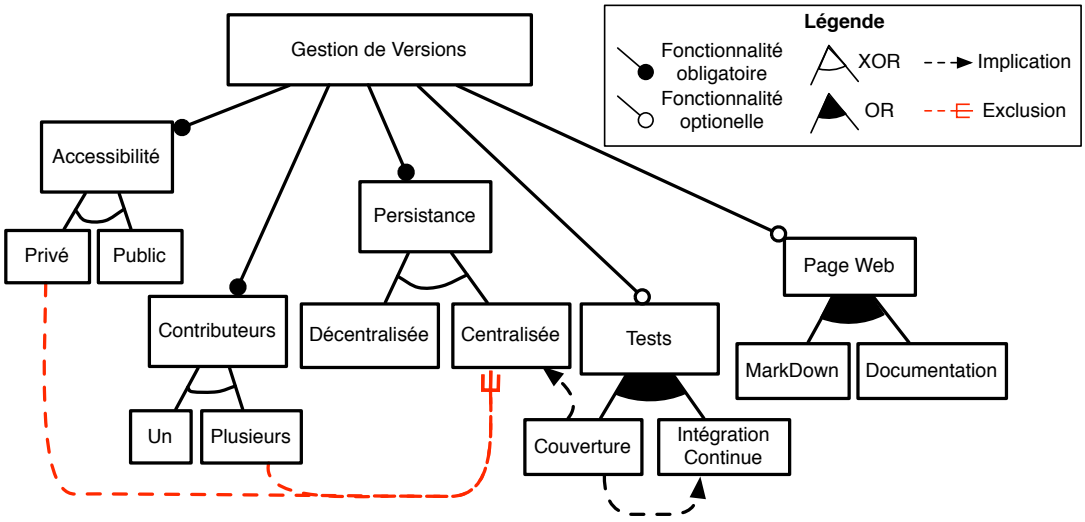


FIGURE 5.5 – *Feature Model* du domaine de la Gestion de Versions.

DEF: CONFIGURATION

Processus itératif de sélection, de désélection ou d'invalidation de fonctionnalités sur une famille logicielle dans le but de converger vers un produit concret valide.

L'objectif est de permettre la configuration d'un produit concret à partir d'un modèle produit par l'expert, *i.e.,* d'une instance valide du méta-modèle de son domaine. Cette configuration est par essence une activité itérative puisqu'elle s'effectue en parallèle de la modélisation d'un système par l'expert. Ainsi nous souhaitons faire avancer la configuration à chaque contribution de l'expert, à partir des choix de modélisation qu'il

effectue, en sélectionnant, déselectionnant ou invalidant une ou plusieurs *features*. Afin de permettre la sélection itérative de fonctionnalités conformes au modèle considéré, nous avons besoin de vérifier si la configuration partielle est valide. Cela revient à vérifier, à chaque étape, s'il existe au moins un produit satisfaisant l'ensemble des fonctionnalités choisies, *i.e.,* l'union de celles choisies aux étapes précédentes et l'étape en cours.

Cette tâche est communément résolue par une analyse sur de la logique propositionnelle [Janota 08], puisque que le FM et ses contraintes attachées peuvent être transformées en une forme normale conjonctive (CNF) [Batory 05], formant une sous classe nommée CNF-FMs. Les problèmes de satisfaisabilité booléenne dérivés des formes CNF-FMs, *i.e.,* la configuration interactive et le calcul d'un espace de solution valide, sont résolus efficacement par un solveur SAT [Mendonca 09]. Nous interrogeons un modèle de variabilité pour décider si les choix effectués pendant la conception d'un modèle par l'expert ont réduit l'espace des solutions à une seul, aucun ou plusieurs produits valides. La configuration est valide si et seulement si la sélection de l'ensemble des *features*, et la désélection des autres *features* mènent à un produit concret existant. La configuration est invalide si certaines *features* sélectionnées sont incompatibles, ou si une *feature* a été invalidée tout en étant requise par une *feature* sélectionnée, ce qui signifie qu'aucun produit ne satisfait cet ensemble de sélections.

Exemple 5.6: **Configuration d'un Gestionnaire de Versions**

Nathalie souhaite pouvoir s'assurer que son gestionnaire de versions est capable d'assurer les fonctionnalités que son projet nécessite. Pour cela, elle s'appuie sur le modèle de fonctionnalité illustré par la FIGURE 5.5, représentant les fonctionnalités caractéristiques de son domaine ainsi qu'un sous-ensemble des contraintes attachées. Lorsqu'elle conçoit dans son modèle un projet d'accès privé pour plusieurs développeur, les fonctionnalités `Privé` et `Plusieurs` sont sélectionnées. Par jeu de contraintes, `Décentralisée` est automatiquement invalidée. A ce stade plusieurs gestionnaires de version du marché sont encore envisageables. Toutefois, lorsqu'elle décide d'activer l'analyse de la couverture de code, le modèle de variabilité détecte qu'aucun produit n'offre cette fonctionnalité sans persistance décentralisée, aucun produit viable n'est alors disponible, la configuration en cours n'est pas valide et Nathalie doit donc revenir sur ses choix pour converger vers un produit de son espace de solutions.
Formellement, la représentation sous forme de FM peut être traduite en une formule logique où chaque *feature* est une variable booléenne. Par exemple, le FM présenté dans la FIGURE 5.5 est équivalent à :
$((Prive \wedge \neg Public) \vee (\neg Prive \wedge Public))$
$\wedge\ ((Un \wedge \neg Plusieurs) \vee (\neg Un \wedge Plusieurs))$
$\wedge\ ((Decentralise \wedge \neg Centralise) \vee (\neg Decentralise \wedge Centralise))$
$\wedge(Couverture \vee \neg Couverture)$
$\wedge(Integration \vee \neg Integration)$
$\wedge(MarkDown \vee \neg MarkDown)$
$\wedge(Documentation \vee \neg Documentation)$
$\wedge((Prive \wedge Plusieurs) \implies \neg Centralise)$
$\wedge(Couverture \implies Centralise)$
$\wedge(Couverture \implies Integration)$

Le besoin ici est de pouvoir calculer à chaque itération si la formule est toujours satisfiable, dans un temps d'exécution raisonnable pour un modèle interactif. Une itération consiste à fixer une valeur booléenne pour une des *features*. En convertissant cette formule sous une forme normale conjonctive comme ci-dessous, nous pouvons réutiliser la capacité des solveurs SAT à calculer sa satisfiaibilité, et ce à chaque étape de la configuration.
$(Prive \vee \neg Prive)$
$\wedge(Prive \vee Public)$
$\wedge(\neg Public \vee \neg Prive)$
$\wedge(\neg Public \vee Public)$
$\wedge(Un \vee \neg Un)$
$\wedge(Un \vee Plusieurs)$
$\wedge(\neg Plusieurs \vee \neg Un)$
$\wedge(\neg Plusieurs \vee Plusieurs)$
$\wedge(Decentralise \vee \neg Decentralise)$
$\wedge(Decentralise \vee Centralise)$
$\wedge(\neg Centralise \vee \neg Decentralise)$
$\wedge(\neg Centralise \vee Centralise)$
$\wedge(Couverture \vee \neg Couverture)$
$\wedge(Integration \vee \neg Integration)$
$\wedge(MarkDown \vee \neg MarkDown)$
$\wedge(Documentation \vee \neg Documentation)$
$\wedge(\neg Prive \vee \neg Plusieurs \vee \neg Centralise)$
$\wedge(\neg Couverture \vee Centralise)$
$\wedge(\neg Couverture \vee Integration)$

Lorsqu'une configuration complète et valide est effectuée, il reste à produire un programme pertinent à partir d'un artefact concret réutilisable, configuré en fonction du contexte. Les approches à base de SPLs s'appuient classiquement sur des méthodes de dérivation de produits [Perrouin 08, Rabiser 10, de Souza 15]. Nous réutilisons donc ce principe en représentant les artefacts sous la forme de *templates* de code à trous paramétrables, en utilisant des String Templates [2]. Un artefact contient alors le corps de code nécessaire à son exécution, expose à la SPL les attributs variants comme des paramètres à fournir en entrée. Le moteur String Template (ST) permet d'instancier un artefact et de renseigner ses attributs.

Exemple 5.7: **Dérivation d'un ticket**

La modélisation du gestionnaire de versions par Nathalie a mené au produit Bitbucket du fait des fonctionnalités qu'elle a choisi. Elle souhaite désormais finir la concrétisation de son modèle en accédant au code correspondant à la mise en place d'un dépôt dans cette technologie. Un template de l'appel à l'API Rest d'Atlassian [a] pour créer un nouveau dépôt est donné en exemple

2. http ://www.stringtemplate.org/

ci-dessous. Les paramètres attendus, encadrés par des '$' dans le template et déclarés au début entre parenthèses, sont remplis en parcourant le modèle produit par Nathalie, ici le nom du projet.

```
1 repo(user, project_name) ::= <<
  curl -k -X POST --user $user$ "https://api.bitbucket.
     org/1.0/repositories" -d "name=$project_name$"
3 >>
```

a. https ://fr.atlassian.com/

A noter qu'un parallèle est à faire avec les trois interfaces proposées dans l'approche *Variation, Customization, Usage* (VCU) [Kienzle 16]. L'utilisation du *feature model* implémente une interface de *Variations* permettant de sélectionner la version la plus approprié en fonction du contexte, *i.e.,* du modèle conçu en amont. Le rôle de l'interface d'adaptation (*Customization*) est rempli par l'utilisation des templates qui permettent de personnaliser l'artefact concret à ce contexte, ici en y injectant les paramètres requis pertinents. Une extension possible de notre approche serait de proposer une interface d'*Usage*, *i.e.,* exposer pour chaque produit considéré ses fonctionnalités réutilisables. Ces informations peuvent être entrées au même moment que la caractérisation des *features* de chaque produit.

5.4 Implémentation

Étant donnée une matrice caractérisant les produits, nous proposons une transformation vers le langage de FM Familiar [Acher 13]. La matrice au format CSV liste les variants en ligne et les produits en colonne. L'expert du domaine indique pour chaque produit s'il satisfait ou non chaque variant avec une valeur booléenne. La transformation est automatique et produit un fichier contenant en ligne un FM par produit décrit. Nous fournissons alors une API au dessus de Familiar afin de manipuler ces FMs en créant un espace de solution sous la forme d'un FM fusionné. Ce dernier prend la forme d'une classe Java `SolutionSpace`, exposant des méthodes pour commencer une configuration, *i.e.,* sélectionner, déselectionner ou invalider des fonctionnalités du FM, connaitre à tout moment le nombre de produits encore disponibles selon l'état courant de la configuration, fusionner cet espace de solution avec un autre, etc. Le constructeur de cette classe est responsable de l'initialisation des FMs dans Familiar ainsi que de la fusion de ceux-ci en un seul modèle de variabilité par l'instruction Familiar `solutionSpace = merge sunion FM_*`.

Exemple 5.8: **Création du modèle de variabilité des gestionnaires de versions**

Pour produire un FM regroupant les solutions technologiques qui s'offrent à elle en matière de gestionnaires de versions, Nathalie commence par catégoriser les solutions qu'elle considère, *i.e.,* Github, Gogs et BitBucket. La caractérisation s'effectue selon les caractéristiques voulues, tirées de sa connaissance métier

TABLE 5.1 – Exemple d'une matrice de caractérisation des produits

Fonctionnalité \ Produit	Github	SVN	Bitbucket	...
Public	✓		✓	
Privé		✓	✓	
Acteur unique	✓	✓	✓	
Plusieurs acteurs	✓	✓		
Décentralisé	✓		✓	
Centralisé		✓		
Tests	✓	✓		
Intégration continue	✓			
Couverture de tests	✓			
Page web	✓			
Markdown	✓			
Documentation	✓			

du domaine, sous la forme d'une matrice exemplifiée par la TABLE 5.1. Il est à noter que la matrice donnée en exemple est volontairement simplifiée. Par exemple, puisque Github, Bitbucket offre des prestations différentes aux comptes gratuits et aux professionnels ou étudiants, chacune de ces formules devrait être un produit différent pour représenter correctement la variabilité de l'espace des solutions.
Cette matrice est donnée en entrée de la transformation automatique qui fournit en résultat un fichier contenant les trois FMs correspondant aux produits concrets ci-dessous.

```
FM_1("Gestion␣de␣Versions": Acces Contributeurs Persistance Tests PageWeb
    Name; Name: Github; Acces : Public Prive; Contributeurs : Un Plusieurs;
     Persistance: Decentralisee; Tests: Couverture "Integration␣Continue";
    PageWeb : Markdown Documentation ;)
FM_2("Gestion␣de␣Versions": Acces Contributeurs Persistance Name; Name:
    Gogs; Acces : Prive; Contributeurs : Un Plusieurs; Persistance:
    Centralisee;)
FM_3("Gestion␣de␣Versions": Acces Contributeurs Persistance Name; Name:
    BitBucket; Acces : Public Prive; Contributeurs : Un; Persistance:
    Decentralisee;)
```

Nathalie peut alors à tout moment créer un "*Espace des solutions*" à travers la classe `SolutionSpace` en fournissant le chemin du fichier des FMs produits. Elle peut alors commencer à réduire son espace de solution en sélectionnant la fonctionnalité `Décentralisé` et voir que le nombre de produits disponibles a diminué.

```
private List<Product> products;

/*
* Launch the evaluation on the file formulaPath, line by line.
* Declare the "atomic" features models in familiar.
*/
public SolutionSpace(String formulaPath) throws IOException {
    products = new ArrayList<>();
    // test the existence of the file
    if (!new File(formulaPath).exists())
      throw new IOException("Invalid␣path␣"+widgetsFormulaPath);
    List<String> inlineProducts = pilot.extractFMsByFile(formulaPath);
    for(String formula : inlineProducts)
```

```
        products.add(new FM(formula));
    this.fmID = pilot.merge(getProductsIDs());
}
```

La configuration d'un produit au sein de cet ensemble de solutions est effectuée en parallèle de la modélisation du système par l'expert. Une fois une configuration valide et complète atteinte, le seul produit concret atteignable est dérivé en retrouvant l'artefact de code correspondant stocké sous la forme de code à trous. Lorsqu'un template est ajouté au système, ses paramètres exposés sont liés aux attributs du méta-modèle du domaine correspondant. Lors de la concrétisation, les attributs de ce template sont remplis à partir des valeurs renseignées par l'expert dans son modèle, comme détaillé dans l'Exemple 5.9. Cette dérivation est automatique, à l'exception du renseignement d'informations nécessitant une intervention manuelle par nature, comme un mot de passe à l'authentification par exemple.

Exemple 5.9: **String Templates pour la dérivation de produits**

En modélisant son système de gestion de tickets, Pascal a convergé vers un produit concret : Jira d'Atlassian. Le ticket qu'il a modélisé est dérivé en un bout de code correspondant à l'appel sur l'API REST de l'outil permettant la création du ticket. La même approche est utilisée pour obtenir les appels nécessaires à la création du dépôts de tickets et à leur mise à jour.

```
issue(key, summary, desc) ::= <<
curl -u admin:admin -X POST --data @data.txt -H
"Content-Type: application/json" http://localhost:8080/
   jira/rest/api/2/issue/

data.txt = {
    "fields": {
        "project": { "key": "$key$" },
                "summary": "$summary$",
                "description": "$desc$",
                "issuetype": { "name": "Task" }
        }
}
>>
```

```
import org.stringtemplate.v4.*;
...
STGroup jiraGroup = new STGroupDir("./jira/",'$', '$');
ST issue = jiraGroup.getInstanceOf("Issue");
issue.add("key", model.getProject().getName());
issue.add("summary", model.getLastIssue().getID());
issue.add("desc", "model.getLastIssue().getDescription
   ());
String productCode = issue.render();
```

5.5 Conclusions

Dans ce chapitre, nous avons présenté notre contribution en matière de gestion de la variabilité technologique de nos domaines, en les outillant avec des lignes de produits logiciels (SPLs). L'existence de contributions indépendantes résolvant partiellement notre problème pose un défi de génie logiciel classique d'interopérabilité. L'originalité de notre approche repose donc dans l'agencement de solutions adaptées à notre contexte en un processus automatisable. L'objectif est d'assister l'expert dans la recherche d'une configuration complète et valide, c'est-à-dire d'un produit utilisable pour concrétiser le système. Les SPLs sont utilisées pour concrétiser un modèle du domaine, en configurant un produit à partir des fonctionnalités attendues. Nous utilisons des *Feature Models* (FMs) qui capturent la variabilité d'un domaine en modélisant les point communs et les variations entre les artefacts concrets d'un espace de solutions considéré. Notre approche permet de converger vers une configuration finale de chaque FM, *i.e.,* qui ne contient plus de variabilité, et valide, *i.e.,* dont l'ensemble des (dé)sélections de fonctionnalité mène à un produit. Afin de construire le catalogue de produits, nous proposons de générer automatiquement un FM atomique par produit à partir de la caractérisation de ses fonctionnalités. Le FM d'un domaine est alors obtenu à partir des FMs atomiques, en réutilisant un opérateur de fusion par union strict assurant par construction la validité du FM résultant. Cet opérateur respecte la Propriété 5.1 d'accessibilité de tous les produits de l'espace de solution et la Propriété 5.2 assurant que toutes les configurations valides du FM sont liées à un produit et donc à un artefact de code réutilisable. Ce procédé permet de capturer tous les produits disponibles, sans contrainte sur leur nombre, ce qui satisfait l'Exigence 5.2, ni sur leur source puisque le vocabulaire de caractérisation est abstrait des solutions technologiques, ce qui satisfait l'Exigence 5.1. Dans notre cas d'application, le vocabulaire des fonctionnalité est extrait du méta-modèle du domaine. Les fonctionnalités du modèle de variabilité peuvent donc être manipulées automatiquement lors de l'instanciation des modèles. La sélection de *features* sur le FM demande de vérifier la satisfiabilité après chaque étape de configuration, *i.e.,* s'il existe au moins un produit encore accessible. Ce calcul repose sur la réutilisation de solveur SAT et la transformation du FM en une formule de logique propositionnelle. L'Exigence 5.3 de semi-automatisation est donc satisfaite par l'automatisation de la production des FMs, de leur fusion en un FM global et ses capacités d'interrogation par les modèles du domaine, seule la description des produits en fonction des fonctionnalités caractéristiques étant manuelle.

CHAPITRE 6

APPLICATION À LA VISUALISATION DE DONNÉES

Table des matières

6.1 Visualisation de données en provenance de capteurs **75**
6.1.1 Présentation des domaines 75
6.1.2 Pertinence du cas d'application 78
6.1.3 Implémentation du prototype 79
6.2 Apport des contributions sur le cas d'application **80**
6.2.1 Intégration des méta-modèles 80
6.2.2 Cohérence de l'architecture 83
6.2.3 Variabilité et concrétisation du système 85

Les chapitres précédents présentent les trois principales contributions de cette thèse, respectivement l'intégration de méta-modèles dans le Chapitre 3, la détection d'incohérences inter-domaines dans le Chapitre 4 et la gestion de la variabilité technologique dans le Chapitre 5. Celles-ci sont illustrées par un exemple fil rouge volontairement simplifié pour mettre l'accent sur la contribution sans subir la complexité d'un cas réel. L'objectif de ce chapitre est de présenter un cas d'étude plus complet, la visualisation de données, qui a servi comme base de validation de nos différentes contributions. La Figure 6.1 visualise nos contributions dans ce cadre applicatif.

6.1 Visualisation de données en provenance de capteurs

6.1.1 Présentation des domaines

Un *dashboard* est défini comme un ensemble de visualisations cohérentes présentant l'information nécessaire à un utilisateur pour l'aider dans sa prise de décisions. Chaque *dashboard* doit donc être conçu en utilisant des visualisations adaptées (1) poursuivant un objectif précis (2) en fonction des données en entrée (3). La conception de *dashboard* nécessite la collaboration d'experts de domaines différents, chacun apportant des connaissances relatives à son domaine d'expertise par exemple, et de manière non limitative :

1. la visualisations de données, ou *Visualisation Design* (VD),
2. l'ingénierie des besoins, ou *Requirement Engineering* (RE),
3. la gestion des données de capteurs, ou *Data Management* (DM).

Chacun de ces domaines possède une complexité intrinsèque qui a fait l'objet de diverses contributions. La conception d'un *dashboard* est donc un écosystème diversifié

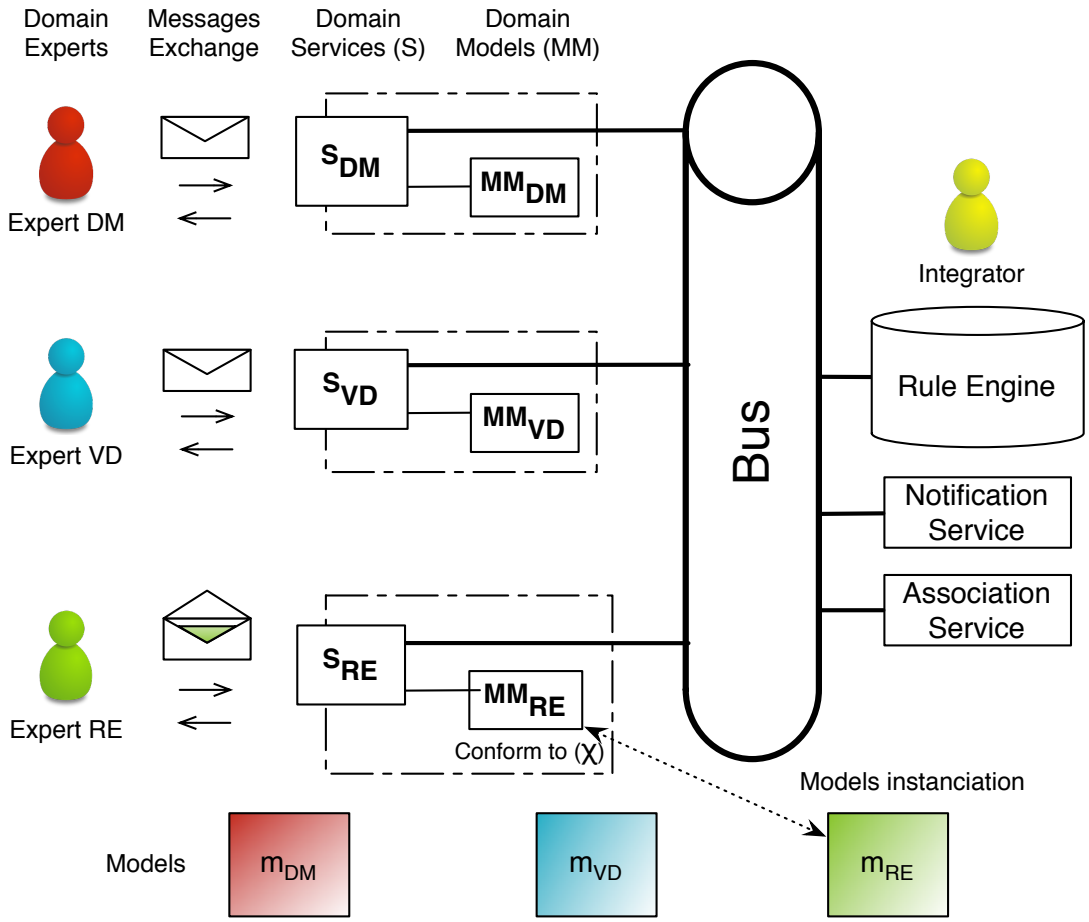


FIGURE 6.1 – Diagramme d'architecture du cas d'application.

d'outils et de langages. Cet aspect de diversité des solutions motive notre hypothèse de réutilisation des langages sans effet de bord structurel, afin de préserver l'interopérabilité avec les outils existant dans cet écosystème. Par exemple le langage RE Gherkin [1] a vocation à être intégré dans Cucumber [Wynne 12], un écosystème de tests d'acceptation. Le rôle et la diversité de chaque domaine sont développés ci-dessous.

Requirement Engineering (RE). L'adéquation entre les objectifs visés par l'utilisateur et les capacités du *dashboard* final repose sur une analyse des besoins. Il existe des outils divers dans le domaine de l'ingénierie des besoins offrant des solutions de modélisation. Certaines méthodologies proposent de modéliser la totalité du système sous la forme d'arbre de tâches [Paternò 97], principalement dans les démarches centrées utilisateur [Kolb 12, Hili 15]. D'autres solutions permettent de décrire les spécifications en utilisant des scénarios utilisateurs comme avec Gherkin [Wynne 12], appelées aussi *user stories* [Wautelet 14], ou via des modèles à états modélisant le comportement comme IBM Rational's DOORS [2]. Ces outils et méthodologies ne sont pas intégrées et ne peuvent pas coopérer. A titre d'exemple, nous proposons d'utiliser la syntaxe abstraite de Gherkin au sein d'un méta-modèle, *cf.* FIGURE 6.2, qui permet la description d'un ensemble de scé-

1. https ://github.com/cucumber/gherkin
2. http ://www-03.ibm.com/software/products/en/ratidoor

narios de visualisation indépendants autour de trois composantes : une situation *Given*, un déclencheur *When* et une réaction *Then*.

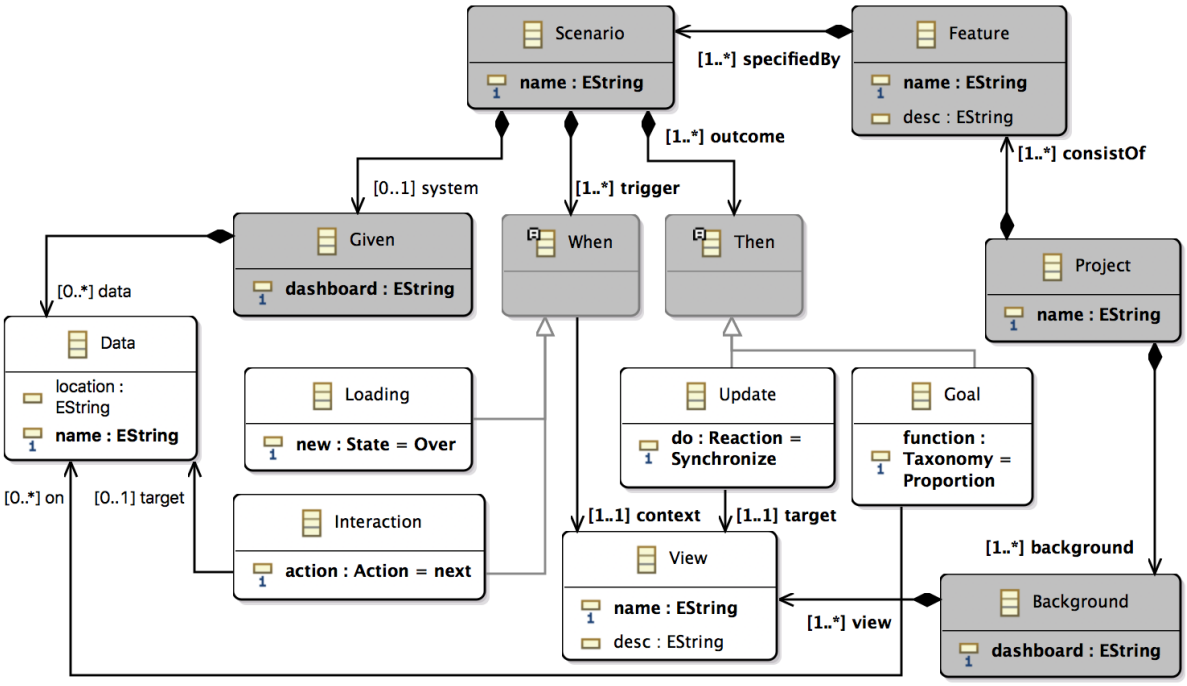


FIGURE 6.2 – Extrait de syntaxe abstraite du domaine RE, adaptée de Gherkin.

Data Management (DM). Les données en provenance de capteurs présentent des formats hétérogènes à gérer et peuvent nécessiter un pré-traitement avant visualisation. Il peut s'agir par exemple d'opérations d'échantillonnage, de sélections de plages temporelles, de conversions d'unités ou de synchronisations entre capteurs. La définition et la manipulation de ces catalogues de données sont réalisées par un expert en réseaux de capteurs via un langage dédié nécessitant une connaissance du parc de capteurs, des formats de données utilisés et des opérations possibles. Il existe de nombreux DSLs couvrant tout ou partie de ces aspects [Liu 06, Shah 03, Botts 07]. Dans notre étude de cas, nous avons fait le choix de réutiliser *Sensor Deployement Language*[3], un langage de gestion d'infrastructure et de réseau de capteurs afin de réutiliser un DSL dédié aux capteurs possédant son propre environnement d'outils externes, ce qui motive notre respect de l'interopérabilité du langage. Celui-ci a été développé dans le cadre du projet de recherche SmartCampus [Cecchinel 14] pour la gestion de politiques de collecte de données [Cecchinel 16], *cf.* FIGURE 6.3.

Visualisation Design (VD). Un *dashboard* vise à synthétiser sous différents diagrammes des données de natures différentes afin de pouvoir observer et éventuellement décider. La pertinence du *dashboard* dépend du choix d'une visualisation plutôt qu'une autre en fonction des capacités d'analyse qu'elle offre, ainsi que l'agencement cohérent des

3. https ://github.com/ILogre/SensorDeployementLanguage

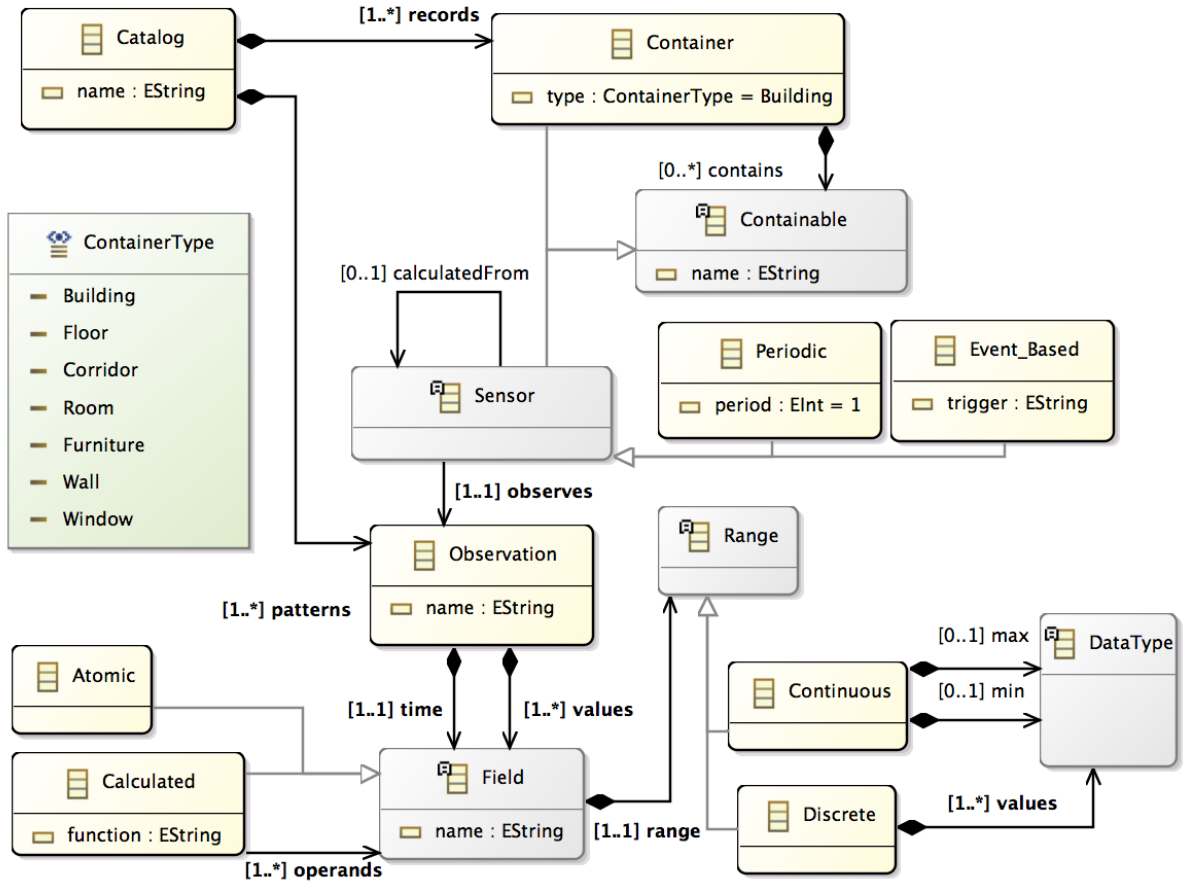


FIGURE 6.3 – Syntaxe abstraite du domaine DM, réutilisée du framework DEPOSIT.

différentes visualisations. Les experts utilisent des bibliothèques graphiques pour concevoir des interfaces via des langages tels qu'IFML [Rossi 13] proposé par l'OMG, des approches par conception d'interfaces abstraites [Calvary 03, García Frey 12, Demeure 08] ou par mash-up [Wilson 12]. Nous mettons à disposition de l'expert un méta-modèle, *cf.* FIGURE 6.4, basé sur une taxonomie des capacités offertes par les visualisations, afin de concevoir un *dashboard* abstrait qui sera par la suite concrétisé par des widgets concrets issus des bibliothèques graphiques. Cette taxonomie a été définie par la communauté de l'infographie et du journalisme par les données [4] précisément pour caractériser la masse de *widgets* existants.

6.1.2 Pertinence du cas d'application

Ce cas d'utilisation met en évidence la nécessité pour plusieurs domaines de collaborer pour concevoir un système unique. Il met aussi en évidence les multiples approches possibles pour chacun de ces sous-domaines qui possèdent, bien souvent leur propre écosystème logiciel. Ce cas d'étude bénéficie donc d'une approche d'intégration permettant la réutilisation des méta-modèles, qui permet d'exposer explicitement les capacités des domaines à un degré d'abstraction compréhensible par tous les experts.

4. http ://datavizcatalogue.com/search.html

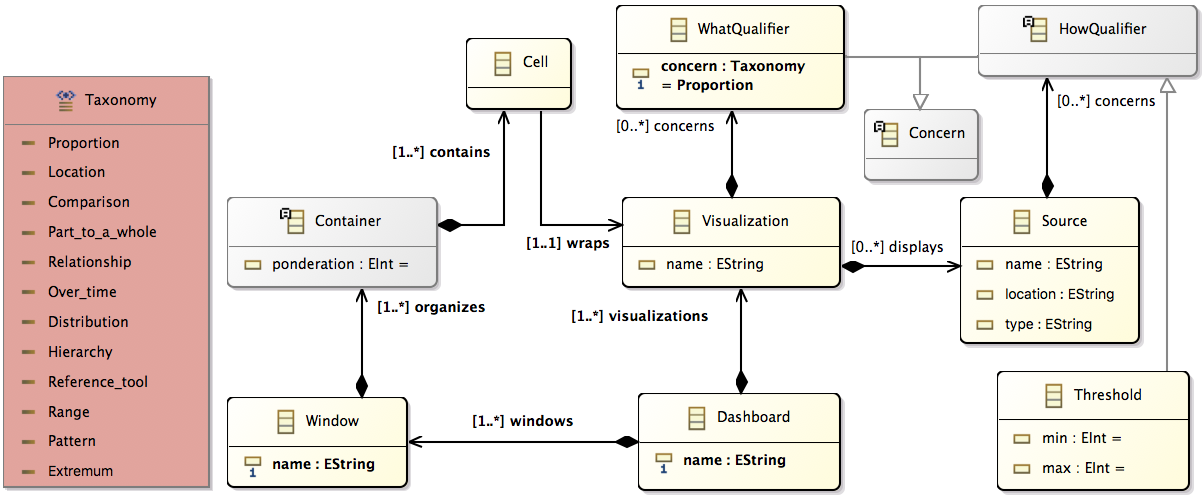


FIGURE 6.4 – Extrait de syntaxe abstraite du domaine VD, basée sur la taxonomie du *data journalism*.

L'interaction entre ces domaines peut être complexe et doit être ajustée en fonction des caractéristiques de chacun des services les encapsulant. Classiquement dans les approches centrées utilisateur, c'est le domaine *RE* qui prend la main, en capturant et modélisant le système final en fonction de l'utilisateur. L'ensemble de capteurs modélisé dans *DM*, possède des limitations physiques quant à leur fréquence et des problématiques de communication réseau intrinsèques, ce domaine ne peut donc pas toujours s'adapter. Externaliser la gestion des interactions permet de définir formellement les échanges entre les domaines, sans impacter la modélisation de ceux-ci. Il est alors possible de concevoir une politique d'interaction entre les domaines selon le système en cours d'étude.

Les domaines *DM* et *VD* embarquent une multitude d'artefacts concrets, indépendamment du méta-modèle utilisé, *i.e.,* respectivement des plateformes de capteurs et des *widgets* de visualisation. Même en se limitant aux solutions apportées par les langages Web pour le domaine *DM*, des dizaines de bibliothèques proposent des *widgets* de visualisations différents. Cet espace des solutions est en croissance continuelle, *e.g.,* la bibliothèque D3.JS [5] proposait 133 implémentations différentes de visualisation en avril 2014, 235 en janvier 2015 et 368 en juin 2017. La concrétisation de ces domaines vers un *dashboard* exécutable profite d'une gestion de cette variabilité technologique caractérisant les capacités de chacune de ces solutions.

Il nous semble évident que la conception d'un *dashboard*, qui est une tâche aujourd'hui courante mais aussi cruciale, au point que de nombreuses sociétés de service proposent des toolkits pour leur construction, est un domaine d'application de nos travaux pertinent pour illustrer la démarche et les potentialités.

6.1.3 Implémentation du prototype

Le prototype réalisé intègre complètement l'approche d'intégration de méta-modèle détaillée dans ce manuscrit, dans le cadre de la construction de *dashboards* pour la visualisation de données issues de capteurs. Nous avons privilégié essentiellement 3 domaines :

5. https ://github.com/d3/d3/wiki/Gallery

RE, *DM* et *VD*. Les syntaxes abstraites et concrètes ont été développées en utilisant EMF [6] et Xtext [7] et sont disponibles [8].

Les représentations EMF de ces domaines sont illustrées par les figures précédentes et détaillées en annexe Annexe A.6, Annexe A.7 et Annexe A.8. Les syntaxes concrètes XText associées à ces méta-modèles sont illustrées en annexes par les figures : Annexe A.3, Annexe A.4 et Annexe A.5 donnant un exemple d'utilisation de ces DSLs. Pour rappel, ces méta-modèles sont supposés existants dans notre approche, *cf.* Chapitre 1.

Selon la démarche que nous proposons, chaque domaine a été encapsulé dans un service disponible sous la forme d'un projet Github distinct. Chacun de ces trois projets, *i.e.,* chaque domaine, contient par conséquent *(i)* l'archive JAR de sa représentation EMF, *(ii)* ses opérations haut-niveau que l'intégrateur à choisi d'exposer et *(iii)* la définition des messages attendus ou émis par chaque opération.

Nous avons fait le choix d'intégrer ces domaines dans un projet Maven séparé possédant des dépendances vers les projets des services de domaine. Selon la méthodologie que nous proposons, le projet Maven d'intégration accède aux différents domaines par les biais des services et non pas directement par l'intermédiaire de l'implémentation EMF. Ce projet est illustré par la Figure 6.1, *i.e.,* la couche de médiation sous la forme d'un bus de messages et un moteur de règles pour lequel nous avons choisi Drools [9]. A partir de ce projet, l'intégrateur doit définir les règles métiers gérant les interactions entre domaines.

La mise en œuvre de ce scénario démonstratif repose sur 31 règles, *i.e.,* 9 règles d'intégration et 22 règles de routage. Nous avons mis en œuvre huit scénarios de complexité croissante qui illustrent (*i*) les possibilités de modélisation d'un système conforme aux méta-modèles EMF et (*ii*) les bénéfices de l'intégration en terme de cohérence inter-domaines.

Cette encapsulation des domaines en service, les mécanismes d'intégration et les scénarios sont développés en Java (~6300 LoC) [10].

6.2 Apport des contributions sur le cas d'application

Les chapitres de contributions précédents (*i.e.,* Chapitre 3, 4 et 5) illustrent leurs apports sur un exemple fil rouge volontairement simplifié de gestion de projet. Cette section vise à démontrer l'applicabilité des contributions de cette thèse sur un cas présentant une réelle complexité du fait des interactions entre les domaines détaillés dans la section Section 6.1. Pour cela les propriétés et les exigences de chaque contribution dans le cadre de la conception de *dashboard* sont détaillées dans les sections suivantes.

6.2.1 Intégration des méta-modèles

Propriétés sur les services : L'objectif de cette partie est de montrer comment l'encapsulation des domaines dans des services répond aux exigences exprimées dans le Chapitre 3, à savoir :

6. https ://eclipse.org/modeling/emf/
7. http ://www.eclipse.org/Xtext/index.htm
8. https ://github.com/ILogre/EMF-VD-DM-RE
9. http ://www.drools.org/
10. https ://github.com/ILogre/Service-Integration-Rules-on-Drools

(*i*) préserver l'intégrité de la syntaxe abstraite du domaine (Exigence 3.1),

(*ii*) permettre d'exposer à l'expert du domaine une interface ayant l'expressivité nécessaire (Exigence 3.2),

(*iii*) pouvoir vérifier la conformité des modèles produits par le service vis-à-vis de la syntaxe abstraite du domaine (Exigence 3.3).

Notre approche d'encapsulation propose à l'expert domaine de réutiliser entièrement la définition du domaine via sa syntaxe abstraite. L'hypothèse de départ est de disposer en entrée des méta-modèles des domaines. Dans ce cas d'utilisation, nous réutilisons trois méta-modèles externes et hétérogènes. RE provient de Gherkhin, DM de SmartCampus et VD d'une taxonomie de classification de visualisation. En appliquant la démarche d'encapsulation en service détaillée dans le Chapitre 3, nous vérifions les exigences rappelées ci-dessus.

L'expert du domaine doit définir l'interface du service qui encapsule le méta-modèle. Pour chaque tâche qu'il réalise sur son sous-système il crée une opération, *e.g.,* définir un nouveau schéma de donnée que peut retourner un capteur. La signature de l'opération est l'attente d'un message en paramètre d'entrée et optionnellement, l'émission d'un autre message en retour. L'expert implémente ensuite cette opération en utilisant des actions atomiques sur le méta-modèle, *i.e.,* créer, consulter, modifier ou supprimer des instances.

Par construction, le service est incapable de modifier la structure du méta-modèle (*cf. i*). Cela permet, par exemple, de réutiliser sa syntaxe concrète pour permettre à l'utilisateur de visualiser le modèle qu'il a conçu via une syntaxe adaptée. Le service exploite les capacités intrinsèques du méta-modèle, par exemple la vérification de la conformité d'une instance vis-à-vis de son méta-modèle (*cf. iii*), qui est proposée par EMF dans notre cas.

L'interface de service exposant les opérations accessibles à l'expert découle de l'usage que cet expert fait du domaine, et plus précisément de sa syntaxe concrète. Pour développer les opérations du service, l'expert du domaine peut accéder aux actions élémentaires sur les concepts du méta-modèle. Notre approche n'impose aucune limite sur le nombre et la granularité des opérations exposées, ce qui permet d'atteindre l'expressivité voulue en fonction du domaine et des usages de cette communauté (*cf. ii*).

Dans le cas du service DM, l'analyse de l'usage du domaine par l'expert a mené à la conception de l'interface explicitée dans l'Exemple 6.1 et dont la structure des messages est détaillée dans la Figure 6.5. Les opérations d'édition permettent à l'expert de construire un modèle valide d'un catalogue de capteurs répartis dans une zone géographique, *e.g.,* les capteurs déployés dans le campus et les formats des données qu'ils collectent. Les opérations de consultation permettent de requêter l'état courant du catalogue en cherchant des informations précises, *e.g.,* la définition formelle d'un format de données déclaré comme `Observation`.

Exemple 6.1: **Interface du service DM**

```
@Service
public class DataManagement extends Service {
    */ Operations pour edition du modele /*
  public static void declareCatalog (DeclareCatalogMsg msg){...}
  public static void buildSensorHostingHierarchy (
      BuildSensorHostingHierarchyMsg msg){...}
  public static void recordPeriodicSensor ( RecordPeriodicSensorMsg msg )
      {...}
```

```
    public static void recordEventBasedSensor ( RecordEventBasedSensorMsg
        msg ){...}
8   public static void sketchPattern ( SketchPatternMsg msg){...}
      */ Operations de consultation du modele /*
10  public static IsDefinedAsw isDefined ( IsDefinedMsg msg ){...}
    public static DescribeSensorAsw describeSensor(DescribeSensorMsg msg)
        {...}
12  public static SearchAllSensorsAsw searchAllSensors(SearchAllSensorsMsg
        msg){...}
    public static DescribeObservationPatternAsw describeObservationPattern(
        DescribeObservationPatternMsg msg){...}
14  public static SearchAllObservationPatternsAsw
        searchAllObservationPatterns(SearchAllObservationPatternsMsg msg)
        {...}
    public static SearchSensorsByObservationPatternAsw
        searchSensorsByObservationPattern (
        SearchSensorsByObservationPatternMsg msg){...}
```

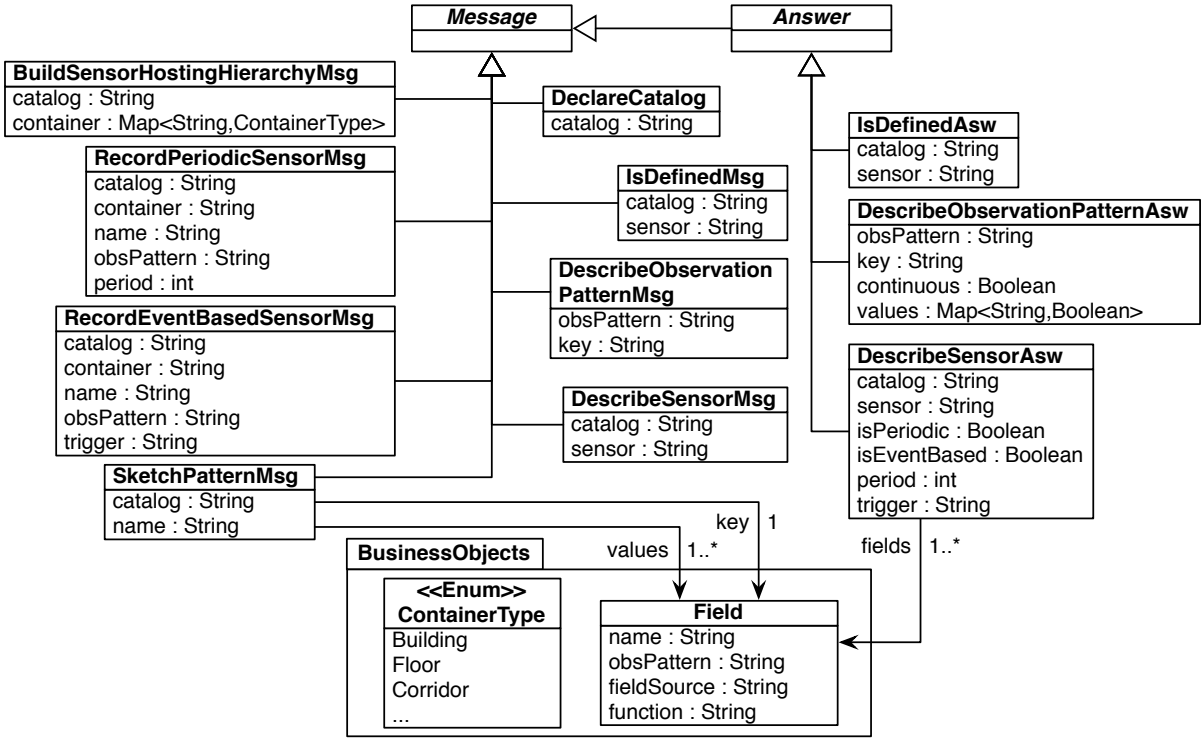


FIGURE 6.5 – Diagramme de classes des Messages échangés avec le service `DataManagement`.

Propriétés sur les opérations : L'objectif de cette partie est de montrer les conséquences du respect des bonnes propriétés attendues par SOA sur l'implémentation des opérations, à savoir :

(*i*) être sans état (PROPRIÉTÉ 3.1),

(*ii*) être idempotent (PROPRIÉTÉ 3.2),

(*iii*) être découplés les uns des autres (PROPRIÉTÉ **??**).

L'absence d'état implique que le service ne maintient pas de référence vers les objets instanciés entre deux invocations (*cf. i*). Cela oblige, lors du développement de l'opération, à retrouver l'instance ciblée par l'invocation, par exemple une instance précise de *dashboard*. Pour cela le message d'invocation doit contenir l'information nécessaire, *e.g.,* l'identifiant du *dashboard* contenant la visualisation que l'on souhaite caractériser. L'exemple sur le cas de l'opération `characterizeVisu()` du service VD est exposé dans l'Annexe A.1. La récupération de la racine du modèle cible par le service VD à partir des informations contenues dans le message d'invocation, permet aux opérations de ne pas maintenir d'état (*cf.* Annexe A.2).

L'idempotence de l'opération est à assurer par l'expert du domaine qui la développe (*cf. ii*). Une solution classique en SOA est de vérifier en accédant au modèle que chaque action élémentaire qui constitue l'opération n'a pas déjà été effectuée avant de l'exécuter [Daigneau 11].

Enfin, les opérations de services ne contiennent aucun appel à une autre opération, du même service ou d'un service différent (*cf. iii*). La responsabilité de déclenchement d'autres opérations pour le maintient de la cohérence est déléguée au moteur d'intégration (*cf.* Section 6.2.2). Chaque service de domaine a été développé comme un projet indépendant, n'ayant aucune dépendance avec les autres services de domaine, comme illustré par la Figure 6.6. Seul le projet d'intégration connait et interagit avec l'ensemble des domaines.

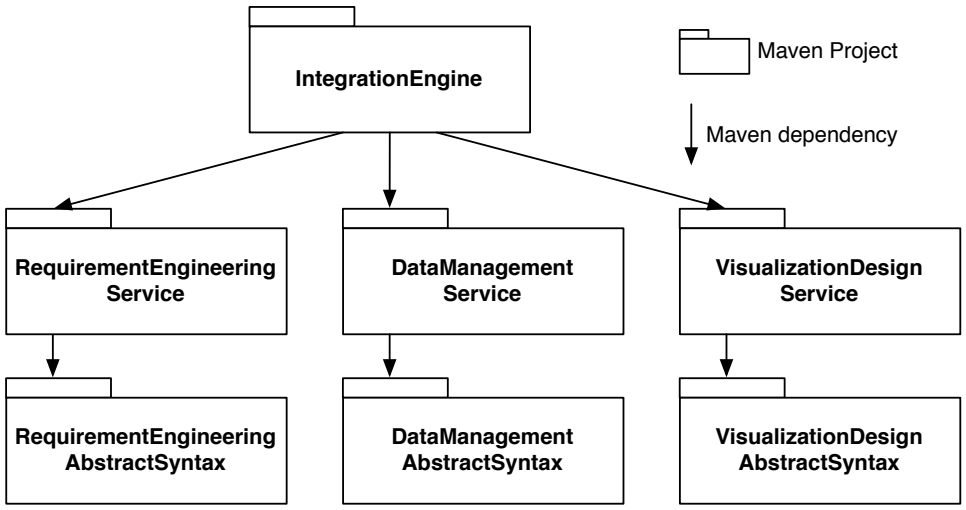


Figure 6.6 – Diagramme de dépendance des projets Maven.

6.2.2 Cohérence de l'architecture

Exigences sur les invocations : La gestion de la communication entre les services permet de détecter les incohérences entre les domaines RE, DM et VD. Pour rappel, nous définissons la cohérence comme l'état de compatibilité entre modèles issus de métamodèles hétérogènes (*cf.* Section 4.1). Nous montrons ici comment la communication entre les services respecte un ensemble d'exigences sur l'utilisation d'un domaine, à savoir :

(*i*) lors d'une invocation explicite de l'expert par message, le système doit assurer l'exécution de l'opération ciblée (Exigence 4.1),

(*ii*) la réalisation d'une action ne doit pas être bloquante pour l'expert ni pour les autres domaines (EXIGENCE 4.2),

(*iii*) la détection d'une incohérence du système doit faire remonter à l'expert l'action qui en est la cause (EXIGENCE 4.3).

Dans notre cas d'application, dès la première itération sur le jeu de règles, une règle de routage a été créée pour chaque opération (*e.g.,* l'EXEMPLE 6.2) assurant l'accessiblité de toutes les opérations (*cf. i*). Depuis, l'ajout de règles d'intégration a rendu certaines règles de routages optionnelles. Par exemple, dans le cas où les règles d'intégration qui invoquent effectivement l'opération dans leur partie droite couvrent la totalité des contextes d'exécutions possibles, *i.e.,* l'union de leurs parties gauches est équivalent à Vrai. Toutefois, puisque les règles de routage n'entrent pas en conflit avec les règles d'intégration, il est conseillé de les conserver au cas où le jeu de règle viendrait à évoluer, pour toujours permettre à l'utilisateur d'atteindre ses opérations.

Exemple 6.2: **Exemple de règle de routage Drools pour l'opération `characterizeVisu()`**

```
rule "Characterize Visualization"
no-loop true
  when
    $msg : CharacterizeVisuMsg (whatQualifiers not contains "Over_time")
  then
    VisualizationDesignAP.characterizeVisu($msg);
end
```

Lorsqu'une incohérence est détectée dans le système, l'action en cause est menée à terme puis le domaine ayant la possibilité de résoudre l'incohérence est notifié avec les informations disponible. Par exemple, la règle d'intégration de l'EXEMPLE 6.3 détecte le cas où l'expert de VD ajoute une ressource à une visualisation qui ne peut être alimentée par aucun capteur modélisé par l'expert de DM.

Dans ce cas, la ressource est effectivement ajoutée à la visualisation conformément au message envoyé par l'expert VD (*cf. i*). Puis le domaine DM est notifié qu'un capteur satisfaisant le besoin de cette visualisation est requis, avec les informations décrivant la ressource (*cf. iii*). En attendant la résolution du conflit par l'expert DM, l'expert VD peut continuer à concevoir sa partie du système en appelant d'autres opérations (*cf. ii*). A noter, ici l'intégrateur a choisit qu'il était du devoir du domaine de DM de fournir les données nécessaires, mais une autre politique peut être conçue où VD doit se satisfaire des capteurs déjà déployés, la remontée d'incohérence est alors faite à VD en renvoyant la liste des capteurs disponibles.

Exemple 6.3: **Règle d'intégration `No Resource matching Data` sur l'opération `PlugData()`**

```
rule "Plug Data - No Resource matching Data"
no-loop true
  when
    $msg : PlugDataMsg ( $dashboard : dashboardName, $dataName : dataName)
    and HasLinkedAsw ( answer == true )
```

```
      from AssociationDP.hasLinked(new HasLinkedMsg($dashboard, "Dashboard
          ", "Catalog"))
    and GetLinkedAsw ($catalog : model)
      from AssociationDP.getLinked(new GetLinkedMsg($dashboard, "Dashboard
          ", "Catalog"))
    and IsDefinedAsw (defined == false)
      from SensorDeploymentDP.isDefined(new IsDefinedMsg($catalog,
          $dataName))
  then
    VisualizationDesignAP.plugData($msg);
    NotificationDP.raiseConsistencyIssue(
      new RaiseConsistencyIssueMsg($dataName,
      "Source",
      ConsistencyIssue.No_sensor_providing_data_linked)
    );
  end
```

Propriétés sur le moteur de règles : L'objectif de cette section concerne l'évolutivité du jeu de règles, en montrant comment la sémantique d'exécution respecte les propriétés suivante :

(*i*) les règles sont indépendante et ne font pas d'hypothèse sur l'existence d'autres règles (Propriété 4.1),

(*ii*) l'ordre d'exécution des règles n'est pas garanti (Propriété 4.2),

(*iii*) la propagation de déclenchement de règle ne crée pas de cycle (Propriété 4.3).

Comme détaillé dans le Chapitre 4, la grammaire de nos règles ne permet pas le déclenchement explicite d'autres règles (*i*). Une règle peut toutefois instancier un message différent du message initial. Ce nouveau message pourra déclencher à son tour des règles. Dans notre cas d'application nous avons choisi l'outil Drools [11] [Browne 09] comme moteur de règles. Drools intègre nativement une gestion des cycles sur les règles disposant de la règle `no-loop true`, permettant de détecter durant l'exécution les cycles infinis, qui seront alors interrompus après le premier déclenchement de chaque règle impliquée (*iii*).

Dans le cas où un message déclencherait plusieurs règles, nous avons choisi de recommander à l'intégrateur de ne pas spécifier une règle prioritaire, ce qui représente le cas nominal proposé par Drools (*ii*). Par exemple, en cas de réception d'un message `PlugDataMsg`, la règle de routage `PlugData` et la règle d'intégration `Ressource Matching Data` (*cf.* Annexe A.4) seront toutes les deux déclenchées sans souci dû à l'ordre grâce à la propriété d'idempotence des opérations. Si l'intégrateur choisit d'établir une priorité sur une règle d'intégration, il conçoit celle-ci avec l'hypothèse qu'elle sera toujours exécutée avant toutes les autres. Lors de l'évolution du jeu de règles, cette hypothèse s'invalide si plusieurs règles désignées prioritaires partagent un contexte d'exécution (*e.g.,* que l'intersection de leurs parties gauche soit vraie). Les règles ne sont alors plus indépendantes et la cohérence du jeu de règles est compromise.

6.2.3 Variabilité et concrétisation du système

Exigences sur la concrétisation : Afin d'être utilisable dans un cas réel, le système de concrétisation doit pouvoir satisfaire les exigences exprimées dans le Chapitre 5, à savoir :

11. http ://www.drools.org/

(*i*) les artefacts concrets considérés sont issus de sources hétérogènes (EXIGENCE 5.1),

(*ii*) la concrétisation doit bénéficier de la multitude d'artefacts à disposition sans souffrir de l'accroissement de ce nombre (EXIGENCE 5.2),

(*iii*) la concrétisation doit être semi-automatisée (EXIGENCE 5.3).

Dans le cas d'application des *dashboards*, les *widgets* de visualisations présentent une forte variabilité, ce qui fait de la visualisation un domaine d'étude pertinent en matière de concrétisation. Nous caractérisons donc chacun des *widgets*, provenant de cinq bibliothèques de visualisations différentes (*i*), selon un vocabulaire de caractéristiques commun (*i*). Un extrait de la caractérisation de la bibliothèque AmCharts est illustré par la FIGURE 6.7 et détaillé en ANNEXE A.1. A noter, certaines bibliothèques proposent plusieurs version d'une même visualisation (*e.g.,* Step Chart), ce qui donne lieu à plusieurs caractérisation et donc à autant d'artefacts concrets, *i.e.,* de *widgets*. Techniquement, les matrices CSV des 73 *widgets* considérés dans notre cas d'application sont traduites en *feature model* atomiques. Ces derniers sont fusionnés en un seul modèle capturant toute la variabilité du domaine, illustré par la FIGURE 6.8 et détaillé en ANNEXE A.2. Chaque concrétisation d'une visualisation abstraite modélisée dans VD revient à une configuration de ce modèle fusionné. Ainsi, le coût de fusion n'est appliqué qu'une seule fois au moment de la caractérisation, tandis que la configuration s'appuie sur des solveurs SAT qui ont été prouvés efficients pour cette tâche (*ii*). Cette concrétisation est automatisée, chaque invocation de l'opération `characterizeVisu()` amenant à la sélection et/ou la dé-sélection d'un ensemble de caractéristiques (*iii*). En plus des caractéristiques issues de la taxonomie présentée dans la SECTION 6.1, nous ajoutons le nom des visualisations caractérisées comme une *feature* du modèle de variabilité pour permettre d'affiner la sélection si plusieurs artefacts concrets sont encore disponibles à la fin de la modélisation, facilitant ainsi la concrétisation des visualisations abstraites.

	A	B	C	D	E	F	G	H	I	J
1		Pie Chart	Funnel Chart	Line Chart	Line Chart	Area Chart	Stacked Area Chart	100% Stacked Area Chart	Step Chart	Step Chart
2	Comparison	Oui	Oui	Non	Oui	Non	Non	Non	Non	Oui
3	Proportion	Oui	Oui	Non	Non	Non	Oui	Oui	Non	Non
4	Relationship	Non	Non	Non	Oui	Non	Oui	Oui	Non	Oui
5	Hierarchy	Non	Non	Non	Non	Non	Non	Non	Non	Non
6	Location	Non	Non	Non	Non	Non	Non	Non	Non	Non
7	Probability	Non	Non	Non	Non	Non	Non	Non	Non	Non
8	PartToAWhole	Oui	Oui	Non	Non	Non	Non	Oui	Non	Non
9	Distribution	Non	Non	Non	Non	Non	Non	Non	Non	Non
10	Patterns	Non	Non	Oui	Oui	Oui	Oui	Oui	Oui	Oui
11	Range	Non	Non	Non	Non	Non	Non	Non	Non	Non
12	OverTime	Non	Non	Oui	Oui	Oui	Oui	Oui	Oui	Oui
13	Scalar	Oui	Oui	Oui	Oui	Oui	Oui	Non	Oui	Oui
14	Discrete	Oui	Oui	Non	Non	Non	Non	Non	Oui	Oui
15	Variations	Non	Non	Oui	Oui	Oui	Oui	Oui	Oui	Oui
16	Extremum	Non	Non	Oui	Oui	Oui	Non	Non	Non	Non

FIGURE 6.7 – Extrait de la matrice de caractérisation d'AmCharts
Caractéristiques de la taxonomie satisfaites par artefact (*widget*).

Propriétés sur la gestion de la variabilité : L'utilisation du *feature model* (FM) décrit ci-dessus dans le cadre de la concrétisation respecte les propriétés suivantes :

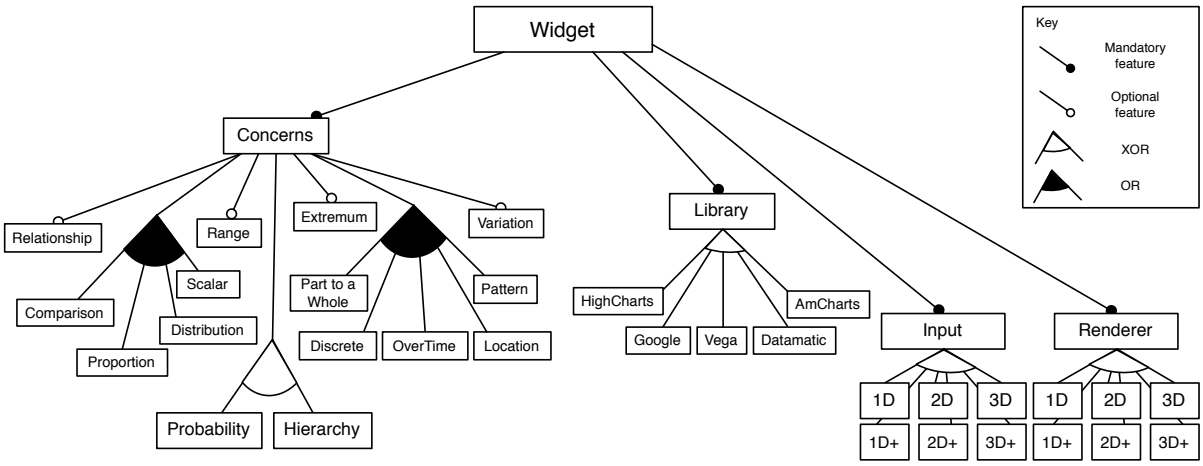


FIGURE 6.8 – *Feature Model* fusionné des 73 *widgets* de visualisation.

(*i*) le modèle de variabilité assure l'accessibilité de chaque artefact concret considéré en entré (PROPRIÉTÉ 5.1),

(*ii*) le modèle de variabilité ne permet pas d'atteindre d'autres cibles de concrétisation que les artefacts concrets considérés en entrée (PROPRIÉTÉ 5.2).

Ces deux propriétés sont assurées par l'utilisation du langage Familiar comme support de déclaration des FMs atomiques produit à partir des matrices de caractérisation, et par l'application de l'opérateur `merge with strict union` défini dans le langage. Le modèle de variabilité attaqué lors de la concrétisation étant constitué de l'union des artefacts concrets, il permet d'atteindre une configuration finale et valide pour chacun et aucune autre.

Le FM obtenu n'est pas trivial comme le montre l'EXEMPLE 6.4, il repose sur 192 contraintes logiques transverses qui sont générées par l'opération de `merge with strict union`. Elles représentent les implications logiques qui ne peuvent pas être représentées graphiquement dans le FM. Par exemple : `(Variations -> OverTime);`. De plus, même la représentation graphique diffère d'un résultat intuitif sur la taxonomie (*concerns* dans le FM). Par exemple, il n'est pas naturel d'opposer la visualisation de la probabilité à la recherche d'une hiérarchie dans les données, or ces deux caractéristiques sont filles d'un XOR dans le FM qui représente les compétences de l'espace des solutions.

Exemple 6.4: **Notation Familiar du FM fusionné et extrait des contraints transverses**

```
fm_merged = FM ( widget: Input Library Concern Name Output ;
Input: ("2D+_i"|"3D_i")? ("2D_i"|"3D+_i")? ;
Library: (highcharts|amcharts|google|datamatic|vega);
Concern: (PartToAWhole|Discrete|OverTime|Patterns|Location)+ (Proportion|
    Comparison|Distribution|Scalar)+ (Probability|Hierarchy)? [Range] [
    Variations] [Extremum] [Relationship] ;
Name: ("MultiSet␣Radar␣Chart"|"Stacked␣Area␣Chart"|"MultiSet␣Stacked␣Area␣
    Chart"|"MultiSet␣Step␣Chart"|"Angular␣Gauge"|Choropleth|"Polar␣Chart"|
    Histogram|"Bubble␣Chart"|Candlestick|"Barley␣Chart"|"MultiSet␣Area␣
    Chart"|"Area␣Chart"|"MultiSet␣Line␣Chart"|ScatterPlot|"Radar␣Chart"|"
    MultiSet␣Bubble␣Chart"|"MultiSet␣Bar␣Chart"|"Arc␣Chart"|Gauge|"Step␣
    Chart"|"Line␣Chart"|Graph|"100\%␣Stacked␣Area␣Chart"|"Bar␣Chart"|Error
```

```
    |"Scatter␣Chart"|TreeMap|"Parallel␣Coords"|"Pie␣Chart"|Timeline|"
    Stacked␣Bar␣Chart"|"MultiSet␣ScatterPlot"|"Funnel␣Chart") ;
Output: ("1D+_o"|"3D_o"|"2D+_o"|"3D+_o"|"2D_o"|"1D_o")+ ;
(Variations -> OverTime);
(Relationship -> !Location);
(OverTime -> !Hierarchy);
(Hierarchy -> !Scalar);
...
```

CHAPITRE 7

VALIDATION

Table des matières

7.1 Conditions d'expérimentation **89**
7.1.1 Objectifs 90
7.1.2 Catégorisation des données 90
7.2 Exp #1 : Comparaison des solutions **91**
7.2.1 Protocole expérimental 91
7.2.2 Observations 92
7.2.3 Discussion et perspectives 93
7.3 Surcoût de l'encaspulation en service **94**
7.3.1 Hypothèses et données 94
7.3.2 Observations 95
7.3.3 Conclusions 96
7.4 Gestion des interactions par des Règles Métier **96**
7.4.1 Hypothèses et données 97
7.4.2 Observations 98
7.4.3 Conclusions 99
7.5 Automatisation de l'approche **100**
7.5.1 Hypothèses 100
7.5.2 Observations 101
7.5.3 Conclusions 101
7.6 Exp #2 : Caractérisation et concrétisation **101**
7.6.1 Présentation de l'expérience 101
7.6.2 Observations 104
7.6.3 Conclusions et discussions 108
7.7 Conclusions **109**

7.1 Conditions d'expérimentation

Le cas d'application de la visualisation de données en provenance de capteurs introduit dans le Chapitre 6 offre la complexité nécessaire à une validation de nos contributions. Il présente les caractéristiques d'un système à la frontière de plusieurs domaines, *e.g., Requirement Engineering* (RE), *Data Management* (DM) et *Visualization Design* (VD), et offre une large variabilité technologique (*e.g., widgets* et capteurs). Dans ce chapitre, nous utilisons la modélisation de tableaux de bord ou *dashboards* pour valider chacune des contributions détaillées au préalable. Nous finissons ce chapitre par une expérience

utilisateur sur le cas d'application permettant de mesurer l'intérêt d'une approche par caractérisation. Cette section présente les objectifs, les hypothèses et les données utilisées par les expérimentations des sections suivantes.

7.1.1 Objectifs

Les sections suivantes présentent les expérimentations et les mesures qui permettent d'évaluer les contributions de cette thèse. Nous utilisons les critères d'évaluation identifiés dans l'état de l'art. L'ajout d'un niveau d'asbtraction représente un risque quant à l'applicabilité d'une approche. Pour cette raison, nous cherchons dans ce chapitre à fournir des données empiriques en terme de coût, d'applicabilité et d'utilisabilité de l'approche. Ainsi cette validation cherche à vérifier les exigences suivantes :

(i) vérifier le besoin de préservation de l'intégrité des méta-modèles sur un projet de conception de *dashboards* (*cf.* Section 7.2),

(ii) quantifier le surcoût de l'encapsulation en service des domaines et mesurer les avantages apportés par l'intégration (*cf.* Section 7.3),

(iii) quantifier l'impact de l'externalisation des interactions entre domaines et mesurer le gain sur le couplage par rapport aux méthodes par fusion (*cf.* Section 7.4),

(iv) quantifier l'effort que les experts et l'intégrateur doivent fournir et vérifier que celui-ci est concentré sur des tâches orientées métier (*cf.* Section 7.3 et Section 7.5),

(v) mesurer le gain lors de la concrétisation du système et vérifier que celle-ci s'effectue sans contrepartie sur la satisfaction de l'utilisateur (*cf.* Section 7.6).

7.1.2 Catégorisation des données

Pour atteindre ces exigences, nous exposons une suite d'expérimentations et de mesures issues de l'implémentation décrite dans la Section 6.1.3. Nous en caractérisons les données comme suit :

— **Modèle du Domaine :** Nous considérons les chiffres indépendants de la technologie utilisée pour concevoir ces modèles, tels que *(i)* les méta-classes : abstraites, énumérées ou normales, *(ii)* les relations d'héritage et d'association et *(iii)* les métriques de couplage.

— **Règles Métiers :** Les règles d'intégration et les règles de routage sont considérées. Les règles d'intégration embarquent la connaissance de l'intégration des domaines, soit par la vérification de la cohérence inter-domaine soit en adressant directement les *proxies* d'autres domaines. Les règles de routage sont simplement responsables de transmettre les messages au domaine pertinent, sans valeur d'intégration intrinsèque.

— **Artefacts Logiciels :** L'hétérogénéité des artefacts logiciels restant (*e.g.,* code métier, composants de communication, API de manipulation de modèles) force l'utilisation d'un élément de comparaison commun : les lignes de code. Nous différencions les lignes de code fonctionnelles nécessitant une intervention *manuelle* et celles qui peuvent être automatiquement *générées* (*e.g.,* par un IDE) ou qui sont *fournies* génériquement avec l'approche. Les lignes de code non-fonctionnelles (*i.e.,* lignes vides, commentaires, messages de débug) ne sont pas pertinentes et sont ignorées dans les mesures.

— ***Widgets* de visualisation :** Nous avons extrait les exemples de cinq bibliothèques [1] [2] [3] [4] [5] qui donnent libre accès aux codes de visualisations. Chacune des variantes de ces visualisations a été caractérisé selon la taxonomie du *Data Journalisme*, résultant en un espace de solution de 73 widgets. Un *feature model* modélisé en FAMILIAR permet l'identification de produits satisfaisant selon un ensemble de caractéristiques sélectionnées.

Nous n'avons pas considéré le nombre d'actions élémentaires effectuées sur les modèles du fait de l'intégration, *i.e.,* celles des opérations déclenchées en supplément par les règles d'intégration. Celles-ci pourraient être comparé aux actions élémentaires nécessaires dans les approches de fusion, par exemple dans le cas où l'on souhaite retrouver un modèle spécifique à un seul domaine à partir d'un modèle conforme au méta-modèle fusionné. Les perspectives (*cf.* CHAPITRE 8) illustrent la pertinence d'une étude comparative de ce type.

7.2 Exp #1 : Comparaison des solutions

Pour vérifier le besoin d'une approche d'intégration des langages de modélisation, nous avons conçu une expérience contrôlée dans le contexte de la conception de *dashboard*. L'hypothèse testée est qu'en forçant l'usage d'un unique langage de modélisation aux utilisateurs, l'utilisation qui en est faite n'est pas adéquat.

7.2.1 Protocole expérimental

L'expérience a été conçue pour des étudiants de master en génie logiciel ayant suivi les cours de modélisations et d'IHM. Puisque l'objet de l'expérience porte sur la diversité des résultats, nous privilégions un panel homogène d'utilisateurs pour ne pas biaiser l'expérience au détriment du nombre de sujets. L'étude a été conduite sur un groupe de quatre utilisateurs en situation réelle, dans le cadre d'un projet sur la visualisation nécessitant la modélisation de *dashboards*. Les données analysées ont donc été produites pour les besoins concrets du projet et non simulées pour cette expérimentation.

Dans les étapes préliminaires, les étudiants ont développé ensemble un projet de visualisation à l'aide de *dashboards*. Quatre scénarios sont proposés :

— le suivi d'activité d'un athlète,

— la planification d'un projet,

— le déploiement d'une application dans le Cloud,

— la consommation d'énergie du chauffage du campus.

Ces scénarios présentent des difficultés équivalentes. Suite à ce développement, il a été demandé aux participants de se séparer et de modéliser individuellement les spécifications de chacun des *dashboards* à l'aide de *Concur Task Trees* (CTT) [Paternò 97], un standard [6] très utilisé en IHM pour l'ingénierie des besoins que nous avons imposé.

1. https ://www.amcharts.com/
2. http ://www.highcharts.com/
3. https ://developers.google.com/chart/interactive/docs/gallery
4. http ://datamatic.io/
5. http ://trifacta.github.io/vega/editor/index.html
6. https ://www.w3.org/2012/02/ctt/

Exemple simplifié d'énoncé : la figure FIGURE 7.1 donne un exemple simpliste à trois tâches, à gauche, et une seule relation à trouver. Le modèle de référence, *i.e.,* la solution que les participants doivent trouver, est illustrée à droite avec la relation d'exclusivité (XOR). Les participants doivent choisir les relations temporelles, horizontales, entre les tâches d'un même nœud (*i.e.,* les tâches sœurs). Les relations hiérarchiques, verticales, de l'arbre étaient données. Les relations définies par CTT sont mises à disposition [7] : *enabling*, *independence*, *suspend/resume*, *concurrent*, *exclusion*, *disabling* and *optional*. Les quatre scénarios considérés dans l'expérience contiennent respectivement 7, 8, 9 et 11 relations temporelles à retrouver.

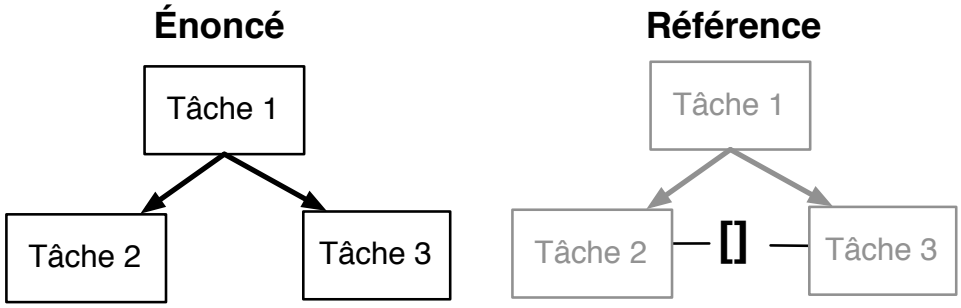


FIGURE 7.1 – Exemple simplifié d'énoncé et de modèle de référence.

Les résultats des participants sont comparés entre eux et avec un modèle de référence conçu par l'examinateur de l'expérience. Nous mesurons la conformité des réponses avec le modèle de référence et la similarité des solutions entre les participants. Sur l'exemple de la figure FIGURE 7.2 un participant a bien choisi l'exclusion ([ ]) mais l'autre a choisi une relation séquentielle (>>). Le participant A à Faux, le participant B a Vrai, nous obtenons une paire "Vrai-Faux". Nous intéresser aux paires de réponses nous permets d'évaluer la diversités des solutions, à la fois par rapport à une solution dite de référence, et vis-à-vis des participants entre eux.

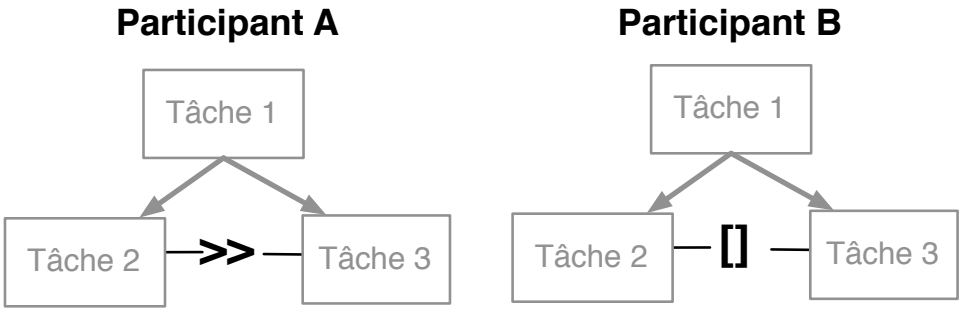


FIGURE 7.2 – Exemple d'une paire de réponse.

7.2.2 Observations

Le bénéfice d'une approche par modélisation est bien établi [Jacobson 99, Stahl 06] dans le cadre du développement d'un système logiciel, en terme d'expressivité, d'abs-

7. Nous omettons volontairement des versions alternatives des relations impliquant de l'échange de données entre tâches afin d'abstraire les résultats de l'ambiguïté de leur sémantique.

Table 7.1 – Conformité des réponses avec le modèle de référence

	Scenarios				
	Activité d'un athlète	Planification de Projet	Déploiement dans le Cloud	Consommation d'énergie	Moyenne Pondérée
Relations	7	8	11	9	
Vrai	43%	25%	72.7%	44.4%	41.7%
Faux	57%	75%	27.3%	55.6%	58.3%

Table 7.2 – Pourcentage de similarité des réponses

Scenarios	Similaire		Different	
	Vrai-Vrai	Faux-Faux	Vrai-Faux	Faux-Faux
Planification de Projet	8.3%	20.8%	33.3%	37.5%
Consommation d'énergie	22.2%	11.1%	44.4%	22.2%
Moyenne pondérée	12.1%	18.2%	36.4%	33.3%

traction, etc. Le tableau 7.1 montre cependant que la modélisation à postériori d'une application par ses propres développeurs diffère du modèle de référence dans plus de 58% des relations utilisées. Les scénarios de l'expérience correspondent aux *dashboards* simples que les participants ont développé précédemment, ils en maîtrisent donc le fonctionnement. Un pourcentage d'erreur aussi haut indique que l'utilisation forcée d'un langage de modélisation mène à des erreurs de conception ou demande de la formation et un temps d'apprentissage, même s'il s'agit d'un standard de l'état de l'art.

Le tableau 7.2 illustre la diversité des réponses. Nous considérons comme "vraies" les relations identiques au modèle de référence et "fausses" celles différent du modèle de référence. Nous séparons les paires "Faux-Faux" en deux cas, selon si les deux participants ont indiqué la même réponse fausse ou deux réponses fausses différentes. Les mesures présentées sont mesurées par paire de réponse pour chaque relation afin de donner une importance égale à toutes les paires similaires et différentes. Il y a un pourcentage comparable de cas où les deux réponses sont fausses (33.3%) et où seul un des deux participants choisit la bonne relation (36.4%). Nous en déduisons que les erreurs ne sont pas induites par un sous-ensemble identifiable de relations ambiguës, mais bien par la compréhension du langage en lui-même. Les participants utilisent la même relation dans seulement presque 30% des cas, et cette relation n'est correcte que dans 12.1% des cas. Dans presque 70% des paires, il y a un désaccord entre les participants, et 88% des paires présentent au moins une réponse fausse.

7.2.3 Discussion et perspectives

Cette expérience illustre la dispersion des solutions atteintes par une équipe projet résultant de l'utilisation d'un langage de modélisation imposé. Ce résultat motive le besoin d'intégration de langages auquel les experts domaines sont formés et la vérification de cohérence durant la modélisation par plusieurs acteurs.

Cette expérience couvre un seul aspect de notre cas d'application, similaire à l'ingénierie des besoins (*RE*). Toutefois, d'autres études montrent que le même principe s'applique dans d'autres domaines. Par exemple, dans la conception de visualisation, les experts du

domaine attestent qu'il n'existe pas de taxonomie absolue des graphiques [Wilkinson 12] et l'imposition d'une même taxonomie mène à un appauvrissement des résultats. Cela qui mène à la co-existence de plusieurs classifications et donc des modélisations hétérogènes.

La plupart des besoins d'intégrations rencontrés nécessitent la coopération entre plus de trois domaines métiers. Par exemple, le projet SmartCampus étend notre cas d'application vers l'interaction de onze domaines au total [Mosser 13]. Cette multiplicité des domaines à couvrir, la variabilité des langages de modélisation disponibles pour chaque aspect et les erreurs de conception induites par l'utilisation forcée d'un langage montrent qu'une approche d'intégration est à la fois naturelle et nécessaire lors de la conception de systèmes complexes.

Bien que cette expérimentation mette en avant les divergences de compréhension d'un langage imposé au sein d'un petit groupe, nous prévoyons de réutiliser ce protocole avec un groupe plus important de sujets afin de préciser ces résultats. De plus, cette expérience devrait être réalisée avec d'autres langages de modélisation d'arbre de tâches présentant une sémantique différente pour éviter le biais des choix de conception de CTT. Enfin, en prévoyant dans le protocole de mesurer la substitution entre chaque type de relation, il serait possible de mesurer l'influence de la proximité sémantique entre les relations (*e.g., indépendence* et *concurrence*) sur la diversité des solutions afin de conclure si le problème repose dans un sous-ensemble de ces relations et de pondérer la divergence des solutions par la distance sémantiques de leurs relations respectives.

7.3 Surcoût de l'encaspulation en service

Cette section s'intéresse à l'influence de la représentation des domaines sous la forme de méta-modèles encapsulés dans des services. Elle a pour but de mesurer le surcoût de la modélisation des domaines en services.

7.3.1 Hypothèses et données

L'approche d'intégration que nous avons choisie nécessite l'encapsulation de chaque modèle du domaine dans un service. Les méta-classes d'un domaine sont pré-existantes, mais l'exposition du service du domaine nécessite un effort de programmation de la part de l'expert du domaine. Cela implique l'implémentation des opérations métier du service, la définition des messages à échanger avec le service, la gestion des exceptions, et au besoin la déclaration d'objets métier. Nous mesurons dans cette section le surcoût en terme de code de l'approche proposée, en comparaison avec les artefacts des modèles du domaine fournis.

Le tableau 7.3 contient le nombre de lignes de code ajoutées manuellement par l'expert pour encapsuler un service. Ces mesures sont détaillées par nature de contribution : *i.e.,* l'implémentation des opérations, la déclaration des objets métier, des messages et des exceptions spécifiques au domaine. Le nombre d'opérations exposées de chaque domaine est indiqué pour permettre d'estimer la taille relative de ceux-ci.

Le tableau 7.4 affiche le volume total de code manipulé dans notre approche. En détail il s'agit du code spécifique à l'approche Service *(i)* nécessitant une implémentation manuelle (calculé dans le tableau précédent) et *(ii)* pouvant être généré par l'approche. Le code généré correspond par exemple au squelette de définition des messages, des objets métiers, des opérations et des services qui est automatisé et guide l'expert lors de l'apport de sa connaissance métier. On compare ces deux colonnes spécifiques à l'encapsulation

TABLE 7.3 – Détail du volume de code de l'encapsulation en service en Lignes de Code (LoC)

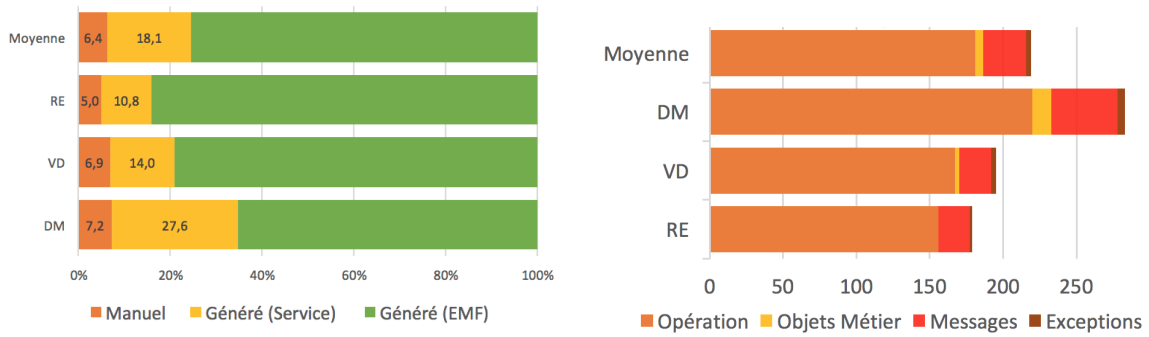
Domaines	**Implémentation Manuelle du Service (LoC)**					Nombre d'Opérations
	Opérations	Objets Métier	Messages	Exceptions	**Total**	
Data Management	220	13	45	5	**283**	11
Visualization Design	167	3	22	3	**195**	6
Requirement Engineering	156	0	21	2	**179**	5

avec *(iii)* le code fournit par l'utilisation d'EMF. Il aide ainsi à mesurer l'impact de l'encapsulation en service sur la quantité globale de code exécutable de chaque domaine. Pour démontrer que notre approche vient avec un faible surcoût, le ratio du volume de code spécifique à notre approche sur le volume global doit être faible.

7.3.2 Observations

Le tableau 7.3 montre que le coût manuel d'encapsulation d'un domaine en fonction de sa taille (*i.e.,* du nombre d'opérations qu'il expose) est comparable avec les métriques classiques de développement de méthodes. Par exemple, un service exposant 11 opérations nécessite seulement 283 lignes de code au total pour l'implémenter. En moyenne, environ 30 lignes de code sont nécessaires par opération. Comme l'illustre la partie droite de la FIGURE 7.3, les opérations (*e.g.,* le code métier) représentent le principal effort de développement par l'expert. Puisque les opérations sont conçues pour être haut-niveau afin de manipuler des modèles, il est naturel d'en obtenir un petit nombre avec une taille moyenne similaire aux méthodes classiques, au contraire des nombreuses méthodes nécessaires à la manipulation de chaque méta-élément dans les méthodes CRUD.

FIGURE 7.3 – Proportion de code généré par rapport au code spécifique à l'intégration à implémenter manuellement, et détail du code manuel.



Comme l'illustre la partie gauche de la FIGURE 7.3, la somme des codes orientés service représente en moyenne moins de 25% de la quantité totale de code exécutable[8].

8. Les packages *.sdk, *.tests, *.ui, *.edit and *.editor générés par EMF ne sont pas pris en compte.

Table 7.4 – Surcoût de l'approche par service en Ligne de Code (LoC)

Domaines	**Implémentation Manuelle du Service**	Code du Service Généré	Code Généré par EMF	Somme	% Spécifique à l'approche
Data Management	**283**	1083	2556	3922	34.83
Visualization Design	**195**	395	2232	2822	20.90
Requirement Engineering	**179**	386	2994	3559	15.87
AVG	219	621	2594	3434	24.47

Dans le pire des cas, pour le plus gros domaine exposant le plus d'opérations sans avoir plus de code EMF, le ratio est proche du tiers du code complet.

7.3.3 Conclusions

D'après ces mesures, le surcoût de l'encapsulation en services représente en moyenne un volume de 25% du code manipulé et 35% dans le pire des cas. Cet effort de développement manuel est concentré sur des problématiques métier, *i.e.,* l'implémentation des opérations. Nous estimons que le coût de cette approche est raisonnable étant donnés les avantages qu'elle apporte, *e.g.,* l'isolation stricte des domaines et la montée en abstraction des actions utilisateur.

Nous observons que le nombre d'opérations exposées par un domaine ne découle pas directement du nombre de méta-éléments qui le compose. La table 7.5, utilisée dans l'expérience suivante, indique les tailles des méta-modèles considérés. Par exemple les méta-modèles des domaines *Visualisation Design* et *Data Management* sont de tailles similaires tandis que ce dernier expose presque le double d'opérations.

Puisque la définition de ces opérations provient de l'utilisation du domaine par son expert, une complexité accidentelle est possible lors de l'encapsulation pour certaines classes de domaines. Si le nombre d'opérations d'un domaine augmente, l'effort à fournir par l'expert reste majoritairement du code métier. Nous prévoyons de mesurer l'impact de l'ajout d'opérations sur les volumes de code lors des futures itérations sur les services du cas d'application pour caractériser l'évolution de cette complexité.

7.4 Gestion des interactions par des Règles Métier

Notre approche propose d'externaliser le couplage des domaines en préservant leur intégrité par intégration et en résolvant leurs interactions à l'aide de règles métier. Nous souhaitons mesurer l'impact de l'isolation des méta-modèles en terme de couplage par rapport aux approches par fusion de méta-modèles. Du fait de la différence de nature des deux approches, il est impossible d'établir un élément de comparaison commun mesurable. Nous proposons donc de mesurer l'impact d'une fusion des méta-modèles RE, VD et VM en terme de couplage, puis de proposer une approche où ce couplage est nul et de mesurer ce que coûte la gestion de la cohérence qui en découle.

7.4.1 Hypothèses et données

Nous souhaitons appliquer une approche de fusion sur les méta-modèles des domaines *RE*, *DM* et *VD* (Figure 6.2, Figure 6.3 et Figure 6.4). Nous proposons tout d'abord quelques mesures de complexité sur ces méta-modèles en entrée. Le tableau 7.5 utilise des métriques classiques d'évaluation des méta-modèles [Fenton 14] en détaillant le nombre et la nature des méta-éléments et leur couplage en utilisant l'indice du *Data Abstraction Coupling* (DAC et DAC') [Li 93]. Le DAC représente le nombre d'attributs au sein d'une classe qui ont une autre classe comme type. Le DAC' quant à lui mesure le nombre de classes différentes qui sont utilisées comme attributs d'une classe. Ces deux métriques sont calculées pour chaque classe non énumérée et nous présentons la valeur moyenne pour chaque domaine pour en évaluer le couplage interne.

Table 7.5 – Métriques sur les méta-modèles des domaines.

Domaines	Méta-Classes			Relations		DAC	DAC'
	Normales	Abstraites	Enum	Héritage	Association		
Data Management	12	5	1	11	12	1.35	1.24
Visualization Design	12	3	2	8	8	1.33	1.13
Requirement Engineering	11	2	7	4	10	2.15	2
Moyenne Pondérée	11.67	3.33	3.33	7.67	10	1.58	1.42

Nous appliquons ensuite un algorithme *Match and merge* sur les trois méta-modèles EMF. Nous commençons par préfixer toutes les méta-classes par leur domaine respectif pour éviter les incompatibilités. Puis nous identifions et fusionnons les concepts connexes. Nous matchons par exemple : VD.Source ↔ DM.Sensor et RE.Goal ↔ VD.WhatQualifier. Le méta-modèle fusionné produit est donné en Annexe A.10.

Le tableau 7.6 mesure la différence de couplage entre l'ensemble des modèles des domaines, où chaque catégorie est la somme des trois méta-modèles, et le résultats d'une fusion de ces méta-modèles afin d'établir une valeur seuil. Par exemple pour la deuxième colonne, les domaines *DM*, *VD* et *RE* ont respectivement 12, 12 et 11 méta-classes classiques, la somme brute est donc de 35 méta-classes tandis qu'une approche par fusion réduit ce nombre à 29, apportant un gain de 17%. Les métriques sur les domaines sont résumées à partir de la table 7.5 pour apporter un élément de comparaison.

Le tableau 7.7 expose pour chaque domaine le nombre d'interactions que ses règles déclenchent. Il est à noter qu'une seule règle peut déclencher des interactions vers plusieurs autres domaines. Nous définissons qu'une règle appartient à un domaine si le message déclencheur (*i.e.,* sa partie gauche) est adressé à une opération du service de ce domaine. Nous nous attendons à observer une concentration intra-domaine sur la diagonale de la matrice, et peu d'interactions inter-domaines de chaque côté. En effet notre approche vise à aider les experts à conserver la maîtrise des conséquences d'invocations d'opérations. Or les conséquences inter-domaines sont les plus aisées à prévoir et il est normal d'avoir du couplage au sein d'un domaine.

Table 7.6 – Comparaison de métriques entre la somme brute des modèles des domaines et l'approche par fusion de modèles

	Méta-Classes			Relations		*DAC*		*DAC′*	
	Classe	Abst.	Enum	Inherit.	Assoc.	SUM	AVG	SUM	AVG
Somme Brute	35	10	10	23	30	71	1.58	64	1.58
Approche par Fusion	29	10	8	21	41	73	2.03	68	1.89
Différentiel Brute/Fusion	-17%	-	-20%	-9%	+36%	+3%	+28%	+6%	+16%

Table 7.7 – Nombre d'interactions provoquées par les règles d'intégration, par domaine

	Domaines Métiers			Services Auxiliaires	
	VD	**DM**	**RE**	Association	Notification
Visualization Design (VD)	10	5	1	5	5
Data Management (DM)	1	13	0	1	1
Requirement Engineering (RE)	2	0	7	2	1
Association	0	0	0	3	0
Notification	0	0	0	0	3

7.4.2 Observations

Nous pouvons voir que la fusion permet de simplifier globalement jusqu'à 20% le nombre de méta-classes ainsi que les relations hiérarchiques d'héritage qui les lient. Toutefois cela vient avec un prix : une augmentation de 36% du nombre d'associations. Les moyennes des *DAC* et *DAC′* démontrent une augmentation significative, respectivement de 28% et 16%, du couplage interne entre ces méta-classes. Cela peut être expliqué par la sémantique de la fusion qui ajoute des relations entre les méta-classes des différents domaines mais diminue le nombre de méta-classes en fusionnant les concepts similaires, donnant un méta-modèle résultat proportionnellement plus dense que ses sous-domaines.

L'analyse de la répartition des règles nécessaires à l'intégration des services du tableau 7.7 nous permet de mesurer l'impact de l'approche par intégration. Avec 31 règles et 58 interactions considérées, la densité diagonale de la matrice montre que la majorité des règles interagissent sur leur propre domaine, vérifiant ainsi une cohérence intra-domaine. Parmi les règles inter-domaines, 13 règles interagissent avec les services auxiliaires (*i.e.,* Association et Notification) et 9 atteignent d'autres services de domaines. Parmi les 58 interactions, seules 15.25% sont inter-domaines.

Les scénarios présentés dans le Chapitre 6 ne nécessitent que 9 règles pour gérer les interactions difficiles, c'est-à-dire inter-domaines. La Figure 7.4 illustre pour chaque domaine en abscisse le volume des interactions de ses règles métiers dédiées à chaque autre domaine. Pour DM par exemple, on voit que peu de ses règles appellent des opérations

de VD alors que l'inverse n'est pas vrai. Cette figure montre bien que chaque domaine proportion majoritaire d'interaction interne (*i.e.*, de sa couleur).

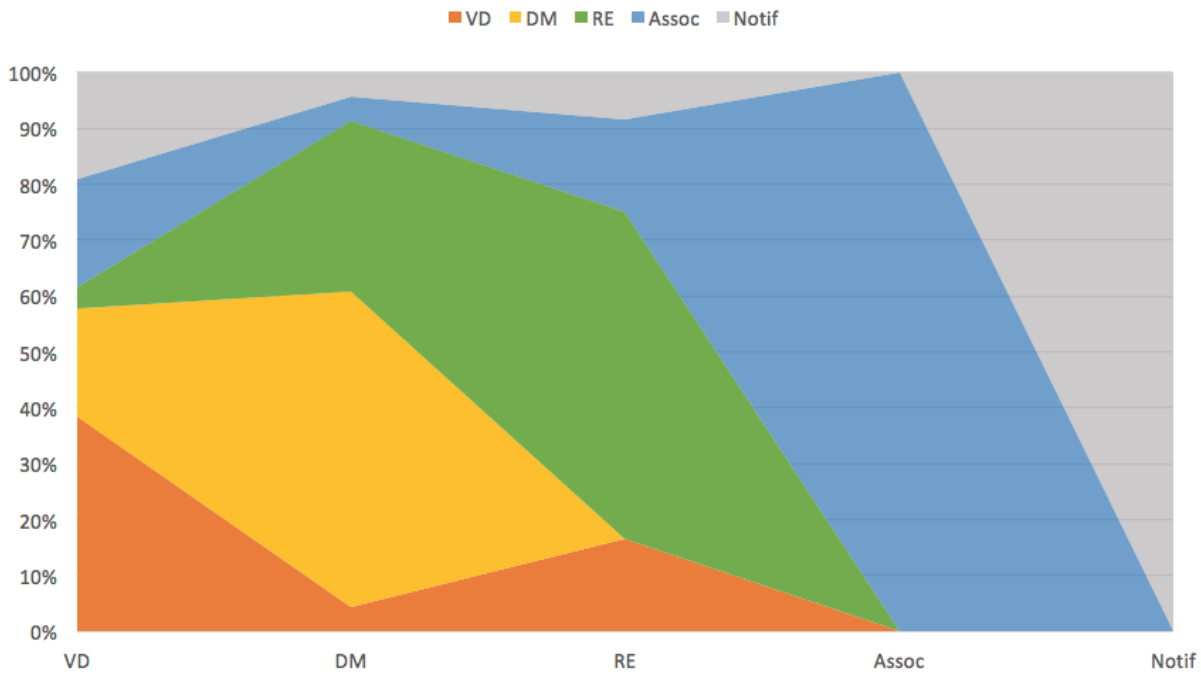


FIGURE 7.4 – Répartition (en %) des domaines ciblés par les interactions par domaine.

7.4.3 Conclusions

Ces résultats indiquent que l'approche d'intégration maintient un découplage entre les domaines en externalisant les interactions transverses dans un moteur conçu précisément pour cette tâche. Du fait de cette externalisation, il n'y a par construction aucun couplage entre les modèles du domaines eux-mêmes contrairement à l'augmentation des métriques DAC et DAC' constatée avec la fusion, la propriété d'isolation est donc respectée.

D'un coté dans le cas du méta-modèle fusionné, la cohérence structurelle assure de ne produire que des modèles conformes aux concepts de tous les domaines considérés, à condition que le méta-modèle soit valide, ce qu'EMF permet de vérifier. De l'autre coté avec l'intégration des méta-modèles, la cohérence des modèles produits est assurée par un moteur externe. Nous avons montré que celle-ci ne requiert que peu d'effort. Elle permet également de remonter au niveau du moteur les règles de cohérence interne au domaine, et ainsi de les abstraire de la représentation spécifique à celui-ci. Ainsi si l'on change la technologie d'implémentation du méta-modèle ou sa syntaxe abstraite par une autre, tant que l'interface du domaine reste la même, la gestion de la cohérence n'a pas à être retouchée.

La séparation des préoccupations de l'approche d'intégration permet donc une maintenance aisée des interactions entre les domaines puisqu'elles prennent la forme de règles formelles concentrées dans un seul moteur. Cette séparation permet également de remplacer facilement un méta-modèle par un autre méta-modèle du même domaine puisque ceux-ci restent isolés. Il est à noter que la pertinence des règles métiers considérées dans les mesures précédentes est un problème inhérent à SOA et en dehors de la portée de cette contribution.

7.5 Automatisation de l'approche

7.5.1 Hypothèses

Le protocole de communication entre les domaines intégrés, présenté dans le Chapitre 3 nécessite des artefacts fonctionnels tels qu'un moteur d'exécution des règles métiers et des *proxies* pour les services. Dans cette section, nous mesurons le coût de mise en place de ces artefacts et la capacité d'automatisation de l'approche. Comme l'approche est conçue pour être générique, nous devons pouvoir fournir ou générer automatiquement tout le code fonctionnel nécessaire à l'intégration. L'approche ne nécessite en entrée de la part des experts que la connaissance métier de l'implémentation des opérations et de la part de l'intégrateur uniquement la connaissance de la sémantique de collaboration.

Le tableau 7.8 indique le volume de code dédié au protocole de communication. Il détaille pour chaque artefact quelle part de code est requise en entrée de la part de l'intégrateur, quelle part est générable à partir des services exposés par les experts, et quelle part est fournie avec l'approche puisque développée pour le cas d'utilisation et générique.

Table 7.8 – Coût de l'intégration en Ligne de Code (LoC)

Package	Manuel		Généré
	Intégrateur	Fourni	
Moteur de Règles	156	24	264
Proxies des Domaines	0	0	254
Objets Métiers	0	70	0
Services Auxiliaires	0	411	109
%	12.1	39.2	48.7

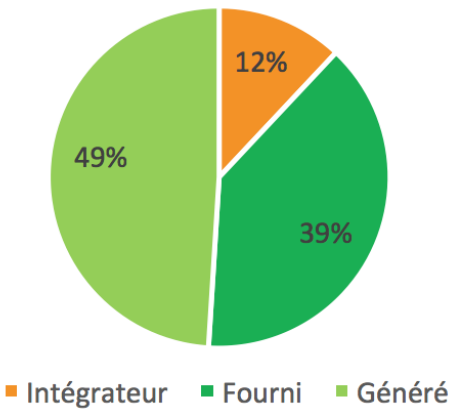


Figure 7.5 – Volume de code a fournir manuellement par l'intégrateur par rapport au code généré ou fourni par l'approche car générique.

7.5.2 Observations

Concernant les artefacts d'intégration, le tableau 7.8 montre que l'approche vient avec 39% du code requis (comprenant les protocoles de communications, la gestion du moteur de règles, les services de Notification et d'Association...) et que près de 49% du reste est généré à partir des interfaces des services (*e.g.,* opérations et structure des messages). Par exemple les *proxies* et les règles de routage sont entièrement dérivées des interfaces de service.

Seules les règles d'intégration sont laissées à la charge de l'intégrateur ce qui représente seulement 12% du volume total du code dédié à l'intégration. En effet, ces règles d'intégration sont l'implémentation de la sémantique de collaboration entre les domaines et ne peuvent donc pas être générées puisque cette sémantique est une donnée d'entrée nécessaire de toute approche de composition.

7.5.3 Conclusions

L'ensemble de ces mesures démontrent que les artefacts de code spécifiques à l'intégration des domaines sont soit fournis par l'approche, soit générés à partir des entrées identifiées : les interfaces des services de domaines et la sémantique de collaboration. Cela illustre le haut niveau d'automatisation que notre proposition permet, et valide sa capacité à décharger les experts des tâches annexes pour qu'ils se concentrent sur des tâches métiers.

La table **??** identifie la classe de problème qui bénéficie le plus de l'utilisation de SOA, à savoir intégrer de nombreux domaines ayant beaucoup d'interactions à gérer. L'effort de développement mesuré nous semble trop élevé pour une intégration sur peu de domaines ne présentant pas ou peu d'intersection. Les apports en terme d'isolation et de cohérence sont favorables respectivement au nombre de domaines, pour faciliter leur évolution et leur remplacement, et à la gestion de leurs interactions, du fait de l'évolutivité de la base de règles.

Table 7.9 – Catégorisation de la classe de problème adressée

	Peu de domaines	**Nombreux domaines**
Peu d'interactions	Peu d'intérêt	Bénéfice de l'isolation
Nombreuses interactions	Bénéfice d'une base de règles évolutive	Classe de problème ciblée par SOA

7.6 Exp #2 : Caractérisation et concrétisation

7.6.1 Présentation de l'expérience

Motivations et contexte : La contribution du Chapitre 5 propose un nouveau moyen de concrétiser un système en cours de conception à partir de caractéristiques et d'un *feature model*. Nous cherchons maintenant à mesurer l'apport de cette contribution sur le cas d'application. Mary Shaw propose une taxonomie des validations en génie logiciel [Shaw 03] qui classe cette question de recherche dans la catégorie des "Méthodes ou moyens de développement". Pour évaluer l'apport, il a été nécessaire de mettre en place

une expérience dédiée. Celle-ci a pour objectif de comparer les méthodes existantes et notre proposition dans un cas réel d'utilisation. L'objectif de cette section est de présenter cette expérience.

Résumé du protocole expérimental : Cette expérience ayant pour but de tester l'acceptation des utilisateurs vis-à-vis d'une solution basée sur la caractérisation, elle a été conçue sous la forme d'un test utilisateur. L'objectif de ce test est de présenter successivement les spécifications textuelles de quatre visualisations d'un *dashboard*, et de faire choisir à l'utilisateur un ensemble de *widgets* concrets satisfaisants à l'aide d'une approche manuelle, ou assistée par une caractérisation. L'approche manuelle consiste à faire choisir parmi les 73 *widgets*, regroupés par bibliothèque. L'utilisateur peut les parcourir autant de fois qu'il veut et indique le sous-ensemble des *widgets* qu'il considère pertinent pour le besoin considéré. L'approche par caractérisation se base sur la taxonomie des capacités offertes par les visualisations définie par la communauté du *Data Journalism* et utilisée dans le domaine *VD* de notre implémentation. Les caractéristique constituant la taxonomie et leur définition sont présentés aux utilisateurs au début du test. Durant la concrétisation par caractérisation, l'utilisateur effectue une itération en ajoutant et/ou enlevant un ensemble de caractéristiques qu'il estime nécessaires pour satisfaire le besoin considéré. L'assistant présente, à chaque fin d'itération, le sous-ensemble des *widgets* satisfaisant ces caractéristiques, calculé à partir d'une configuration dans le *feature model*. L'utilisateur peut décider à chaque fin d'itération que ce sous-ensemble le satisfait ou qu'il souhaite démarrer une nouvelle itération.

Exemple de déroulement de l'expérience : Nous proposons de détailler un exemple factice d'expérience afin de guider le lecteur dans la compréhension du protocole et des observations.

(*0*) Dorian se présente pour passer l'expérience. L'assistant tire au sort qu'il suivra le protocole B (*cf.* Annexe B.2).

(*1.0*) L'assistant présente à Dorian 73 widgets classés par bibliothèque, ainsi qu'une taxonomie (Annexe B.4) caractérisant les capacités des visualisations. Dorian s'y familiarise pendant 10 minutes en posant des questions sur les termes qu'il ne comprends pas.

(*1.1*) L'assistant présente à Dorian un premier exercice : "Afin de se faire une idée de la contribution relative de chaque étudiant membre d'une même équipe, il faut proposer une visualisation du nombre de tickets gérés par chacun des membres de l'équipe en relation avec le volume total de tickets à gérer.".

(*1.2*) Lors de sa première itération, Dorian choisit en une minute les caractéristiques *Proportion* et *Temporisation*. L'assistant lui présente les 16 *widgets* correspondants. L'ensemble ne semble pas satisfaisant à Dorian.

(*1.3*) Lors de sa seconde itération, Dorian choisit en trente secondes d'ajouter les caractéristiques *Discrète* et *Part d'un tout* et de supprimer *Proportion*. L'assistant lui présente les 7 *widgets* correspondants, 3 sont nouveaux et 4 étaient présents dans le premier ensemble, les autres ont été éliminés. Dorian est satisfait de cet ensemble de *widgets*. Il sélectionne parmi cet ensemble les *widgets* qu'il va utiliser.

(*1.4*) L'assistant présente un nouvel exercice et Dorian répète le protocole précédent jusqu'à être satisfait de l'ensemble de *widgets* proposé. Il sélectionne comme précédemment les *widgets* qu'il va utiliser.

(*2.0*) L'assistant enlève la taxonomie à Dorian.

(*2.1*) L'assistant présente à Dorian le troisième exercice : "L'objectif de cette troisième visualisation est d'illustrer le cycle de vie des tickets qui passent par les statuts *To Do*, *In Progress*, *Done*) en fonction du temps pour détecter les mauvaises habitudes de gestion des tickets et l'influence du temps".

(*2.2*) Dorian parcourt les *widgets* et annonce à l'assistant celles qui satisfont le besoin : les *Area Chart* de toutes les bibliothèques et la *100% Stacked* d'AmCharts. On répète l'étape 2 avec le quatrième exercice de visualisation.

(*3.0*) L'assistant redonne la taxonomie et le troisième exercice de visualisation à Dorian. Ce dernier doit chercher à la satisfaire par caractérisation en ayant en mémoire les *widgets* qu'il a choisit manuellement.

(*3.1*) Dorian choisit les caractéristiques *Part D'un Tout*, *Temporisation* et *Variation*. L'assistant lui présente les 5 *widgets* correspondant, Dorian choisit de s'arrêter à cette première itération puisqu'il en retrouve 4 parmi ceux qu'il avait en tête.

Propriétés de l'expérience : Cette expérience a été conçue pour être reproductible. Le protocole expérimental, le scénario d'exemple considéré et les définitions sont détaillés en annexe (*cf.* Annexe B). Les données brutes résultant de l'expérience sont également données en annexe (*cf.* Annexe B.5).

L'expérience s'appuie sur des données d'entrées librement accessibles : 73 *widgets* ont été considérés, extraits de 5 bibliothèques de visualisations différentes [9] [10] [11] [12] [13] dont les exemples et le code sont librement accessibles. La taxonomie utilisée pour caractériser les fonctions de visualisation est détaillée en annexe et la création du modèle de variabilité utilise Familiar [14] dont un exemple d'utilisation pour les *widgets* est disponible [15].

Puisque l'objet de l'expérience est la comparaison de deux méthodes, nous devons éviter le biais lié à la différence entre les populations tests affectées à chaque méthode. Pour cela, chaque utilisateur utilise les deux méthodes. De plus, l'expérience a été conduite auprès de 12 utilisateurs présentant un niveau d'éducation uniforme dans des domaines connexes, préparant une thèse en Génie Logiciel.

L'ordre d'utilisation des méthodes par chaque utilisateur peut induire un biais dans les résultats. Pour pallier à cela, nous séparons la population test en deux groupes A et B, affectés respectivement au protocole A (*cf.* Annexe B.1) et au protocole B (*cf.* Annexe B.2), couvrant ainsi l'utilisation de la méthode manuelle puis de la méthode par caractérisation, et inversement. L'affectation à un groupe est aléatoire et les utilisateurs ne sont pas prévenus de l'existence d'un autre protocole.

Afin d'éviter la nécessité d'apprentissage d'une Interface Homme-Machine, l'utilisateur ne manipule pas et ne voit pas l'interface de manipulation du *feature model*. Pour les deux méthodes, le choix des *widgets* se fait en interagissant avec l'assistant.

Afin de comparer ces méthodes, nous mesurons le temps nécessaire à chacune et la satisfaction de l'utilisateur vis-à-vis de l'ensemble de *widget* retourné par notre approche.

9. https ://www.amcharts.com/
10. http ://www.highcharts.com/
11. https ://developers.google.com/chart/interactive/docs/gallery
12. http ://datamatic.io/
13. http ://trifacta.github.io/vega/editor/index.html
14. https ://github.com/FAMILIAR-project/familiar-documentation
15. https ://github.com/ILogre/WidgetFamiliarPilot

Dans l'approche manuelle, l'utilisateur choisit directement ses *widgets*, sa satisfaction vis-à-vis de l'ensemble qu'il sélectionne est donc toujours de 100%. Établir un oracle qui évaluerait la pertinence de cet ensemble en fonction du besoin exprimé introduirait un biais, puisqu'il n'existe pas de visualisation absolue qui satisfasse tous les utilisateurs pour un besoin donné. Nous mesurons toutefois un retour d'expérience qualitatif de la part des sujets qui utilisent d'abord la méthode manuelle puis celle par caractérisation sur le même besoin pour évaluer si leur satisfaction vis-à-vis de l'ensemble sélectionné manuellement a été influencé par de nouvelles propositions automatiques.

7.6.2 Observations

Quantitatif - Utilisation de la méthode par caractérisation : Nous cherchons à mesurer l'utilisation de la méthode par caractérisation pour évaluer ses performances. Pour cela, nous présentons quatre mesures au sein de la Table 7.10 :

— le temps moyen nécessaire à une itération, pour mesurer la difficulté de choix des caractéristiques,

— le nombre moyen d'itérations réalisées par l'utilisateur pour obtenir un ensemble de *widgets* satisfaisants,

— le pourcentage moyen du catalogue de *widgets* restant à la fin des itérations pour mesurer l'efficacité de sélection,

— le pourcentage moyen de *widgets* satisfaisants parmi ceux restant à la fin des itérations pour mesurer la pertinence de la sélection.

La Figure 7.6 illustre les cinq quartiles pour chacune de ces mesures. Par exemple nous pouvons y voir notamment que le temps médian par itération est inférieur de 42 secondes, un temps minimum à 5 secondes et maximum à 84 secondes, où 50% des résultats se trouvent entre 24 et 46 secondes. La Figure 7.7 propose pour les trois dernières mesures la proportion de résultats en segments. Par exemple, concernant le nombre d'itérations effectuées, nous pouvons voir que la moitié de la population n'a utilisé qu'une ou deux itérations. Ces résultats nous permettent d'évaluer la facilité d'utilisation de la méthode par caractérisation, nécessitant seulement $2,5$ itérations par visualisation avec un temps médian de 42 secondes par itération. Ces itérations réduisent l'ensemble de solutions à 12% de son volume total avec une satisfaction médiane de 81% sur ce sous-ensemble.

Nous souhaitons désormais apporter une analyse plus fine afin de préciser les différents cas de tests contenus dans ces résultats médians. En effet, les valeurs dont elles sont tirées correspondent à des situations différentes d'utilisation, par exemple nous pouvons penser que la première visualisation à traiter prend plus de temps alors que le traitement d'une visualisation par caractérisation est facilité si l'on a parcouru l'ensemble des *widgets* plusieurs fois avant (*i.e.,* cas du protocole A). Nous détaillons les données en six segments, représentant chacun une situation où un utilisateur a eu besoin d'utiliser la méthode par caractérisation. Nous séparons ainsi les résultats selon la visualisation considérée et l'ordre d'utilisation des méthodes par l'utilisateur. Il s'agit des segments notés S_i, où i prend valeur dans :

1. *i.e.,* le traitement de la visualisation 3 par le protocole A,
2. *i.e.,* le traitement de la visualisation 4 par le protocole A,
3. *i.e.,* le traitement de la visualisation 1 par le protocole A,

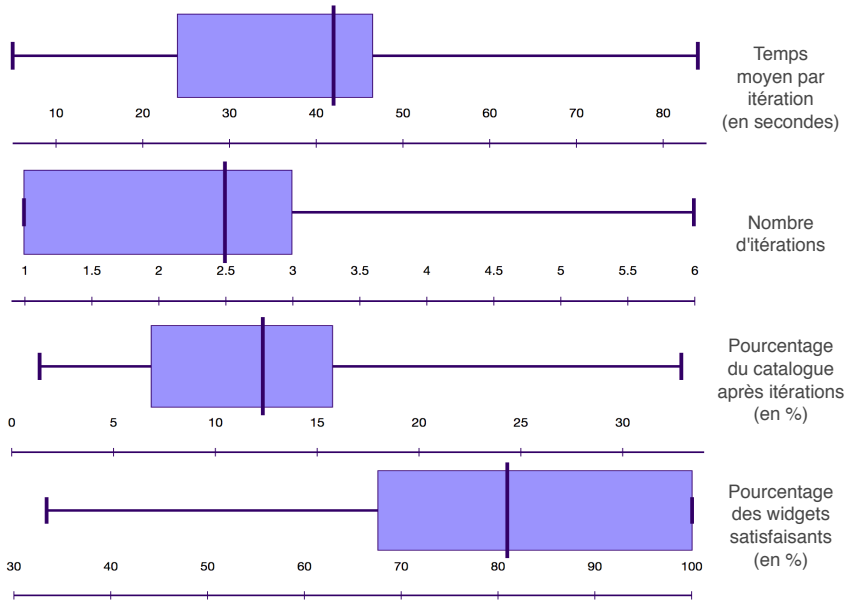
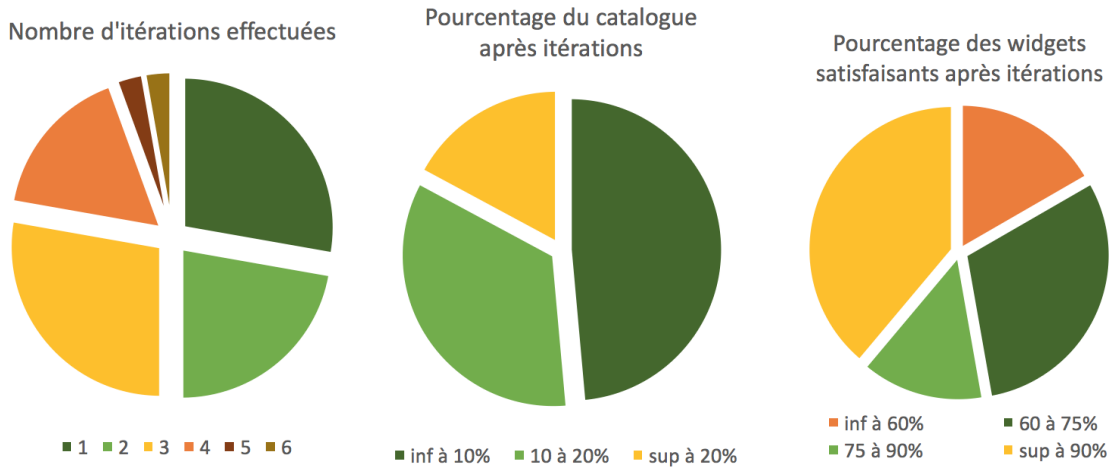


FIGURE 7.6 – Mesures par caractérisation en quartiles (min, Q1, médiane, Q3, max).

FIGURE 7.7 – Mesures d'utilisation de la méthode par caractérisation



4. *i.e.,* le traitement de la visualisation 1 par le protocole B,
5. *i.e.,* le traitement de la visualisation 2 par le protocole B,
6. *i.e.,* le traitement de la visualisation 3 par le protocole B.

Comme affiché dans la TABLE 7.10, chacune des moyennes de segment est comprise entre 49 et 23 secondes, le temps moyen d'une itération étant de 39 secondes. On note une différence entre les populations qui commencent par la caractérisation (S_4 et S_5) et les caractérisations après avoir parcouru manuellement le catalogue (S_1 et S_2) qui prennent plus de temps. Ceci peut être expliqué par la résistance au changement lors de l'inversion des méthodes. Cependant toutes les moyennes des segments sont proches de la médiane affichée précédemment dans la FIGURE 7.6.

Les utilisateurs ont besoin en moyenne de $2,53$ itérations pour obtenir un sous-ensemble du catalogue qu'ils jugent satisfaisant. Ici encore, les moyennes des segments sont toutes proches de la médiane générale. Le sous-ensemble résultant représente en

TABLE 7.10 – Moyennes des résultats par segment de l'utilisation de la méthode par caractérisation

Mesures \ Segment	**Moy** S_1	**Moy** S_2	**Moy** S_3	**Moy** S_4	**Moy** S_5	**Moy** S_6	**MOY**
Temps moyen par itération	0'47"	0'49"	0'23"	0'38"	0'39"	0'38"	0'39"
Nombre moyen d'itérations	3,5	2	2	3,33	3	2	2,53
Pourcentage du catalogue après itérations	9%	14%	15%	16%	8%	12%	13%
Pourcentage de *widgets* satisfaisants après itérations	83%	81%	80%	76%	76%	82%	80%

moyenne 13% du catalogue de *widgets* proposés et est constitué à 80% de *widgets* jugés satisfaisants, chacune des moyennes de segment étant compris entre 76% et 83% de satisfaction.

La segmentation des données montre l'influence de la difficulté croissante des visualisations de test sur le temps nécessaire : le temps de traitement de la visualisation 1 (S_3 et S_4) est inférieur celui de la visualisation 2 (S_5), lui même inférieur à celui de la visualisation 3 (S_1 et S_6) lui même inférieur à celui de la visualisation 4 (S_2). Toutefois cela ne semble pas avoir d'effet notable sur le nombre d'itérations moyen nécessaires ni sur la satisfaction vis-à-vis du sous-ensemble de *widgets* atteint, nous concluons que cette méthode par caractérisation s'adapte bien aux six situations de test considérées.

Quantitatif - Comparaison entre les deux méthodes : Nous cherchons à mesurer la différence entre l'utilisation des deux méthodes. Pour cela nous mesurons les performances moyennes des méthodes manuelle et par caractérisation ainsi que le gain relatif sur chacune des visualisations de test pour établir une comparaison. La FIGURE 7.8 affiche les temps moyens réalisés pour satisfaire chacune des visualisations à l'aide des deux méthodes. La TABLE 7.11 détaille les résultats. La colonne de gain est calculée à partir des temps moyens de deux méthodes pour chaque visualisation selon la formule : $(Temps_{manuel} - Temps_{caracterisation})/Temps_{manuel}$. Pour s'abstraire du biais de la familiarisation avec une visualisation adressé dans la TABLE 7.12, les données correspondent uniquement aux étapes 1 et 2 des protocoles expérimentaux, sans considérer l'étape 3. Il est à noter que du fait de la mise en place de deux protocoles expérimentaux, la moitié des utilisateurs a commencé par la méthode manuelle tandis que l'autre moitié a commencé par la caractérisation, il n'y a donc pas d'influence d'ordre dans les résultats présentés. Nous pouvons voir que pour toutes les visualisations les utilisateurs ont mis en moyenne moins de temps en utilisant la méthode par caractérisation, impliquant un gain moyen supérieur à 50%, le gain étant près de 20% dans le pire des cas, et plus de 68% dans le meilleur des cas.

Quantitatif - Apprentissage : Nous cherchons à mesurer l'effet d'apprentissage, c'est-à-dire la variation (espérée positive) des performances d'un utilisateur en fonction du temps. Pour cela nous mesurons le gain de temps induit par l'utilisation répétée de cha-

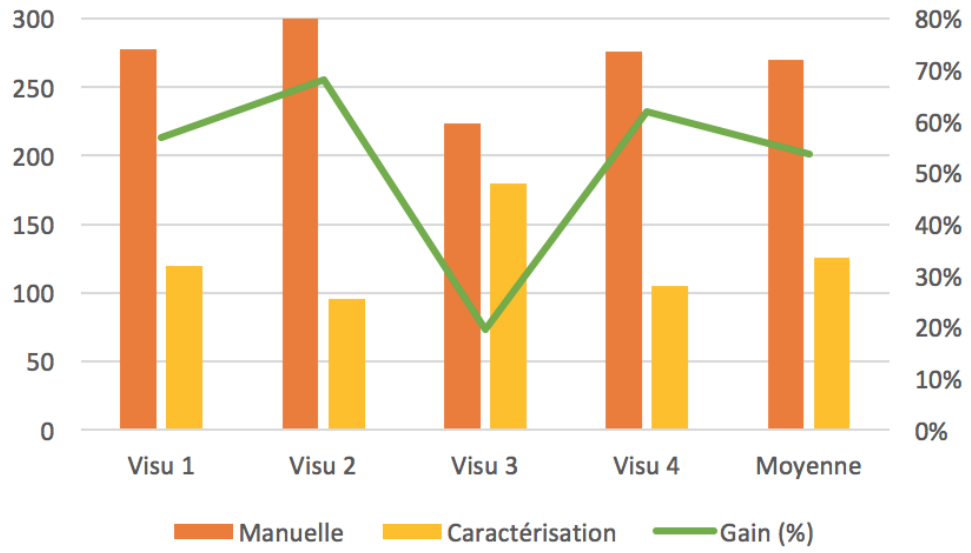


FIGURE 7.8 – Comparaison des temps moyens (en s) de traitement des visualisations et gain (en %) entre les méthodes.

TABLE 7.11 – Comparaison des temps moyens d'utilisation des méthodes par visualisation

Visualisations	Manuel (moy)	Caractérisation (moy)	Gain (%)
Visu 1	4'38"	2'00"	56,9%
Visu 2	5'02"	1'36"	68,3%
Visu 3	3'44"	3'00"	19,7%
Visu 4	4'36"	1'45"	62%
Moyenne	4'30"	2'05"	54%

cune des méthodes afin d'avoir une intuition sur l'effet d'apprentissage de chacune des méthodes et sur l'utilisation de la méthode par caractérisation par des utilisateurs habitués à la méthode manuelle. La TABLE 7.12 affiche, par méthode, le temps de traitement de la première visualisation traitée, de la seconde, puis le gain de temps calculé selon la formule : $(Temps_1 - Temps_2)/Temps_1$. La première ligne de données se lit : "La population ayant suivi le protocole A a mis en moyenne 4 minutes et 38 secondes à satisfaire la visualisation 1 manuellement puis 5 minute et 2 secondes à satisfaire la visualisation 2 manuellement, ce qui représente un perte d'efficacité de 9%". On peut voir que la seconde utilisation de la méthode par caractérisation diminue le temps moyen nécessaire de 20 à 42%, tandis que la seconde utilisation de la méthode manuelle voit le temps nécessaire augmenter, entre 9 et 23%. Les deux dernières lignes exposent les résultats moyens d'utilisation de la méthode par caractérisation sur une visualisation traitée manuellement au préalable. Cette mesure donne une intuition du gain apporté par la méthode de caractérisation à des utilisateurs ayant pour habitude de rechercher manuellement, *i.e.,* ayant déjà en tête un ensemble de *widget* satisfaisant à atteindre. Dans ces conditions, les utilisateurs mettent 4 à 5 fois moins de temps pour retrouver un ensemble de *widgets* qu'ils jugent satisfaisants.

TABLE 7.12 – Gain de temps mesuré par l'utilisation répétée de chaque méthode

	Population	Temps 1	Temps 2	Gain (%)
Méthode Manuelle	Pop. A	(visu 1) 4'38"	(visu 2) 5'02"	- 9 %
	Pop. B	(visu 3) 3'44"	(visu 4) 4'36"	- 23 %
Méthode Caractérisation	Pop. A	(visu 3) 3'00"	(visu 4) 1'45"	42 %
	Pop. B	(visu 1) 2'00"	(visu 2) 1'36"	20 %
Manuelle puis Caractérisation	Pop. A	(visu 1 manuelle) 4'38"	(visu 1 carac.) 0'48"	84%
	Pop. B	(visu 3 manuelle) 3'44"	(visu 3 carac.) 1'08"	75,5 %

7.6.3 Conclusions et discussions

Quantitatif - Interprétation des résultats : Cette expérience permet de mesurer l'utilisation d'une méthode de choix d'artefacts concrets par caractérisation et de la comparer avec une méthode de choix manuelle, *i.e.,* l'alternative actuelle de choix de *widgets* pour la conception d'un *dashboard*.

Les résultats présentés montrent qu'en utilisant une taxonomie avec laquelle ils ne sont pas familiers les utilisateurs convergent vers un sous-ensemble de *widgets* dans un temps raisonnable, en moyenne 2 minutes et 5 secondes. Le nombre d'itération estimé nécessaire par les utilisateurs reste faible, près de $2,5$, chacune étant effectuée en 39 secondes. Le sous-ensemble de *widget* atteint par l'utilisateur représente moins d'1/6ème du catalogue présenté et est constitué à 80% d'éléments satisfaisants. Nous défendons donc que le niveau de difficulté de cette méthode est adapté à la tâche et qu'elle permet effectivement d'atteindre un ensemble d'artefacts pertinents.

En matière de comparaison, cette expérience démontre que l'utilisation de la méthode par caractérisation est plus rapide qu'un choix de *widgets* manuel, et ce pour toutes les visualisations de test, en moyenne de 52% du temps dédié. De plus, le surcoût de temps induit par l'apprentissage de la taxonomie tend à décroitre, ce qui est illustré par la diminution du temps passé entre la première et la seconde visualisation satisfaites par cette méthode, en moyenne de 33%, tandis que le temps d'utilisation de la méthode manuelle s'accroît. Ces résultats suggèrent qu'après une période d'apprentissage du vocabulaire de caractérisation, la méthode serait encore plus efficace que mesuré par cette expérience et que cet effet est déjà visible après seulement deux occurrences, suggérant un apprentissage rapide.

Retour d'expérience qualitatif et perspectives : Le protocole expérimental a été établi pour recueillir un retour utilisateur qualitatif en fin de test afin de capturer les avantages et inconvénients de la caractérisation sous la forme d'une réponse libre. La TABLE 7.13 compile les résultats exprimés le plus souvent après l'utilisation de chacune des méthodes sur des visualisation différentes et après la comparaison des deux méthodes sur une même visualisation. On y trouve la confirmation de la rapidité mesurée ci-dessus et une validation de la pertinence d'une approche par caractérisation. De plus, la moitié

des utilisateurs ont déclaré que le sous-ensemble résultant de leurs itérations proposait des *widgets* qu'ils n'auraient pas choisi manuellement, par défaut de compréhension, mais qu'ils ont estimé satisfaisant après réflexion. L'inconvénient concernant l'apprentissage est contre-balancé par la mesure de diminution du temps nécessaire exposé ci-dessus.

TABLE 7.13 – Avantages et inconvénients exprimés lors du retour utilisateurs et fréquence d'occurrence

	Retour utilisateur	**Réponses dans ce sens**
Avantages	Proximité cognitive entre l'expression des besoins et les caractéristiques	10 / 12
	Rapidité d'obtention de *widgets* satisfaisants	8 / 12
	Découverte de nouvelles visualisations	6 / 12
Inconvénients	Nécessité de maîtrise du vocabulaire Apprentissage nécessaire	6 / 12
	Validité de la caractérisation actuelle des *widgets* proposés	6 / 12

Certains utilisateurs ont été surpris de constater au cours des itérations qu'un *widget* qu'ils avaient en tête ne satisfaisait pas une caractéristique choisie. Il est à noter que l'expérience n'a pas vocation à défendre la pertinence de la caractérisation, ici ad hoc, des *widgets* proposés. L'approche défendue dans le CHAPITRE 5 suppose que les caractéristiques de chaque artefact concret sont exprimés par un expert du domaine, ce qui est une limite de cette expérience. Comme détaillé dans le CHAPITRE 8, nous prévoyons de raffiner la caractérisation des *widgets* de ce cas d'utilisation à l'aide d'un site web proposé à la communauté de la visualisation de données. La pertinence de cette caractérisation aura pour effet de renforcer la pertinence du sous-ensemble atteint par l'utilisateur, or la FIGURE 7.9 illustre que la satisfaction moyenne est dépendante du nombre de *widgets* constituant ce sous-ensemble, une sélection plus précise devrait améliorer la satisfaction mesurée par cette expérience.

7.7 Conclusions

Ces expérimentations nous ont permis de montrer que :

(i) l'utilisation forcée d'un DSML mène à une dispersion des utilisations des éléments du langage et induit des problèmes de cohérence dans le système (*cf.* SECTION 7.2). Cela encourage à offrir à chaque expert la possibilité d'intégrer son domaine et de régler les interactions à un niveau d'abstraction supérieur,

(ii) l'encapsulation des domaines en service implique un surcoût moyen de 25% en volume de code (*cf.* SECTION 7.3) pour permettre de les isoler tout en exposant le comportement pertinent. Le code de gestion des interactions est fourni ou généré automatiquement à 88%, laissant uniquement les règles d'intégration à la charge de l'intégrateur (SECTION 7.5),

(iii) s'abstraire de l'augmentation du couplage induit par la fusion de méta-modèles à l'aide de règles métier permet de remonter la gestion des interactions à un niveau d'abstraction qui améliore l'évolutivité. La coordination entre le différents méta-modèle se fait par un nombre limité de règles (*cf.* SECTION 7.4),

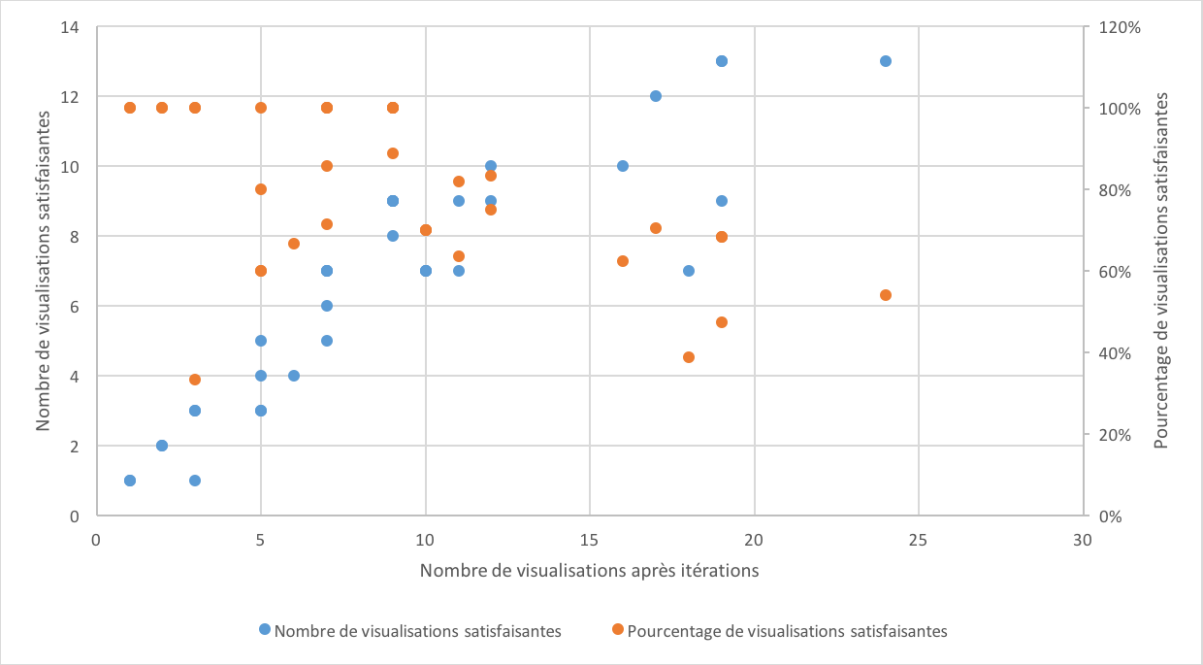


FIGURE 7.9 – Influence du nombre de visualisations après itération sur la satisfaction.

(iv) les experts et l'intégrateur fournissent un effort de développement orientées métier et profitent des capacités de génération et des artefacts fournis par l'approche (*cf.* SECTION 7.3 et SECTION 7.5),

(v) la méthode par caractérisation diminue le temps dédié à la concrétisation en moyenne de 52% par rapport à un choix de *widgets* manuel. Elle mène, en peu d'itérations, à réduire le catalogue présenté à 1/6ème de l'espace de solutions, menant à 80% d'éléments satisfaisants (*cf.* SECTION 7.6).

CHAPITRE 8

CONCLUSIONS ET PERSPECTIVES

8.1 Conclusions

Ce document propose une approche permettant de préserver l'isolation des domaines lors de la conception de système de systèmes. Cet objectif présente des défis relatifs *(i)* à l'hétérogénéité des domaines impliqués, dont les représentations ne sont pas conformes au même méta-modèle, *(ii)* aux interactions entre ces domaines qui impliquent de détecter les incohérences inter-domaines durant la conception, et *(iii)* à la multitude d'artefacts concrets disponibles pour concrétiser le système. Nous avons présenté trois contributions couvrant les différents niveaux d'abstractions considérés. La FIGURE 8.1 donne un aperçu global de l'architecture proposée dans ce document pour répondre à ces trois défis principaux, à savoir l'isolation des représentations des domaines, la cohérence inter-domaine des modèles et la gestion des artefacts disponibles pour concrétiser le système.

Au niveau des méta-modèles, nous tirons profit des bonnes propriétés des Architectures Orientées Services (SOA) en encapsulant les domaines comme des services, détaillés dans le CHAPITRE 3. L'exposition d'une interface d'opérations idempotentes hauts niveaux fonctionnant sans état, nous permet de préserver l'isolation de chaque modèle du domaine tout en permettant aux experts de contribuer au système en utilisant des actions au niveau d'abstraction proche de leur métier. Cette modélisation en service permet également de réutiliser chaque méta-modèle tel quel. Ainsi, l'outillage pré-existant du domaine est automatiquement compatible et le service du domaine résultant peut bénéficier, par exemple, des capacités de vérification, de validation, de simulation. Les méta-modèles utilisés par les experts sont donc isolés ce qui facilite la gestion de leur évolution.

Afin de gérer la cohérence au niveau des modèles, nous définissons un rôle d'intégrateur et lui permettons de concevoir et de maintenir une base de règles métier pertinentes, comme présenté dans le CHAPITRE 4. Le but de ces règles est de gérer les interactions entre domaines et de détecter les incohérences durant celles-ci. Ces règles métiers n-aires sont définies au niveau méta et utilisées sur les modèles. Elles permettent de faire remonter chaque incohérence aux experts concernés en exprimant les informations contextuelles disponibles dans son domaine afin de faciliter leur résolution. Le moteur d'intégration assure une sémantique d'exécution des règles sans ordre, sans cycle et permet la définition de règles indépendantes, facilitant ainsi le maintien de la base de règles par l'intégrateur. La cohérence inter-domaine des modèles produits par les experts est donc vérifiée à tout instant.

Au niveau de l'espace des solutions, la variabilité technologique des artefacts concrets est gérée par l'utilisation de Lignes de Produits Logiciels (SPLs), comme détaillé dans le CHAPITRE 5. Des *Feature Models* (FM) capturent la variabilité des concepts à concrétiser, modélisant les point communs et les variations entre les artefacts d'implémentation.

Nous proposons une approche automatisée permettant de générer un FM atomique pour chaque produit à partir de la caractérisation de ses capacités exprimée par l'expert, puis de les fusionner en un seul FM représentant le catalogue de produit. L'opérateur fusion des FMs utilisé assure que le catalogue résultant dispose des capacités de tous les produits en entrée et seulement de celles-ci. Ce procédé permet de capturer la multitude de produits hétérogènes que constituent l'espace des solutions. Nous assistons ainsi l'expert dans la recherche d'une configuration complète et valide, c'est à dire d'un produit pertinent pour concrétiser chaque partie du système.

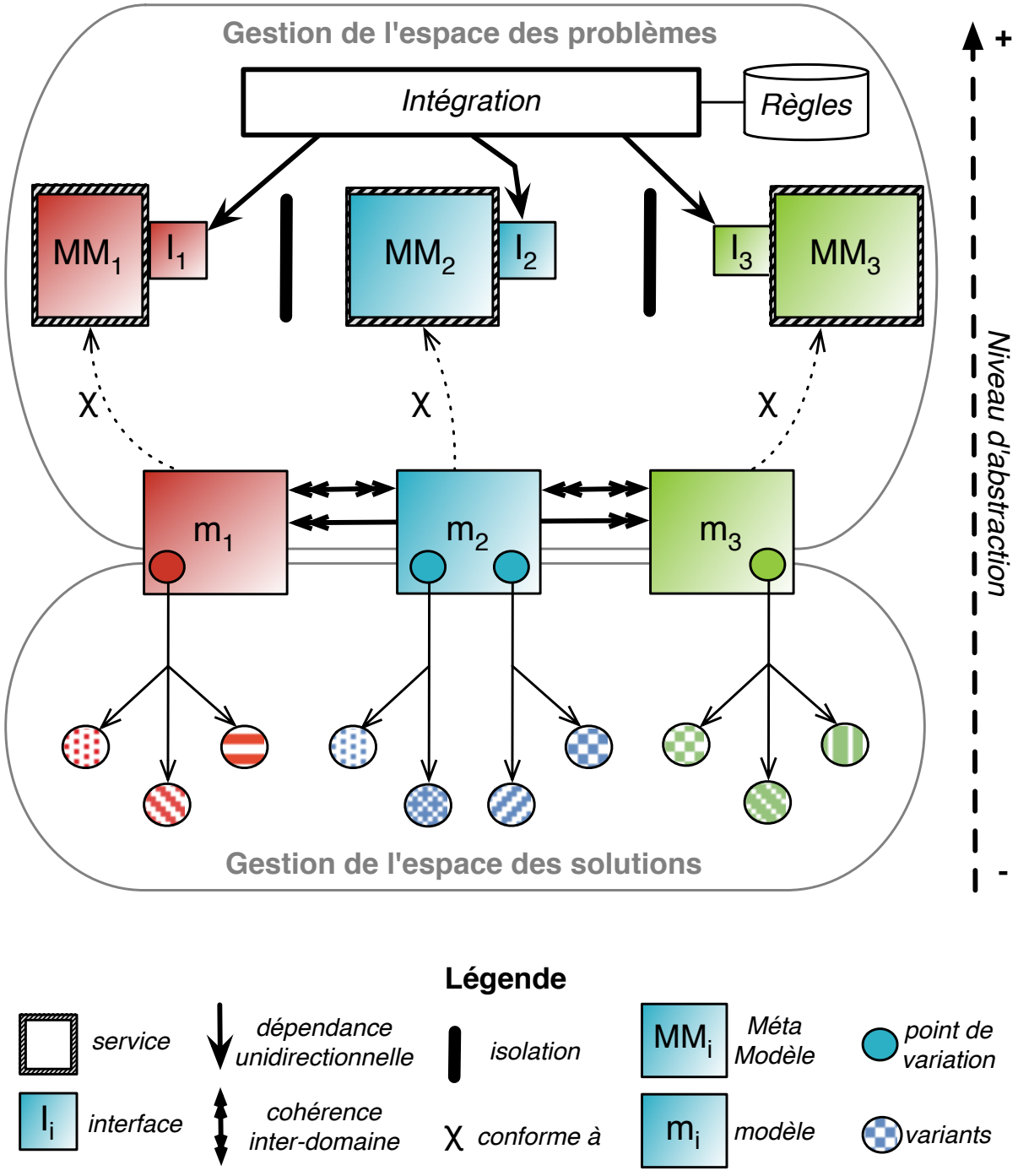


FIGURE 8.1 – Architecture de conception d'un système à trois domaines hétérogènes.

L'application de ces contributions est détaillée sur le cas de la visualisation de données en provenance de capteurs dans le Chapitre 6 où les propriétés exprimées sont instanciées sur un cas pertinent. La validation de l'approche, présentée dans le Chapitre 7, repose sur un ensemble d'expérimentation couvrant les trois contributions où nous quantifions le coût de la préservation de l'isolation par l'utilisation des services. Le surcout moyen de l'encapsulation en service des domaines est mesuré à 25% du volume de code total, soit le développement des interfaces par les experts. Concernant le protocole de communication, seules les règles entre domaine sont fournies par l'intégrateur, soit seulement 12% du code du moteur d'intégration. Enfin, une expérience utilisateur mesure le gain de temps moyen de 52% lors de la concrétisation du système, où en peu d'itération l'approche permet de réduire le catalogue à 1/6ème de l'espace des solutions, menant à 80% d'éléments satisfaisants.

L'approche d'intégration de méta-modèles hétérogènes présentée dans ce document permet donc l'utilisation de chaque domaine par son expert en stricte isolation, tout en détectant les incohérences inter-domaines grâce aux règles métier de l'intégrateur et en facilitant la configuration d'un système final conforme aux modèles.

8.2 Perspectives

8.2.1 Généralisation de la validation

La première perspective que nous proposons est une étude empirique de l'impact de l'intégration dans les systèmes logiciels. Il serait intéressant de confronter une approche d'intégration de méta-modèles à des domaines issus de l'industrie pour vérifier l'utilisation et l'utilité des interfaces de services et le passage à l'échelle du moteur de règles d'intégration. Cela nous permettrait également de mesurer l'effort nécessaire lors de l'ajout ou du remplacement d'un domaine au sein d'un système complexe déjà fonctionnel. A plus court terme, nous proposons de retravailler l'expérience présentée au sein de ce document.

L'expérience utilisateur exposée en Section 7.6 permet de mesurer le gain de temps lors de la concrétisation en utilisant une approche par caractérisation. Afin d'assurer la représentativité du panel de test, nous souhaitons augmenter le nombre de participants mais également nous inspirer des résultats de cette première expérience pour définir plusieurs profils utilisateurs. En effet, nous pouvons découper notre panel actuel en trois classes de maîtrise :

1. une population ❶ utilisant couramment des modèles, des méta-modèles ou des langages,
2. une population ❷ utilisant couramment de la visualisation de données sans outils de modélisation,
3. une population ❸ éloignée professionnellement des modèles et de la visualisation.

Bien que l'échantillon de chaque population est réduit, nous notons des différences de résultats dans ces populations. La population ❸ présente les plus grands écarts de performance (temps de concrétisation entre 47 secondes et 8 minutes et 26 secondes) et relève fréquemment une difficulté à faire le lien entre les caractéristiques et les visualisations résultante lors de la phase de retours qualitatifs. La population ❷ présente d'excellents gains de temps (82.6% et 83.7%) en passant à la méthode par caractérisation, mais a été la plus critique quand au vocabulaire de la taxonomie, suggérant la possibilité d'apporter son propre langage de caractérisation, ce qui renforce l'hypothèse d'isolation de

intégration. En définissant plus précisément un profil utilisateur pour notre expert, , et en menant cette expérience sur un panel représentatif de ce profil nous pourrions affiner les résultats en s'abstrayant des problématiques pré-requises à l'approche.

Nous souhaitons également utiliser notre approche sur un second cas d'application. Les trois contributions de ce document sont indépendantes de la visualisation de données, l'utilisation d'un autre domaine permettrait de renforcer la généricité de l'approche. Par exemple, le cas d'application de la gestion de projet, dans une forme complète et non embryonnaire comme notre fil rouge, pourrait bénéficier des interactions entre les différents acteurs d'un projet et faire ressortir le gain de la détection d'incohérence, par exemple entre la gestion de ticket et les versions de code, ou l'adéquation entre les scénarios de spécifications écrits via Gherkin Cucumber [1] et l'état de l'application à un temps t.

8.2.2 Analyse de Satisfaction des Spécifications par le Système Produit

La problématique de ce document concerne la gestion de la cohérence et de l'hétérogénéité des domaines lors de leur intégration. La cohérence des systèmes soumis à la séparation des préoccupations est un problème vaste qui nécessite un approfondissement. Nous pouvons par exemple considérer une utilisation plus directe par l'utilisateur final qui personnalise son *dashboard*. Dans le cadre du projet Smart Campus, nous avons cherché à capturer les besoins en visualisation des usagers du campus. Nous avons obtenu 15 *dashboards* différents pour 18 personnes interrogées.

Pour mettre l'utilisateur au centre de la conception de son *dashboard*, il est nécessaire qu'il interagisse à la frontière des domaines *RE*, *VD* et *DM* de notre cas d'application. L'utilisateur final peut compléter des modèles pré-établis par les experts, en s'appuyant sur une visualisation "au plus tôt" du rendu de son système. A l'inverse, l'utilisateur peut établir un modèle de base que les experts rendent opérationnel par la suite, ses choix impacteraient alors le choix des DSLs utilisés. Ces ambitions nécessitent un travail tant au niveau des modèles, en allant vers la concrétisation continue du système avec des modèles d'incertitude ; qu'au niveau méta, *e.g.,* via une réflexion sur les types de langages, leurs familles, leurs compatibilités et leur proximité pour voir lesquels sont inter-changeables.

Dans ce document, nous avons vu une approche de conception de système descendante, partant de l'intégration des méta-modèles, passant par la cohérence des modèles puis la concrétisation vers les artefacts d'implémentation. Dans ce contexte, une perspective intéressante à moyen terme à ce travail serait la mise en place d'une boucle de retour de l'adéquation entre le système généré et les spécifications.

Dans les approches MDE le code fonctionnel ne découle pas automatiquement du modèle de besoin. Or, le cycle de vie d'un système implique des modifications au niveau du code, dédiées à la personnalisation du système à son utilisateur final et à son adaptation à plusieurs conditions d'utilisation. L'idée serait de prendre en entrée un système utilisé à un temps t^i et un modèle expriment des spécifications sur ce système, conçu à un temps t^j, où $j < i$, pour s'assurer que l'évolution du système est toujours compatible avec la conception initiale.

Considérons notre cas d'application de la visualisation de données. D'un côté, le modèle du domaine *Requirement Engineering* permet des experts d'exprimer des spécifica-

1. https ://cucumber.io/docs/reference

tions indépendantes les unes des autres. De l'autre côté, le système du *dashboard* est généré en web, soit en un flot de contrôle. L'utilisation d'un modèle pivot permettrait de comparer ces deux espaces. Celui-ci peut nous permettre de vérifier trois points :

1. L'ensemble de spécification est-il satisfiable, *i.e.,* existe-t-il des spécifications incompatibles ?
2. Le système généré satisfait-il l'ensemble de ces spécifications, *i.e.,* existe-t-il un chemin d'exécution possible pour chaque spécification ?
3. Recommander une correction en identifiant les parties fautives dans les deux espaces.

Une piste de solution que nous explorons est d'utiliser des automates comme modèle pivot. En effet un flot de contrôle peut-être exprimé facilement comme un automate, et les spécifications peuvent être vues comme des automates atomiques composables. Appliqué aux *dashboards*, un système est alors un ensemble d'`États` et de `Transitions`. D'après les objectifs annoncés ci-dessus, nous avons besoin de représenter les capacités de chaque visualisation, *e.g.,* les termes de la taxonomie *data journalism* qu'elle satisfait, et les données qu'elle expose. Nous proposons un modèle où un `État` représente une visualisation, et une `Transition` représente la consommation d'une capacité, d'une donnée ou le passage à une autre visualisation. Un `Scénario` est alors un chemin valide dans l'automate.

La logique temporelle permettrait de composer les `Scénarios` atomiques des spécifications en un seul automate pour vérifier sa validité et de vérifier la satisfaction des spécifications dans l'automate du système.
Soit un Système $D_{effectif}$ dérivé d'un *dashboard* et un ensemble $S = \{s^1, \ldots, s^n\}$,

$\forall s^i \in S,\ s^i \subset D_{effectif}$

$\bigwedge \exists\ System\ D_{absolu}, \forall s^i, s^i \subset D_{absolu}$

$\bigwedge D_{effectif} \subset D_{absolu}.$

Une des difficultés réside dans l'ambiguïté de la documentation des bibliothèques de visualisation utilisées pour générer un *dashboard*. En effet, afin de produire automatiquement un automate à partir d'un système exécutable il est nécessaire de transformer le flot de contrôle en `États` et `Transitions`. Or, l'absence de système de typage stricte en Javascript et rend le recouvrement du type de visualisation difficile, interdisant ainsi de retrouver ses capacités. Par exemple, l'API de HichCharts autorise la déclaration des données de six manières différentes et incompatibles, dont certaines impliquent la perte d'identification des dimensions (*e.g.,* température en fonction du temps) et d'autres l'absence de lien entre chaque dimension et la façon dont elle est visualisée.

De plus, l'expression des relations temporelles entre les spécifications atomiques est complexe. L'enchaînement des visualisations d'un *dashboard* est statique et connu, pouvant être exprimées par les relations de séquence, d'exclusion et de parallélisme. Les spécifications conçue indépendamment nécessitent une plus grande expressivité puisque certaines relations ne sont pas spécifiées. L'utilisation des intervalles de Allen [Allen 83] définissant 13 relations différentes et exprimant le résultat de leur composition deux à deux est une piste de résolution.

Annexes

ANNEXE A

MODÈLES ET CODE DU CAS D'APPLICATION

A.1 Extraits de code des services

Exemple A.1: **Extrait de l'opération `characterizeVisu()`**

```
@Operation
public static void characterizeVisu(CharacterizeVisuMsg msg)
throws UnknownDashboardException {
  String dashboardName = msg.getDashboardName();
  String visuName = msg.getVisuName();
  String[] concerns = msg.getWhatQualifiers();

  Dashboard preexisting = getDashboard(dashboardName);
  if (preexisting != null) {
    Visualization v =
      VisualizationDesignLanguageFactory.eINSTANCE.createVisualization()
          ;
    v.setName(visuName);
    for (String s : concerns) {
      WhatQualifier c =
        VisualizationDesignLanguageFactory.eINSTANCE.createWhatQualifier
            ();
      c.setConcern(Taxonomy.valueOf(s.toUpperCase()));
      v.getConcerns().add(c);
    }
    preexisting.getVisualizations().add(v);

    StringBuilder qualif = new StringBuilder();
    for(String s : msg.getWhatQualifiers())
      qualif.append(s+" ");
    validated = false;
    Log.send("Dashboard "+dashboardName+" contains "
      +visuName+" which shows "+qualif.toString() );
      }
  ...
}
```

Exemple A.2: **Extrait de la méthode privée `loadModel()` du service VD**

```
public static Catalog loadModel(String name) {
  // load the xmi file
  XMIResource resource = new XMIResourceImpl(URI.createURI("resources/"
      + name + ".xmi"));
  try { resource.load(null); }
  catch (IOException e) {
    ... //remontée d'erreur
    return null;
  }
```

```
9
    // get the root of the model
11  return (Catalog) resource.getContents().get(0);
  }
```

Exemple A.3: **Extrait de la gestion de l'idempotence des opérations du service VD**

```
@Service
2 public class VisualizationDesign extends Service {
    static private Map<String, List<Message> > previousMessages = new
        HashMap<String, List<Message>>();
4
    // initialize the message's history queues
6   static
    {
8     for(String operationName : VisualizationDesign.Operations)
        previousMessages.put(operationName, new ArrayList<Message>());
10  }

12  @Operation
    public static void declareDashboard(DeclareDashboardMsg msg) {
14    if(!previousMessages.get("getDashboard").contains(msg)){
        ...
16      previousMessages.get("getDashboard").add(msg);
      }
18  }
    ...
```

Exemple A.4: **Règle d'intégration `Resource matching Data` sur l'opération `PlugData()`**

```
1 rule "Plug␣Data␣-␣Resource␣matching␣Data"
no-loop true
3   when
    $msg : PlugDataMsg ( $dashboard : dashboardName, $dataName : dataName)
5   and HasLinkedAsw ( answer == true )
      from AssociationDP.hasLinked(new HasLinkedMsg($dashboard, "Dashboard",
          "Catalog"))
7   and GetLinkedAsw ($catalog : model)
      from AssociationDP.getLinked(new GetLinkedMsg($dashboard, "Dashboard",
          "Catalog"))
9   and IsDefinedAsw (defined == true)
      from SensorDeploymentDP.isDefined(new IsDefinedMsg($catalog, $dataName
          ))
11  then
      VisualizationDesignAP.plugData($msg);
13    AssociationDP.link(new LinkMsg($dataName,"Source",$dataName, "Sensor",
          true));
    end
```

		Pie Chart	Funnel Chart	Line Chart	Line Chart	Area Chart	Stacked Area Chart	100% Stacked Area Chart	Step Chart	Step Chart	Bar Chart	Bar Chart	Histogram	Bubble Chart	Bubble Chart	Scatter Plot	Scatter Plot	Angular Gauge	Radar Chart	Radar Chart
2	Comparison	Oui	Oui	Non	Oui	Non	Non	Non	Non	Oui	Oui	Oui	Oui	Non	Oui	Non	Oui	Non	Oui	Oui
3	Proportion	Oui	Oui	Non	Non	Non	Oui	Oui	Non	Non	Non	Non	Non	Oui	Oui	Non	Non	Non	Oui	Non
4	Relationship	Non	Non	Non	Oui	Non	Oui	Oui	Non	Oui	Oui	Oui	Non	Non	Oui	Oui	Oui	Non	Oui	Oui
5	Hierarchy	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non
6	Location	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non
7	Probability	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Oui	Oui	Oui	Non	Non	Non	Non	Non
8	PartToAWhole	Oui	Oui	Non	Non	Non	Non	Oui	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non
9	Distribution	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Oui	Oui	Oui	Oui	Oui	Oui	Non	Oui	Non
10	Patterns	Non	Non	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Non	Oui	Oui
11	Range	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Oui	Non	Non	Non	Non	Non	Non	Non
12	OverTime	Non	Non	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Non	Non	Oui	Oui	Oui	Oui	Oui	Oui	Non	Non
13	Scalar	Oui	Oui	Oui	Oui	Oui	Oui	Non	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Non	Non	Oui	Oui	Non
14	Discrete	Oui	Oui	Non	Non	Non	Non	Non	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Non	Oui	Oui
15	Variations	Non	Non	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Non	Non	Oui	Non	Non	Non	Non	Non	Non	Non
16	Extremum	Non	Non	Oui	Oui	Oui	Non	Non	Non	Non	Oui	Oui	Oui	Oui	Oui	Non	Non	Non	Oui	Oui
17	"2D_i"	Oui	Oui	Oui	Non	Oui	Non	Non	Oui	Non	Oui	Non	Oui	Non	Non	Oui	Non	Oui	Oui	Non
18	"2D+_i"	Non	Non	Non	Oui	Non	Oui	Oui	Non	Oui	Non	Oui	Non	Non	Non	Non	Oui	Non	Oui	Oui
19	"3D_i"	Non	Non	Non	Non	Non	Non	Oui	Non	Oui	Non	Oui	Non	Oui	Non	Non	Non	Non	Non	Non
20	"3D+_i"	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Oui	Non	Non	Non	Non	Non
21	"1D_o"	Oui	Oui	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Oui	Oui	Oui	Non
22	"1D+_o"	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Oui
23	"2D_o"	Non	Non	Oui	Non	Oui	Non	Non	Oui	Non	Oui	Non	Oui	Non	Non	Non	Non	Non	Non	Non
24	"2D+_o"	Non	Non	Non	Oui	Non	Oui	Non	Non	Oui	Non	Oui	Non	Non	Non	Non	Non	Non	Non	Non
25	"3D_o"	Non	Non	Non	Non	Non	Non	Oui	Non	Non	Non	Non	Non	Oui	Non	Oui	Non	Non	Non	Non
26	"3D+_o"	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Non	Oui	Non	Oui	Non	Non	Non

FIGURE A.1 – Matrice de caractérisation d'AmCharts

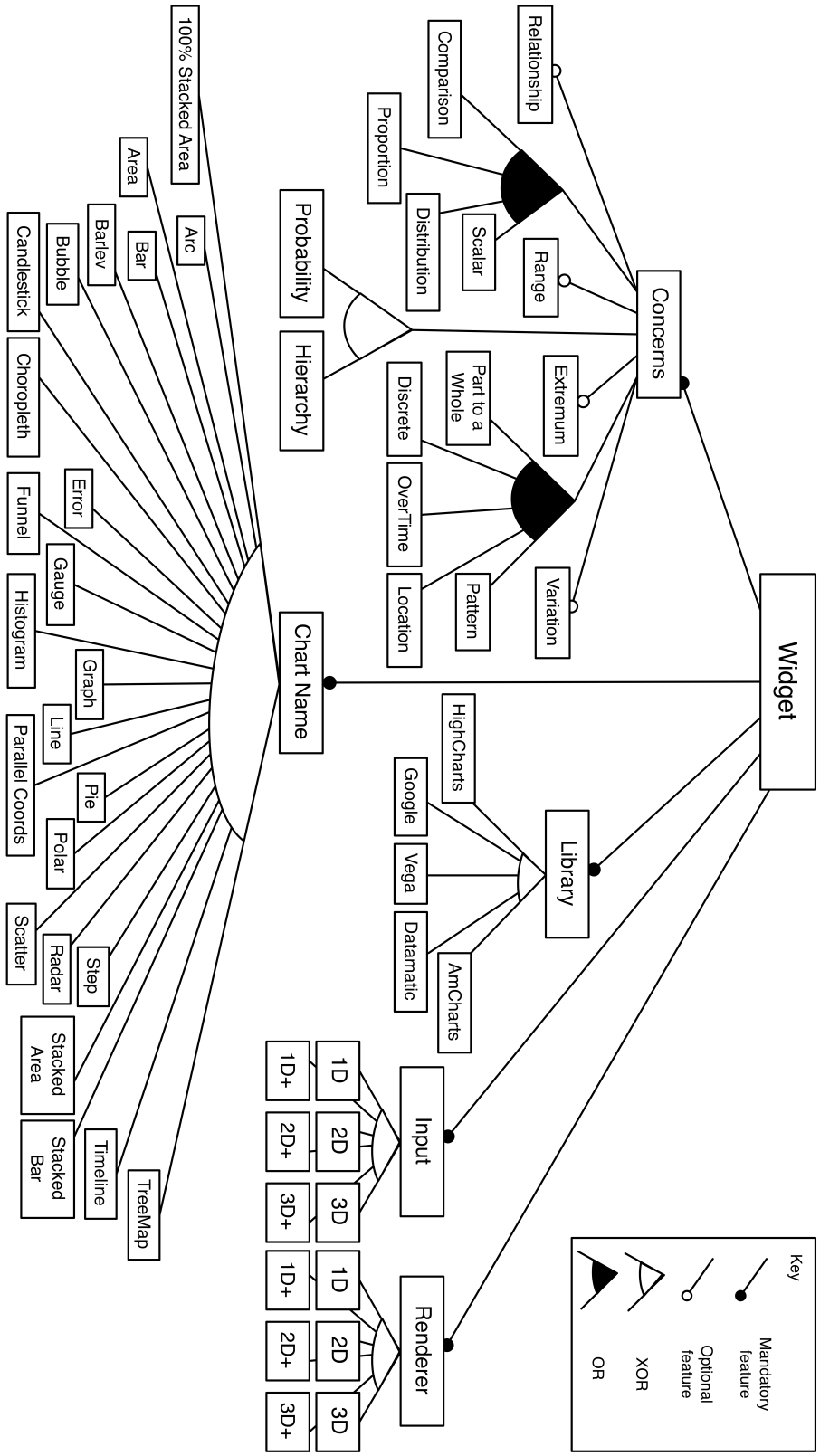


FIGURE A.2 – Feature Model fusionné des 73 widgets de visualisation

A.2 Syntaxes concrètes

```
Project: WorkingProgress
Background:
Given dashboard WorkingCondition
with views
overview to 'get a general idea about the office environment'
summer to 'monitor the air conditioning and the pollution'
winter to 'monitor the heating system and the lights'
surounding to 'analyze the noise impact on work'

Feature: overallNavigability in order to 'set the transition between the views'

Scenario: 'From the beginning'
When overview is Current
Then summer is Enable
And winter is Enable
And surounding is Enable

Scenario: 'In summertime'
When summer is Current
Then winter is Disable

Scenario: 'If snowflake'
When winter is Current
Then summer is Disable

Feature: interaction in order to 'handle the behavior between visualization'

Scenario: 'Synchronization of time'
When range time selected in surounding
Then surounding is Synchronization
```

FIGURE A.3 – Exemple d'implémentation EMF de la syntaxe concrète du domaine RE, réutilisée de Gherkin.

A.3 Syntaxes Abstraites

```
Dashboard WorkingProgress :

Visualizations :
  MiniMap shows Location of
  allsensors of type listing locate at office443 displayed as Icon

  RawValues of
  indoorTemperature of type temperature locate at office443  displayed as Scalar;
  doorStatus of type enumerate locate at office443 displayed as Scalar;
  windowStatus of type enumerate locate at office443  displayed as Scalar

  CorridorNoiseImpact shows Over_time, Comparison, Range
  of  door displayed as Discrete; corridorNoise displayed as Continuous, Threshold (max 500)

  DoorDailyState shows Proportion, Part_to_a_whole
  of  door displayed as Structural

  OutsideNoiseImpact shows Over_time, Comparison, Range
  of  window displayed as Discrete; outsideNoise displayed as Continuous, Threshold (max 500)

  WindowDailyState shows Proportion, Part_to_a_whole
  of  window  displayed as Structural

  DoorDailyActivity shows Over_time, Distribution
  of  doorOpened displayed as Color green; doorClosed displayed as Color red

  WindowDailyActivity shows Over_time, Distribution
  of  windowOpened displayed as Color green; windowClosed displayed as Color red

  OfficeHeating shows Comparison, Relationship, Over_time, Proportion
  of  indoorTemperature displayed as Continuous; outsideTemperature displayed as Continuous;
      radiator displayed as Color green, Color red

  EnergyWaste shows Comparison, Relationship, Over_time
  of  window_openning displayed as Discrete; airConditionningStatus displayed as Discrete

  OfficeCooling shows Comparison, Over_time
  of  indoorTemperature displayed as Continuous, Threshold (min 19 max 29);
      outsideTemperature displayed as Continuous

  AirConditioningState
  of  airConditioning displayed as Structural, Color green, Color red

  LightState
  of  light displayed as Structural, Color green, Color red

  RadiatorState
  of  radiator displayed as Structural, Color green, Color red

  WindowState
  of  window displayed as Structural, Color green, Color red


  Positioning :
  view overview :
  Column sized 1 : [ RawValues ]   |   Column sized 2 : [ MiniMap ]

  view surrounding :
  Column : [ CorridorNoiseImpact; DoorDailyState; DoorDailyActivity ]
         |   Column : [ OutsideNoiseImpact; WindowState; WindowDailyActivity ]

  view summer :
  Line sized 1 : [AirConditioningState; WindowDailyState]   |   Line sized 3: [EnergyWaste]

  view winter :
  Line sized 1 : [RadiatorState;LightState]   |   Line sized 3 : [OfficeHeating]
```

FIGURE A.4 – Exemple d'implémentation EMF de la syntaxe concrète du domaine VD, réutilisant la taxonomie du data journalisme.

FIGURE A.5 – Exemple d'implémentation MPS de la syntaxe concrète du domaine SD, réutilisée du framework DEPOSIT.

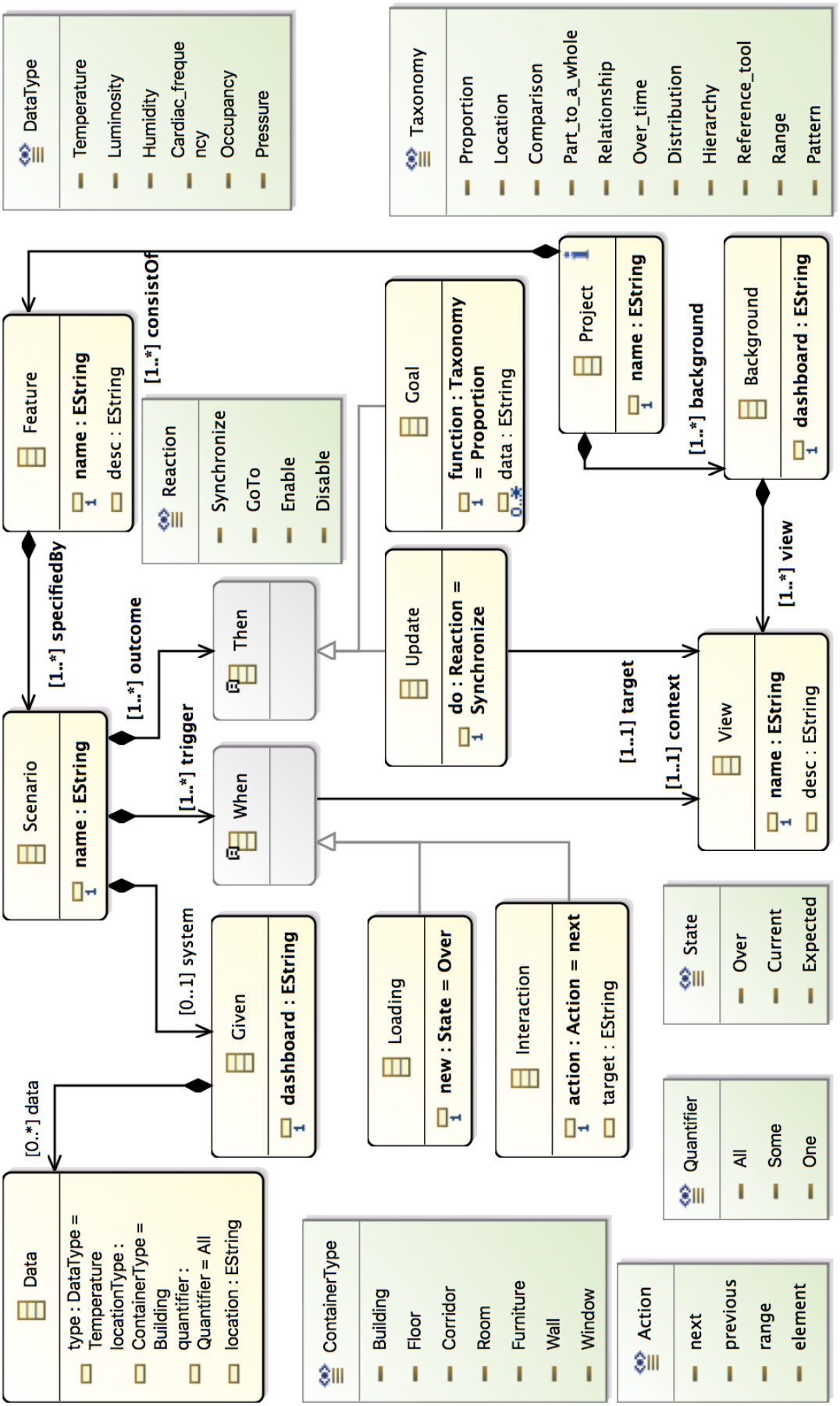


FIGURE A.6 – Exemple de syntaxe abstraite du domaine RE, adaptée de Gherkin.

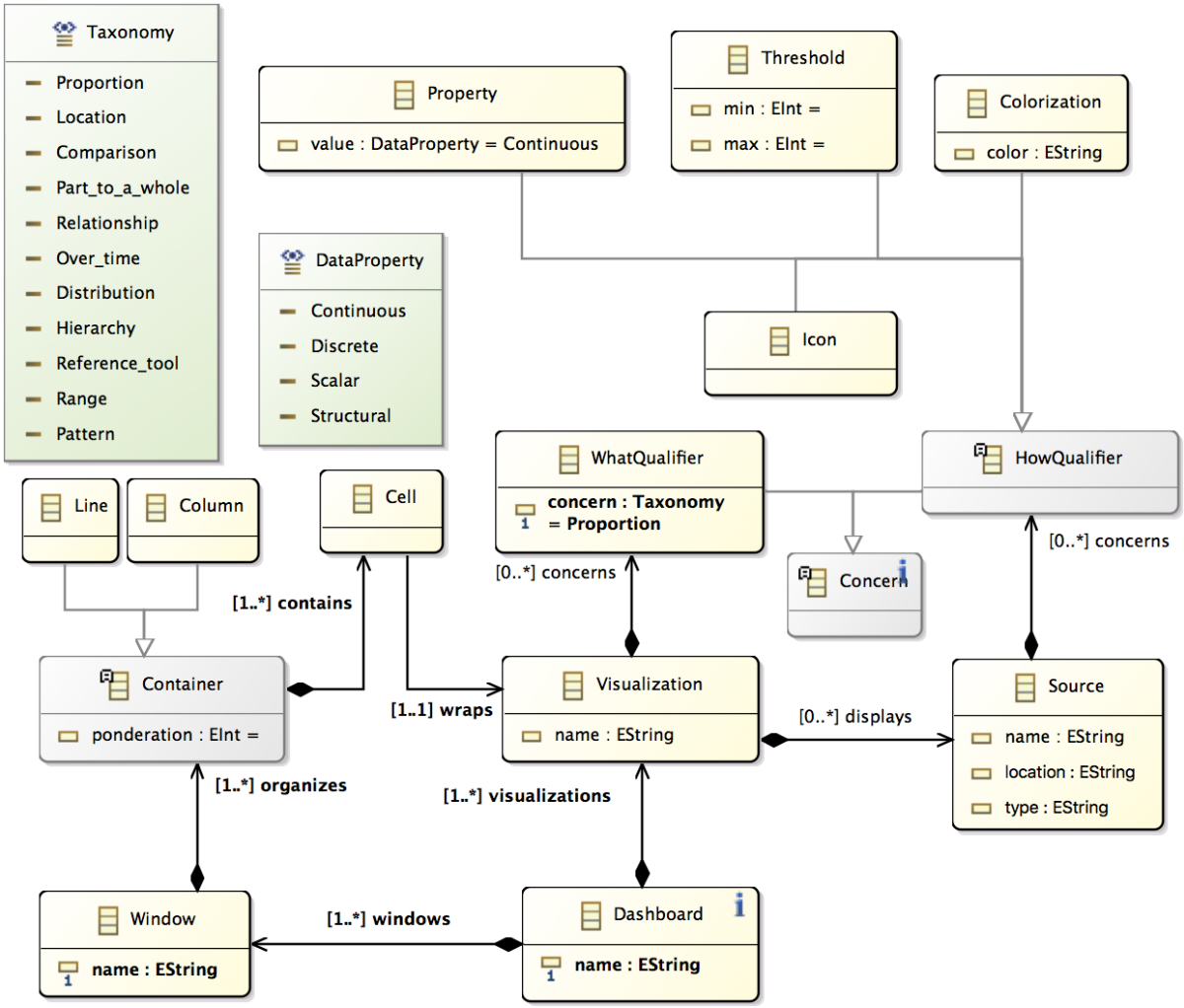


FIGURE A.7 – Exemple de syntaxe abstraite du domaine VD, réutilisant la taxonomie du data journalisme.

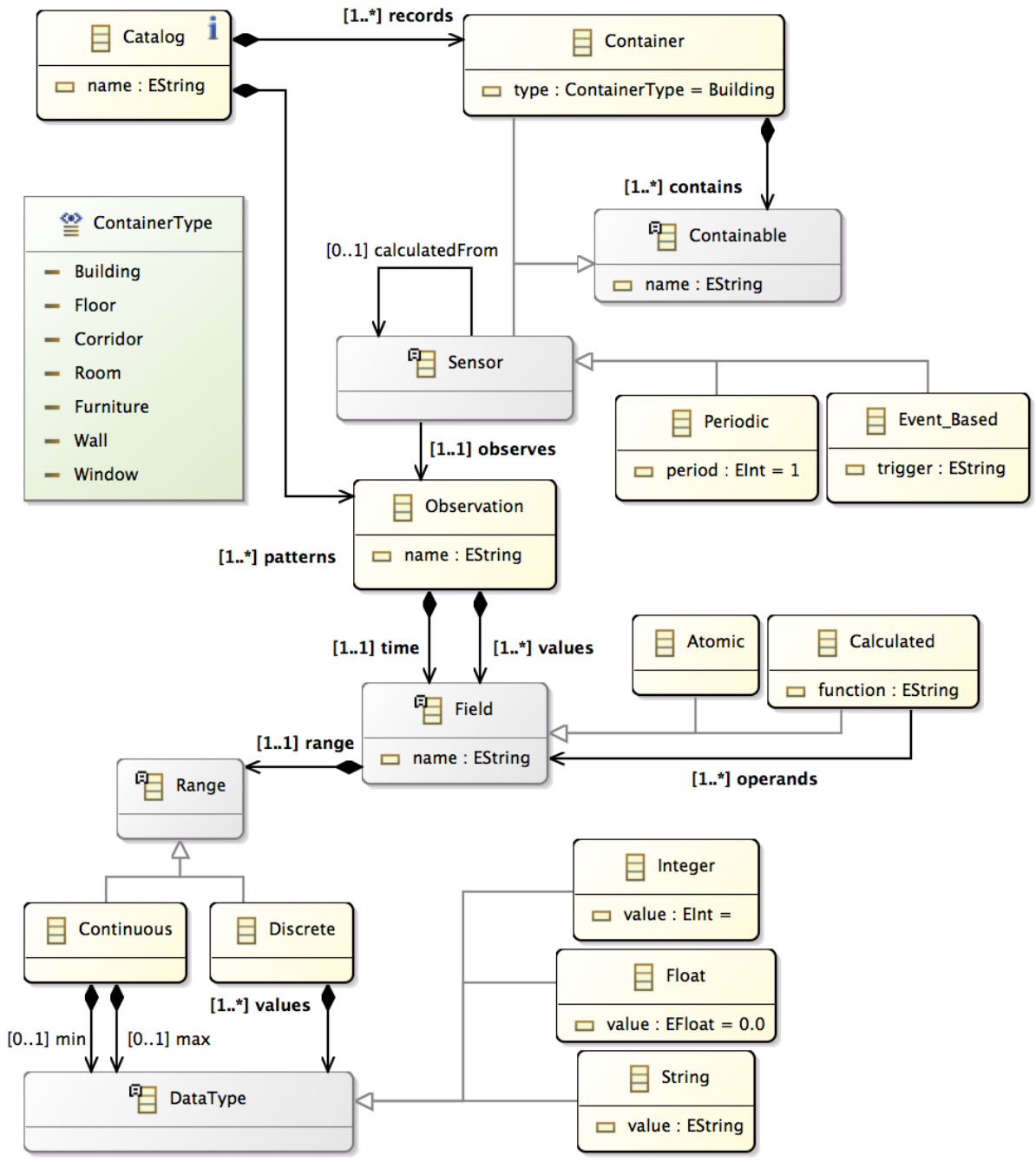
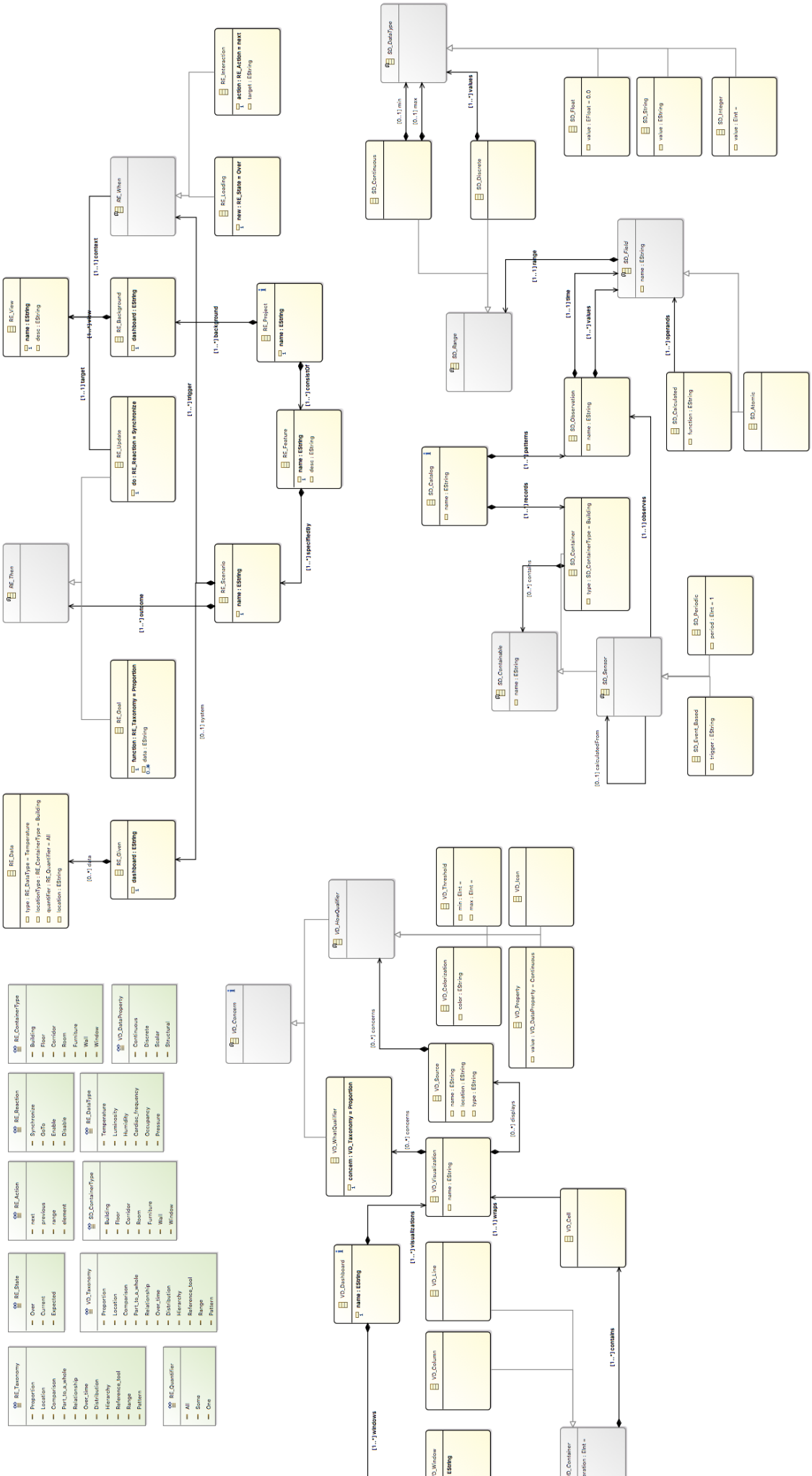
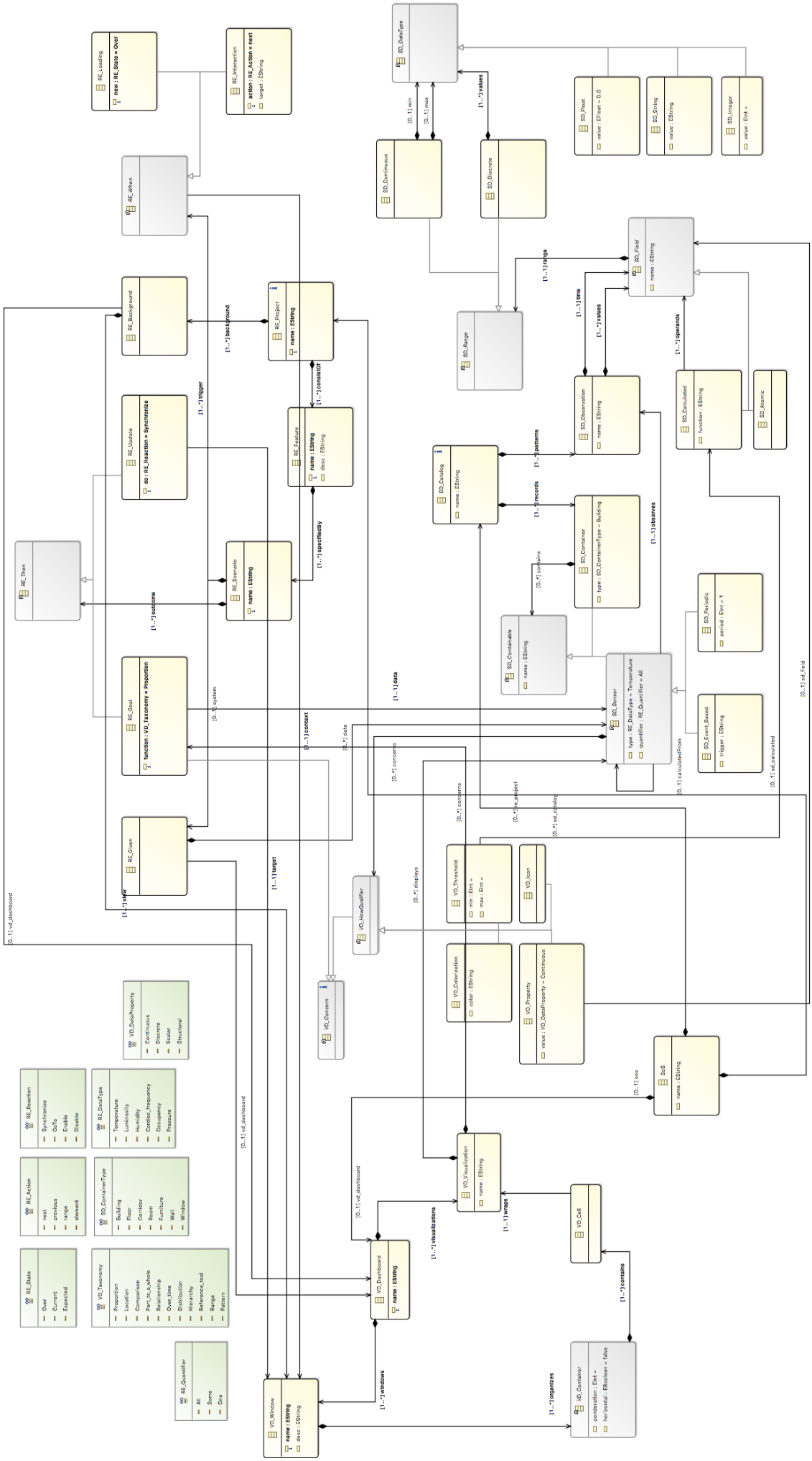


FIGURE A.8 – Exemple de syntaxe abstraite du domaine SD, réutilisée du framework DEPOSIT.




RE_Loading
new : RE_State = Over
RE_Interaction
action : RE_Action = next
target : EString
SD_DataType
SD_Continuous
SD_Discrete
SD_Float
value : EFloat = 0.0
SD_String
value : EString
SD_Integer
value : EInt =
RE_When
RE_Background
RE_Project
name : EString
SD_Range
SD_Field
name : EString
RE_Update
do : RE_Reaction = Synchronize
RE_Feature
name : EString
desc : EString
SD_Observation
name : EString
SD_Calculated
function : EString
SD_Atomic
SD_Catalog
name : EString
SD_Container
type : SD_ContainerType = Building
RE_Then
RE_Scenario
name : EString
SD_Containable
name : EString
SD_Periodic
period : EInt = 1
RE_Goal
function : VD_Taxonomy = Proportion
SD_Sensor
type : RE_DataType = Temperature
quantifier : RE_Quantifier = All
SD_Event_Based
trigger : EString
RE_Given
VD_HowQualifier
VD_Threshold
min : EInt =
max : EInt =
VD_Icon
VD_Concern
VD_Colorization
color : EString
VD_Property
value : VD_DataProperty = Continuous
SoS
name : EString
VD_DataProperty
Continuous
Discrete
Scalar
Structural
RE_Reaction
Synchronize
GoTo
Enable
Disable
RE_DataType
Temperature
Luminosity
Humidity
Cardiac_frequency
Occupancy
Pressure
RE_Action
next
previous
range
element
SD_ContainerType
Building
Floor
Corridor
Room
Furniture
Wall
Window
RE_State
Over
Current
Expected
VD_Taxonomy
Proportion
Location
Comparison
Part_to_a_whole
Relationship
Over_time
Distribution
Hierarchy
Reference_tool
Range
Pattern
VD_Visualization
name : EString
VD_Cell
VD_Dashboard
name : EString
RE_Quantifier
All
Some
One
VD_Window
name : EString
desc : EString
VD_Container
ponderation : EInt =
horizontal : EBoolean = false
[0..1] vd_dashboard
[1..1] target
[1..1] data
[0..1] sd_field
[0..1] sos
[1..1] wraps
[1..1] visualizations
[1..1] windows
[1..1] organizes
[1..1] contains
[0..1] sd_calculated
[0..1] calculatedFrom
[0..1] sd_catalog
[0..1] re_project
[0..1] concerns
[0..1] data
[0..1] system
[1..1] outcome
[1..1] trigger
[1..1] background
[1..1] specifiedBy
[1..1] consistOf
[1..1] time
[1..1] range
[1..1] values
[1..1] operands
[1..1] patterns
[1..1] records
[1..1] observes
[0..1] min
[0..1] max
[1..1] values
[0..1] contains
[0..1] system
[1..1] context
[0..1] data
[0..1] concerns
[0..1] displays
[0..1] vd_dashboard

ANNEXE B

EXPÉRIENCE : CARACTÉRISATION ET CONCRÉTISATION

Cette partie des annexes est dédiée à l'expérience décrite dans le Chapitre 6. On y détaille les deux versions du protocole expérimental suivi par les testeurs ainsi que les données brutes. Le paragraphe de définition du contexte et de définitions est partagé par les deux versions. Celles-ci diffèrent par l'ordre dans lequel sont testées les deux méthodes de choix de widgets. Les spécifications du dashboard de test et la taxonomie sont communes aux deux versions.

Contexte et définitions : Un dashboard est constitué de plusieurs visualisations juxtaposées. Il existe plusieurs centaines de visualisations, mises à disposition par des dizaines de bibliothèques de visualisations. Chaque bibliothèque propose son implémentation réutilisable d'un type de visualisation, appelé un widget, offrant parfois des possibilités différentes des autres bibliothèques. Certaines bibliothèques offrent des informations contextuelles sur les visualisations au passage de la souris, pour mettre en exergue une valeur par exemple.

Votre objectif est de définir, pour chaque visualisation, le ou les widgets qui répondent le mieux aux besoins exprimés. Il n'existe pas de dashboard absolu. L'expérience porte sur le processus de choix qui vous amène à un ensemble de visualisations qui vous conviennent dans un contexte donné.

B.1 Protocole Expérimental - Version A

Etape 1 - *Méthode «manuelle»*

L'assistant vous fourni : une liste de 73 widgets, tirés de 5 bibliothèques.

A faire : Familiarisez-vous seul pendant 5 minutes avec les widgets à disposition.

L'assistant vous fourni : les spécifications d'un dashboard pour le suivi de projets étudiants.

A faire : Choisissez le ou les widgets qui répondent le mieux aux besoins exprimés pour les deux premières visualisations. Si vous le pouvez, formulez à voix haute la raison de votre choix dans l'élimination ou la sélection d'un ensemble de widgets.

Etape 2 - *Méthode «taxonomie»*

L'assistant vous fourni : une taxonomie caractérisant les capacités des visualisations.

A faire : Familiarisez-vous pendant 5 minutes avec la définition des caractéristiques de la taxonomie. Vous pouvez poser des questions à l'assistant.

Pour associer un ensemble de widgets aux prochaines visualisation, vous pouvez choisir un ensemble de caractéristiques de la taxonomie, l'assistant de l'expérience vous indiquera les widgets satisfaisant ces caractéristiques. Vous pouvez itérer autant de fois que souhaité en sélectionnant ou désélectionnant des caractéristiques pour raffiner votre liste. À tout moment, vous pouvez vous décider sur un ensemble de widgets qui correspondent aux besoins exprimés.

Si vous le pouvez, formulez à voix haute la raison de votre choix dans la sélection ou la désélection d'une caractéristique.

Etape 3 : Quel processus de choix avez-vous préféré ? Pourquoi ? Utilisez la méthode «taxonomie» sur une visualisation que vous avez traité «manuellement». Comparez vos expériences.

B.2 Protocole Expérimental - Version B

Etape 1 - *Méthode «taxonomie»*

L'assistant vous fourni :

— Une liste de 73 widgets, tirés de 5 bibliothèques différentes

— Une taxonomie caractérisant les capacités des visualisations

A faire : Familiarisez-vous pendant 10 minutes avec les widgets à disposition et avec la définition des caractéristiques de la taxonomie.

L'assistant vous fourni : les spécifications d'un dashboard pour le suivi de projets étudiants.

A faire : Pour associer un ensemble de widgets aux deux premières visualisation, vous pouvez choisir un ensemble de caractéristiques de la taxonomie, l'assistant de l'expérience vous indiquera les widgets satisfaisant ces caractéristiques. Vous pouvez itérer autant de fois que souhaité en sélectionnant ou désélectionnant des caractéristiques pour raffiner votre liste. À tout moment, vous pouvez vous décider sur un ensemble de widgets qui correspondent aux besoins exprimés. Si vous le pouvez, formulez à voix haute la raison de votre choix dans la sélection ou la désélection d'une caractéristique.

Etape 2 - *Méthode «manuelle»*

L'assistant vous retire la taxonomie caractérisant les capacités des visualisations.

A faire : Vous devez choisir le ou les widgets qui répondent le mieux aux besoins exprimés pour les deux dernières visualisations. Si vous le pouvez, formulez à voix haute la raison de votre choix dans l'élimination ou la sélection d'un ensemble de widgets.

Etape 3 : Quel processus de choix avez-vous préféré ? Pourquoi ? Utilisez la méthode «taxonomie» sur une visualisation que vous avez traité «manuellement». Comparez vos expériences.

B.3 Spécification du dashboard de suivi d'une équipe de projet étudiant

Ce dashboard consiste en quatre visualisations. Vous travaillez sur une seule à la fois. L'équipe d'étudiants à suivre compte cinq membres ou moins.

Etape 1 - *Visualisations 1 et 2 :*

1. La première visualisation doit permettre de visualiser le nombre de tickets gérés par chaque membre de l'équipe, par rapport au volume total de tickets, afin de se faire une idée de la contribution relative de chaque étudiant.
2. La seconde visualisation montre, pour une période de temps donnée, le nombre moyen de jours pendant lesquels les tickets résolus à cette date sont restés ouverts, pour analyser la répartition de la dette technique et les valeurs aberrantes.

Etape 2 - *Visualisations 3 et 4 :*

3. Le troisième visualisation illustre l'évolution des statuts des tickets (To Do, In Progress, Done) en fonction du temps pour détecter des mauvaises habitudes de gestion des tickets et l'influence du temps.
4. La quatrième visualisation affiche le volume courant relatif des tickets affectés à chaque membre de l'équipe, pour chaque type de statut (To Do, In Progress, Done).

B.4 Taxonomie et définition des termes

Source : http ://datavizcatalogue.com/search.html
Comparaison : aide à montrer les différences ou les similarités entre les valeurs.
Proportion : utilise la taille ou des zones pour montrer les différences ou les similarités entre toutes les valeurs ou par rapport au tout.
Relation : met en valeur les relations entre les valeurs ou l'influence de l'un sur les autres.
Hiérarchie : montre comment les données sont organisées et/ou ordonnées entre elles.
Localisation : place les données dans un contexte géographique.
Probabilité : permet l'analyse probabiliste des valeurs affichées.
Part d'un tout : montre une ou des partie(s) par rapport au tout. Souvent utilisé pour montrer comment quelque chose est divisé.
Distribution : affiche les données en fréquence, comment les données sont réparties ou groupées au sein d'un intervalle.
Patterns : aide à révéler les schémas ou patterns dans les données pour trouver du sens.
Gamme : affiche les variations entre les limites maximum et minimum sur une échelle.
Temporisation : affiche les données en fonction d'une période de temps donnée.
Scalaire : autorise l'affichage de la valeur numérique brute.
Discrète : affiche spécifiquement des données discrètes, non continues.
Variation : met en valeur les variations de données par rapport aux précédentes et aux suivantes.
Extrêmes : facilite l'identification des valeurs maximum et/ou minimum.

B.5 Résultats bruts

Etape 1						Etape 2											
Visu 1		Visu 2				Visu 3						Visu 4					
Temps	Nb visus	Temps	Nb visus	delta temps 1 / temps 2	Temps	Nb visus OK	Nb visus tot	Ratio res / ok	Ratio res / itérations	Nb itérations	Temps moyen par itération	Temps	Nb visus OK	Nb visus tot	Ratio res / ok	Ratio res / itérations	Nb itérations	Temps moyen par itération	delta temps 4 / temps 3
0:02:38	18	0:02:24	10	9%	0:00:49	3	3	4%	100%	3	0:00:16	0:00:33	10	7	14%	70%	1	0:00:33	33%
0:07:32	9	0:08:45	19	-16%	0:02:46	11	7	15%	64%	4	0:00:42	0:01:56	24	13	33%	54%	3	0:00:39	30%
0:03:34	13	0:04:44	10	-33%	0:02:07	7	7	10%	100%	3	0:00:42	0:01:27	1	1	1%	100%	2	0:00:44	31%
0:05:54	2	0:06:28	7	-10%	0:08:26	7	6	10%	86%	6	0:01:24	0:03:46	9	8	12%	89%	3	0:01:15	55%
0:03:40	11	0:03:33	13	3%	0:00:56	7	5	10%	71%	1	0:00:56	0:00:46	7	7	10%	100%	1	0:00:46	18%
0:04:31	10	0:04:20	13	4%	0:02:55	5	4	7%	80%	4	0:00:44	0:02:01	12	9	16%	75%	2	0:01:00	31%
0:04:38	**10,5**	**0:05:02**	**12**	**-9%**	**0:03:00**	**6,67**	**5,3**	**9%**	**80%**	**3,5**	**0:00:51**	**0:01:45**	**10,5**	**7,5**	**14%**	**71%**	**2**	**0:00:52**	**42%**

FIGURE B.1 – Étapes 1 et 2 des participants au protocole A

| Etape 3 | | |

Etape 1																Etape 2								
Visu 1								Visu 2								Visu 3		Visu 4						
Temps visus	Nb visus	Nb visus OK	Ratio res / tot	Ratio ok / res	Nb itérations	Temps moyen par itération		Temps visus	Nb visus	Nb visus OK	Ratio res / tot	Ratio ok / res	Nb itérations	Temps moyen par itération	delta	Temps	Nb visus	Temps	Nb visus	delta				delta
																				temps 4 / temps 3				temps 2 / temps 1
0:01:25	9	9	12%	100%	4	0:00:21		0:00:06	12	10	16%	83%	1	0:00:06	93%	0:03:21	4	0:02:39	4	21%				
0:00:47	19	9	26%	47%	1	0:00:47		0:02:22	6	4	8%	67%	2	0:01:11	-202%	0:05:23	5	0:04:16	12	21%				
0:01:23	19	13	26%	68%	4	0:00:21		0:02:15	3	1	4%	33%	3	0:00:45	-63%	0:03:11	3	0:06:29	5	-104%				
0:02:29	5	3	7%	60%	3	0:00:50		0:02:17	2	2	3%	100%	3	0:00:46	8%	0:01:34	3	0:03:29	4	-122%				
0:02:46	11	9	15%	82%	4	0:00:42		0:01:53	10	7	14%	70%	5	0:00:23	32%	0:04:08	10	0:05:05	6	-23%				
0:03:10	9	9	12%	100%	4	0:00:47		0:00:42	3	3	4%	100%	1	0:00:42	78%	0:04:47	6	0:05:36	4	-17%				
0:02:00	**12**	**8,7**	**16%**	**72%**	**3,33**	**0:00:36**		**0:01:36**	**6**	**4,5**	**8%**	**75%**	**3**	**0:00:38**	**20%**	**0:03:44**	**5,17**	**0:04:36**	**5,83**	**-23%**				

FIGURE B.3 – Étapes 1 et 2 des participants au protocole B

Etape 3															Delta retours	
					Méthode manuelle -> taxonomie											
Retours méthode taxonomie	Retours méthode manuelle	N° visu	Temps	Nb visus res	Nb visus OK	Ratio res / tot	Ratio ok / res	Nb visu manue l / taxo	Nombre itérations	Temps moyen par itération	Ratio temps taxo / manuel	Retours méthode taxonomie	Retours méthode manuelle
+ choix plus facile + expression besoin - besoin reflexion	+ découvrir - lent - subjectif (peut se tromper)	3	0:00:52	5	5	7%	100%	25%	1	0:00:52	67,30%	+ rapide - apprentissage	- passe a côté inconnu
+ émerger des idées - maîtriser les termes	+ optimal	3	0:00:42	2	2	3%	100%	-83%	1	0:00:42	83,59%	+ rapide + bonne taxonomie	
- widget <-> taxonomie + découvrir + bonne taxonomie	+ trouver les connues	3	0:00:36	16	10	22%	63%	100%	2	0:00:18	90,75%	- besoin de rollback	
- devoir passer par des mots	+ sait déjà ce qu'il veut	3	0:00:34	5	3	7%	60%	-25%	1	0:00:34	83,73%		
+ résultats pertinents inconnus + se poser des question	+ systématique - fait des aller retours	3	0:01:44	17	12	23%	71%	100%	3	0:00:35	65,90%	~ catégorisation	
+ rapide + bon résultats - besoin de maîtriser les mots	+ simple - long - linéaire - perte intérêt	3	0:02:18	7	7	10%	100%	75%	3	0:00:46	58,93%	+ gain de temps + bon mots - travail	
		0:01:08	**8,67**	**6,50**		**12%**	**75%**	**32%**	**2**	**0:00:37**	**75,45%**		

FIGURE B.4 – Étapes 3 des participants au protocole B

BIBLIOGRAPHIE

[Acher 09] Mathieu Acher, Philippe Collet, Philippe Lahire & Robert B. France. *Composing Feature Models*. In Mark van den Brand, Dragan Gasevic & Jeff Gray, editeurs, SLE, volume 5969 of *Lecture Notes in Computer Science*, pages 62–81. Springer, 2009.

[Acher 11] Mathieu Acher. *Managing, multiple feature models : foundations, languages and applications*. PhD thesis, Nice, 2011.

[Acher 12] Mathieu Acher, Anthony Cleve, Gilles Perrouin, Patrick Heymans, Charles Vanbeneden, Philippe Collet & Philippe Lahire. *On extracting feature models from product descriptions*. In Ulrich W. Eisenecker, Sven Apel & Stefania Gnesi, editeurs, VaMoS, pages 45–54. ACM, 2012.

[Acher 13] Mathieu Acher, Philippe Collet, Philippe Lahire & Robert B. France. *FAMILIAR : A domain-specific language for large scale management of feature models*. Sci. Comput. Program., vol. 78, no. 6, pages 657–681, 2013.

[Allen 83] James F Allen. *Maintaining knowledge about temporal intervals*. Communications of the ACM, vol. 26, no. 11, pages 832–843, 1983.

[Arango 89] Guillermo Arango. *Domain analysis : from art form to engineering discipline*. In ACM Sigsoft software engineering notes, volume 14, pages 152–159. ACM, 1989.

[Arsanjani 04] Ali Arsanjani. *Service-oriented modeling and architecture*. IBM developer works, pages 1–15, 2004.

[Atkinson 08] Colin Atkinson, Dietmar Stoll & Philipp Bostan. *Orthographic software modeling : a practical approach to view-based development*. In International Conference on Evaluation of Novel Approaches to Software Engineering, pages 206–219. Springer, 2008.

[Atkinson 15] Colin Atkinson, Christian Tunjic & Torben Möller. *Fundamental Realization Strategies for Multi-view Specification Environments*. In Enterprise Distributed Object Computing Conference (EDOC), 2015 IEEE 19th International, pages 40–49. IEEE, 2015.

[Baniassad 04] Elisa Baniassad & Siobhan Clarke. *Theme : An approach for aspect-oriented analysis and design*. In Proceedings of the 26th International Conference on Software Engineering, pages 158–167. IEEE Computer Society, 2004.

[Barais 08] Olivier Barais, Jacques Klein, Benoit Baudry, Andrew Jackson & Siobhan Clarke. *Composing multi-view aspect models*. In Composition-Based Software Systems, 2008. ICCBSS 2008. Seventh International Conference on, pages 43–52. IEEE, 2008.

[Batory 05] D. Batory. *Feature Models, Grammars, and Propositional Formulas*. In Proc. of SPLC'2005, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.

[Benavides 10] David Benavides, Sergio Segura & Antonio Ruiz-Cortés. *Automated analysis of feature models 20 years later : A literature review*. Information Systems, vol. 35, no. 6, pages 615–636, 2010.

[Black 10] Andrew P Black, Oscar Nierstrasz, Stéphane Ducasse & Damien Pollet. Pharo by example. Lulu. com, 2010.

[Boardman 06] John Boardman & Brian Sauser. *System of Systems-the meaning of of*. In System of Systems Engineering, 2006 IEEE/SMC International Conference on, pages 6–pp. IEEE, 2006.

[Bona 11] Daniele Di Bona, Giuseppe Lo Re, Giovanni Aiello, Adriano Tamburo & Marco Alessi. *A methodology for graphical modeling of business rules*. In Computer Modeling and Simulation (EMS), 2011 Fifth UKSim European Symposium on, pages 102–106. IEEE, 2011.

[Botts 07] Mike Botts & Alexandre Robin. *OpenGIS Sensor Model Language (SensorML) Implementation Specification*. Technical report, OGC, July 2007.

[Boucké 10] Nelis Boucké, Danny Weyns & Tom Holvoet. *Composition of architectural models : Empirical analysis and language support*. Journal of Systems and Software, vol. 83, no. 11, pages 2108–2127, 2010.

[Boulanger 08] Frédéric Boulanger & Cécile Hardebolle. *Simulation of multi-formalism models with ModHel'X*. In Software Testing, Verification, and Validation, 2008 1st International Conference on, pages 318–327. IEEE, 2008.

[Boyer 11] Mr Jérôme Boyer & Hafedh Mili. Agile business rule development. Springer, 2011.

[Browne 09] Paul Browne. JBoss Drools Business Rules. Packt Publishing Ltd, 2009.

[Bruneliere 15] Hugo Bruneliere, Jokin Garcia, Philippe Desfray, Djamel Eddine Khelladi, Regina Hebig, Reda Bendraou & Jordi Cabot. *On Lightweight Metamodel Extension to Support Modeling Tools Agility*. In European Conference on Modelling Foundations and Applications, pages 62–74. Springer, 2015.

[Bruneton 04] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma & Jean-Bernard Stefani. *An open component model and its support in java*. In International Symposium on Component-based Software Engineering, pages 7–22. Springer, 2004.

[Bryant 15] Barrett Bryant, Jean-Marc Jézéquel, Ralf Lämmel, Marjan Mernik, Martin Schindler, Friedrich Steinmann, Juha-Pekka Tolvanen, Antonio Vallecillo & Markus Völter. *Globalized Domain Specific Language Engineering*. In Globalizing Domain-Specific Languages, pages 43–69. Springer, 2015.

[Buck 94] Joseph T Buck, Soonhoi Ha, Edward A Lee & David G Messerschmitt. *Ptolemy : A framework for simulating and prototyping heterogeneous systems*. 1994.

[Calvary 03] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon & Jean Vanderdonckt. *A Unifying Reference Framework for multi-target user interfaces.* Interacting with Computers, vol. 15, no. 3, pages 289–308, 2003.

[Cecchinel 14] Cyril Cecchinel, Matthieu Jimenez, Sébastien Mosser & Michel Riveill. *An architecture to support the collection of big data in the internet of things.* In Services (SERVICES), 2014 IEEE World Congress on, pages 442–449. IEEE, 2014.

[Cecchinel 16] Cyril Cecchinel, Sebastien Mosser & Philippe Collet. *Automated Deployement of Data Collection Policies over Heterogeneous Shared Sensing Infrastructures.* Asia-Pacific Software Engineering Conference (APSEC), 2016.

[Cheng 15] Betty HC Cheng, Benoit Combemale, Robert B France, Jean-Marc Jézéquel & Bernhard Rumpe. *On the Globalization of Domain-Specific Languages.* In Globalizing Domain-Specific Languages, pages 1–6. Springer, 2015.

[Clavreul 11] Mickael Clavreul, Sébastien Mosser, Mireille Blay-Fornarino & Robert B France. *Service–oriented Architecture Modeling : Bridging the Gap Between Structure and Behavior.* In Model Driven Engineering Languages and Systems, pages 289–303. Springer, 2011.

[Clements 01] P. Clements & L. M. Northrop. Software product lines : Practices and patterns. Addison-Wesley Professional, 2001.

[Czarnecki 04] Krzysztof Czarnecki, Simon Helsen & Ulrich Eisenecker. *Staged configuration using feature models.* In International Conference on Software Product Lines, pages 266–283. Springer, 2004.

[Daigneau 11] Robert Daigneau. *Service Design Patterns : Fundamental Design Solutions for SOAP.* WSDL and RESTful Web, 2011.

[de Souza 15] Leandro Oliveira de Souza, Pádraig O'Leary, Eduardo Santana de Almeida & Sílvio Romero de Lemos Meira. *Product derivation in practice.* Information and Software Technology, vol. 58, pages 319–337, 2015.

[Degueule 15] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais & Jean-Marc Jézéquel. *Melange : A meta-language for modular and reusable development of dsls.* In Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering, pages 25–36. ACM, 2015.

[Demeure 08] Alexandre Demeure, Gaëlle Calvary & Karin Coninx. *COMET(s), A Software Architecture Style and an Interactors Toolkit for Plastic User Interfaces.* In DSV-IS, pages 225–237, 2008.

[Di Natale 14] Marco Di Natale, Francesco Chirico, Andrea Sindico & Alberto Sangiovanni-Vincentelli. *An MDA approach for the generation of communication adapters integrating SW and FW components from Simulink.* In International Conference on Model Driven Engineering Languages and Systems, pages 353–369. Springer, 2014.

[Di Ruscio 10] Davide Di Ruscio, Ivano Malavolta, Henry Muccini, Patrizio Pelliccione & Alfonso Pierantonio. *Developing next generation ADLs through MDE*

techniques. In Software Engineering, 2010 ACM/IEEE 32nd International Conference on, volume 1, pages 85–94. IEEE, 2010.

[Dijkman 08] Remco M Dijkman, Dick AC Quartel & Marten J van Sinderen. *Consistency in Multi-Viewpoint Design of Enterprise Information Systems*. Information and Software Technology, vol. 50, no. 7, pages 737–752, 2008.

[Dijkstra 76] Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, Edsger Wybe Dijkstra, Etats-Unis Informaticien & Edsger Wybe Dijkstra. A discipline of programming, volume 1. prentice-hall Englewood Cliffs, 1976.

[DoDAF 03] DoDAF. *Department of Defense Architecture Framework*. Technical report, 2003.

[FEAF 99] FEAF. *Federal enterprise architecture framework version 1.1*. Technical report, 1999.

[Fenton 14] Norman Fenton & James Bieman. Software metrics : a rigorous and practical approach. CRC Press, 2014.

[Few 06] Stephen Few. Information Dashboard Design. O'Reilly, 2006.

[Finkelstein 92] Anthony Finkelstein, Jeff Kramer, Bashar Nuseibeh, Ludwik Finkelstein & Michael Goedicke. *Viewpoints : A framework for Integrating Multiple Perspectives in System Development*. International Journal of Software Engineering and Knowledge Engineering, vol. 2, no. 01, pages 31–57, 1992.

[Fischer 12] Klaus Fischer, Dima Panfilenko, Julian Krumeich, Marc Born & Philippe Desfray. *Viewpoint-Based Modeling-Towards Defining the Viewpoint Concept and Implications for Supporting Modeling Tools*. In EMISA, pages 123–136, 2012.

[Fleurey 07] Franck Fleurey, Benoit Baudry, Robert France & Sudipto Ghosh. *A generic approach for automatic model composition*. In International Conference on Model Driven Engineering Languages and Systems, pages 7–15. Springer, 2007.

[Forgy 82] Charles L Forgy. *Rete : A fast algorithm for the many pattern/many object pattern match problem*. Artificial intelligence, vol. 19, no. 1, pages 17–37, 1982.

[Fowler 02] Martin Fowler. Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Fowler 10] Martin Fowler. Domain-Specific Languages. Pearson Education, Boston, MA, USA, 2010.

[France 07] Robert France, Franck Fleurey, Raghu Reddy, Benoit Baudry & Sudipto Ghosh. *Providing support for model composition in metamodels*. In Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International, pages 253–253. IEEE, 2007.

[García Frey 12] Alfonso García Frey, Eric Ceret, Sophie Dupuy-Chessa, Gaëlle Calvary & Yoann Gabillon. *UsiComp : an extensible model-driven composer*. In EICS, pages 263–268, 2012.

[Golra 16a] Fahad R Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guerin & Christophe Guychard. *Addressing modularity for heterogeneous multi-model systems using model federation*. In Companion Proceedings of

the 15th International Conference on Modularity, pages 206–211. ACM, 2016.

[Golra 16b] Fahad R Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guerin & Christophe Guychard. *Continuous Requirements Engineering Using Model Federation.* In Requirements Engineering Conference (RE), 2016 IEEE 24th International, pages 347–352. IEEE, 2016.

[Gómez 12] Verónica Uquillas Gómez, Stéphane Ducasse & Theo D'Hondt. *Ring : a unifying meta-model and infrastructure for Smalltalk source code analysis tools.* Computer Languages, Systems & Structures, vol. 38, no. 1, pages 44–60, 2012.

[Gronback 09] Richard C Gronback. Eclipse modeling project : a domain-specific language (dsl) toolkit. Pearson Education, 2009.

[Große-Rhode 13] Martin Große-Rhode. Semantic integration of heterogeneous software specifications. Springer Science & Business Media, 2013.

[Haber 11] Aleksandar Haber, Holger Rendel, Bernhard Rumpe, Ina Schaefer & Frank Van Der Linden. *Hierarchical variability modeling for software architectures.* In Software Product Line Conference (SPLC), 2011 15th International, pages 150–159. IEEE, 2011.

[Haber 14] Arne Haber, Jan Oliver Ringert & Bernhard Rumpe. *Montiarc-architectural modeling of interactive distributed and cyber-physical systems.* arXiv preprint arXiv :1409.6578, 2014.

[Halle 01] Barbara Von Halle & G Ronald. Business rules applied : building better systems using the business rules approach. John Wiley & Sons, Inc., 2001.

[Harrand 16] Nicolas Harrand, Franck Fleurey, Brice Morin & Knut Eilif Husa. *ThingML : a language and code generation framework for heterogeneous targets.* In Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, pages 125–135. ACM, 2016.

[Herrington 03] Jack Herrington. Code generation in action. Manning Publications Co., 2003.

[Hili 15] Nicolas Hili, Yann Laurillau, Sophie Dupuy-Chessa & Gaëlle Calvary. *Innovative key features for mastering model complexity : flexilab, a multimodel editor illustrated on task modeling.* In Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, pages 234–237. ACM, 2015.

[Hilliard 12] Rich Hilliard, Ivano Malavolta, Henry Muccini & Patrizio Pelliccione. *On the composition and reuse of viewpoints across architecture frameworks.* In Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on, pages 131–140. IEEE, 2012.

[Höfferer 07] Peter Höfferer. *Achieving Business Process Model Interoperability Using Metamodels and Ontologies.* In ECIS, pages 1620–1631, 2007.

[Hohpe 04] Gregor Hohpe & Bobby Woolf. Enterprise integration patterns : Designing, building, and deploying messaging solutions. Addison-Wesley Professional, 2004.

[Iacob 12] M.E. Iacob, Dr. H. Jonkers, M.M. Lankhorst, E. Proper & Dr.ir. D.A.C. Quartel. *ArchiMate 2.0 Specification : The Open Group*. Van Haren Publishing, 2012.

[Jackson 90] Michael Jackson. *Some complexities in computerbased systems and their implications for system development*. In CompEuro'90. Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering, pages 344–351. IEEE, 1990.

[Jacobson 99] Ivar Jacobson, Grady Booch, James Rumbaugh, James Rumbaugh & Grady Booch. The unified software development process, volume 1. Addison-wesley Reading, 1999.

[Janota 08] Mikolas Janota. *Do SAT solvers make good configurators ?* In SPLC (2), pages 191–195, 2008.

[Jézéquel 12] Jean-Marc Jézéquel. *Model-driven engineering for software product lines*. ISRN Software Engineering, vol. 2012, 2012.

[Jézéquel 15] Jean-Marc Jézéquel, Benoit Combemale, Olivier Barais, Martin Monperrus & François Fouquet. *Mashup of metalanguages and its implementation in the kermeta language workbench*. Software & Systems Modeling, vol. 14, no. 2, pages 905–920, 2015.

[Josuttis 07] Nicolai M Josuttis. SOA in Practice : the Art of Distributed System Design. " O'Reilly Media, Inc.", 2007.

[Jouault 06] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev & Patrick Valduriez. *ATL : a QVT-like transformation language*. In Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, pages 719–720. ACM, 2006.

[Kang 90] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak & A Spencer Peterson. *Feature-oriented domain analysis (FODA) feasibility study*. Technical report, DTIC Document, 1990.

[Kang 98] K. Kang, S. Kim, J. Lee, K. Kim, E. Shin & M. Huh. *FORM : A feature-oriented reuse method with domain-specific reference architectures*. Annals of Software Engineering, vol. 5, no. 1, pages 143–168, 1998.

[Keating 03] Charles Keating, Ralph Rogers, Resit Unal, David Dryer, Andres Sousa-Poza, Robert Safford, William Peterson & Ghaith Rabadi. *System of systems engineering*. Engineering Management Journal, vol. 15, no. 3, pages 36–45, 2003.

[Kelly 08] Steven Kelly & Juha-Pekka Tolvanen. Domain-specific modeling : enabling full code generation. John Wiley & Sons, 2008.

[Kiczales 97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier & John Irwin. *Aspect-oriented programming*. ECOOP'97—Object-oriented programming, pages 220–242, 1997.

[Kienzle 09] Jörg Kienzle, Wisam Al Abed & Jacques Klein. *Aspect-oriented multi-view modeling*. In Proceedings of the 8th ACM international conference on Aspect-oriented software development, pages 87–98. ACM, 2009.

[Kienzle 16] Jörg Kienzle, Gunter Mussbacher, Omar Alam, Matthias Schöttle, Nicolas Belloir, Philippe Collet, Benoit Combemale, Julien DeAntoni, Jacques Klein & Bernhard Rumpe. *VCU : The Three Dimensions of Reuse*. In International Conference on Software Reuse, pages 122–137. Springer, 2016.

[Klein 12] Jacques Klein, Max E Kramer, Jim RH Steel, Brice Morin, Jörg Kienzle, Olivier Barais & Jean-Marc Jézéquel. *On the formalisation of geko : a generic aspect models weaver*. On the Formalisation of GeKo : a Generic Aspect Models Weaver (Tech Report), pages 1–15, 2012.

[Kolb 12] Jens Kolb, Manfred Reichert & Barbara Weber. *Using concurrent task trees for stakeholder-centered modeling and visualization of business processes*. In International Conference on Subject-Oriented Business Process Management, pages 237–251. Springer, 2012.

[Kolovos 06] Dimitrios S Kolovos, Richard F Paige & Fiona Polack. *Merging models with the epsilon merging language (EML)*. In MoDELS, volume 6, pages 215–229. Springer, 2006.

[Krahn 10] Holger Krahn, Bernhard Rumpe & Steven Völkel. *MontiCore : a framework for compositional development of domain specific languages*. International journal on software tools for technology transfer, vol. 12, no. 5, pages 353–372, 2010.

[Kramer 13] Max E Kramer, Erik Burger & Michael Langhammer. *View-centric engineering with synchronized heterogeneous models*. In Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling, page 5. ACM, 2013.

[Kurpjuweit 07] Stephan Kurpjuweit & Robert Winter. *Viewpoint-based Meta Model Engineering*. In EMISA, volume 143, page 2007, 2007.

[Larsen 15] Matias Ezequiel Vara Larsen, Julien Deantoni, Benoit Combemale & Frédéric Mallet. *A behavioral coordination operator language (BCOoL)*. 2015.

[Li 93] Wei Li & Sallie Henry. *Object-oriented metrics that predict maintainability*. Journal of systems and software, vol. 23, no. 2, pages 111–122, 1993.

[Liu 06] Haiyang Liu, San-Yih Hwang & Jaideep Srivastava. *PSRA : a data model for managing data in sensor networks*. In IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC'06), volume 1, pages 8–pp. IEEE, 2006.

[Logre 15] Ivan Logre, Sébastien Mosser & Michel Riveill. *Composition Challenges for Sensor Data Visualization*. In 14th International Conference on Modularity(MODULARITY 2015), poster, Companion, pages 25–26, Fort Collins, CO, USA, March 2015. ACM.

[Malavolta 13] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione & Antony Tang. *What industry needs from architectural languages : A survey*. IEEE Transactions on Software Engineering, vol. 39, no. 6, pages 869–891, 2013.

[Manyika 15] James Manyika. The internet of things : Mapping the value beyond the hype. 2015.

[Mei 02] Hong Mei, Feng Chen, Qianxiang Wang & Yaodong Feng. *ABC/ADL : An ADL supporting component composition*. In International Conference on Formal Engineering Methods, pages 38–47. Springer, 2002.

[Mendonca 09] Marcilio Mendonca, Andrzej Wasowski & Krzysztof Czarnecki. *SAT-based analysis of feature models is easy*. In Proceedings of the 13th International Software Product Line Conference, pages 231–240. Carnegie Mellon University, 2009.

[Morin 08] Brice Morin, Jacques Klein, Olivier Barais & Jean-Marc Jézéquel. *A generic weaver for supporting product lines*. In Proceedings of the 13th international workshop on Early Aspects, pages 11–18. ACM, 2008.

[Morin 17] Brice Morin, Nicolas Harrand & Franck Fleurey. *Model-Based Software Engineering to Tame the IoT Jungle*. IEEE Software, vol. 34, no. 1, pages 30–36, 2017.

[Mosser 13] Sébastien Mosser, Ivan Logre, Nicolas Ferry & Philippe Collet. *From Sensors to Visualization Dashboards : Need for Language Composition*. In Proceedings of the 2nd International Workshop on Globalization of Modeling Languages at MODELS, page 6, 2013.

[Muller 05] Pierre-Alain Muller, Franck Fleurey & Jean-Marc Jézéquel. *Weaving executability into object-oriented meta-languages*. In International Conference on Model Driven Engineering Languages and Systems, pages 264–278. Springer, 2005.

[Musset 06] Jonathan Musset, Étienne Juliot, Stéphane Lacrampe, William Piers, Cédric Brun, Laurent Goubet, Yvan Lussaud & Freddy Allilaire. *Acceleo user guide*. See also http ://acceleo. org/doc/obeo/en/acceleo-2.6-user-guide. pdf, vol. 2, 2006.

[Nalepa 09] Grzegorz J Nalepa & Maria A Mach. *Business rules design method for Business Process Management*. In Computer Science and Information Technology, 2009. IMCSIT'09. International Multiconference on, pages 165–170. IEEE, 2009.

[Nierstrasz 92] Oscar Nierstrasz, Simon Gibbs & Dennis Tsichritzis. *Component-oriented software development*. Communications of the ACM, vol. 35, no. 9, pages 160–165, 1992.

[Papazoglou 03] Mike P Papazoglou. *Service-oriented computing : Concepts, characteristics and directions*. In Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on, pages 3–12. IEEE, 2003.

[Papazoglou 08] Michael Papazoglou. Web services : principles and technology. Pearson Education, 2008.

[Parnas 72] David Lorge Parnas. *On the criteria to be used in decomposing systems into modules*. Communications of the ACM, vol. 15, no. 12, pages 1053–1058, 1972.

[Paternò 97] Fabio Paternò, Cristiano Mancini & Silvia Meniconi. *ConcurTaskTrees : A Diagrammatic Notation for Specifying Task Models*. In INTERACT, pages 362–369, 1997.

[Peltz 03] Chris Peltz. *Web services orchestration and choreography.* Computer, no. 10, pages 46–52, 2003.

[Perrouin 08] Gilles Perrouin, Jacques Klein, Nicolas Guelfi & Jean-Marc Jézéquel. *Reconciling automation and flexibility in product derivation.* In Software Product Line Conference, 2008. SPLC'08. 12th International, pages 339–348. IEEE, 2008.

[Perrouin 12] Gilles Perrouin, Gilles Vanwormhoudt, Brice Morin, Philippe Lahire, Olivier Barais & Jean-Marc Jézéquel. *Weaving variability into domain metamodels.* Software & Systems Modeling, vol. 11, no. 3, pages 361–383, 2012.

[Pohl 05] Klaus Pohl, Günter Böckle & Frank J. van der Linden. Software product line engineering : Foundations, principles and techniques. Springer-Verlag, 2005.

[Rabiser 10] Rick Rabiser, Paul Grünbacher & Deepak Dhungana. *Requirements for product derivation support : Results from a systematic literature review and an expert survey.* Information and Software Technology, vol. 52, no. 3, pages 324–346, 2010.

[Reddy 06] Y Raghu Reddy, Sudipto Ghosh, Robert B France, Greg Straw, James M Bieman, Nathan McEachen, Eunjee Song & Geri Georg. *Directives for composing aspect-oriented design class models.* In Transactions on Aspect-Oriented Software Development I, pages 75–105. Springer, 2006.

[Riebisch 03] Matthias Riebisch. *Towards a more precise definition of feature models.* Modelling Variability for Object-Oriented Product Lines, pages 64–76, 2003.

[Romero 09] José Raùl Romero, Juan Ignacio Jaen & Antonio Vallecillo. *Realizing correspondences in multi-viewpoint specifications.* In Enterprise Distributed Object Computing Conference, 2009. EDOC'09. IEEE International, pages 163–172. IEEE, 2009.

[Rossi 13] Gustavo Rossi. *Web Modeling Languages Strike Back.* IEEE Internet Computing, vol. 17, no. 4, pages 4–6, 2013.

[Rozanski 05] Nick Rozanski & Eóin Woods. Software Systems Architecture : Working With Stakeholders Using Viewpoints and Perspectives. Addison-Wesley Professional, 2005.

[Rozanski 11] Nick Rozanski & Eóin Woods. Software systems architecture : working with stakeholders using viewpoints and perspectives. Addison-Wesley, 2011.

[Schobbens 06] Pierre-Yves Schobbens, Patrick Heymans & Jean-Christophe Trigaux. *Feature diagrams : A survey and a formal semantics.* In 14th IEEE International Requirements Engineering Conference (RE'06), pages 139–148. IEEE, 2006.

[Schottle 13] Matthias Schottle & Jörg Kienzle. *On the challenges of composing multi-view models.* In GEMOC workshop, 2013.

[Schöttle 15] Matthias Schöttle, Nishanth Thimmegowda, Omar Alam, Jörg Kienzle & Gunter Mussbacher. *Feature modelling and traceability for concern-driven software development with TouchCORE.* In Companion Procee-

dings of the 14th International Conference on Modularity, pages 11–14. ACM, 2015.

[Schöttle 16] Matthias Schöttle, Omar Alam, Jörg Kienzle & Gunter Mussbacher. *On the modularization provided by concern-oriented reuse.* In Companion Proceedings of the 15th International Conference on Modularity, pages 184–189. ACM, 2016.

[Shah 03] Rahul C Shah, Sumit Roy, Sushant Jain & Waylon Brunette. *Data mules : Modeling and analysis of a three-tier architecture for sparse sensor networks.* Ad Hoc Networks, vol. 1, no. 2, pages 215–233, 2003.

[Shaw 03] Mary Shaw. *Writing good software engineering research papers.* In Software Engineering, 2003. Proceedings. 25th International Conference on, pages 726–736. IEEE, 2003.

[Sriplakich 08] Prawee Sriplakich, Xavier Blanc & Marie-Pierre Gervals. *Collaborative software engineering on large-scale models : requirements and experience in modelbus.* In Proceedings of the 2008 ACM symposium on Applied computing, pages 674–681. ACM, 2008.

[Stahl 06] Thomas Stahl, Markus Voelter & Krzysztof Czarnecki. Model-driven software development : technology, engineering, management. John Wiley & Sons, 2006.

[Steinberg 09] David Steinberg, Frank Budinsky, Marcelo Paternostro & Ed Merks. Emf : Eclipse modeling framework 2.0. Addison-Wesley Professional, 2nd edition, 2009.

[Svahnberg 05] M. Svahnberg, J. van Gurp & J. Bosch. *A taxonomy of variability realization techniques : Research Articles.* Softw. Pract. Exper., vol. 35, no. 8, pages 705–754, 2005.

[Tang 04] Antony Tang, Jun Han & Pin Chen. *A comparative analysis of architecture frameworks.* In Software Engineering Conference, 2004. 11th Asia-Pacific, pages 640–647. IEEE, 2004.

[Taylor 11] James Taylor. Decision management systems : a practical guide to using business rules and predictive analytics. Pearson Education, 2011.

[TOGAF 09] TOGAF. *The Open Group Architecture Framework Version 9.* Technical report, 2009.

[Vara Larsen 16] Matias Ezequiel Vara Larsen. *BCool : the Behavioral Coordination Operator Language.* Theses, Université de Nice Sophia Antipolis, April 2016.

[Voelter 12] Markus Voelter & Vaclav Pech. *Language modularity with the MPS language workbench.* In Software Engineering (ICSE), 2012 34th International Conference on, pages 1449–1450. IEEE, 2012.

[Wautelet 14] Yves Wautelet, Samedi Heng, Manuel Kolp & Isabelle Mirbel. *Unifying and extending user story models.* In International Conference on Advanced Information Systems Engineering, pages 211–225. Springer, 2014.

[Whittle 09] Jon Whittle, Praveen Jayaraman, Ahmed Elkhodary, Ana Moreira & João Araújo. *MATA : A unified approach for composing UML aspect models based on graph transformation.* In Transactions on aspect-oriented software development VI, pages 191–237. Springer, 2009.

[Wilkinson 12] Leland Wilkinson. The grammar of graphics. Springer, 2012.

[Wilson 12] Scott Wilson, Florian Daniel, Uwe Jugel & Stefano Soi. *Orchestrated User Interface Mashups Using W3C Widgets*. In Andreas Harth & Nora Koch, editeurs, Current Trends in Web Engineering, volume 7059 of *Lecture Notes in Computer Science*, pages 49–61. Springer Berlin Heidelberg, 2012.

[Withey 96] James Withey. *Investment Analysis of Software Assets for Product Lines.* Technical report, DTIC Document, 1996.

[Wynne 12] Matt Wynne & Aslak Hellesoy. The cucumber book : behaviour-driven development for testers and developers. Pragmatic Bookshelf, 2012.

[Zachman 03] John A. Zachman. *The Zachman Framework For Enterprise Architecture : A Primer for Enterprise Engineering and Manufacturing*. Zachman International, 2003.

[Živković 15] Srđan Živković & Dimitris Karagiannis. *Towards metamodelling-in-the-large : Interface-based composition for modular metamodel development*. In International Conference on Enterprise, Business-Process and Information Systems Modeling, pages 413–428. Springer, 2015.

RÉSUMÉ

La complexité croissante des systèmes logiciels pousse à la séparation des préoccupations, i.e., permettre aux concepteurs de considérer les sous-systèmes en isolation tout en conservant une vision globale du système. La variabilité des domaines impliqués dans la modélisation de ce type de système implique donc *(i)* un effort de composition des méta-modèles hétérogènes représentant ces domaines, *(ii)* une gestion de la cohérence inter-domaine des modèles produits en isolation et *(iii)* une gestion de la multiplicité des cibles atteignables dans l'espace des solution de chacun des domaines. Pour relever ces défis, nous présentons dans cette thèse une approche couvrant trois contributions : - une approche de composition respectant l'isolation des domaines en tirant profit des méthodes d'intégration des Architectures Orientées Services (SOA). Les méta-modèles sont encapsulés dans des services, exposant le comportement pertinent via une interface à destination des experts du domaine ; - un moteur de règles métiers qui assure la gestion des interactions entre domaines et permet de détecter les incohérences inter-domaine et de faire remonter aux experts les informations nécessaires à leur résolution ; - une modélisation de la variabilité des produits par caractérisation qui permet de concrétiser les sous-systèmes vers des artefacts concrets. Les contributions sont appliquées sur le cas de la visualisation de données en provenance de capteurs. Nous vérifions *(i)* le besoin de préservation de l'intégrité des méta-modèles sur un projet de conception de dashboards, puis nous quantifions *(ii)* le surcoût de l'encapsulation en service des domaines par rapport aux avantages apportés par l'intégration, *(iii)* l'impact de l'externalisation des interactions entre domaines, *(iv)* l'effort que les experts et l'intégrateur doivent fournir. Enfin, nous avons procédé à une expérience utilisateur afin de mesurer le gain lors de la concrétisation du système et son impact sur la satisfaction vis à vis des visualisations résultantes.

ABSTRACT

The growing complexity of software engineering leads to the use of separation of concerns, i.e. enable to consider manageable sub-systems while keeping an overview of the whole system. The domain variability involved in these system design imply : *(i)* to compose multiple heterogeneous metamodels dedicated to each domain, *(ii)* to handle cross-domain consistency of the model produced in isolation, *(iii)* and to tame the multiplicity of concrete artefact available in the solution space of each domain. To adress these challenges, we offer in this thesis an approach encompassing three contributions : - an isolation-compliant composition which benefits from Service Oriented Architecture (SOA) integration. Each domain metamodel is embedded in a service exposing the relevant behavior through an interface designed and used by domain experts ; - a business rule engine handling the interaction between domains and detecting cross-domain inconsistency to provide relevant feedback to resolve it ; - a feature-based characterization of the products variability allowing to concretize each sub-system toward concrete artifacts. The contributions are applied on the sensor data visualization use case. We validate *(i)* the need for domain isolation preservation on a dashboard design project, then we quantify *(ii)* the overhead of the service encapsulation, *(iii)* the impact of the externalization of domain interactions, *(iv)* the effort required from the experts and the integrator. Finally, we proceed to a user experiment to measure the gain during the concretization of a sub-system, and the impact on the user satisfaction on the resulting visualisation widgets.