FML : un langage de fédération de modèles pour l'interopérabilité sémantique de sources d'information hétérogènes

Sylvain Guérin

COLLEGES SCIENCES
BRETAGNE POUR L'INGENIEUR
LOIRE ET LE NUMERIQUE

ENSTA BRETAGNE

THESE DE DOCTORAT DE

L'ECOLE NATIONALE SUPERIEURE
DE TECHNIQUES AVANCEES BRETAGNE

ECOLE DOCTORALE N° 648
Sciences pour l'Ingénieur et le Numérique
Spécialité : *Informatique*

Par
Sylvain GUÉRIN

FML : un langage de fédération de modèles pour l'interopérabilité sémantique de sources d'information hétérogènes

Thèse présentée et soutenue à l'ENSTA Bretagne (Brest), le 30 novembre 2023
Unité de recherche : Laboratoire Lab-STICC, UMR 6285

Rapporteurs avant soutenance :

Jean-Marc Jézéquel	Professeur, Université de Rennes 1
Vincent Englebert	Professeur, Université de Namur, Belgique

Composition du Jury :

Présidente :	Régine Laleau	Professeure, Université Paris-Est
Examinateurs :	Jean-Marc Jézéquel	Professeur, Université de Rennes 1
	Vincent Englebert	Professeur, Université de Namur, Belgique
	Sophie Ebersold	Professeure, Université de Toulouse
	Pierre-Alain Müller	Professeur, Université de Haute-Alsace, Mulhouse-Colmar
Dir. de thèse :	Antoine Beugnard	Professeur, IMT Atlantique, Brest
Co-dir. de thèse :	Joël Champeau	Maître de Conférence, ENSTA Bretagne, Brest

Invité(s)

Hans Vangheluwe	Professeur, University of Antwerp, Belgique
Dominique Blouin	Ingénieur de recherche, Télécom Paris

Abstract et résumé

Abstract

Modelling is a universal practice at the root of human thought, for designing, understanding, calculating, imagining, analyzing and communicating. Model Driven Engineering (MDE) aims to systematize the use of models in all activities related to the whole life cycle of systems and software (development, deployment, integration, maintenance and evolution). From this context emerges the need to coherently manage multiple models (or information sources interpreted as models), where each model comes with its own autonomy and lifecycle. The core contribution of this thesis is founded on the formalization of an approach called "model federation", which aims to explicitly reify the links between federated models, while associating behaviour with those links. More specifically, we propose a modelling language called FML, which is both a language for conceptualization and reification of interpretation, but also features a referencing mechanism that enables federation links to point on heterogeneous data sources. FML allows both to react to the behaviour and evolution of federated models and to program behaviours acting on these models. We propose a tooling environment for the language within a software infrastructure called Openflexo. Finally, we validate the approach on four use cases, research prototypes and industrial experiments.

Résumé

La modélisation est une pratique universelle à la base de la pensée humaine, pour concevoir, comprendre, calculer, imaginer, analyser, communiquer. L'Ingénierie Dirigée par les Modèles (IDM) se propose de systématiser l'utilisation de modèles dans toutes les tâches liées au cycle de vie des systèmes et des logiciels (développement, déploiement, intégration, maintenance et évolution). De ce contexte émerge le besoin de gérer en cohérence de multiples modèles (ou des sources d'information interprétées comme des modèles), qui ont chacun leur autonomie et leur propre cycle de vie. Le coeur de la contribution de cette thèse repose sur la formalisation d'une approche que nous nommons "fédération de modèles" et qui se propose de réifier explicitement les liens entre les modèles fédérés, et de leur associer un comportement. Plus précisément, nous proposons un langage de modélisation appelé FML, qui est à la fois un langage permettant la conceptualisation et la réification de l'interprétation, mais qui est également doté d'un mécanisme de désignation qui permet l'établissement de liens de fédération vers des sources de données hétérogènes. FML permet, à la fois, de réagir aux comportements et évolutions des modèles fédérés et de programmer des comportements agissant sur ces modèles. Nous proposons un environnement outillé du langage au sein d'une infrastructure logicielle appelée Openflexo. Enfin, nous validons l'approche sur quatre cas d'utilisation, prototypes de recherche et expérimentations industrielles.

Remerciements

Une des grandes difficultés de la rédaction de ce mémoire a été le passage du "nous" au "je". En effet, ce travail est pour une grande part le résultat d'une équipe et d'un travail collectif, autour de l'aventure Openflexo et ce depuis maintenant plus de dix ans.

Tout au long de ce travail, et plus généralement tout au long de mon parcours professionnel, j'ai eu la chance de bénéficier d'un entourage enthousiaste et bienveillant. A toutes ces personnes, je tiens à exprimer toute ma gratitude et mon amitié. Merci notamment à Antoine et Joël d'avoir accepté de diriger cette thèse, et merci à tous ceux qui m'ont prodigué conseils, encouragements et relectures attentives, et notamment Fabien, Jean-Christophe, Salvador, Vincent et Luka.

Je tiens aussi à remercier toutes les personnes qui ont initié, participé, contribué au projet Openflexo. Je pense notamment à Dom qui a mis beaucoup d'énergie pour faire exister la structure, à Christophe, Vincent, Jean-Charles, Julien, et également à Ali, Gilles, Fahad, Ihab et Philippe, ainsi qu'aux nombreux contributeurs de la plate-forme logicielle. J'ai une pensée particulière pour Guillaume et les nombreux moments qu'on a passés à concevoir et organiser des abstractions logicielles.

Merci à Jean-Marc Jezequel et Vincent Englebert d'avoir accepté d'évaluer ce mémoire.

Merci également à tous mes collègues et amis de l'ENSTA Bretagne pour tous ces moments partagés autour d'une belle aventure humaine.

Evidemment un grand merci à ma famille et mes amis pour leurs soutiens, leurs encouragements et leurs présences à mes côtés durant toutes ces années. A vous Louise, Margot et Marianne : merci d'avoir été si patientes !

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction et contexte

1.1 Contexte

Il est aujourd'hui communément admis que le développement, systématique et profond, du numérique a bouleversé nos sociétés et constitue une modification sans précédent de notre environnement culturel. Tous les individus et toutes les organisations, de quelque nature qu'elles soient, ont à faire face à la gestion d'une quantité importante d'informations et de données. Les activités humaines doivent composer avec la gestion d'une grande quantité d'informations, de sources multiples et diverses, dans des formalismes et des formats d'encodage et de stockage hétérogènes. Ces informations sont parfois redondantes, parfois contradictoires, très souvent sujettes à une forme d'interprétation, ou tout au moins au choix d'un référentiel d'interprétation.

Dans ce contexte, le déploiement de la pensée humaine au cœur de cet environnement numérique foisonnant constitue un défi au cerveau humain, dès lors que l'on fait le constat que les capacités cognitives de nos cerveaux ne suivent pas une loi de Moore, et ne peuvent pas appréhender une trop grande quantité d'informations diverses.

La modélisation constitue un sujet d'étude intéressant si on le plonge dans cette perspective. La modélisation est une pratique universelle, à la base de la pensée humaine. Dans la vie de tous les jours, le langage est un modèle de notre monde ; il permet d'en parler, de le décrire, de raisonner dessus et d'en sortir en imaginant d'autres réalités, à venir ou non. En science, les mathématiques modélisent en proposant un autre langage pour décrire une autre réalité, celles des nombres, de la géométrie et plus généralement de la physique du monde au travers du calcul. En informatique, la démarche est identique. Pour modéliser, on utilise des langages pour décrire un calcul ou un système existant ou à venir. La modélisation est également à la base de la coopération entre êtres humains.

On parle évidemment beaucoup de modélisation en ingénierie, qui est un cadre naturel de la manipulation et du partage d'abstractions. D'une certaine façon, cette pratique n'est pas réservée aux ingénieurs, et nous formulons cette assertion que tout le monde modélise sans s'en apercevoir, comme *Monsieur Jourdain* [qui] *fait de la prose sans le savoir* [162]. En effet, dans un monde presque entièrement numérisé, nous manipulons tous, tous les jours, une grande quantité d'artefacts numériques, ne serait-ce qu'à travers des logiciels de traitement de textes, de tableurs, de bases de données cachées derrière des formulaires en ligne, etc. L'écriture est progressivement délaissée pour laisser place à la donnée nu-

mérique comme support généralisé de la conceptualisation, avec le double problème de l'implicite lié au sens qu'on donne à l'information, et son corollaire : le consensus à établir entre plusieurs êtres humains pour partager cette interprétation.

1.2 Ingénierie et modélisation

N'importe quelle ingénierie requiert d'analyser, de concevoir, de comprendre, de calculer, d'imaginer, de communiquer, etc. Toutes ces activités ont ceci en commun qu'elles mettent en jeu des modèles. L'Ingénierie Dirigée par les Modèles (IDM) témoigne de la généralisation des pratiques de modélisation. Les pratiques usuelles pour ces activités de modélisation impliquent l'utilisation d'un nombre limité de formalismes, standardisés ou spécifiques, et liés aux préoccupations de l'activité.

Parmi les modèles couramment considérés, on distingue les modèles prescriptifs des modèles descriptifs. Les modèles prescriptifs sont applicables à un contexte donné, pour un ensemble de préoccupations donné. Ils sont liés à un métamodèle, en général figé, qui permet d'exposer un référentiel partagé, en général outillé. Les modèles prescriptifs présentent l'avantage d'exprimer des connaissances non-ambiguës relativement au domaine considéré (mais parfois implicites [1]). Ils sont donc facilement opérationnalisables et exploitables par la machine, mais ont l'inconvénient d'une expressivité limitée, ou alors au prix d'un détournement de l'outillage associé, et donc d'un risque de mauvaise interprétation, par la machine ou par un être humain. Les modèles prescriptifs servent à concevoir, et sont en général utilisés pendant les phases d'analyse, de conception, de mise en oeuvre et d'évaluation. Les modèles descriptifs rendent compte d'une réalité existante. Ces modèles, tels que les ontologies ou les documents textuels, sont moins formels, moins contraints, ouverts et extensibles. Leur avantage est leur relative exhaustivité. Leur inconvénient est leur moindre capacité à être traité automatiquement, leur sensibilité à l'interprétation et le risque d'ambiguïté associé. Dans le monde de la conception, les modèles descriptifs servent généralement à documenter et qualifier les systèmes existants, et sont utilisés par exemple dans les phases de rétroingénierie.

Dans la pratique, beaucoup d'autres artefacts informatique jouent (ou pourraient jouer) le rôle de modèle : documents textuels, traitement de textes, tableurs, base de données, présentations, diagrammes. Pour être considérés comme des modèles, ces données requièrent une **interprétation** et véhiculent donc un certain **implicite**. Ces usages, s'ils sont fréquents et réels, ne sont généralement pas outillés [2], et sont à la charge d'un cerveau humain.

Un modèle est conçu pour une préoccupation donnée (ou un petit nombre de préoccupations). Aucune ingénierie n'a à gérer qu'une seule préoccupation. La conséquence est donc de devoir assurer la **gestion de modèles multiples** (avec sans doute un certain niveau de recouvrement). Par ailleurs, de nombreuses ingénieries font intervenir et collaborer de **nombreuses expertises métiers**, ce qui implique de faire interopérer de nombreuses sources de données, modèles et outils dans des formalismes divers. Enfin, tous ces modèles et toutes ces sources de données ne sont pas des artefacts figés. Ils **évoluent** tous dans le cadre des différents utilisateurs et outils impliqués dans le cycle de vie de chacun des domaines métiers.

1. Un plan d'architecture d'immeuble est prescriptif bien que souvent il n'y ait pas de métamodèle explicite.
2. Ils le sont parfois à la faveur d'une opération de copier/coller !

Ces réflexions amènent certaines questions : *Comment mettre en relation (et si possible gérer de façon consistante) de multiples modèles ou sources d'information ? Comment gérer l'hétérogénéité de ces modèles ou de ces sources d'information ? Comment donner du sens à la donnée ? Quelles sont les conditions d'interprétation d'une source d'information comme un modèle ? Comment réifier cette interprétation pour la partager avec d'autres ? Comment faire en sorte que des modèles connectés puissent évoluer de manière autonome ?*

1.3 Problématique

L'exposé du contexte nous conduit à travailler sur une problématique de **gestion coordonnée et consistante d'un ensemble de sources d'information dynamiques et autonomes**, en nous concentrant sur les aspects d'**interopérabilité sémantique**. Dans la littérature qui traite du domaine de la modélisation et plus particulièrement des approches multi-modèles, trois grandes techniques sont identifiées (section 2.3.4). Parmi ces dernières, nous identifions la **fédération de modèles** en tant qu'approche pertinente au regard de notre problématique, pour ses capacités à réifier des liens entre des modèles autonomes. Cette fédération doit pouvoir s'opérer relativement à des représentations variées, à différents niveaux d'abstraction, à des sources d'information et de formats hétérogènes, et en combinant des paradigmes divers. Dans ce contexte, fédérer consiste à pouvoir manipuler de multiples sources d'information qui sont structurées, organisées et qui évoluent de façon autonome, avec pour chaque source d'information, des préoccupations initiales *a priori* sans lien avec celles de la fédération.

Plus particulièrement, nos objectifs de recherche s'articulent autour de trois grands enjeux :

— la problématique de l'**interprétation** de la donnée (enjeu E1),

— la prise en compte de l'**hétérogénéité** (enjeu E2),

— la nécessité de manipuler des **modèles vivants** (enjeu E3).

À ces trois enjeux s'ajoutent des contraintes et des exigences supplémentaires, et notamment le respect de l'autonomie du cycle de vie des sources d'informations fédérées, la gestion d'une connectivité intermittente, le principe de non-intrusivité et la gestion de la source de vérité.

1.4 Approche mise en œuvre

L'objectif général des travaux présentés dans le cadre de ce manuscrit visent à proposer une **approche générique pour la mise en œuvre de la « fédération de modèles »**, que l'on définit ici comme une solution d'assemblage de modèles permettant **l'interopérabilité sémantique de sources d'information hétérogènes et autonomes**.

Notre première question de recherche (enjeu E1) porte sur la problématique de donner un sens explicite à l'information (pour être capable de le partager avec d'autres). À rebours des approches qui proposent l'établissement de (méta-)modèles pivot (ou qui proposent de s'aligner sur des standards existants), nous proposons un mécanisme de **réification** d'une interprétation **explicite** et **contextuelle** de la donnée. Cette réification permet une interprétation locale non ambiguë (qui permet le traitement automatique), et contextualisée

(c'est-à-dire que cette interprétation sera faite dans un contexte donné). Cette interprétation est réifiée au sein d'une couche conceptuelle, qui permet d'externaliser le sens donné à l'information, pour construire des modèles *ad hoc*, avec un découplage total entre la source d'information et sa sémantique contextuelle.

Ceci nous amène à notre deuxième objectif de recherche (enjeu E2), lié à l'hétérogénéité des modèles. Dans ce cadre, nous proposons un **mécanisme de désignation** qui permet l'établissement de liens entre une couche conceptuelle qui porte la sémantique, et des espaces technologiques arbitraires. La capacité à établir ces liens et à référencer l'information en traversant les espaces technologiques offre par ailleurs des possibilités très intéressantes en terme de traçabilité. Un autre aspect important de ces liens réside dans le fait qu'ils ne sont pas que structurels, mais qu'ils sont associés aussi à des **aspects comportementaux** liés à l'évolution de la donnée désignée (enjeu E3).

Au travers du langage de modélisation FML (*Federation Modelling Language*), nous proposons **une approche unifiée de l'environnement de modélisation**, incluant toutes les sources d'informations (existantes ou en cours de conception), la conceptualisation et l'outillage associé. Nous proposons un partitionnement de l'environnement de modélisation en espace technique, espace conceptuel, et espace d'outillage, comme illustré sur la figure 1.1. L'espace conceptuel permet de conceptualiser à partir des sources d'informations et des modèles définis dans l'espace technique. Un aspect important réside dans le fait que cette conceptualisation n'est pas que structurelle mais permet également la définition de **comportements**, qui agissent sur les modèles, et réagit à l'évolution de ces modèles (enjeu E3). Cette conceptualisation permet ensuite de «nourrir» l'espace d'outillage qui l'exploite en tant que source sémantique de données.

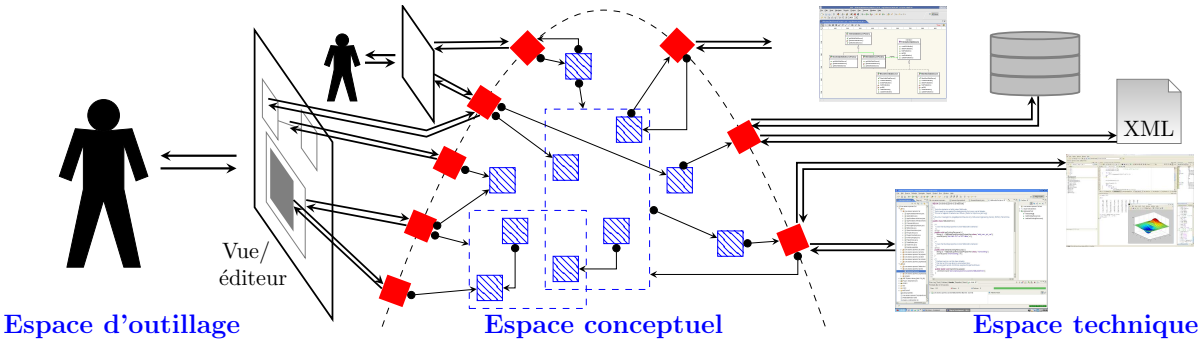

FIGURE 1.1 – Partitionnement de l'espace de modélisation

Un dernier axe de notre approche s'attache à la pragmatique liée à l'utilisation du langage FML. La méthologie générale consiste à construire des modèles à partir de sources d'information diverses, ou à partir d'autres modèles. Nous nous proposons de considérer ces modèles comme des composants génériques, construits avec une préoccupation donnée, exposés sous la forme de contrats, et dotés de comportements. L'approche est **compositionnelle**, en ce sens qu'elle permet de construire des composants qui vont être assemblés selon des principes de modularité et de séparation des préoccupations. Notre proposition s'appuie plus généralement sur les principes de la programmation orientée objet (indépendance conceptuelle, réutilisation, composition, encapsulation, héritage, polymorphisme, gestion de la contenance, abstractions - concepts, propriétés, comportements - surcharge, redéfinition).

1.5 Contributions

Formalisation de l'approche "fédération de modèles"

À partir de l'état de l'art et de l'exposé de nos objectifs de recherche, une première contribution porte sur l'explicitation et la formalisation de l'approche **fédération de modèles** (chapitre 3). Nous avons positionné cette approche par rapport à de multiples vocables et domaines de recherche, notamment par la proposition d'un diagramme de caractéristiques des approches "fédératives". Cette contribution est complétée par la proposition du *framework* MF^2 qui précise la sémantique opérationnelle de l'approche.

Le langage FML

La deuxième contribution constitue le cœur scientifique de cette thèse. Elle repose sur la proposition du langage FML (chapitre 4), en tant que langage de métamodélisation permettant de réifier des liens de fédération et de leur associer un comportement. FML est à la fois un langage permettant la conceptualisation et la réification de l'interprétation, mais il est également doté d'un mécanisme de désignation qui permet l'établissement de liens de fédération vers des sources de données hétérogènes. C'est en outre un langage de modélisation qui définit et embarque son propre comportement. Nous décrivons le langage et donnons un aperçu de sa syntaxe textuelle et de sa sémantique opérationnelle. Nous montrons comment cette proposition constitue une réponse à nos objectifs de recherche.

L'infrastructure logicielle Openflexo

Une troisième contribution concerne la proposition de l'infrastructure logicielle Openflexo (chapitre 5), qui se présente comme une bibliothèque de composants logiciels qui permettent la mise en œuvre du langage FML et de la fédération de modèles. Cette infrastructure implante un environnement de développement intégré de FML, ainsi qu'un moteur d'exécution et un grand nombre d'adaptateurs technologiques, permettant de manipuler des sources de données dans des technologies diverses. Ce *framework* s'accompagne d'une méthodologie outillée permettant son utilisation pour produire et instancier des logiciels métiers.

Validation sur différents cas d'étude

Le langage FML et l'infrastructure Openflexo ont été utilisés dans le contexte de différents cas d'utilisation, allant du prototype de recherche au déploiement dans un cadre industriel (chapitre 6). Nous avons pu notamment valider l'utilisabilité du langage FML au regard d'approches méthodologiques diverses. Nous avons notamment validé sur plusieurs projets les aspects abstraction/généricité, modularité et réutilisation du langage et son intérêt dans le cadre du développement de solutions d'intégration de travaux et de technologies hétérogènes.

La modélisation libre (*Free Modelling)*

Un dernier cas d'utilisation traite d'une proposition scientifique que nous avons appelée la modélisation libre (section 6.4), qui se propose de traiter le typage de modèles d'un point de vue ontologique plutôt que linguistique. Le démonstrateur développé permet d'assister des utilisateurs dans la capture et l'émergence d'un modèle métier à partir d'exemples et de diagrammes.

1.6 Contexte des travaux

Les travaux présentés dans le cadre de cette thèse s'appuient sur une période de plus de dix années de collaborations entre des partenaires industriels, des équipes de recherche, et une société privée (la SCIC Openflexo) qui a un temps exploité commercialement l'infrastructure logicielle Openflexo.

Les équipes de recherche actuellement impliquées dans le projet Openflexo concernent principalement l'ENSTA Bretagne et l'IMT Atlantique, notamment au travers de l'équipe P4S [3] du laboratoire LabSTICC [4]. Cette équipe comprend des chercheurs issus de l'ENSTA Bretagne, de l'IMT Atlantique et de l'Université de Bretagne Occidentale (UBO). De nombreux projets de recherche avec des partenaires scientifiques (LACL, Telecom Sud Paris, Telecom Paris, Université de Rennes, IRT SystemX, etc.) et industriels (ClearSy, Telindus, Thalès, Airbus, DGA, ESA, etc.) ont permis de nourrir la réflexion sur les choix de conception et de consolider l'architecture logicielle de l'infrastructure. Citons notamment le projet ANR *Formose*, le projet *SSE4Space* pour l'ESA avec Telindus, le projet *OneWay* avec Airbus, et d'autres projets nationaux pour la DGA.

La Société Coopérative d'Intérêt Collectif (SCIC) Openflexo (2014-2019) témoigne quant à elle d'une expérience originale. Cette entreprise a bénéficié d'un statut juridique très particulier, à mi-chemin entre le monde de l'entreprise et celui des associations. Son objet était une exploitation commerciale de l'infrastructure logicielle Openflexo dans les valeurs de l'ESS (Economie Sociale et Solidaire). Plus qu'une simple licence open-source, le modèle économique sous-jacent reposait sur une grande coopération entre les différents acteurs de la chaîne économique (salariés, scientifiques, clients et fournisseurs).

Au fil des années, le projet Openflexo a bénéficié de cet écosystème original entre le monde de l'entreprise et le monde de la recherche, qui se nourrissent l'un-l'autre. L'infrastructure logicielle s'est construite d'une part autour de problématiques scientifiques (intégration continue de la recherche en IDM), et d'autre part autour de besoins pragmatiques, techniques et industriels (construction de solutions opérationnelles pour des clients). Cette infrastructure représente aujourd'hui une importante base de code (autour de 800 000 lignes de code Java).

Cette thèse s'articule autour des problématiques scientifiques qui ont émergé au cours de tous ces travaux.

3. *Processes for Safe and Secure Software and Systems*, cf https://p4s.enstb.org/
4. UMR 6285, https://labsticc.fr/fr

1.7 Structure du manuscrit

Ce manuscrit est structuré en 7 principaux chapitres.

Après cette partie introductive (chapitre 1), le chapitre 2 dresse un état de l'art du domaine qui nous mène de la connaissance à la modélisation et aux problématiques liées à l'interopérabilité sémantique de modèles hétérogènes et autonomes. Cet état de l'art sera aussi l'occasion d'une mise en perspective de ces aspects dans une vision sémiotique du traitement de l'information.

Le chapitre 3 précise la problématique et les objectifs de recherche. Cette problématique est illustrée par cinq scénarios de modélisation typiques qui couvrent un certain nombre de pratiques courantes de l'IDM, et qui vont servir de fil conducteur à la présentation de nos travaux. Nous proposons une formalisation au travers d'un métamodèle pour l'approche **fédération de modèles**, avant de montrer sa mise en œuvre sur les cinq scénarios de référence. Nous concluons en évoquant le framework MF^2 qui propose une sémantique opérationnelle pour cette approche. MF^2 est présenté en détail dans l'annexe A.

Le chapitre 4 constitue le cœur de notre contribution et présente le langage FML, ses concepts et un aperçu de sa syntaxe textuelle. Les annexes B et C donnent respectivement des définitions formelles du langage et précisent sa sémantique. Ce chapitre se poursuit par la présentation de la mise en œuvre du langage sur les cinq scénarios prototypiques.

Le chapitre 5 décrit l'infrastructure logicielle Openflexo, en tant qu'implantation Java open-source du langage FML. Son architecture et ses composants logiciels sont précisés, ainsi que les principes de construction d'une application avec Openflexo.

Le chapitre 6 décrit les quatre cas d'étude évoqués dans les contributions. Le premier cas d'utilisation (le projet *Formose*, section 6.1) porte sur un projet de recherche en ingénierie des exigences, impliquant un consortium de partenaires industriels et académiques. Le deuxième exemple (*MULTI Process Challenge*, section 6.2) a pour cadre un *challenge* scientifique. Il a permis de confronter le langage FML à la modélisation multi-niveaux et au développement d'un prototype outillant la solution. Le troisième cas d'utilisation (*SSE4Space*, section 6.3) a permis la validation effective de l'approche dans un contexte industriel. La contribution concernant la modélisation libre, pour la capture et l'émergence d'un modèle métier à partir d'exemples et de diagrammes, est traitée dans la section 6.4.

Le chapitre 7 conclut ce manuscrit en résumant les contributions et présente les perspectives de nos travaux.

Chapitre 2

Etat de l'art

Le titre de cette thèse évoque la notion de *modèle*. Le sens de ce mot très courant renvoie cependant à de nombreuses significations et usages, tant dans la langue française que dans différents domaines scientifiques et technologiques. Si l'abstraction fonde la nature même de la pensée humaine, et conduit *in fine* à la modélisation, il n'est peut-être pas inutile de s'intéresser à cette construction intellectuelle bien en amont. En effet, l'observation du monde, l'acquisition de connaissances et la coopération entre êtres humains précèdent la construction de sens, et justifient la conception de modèles, en tant que base conceptuelle à partir de laquelle l'activité humaine peut se déployer.

Nous développerons cet argumentaire dans la section 2.1, en proposant une vision sémiotique de l'interprétation de données numériques. Nous clarifierons dans un second temps et dans la section 2.2 la notion de modèle, avant de la mettre en perspective dans le contexte de l'*Ingénierie Dirigée par les Modèles* (IDM). La section 2.3 introduit la problématique levée par l'existence de nombreux modèles et la nécessité d'établir des liens d'interopérabilité sémantique entre eux. À cette occasion, nous justifierons ce besoin et les motivations liées avant de nous pencher sur les principales approches existantes. Nous nous attarderons dans la section 2.4 sur les approches dites "fédératives", qui se proposent d'établir un couplage faible entre des modèles autonomes avec des liens de fédération explicitement réifiés et nous proposerons un tour d'horizon des approches existantes. Une dernière section 2.5 sera consacrée aux approches de métamodélisation, et nous conclurons sur l'exposé du contexte et de la problématique générale de nos travaux.

Sommaire

2.1 De la connaissance à la modélisation

2.1.1 Le partage de connaissances à la base de la coopération humaine

Dialoguer, communiquer, décrire ou élaborer une théorie, imaginer, construire, concevoir un mécanisme et l'expliquer, calculer, raisonner : autant d'activités qui sont pratiquées en permanence par tous les êtres humains. Au centre de ces activités intellectuelles se pose la question de la connaissance, indispensable à la production intellectuelle.

Toute activité de production intellectuelle impliquant plusieurs individus requiert de s'appuyer sur un certain nombre de connaissances communes, reconnues et légitimées par l'ensemble des parties prenantes à la dite-activité [37].

Cette somme de connaissances qui doivent être partagées comprend notamment :

— des faits, des concepts, des règles générales (théories, axiomes, théorèmes) ;

— des manières de manipuler ces faits et ces concepts, pour les assembler, pour les manipuler ou conduire des raisonnements ;

— des moyens permettant d'échanger de l'information et de communiquer, ce qui implique la définition de symboles (mots, acronymes, idéogrammes, dessins, représentations, etc.) et plus généralement de toute la symbolique associée. La notion de langage s'appuie sur cette symbolique.

Pour que le processus d'activité collectif fonctionne bien, il faut en outre que tous les participants partagent des objectifs communs, et un certain nombre de valeurs qui permettent de caractériser les faits et leur manipulation [187].

La base de connaissances ainsi définie comprend nécessairement des éléments explicitement définis (en référence à des productions concrètes réifiées : dictionnaires, encyclopédies, glossaires, documents techniques et documents de références, lois, articles et traités scientifiques, théories, etc.), mais aussi de beaucoup d'éléments implicites. Ces connaissances implicites peuvent être soit propres à des éléments culturels partagés par les différents participants (origine, langue, nationalité, entreprise), soit elles sont issues du domaine d'étude considéré. Elles peuvent être en outre héritées indirectement d'autres connaissances - explicites ou implicites -.

2.1.2 De la connaissance au modèle : modéliser et interpréter

Le développement de l'activité humaine est intrinsèquement lié à l'usage d'outils. L'activité intellectuelle ne constitue pas une exception, et ce sont les modèles et les langages qui sont les principaux instruments méthodologiques au service de la connaissance, et qui participent à sa recherche, à sa découverte, sa validation, sa communication, son apprentissage et son évolution. La modélisation est en effet une pratique universelle. Dans la vie de tous les jours, elle se manifeste par l'usage du langage qui est un modèle de notre monde et qui nous permet d'en parler, de le décrire, de raisonner dessus et d'en sortir en imaginant d'autres réalités, à venir ou non. Le monde des mathématiques et plus généralement celui des sciences offrent d'autres langages qui permettent de décrire et de manipuler d'autres réalités : celle des nombres, de la géométrie et plus généralement de la physique du monde au travers de la manipulation de concepts et du calcul sur ces abstractions.

Guy Caplat définit les modèles comme des constructions élaborées par l'humain à des fins particulières [45]. Nous reviendrons plus loin et en détail sur la notion de modèles, qui prennent en pratique des formes très différentes (une carte routière, une formule mathématique, un texte, un diagramme, un programme informatique), mais qui ont en commun le fait d'être porteurs d'information et de véhiculer de la connaissance. La notion de sujet d'étude peut exister préalablement au modèle qui a pour objectif de le décrire, ou bien au contraire de spécifier ce même sujet d'étude (par exemple un système). Dans un cas comme dans l'autre, le modèle est une "représentation" dépositaire d'une signification, un moyen d'échanger et de partager des connaissances [45].

Imaginer et construire un objet ou un système, élaborer une théorie, comprendre un phénomène, calculer, analyser, tester, communiquer, anticiper : à chaque fois il s'agit de construire des modèles. Inversement, nous utilisons des modèles dans notre vie de tous les jours pour consulter des informations, utiliser des appareils technologiques, collaborer au sein d'une équipe, pour comprendre et élaborer un sens.

Pour tout ce qui concerne la sphère intellectuelle, les interactions entre êtres humains mettent forcément en jeu deux activités duales et centrées autour des modèles : la première consiste à matérialiser un point de vue, à convertir une idée en un objet tangible, la seconde est le mécanisme inverse de traduction d'une représentation physique et codifée en une représentation "mentale" chez le receveur de la connaissance. La figure 2.1 illustre l'activité de modélisation et son activité d'interprétation duale mise en œuvre dans le cadre d'une interaction entre deux humains.

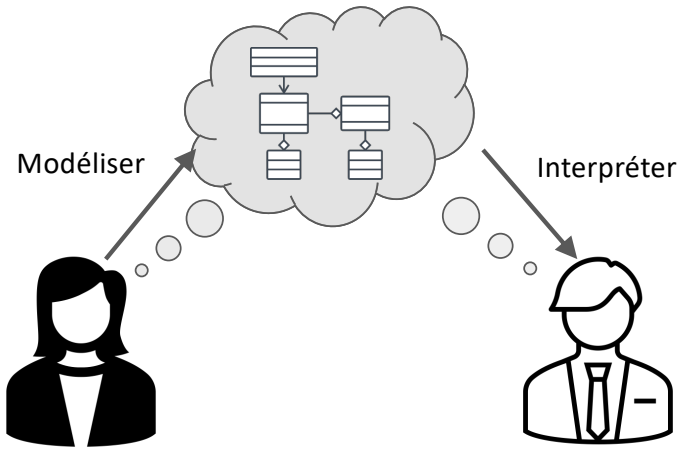

FIGURE 2.1 – Modéliser et interpréter des modèles

L'interprétation d'un modèle correspond à l'élaboration d'un sens, c'est-à-dire en la construction d'une image mentale signifiante à partir d'une représentation concrète. Ceci pose le problème de savoir si l'interprétation d'un modèle est fidèle (ou au moins compatible) à l'intention de la personne qui a conçu ce même modèle. En effet, pour que la coopération entre différents être humains autour d'une activité de production intellectuelle fonctionne, il faut nécessairement que tous partagent ces connaissances. Évidemment, il est illusoire d'imaginer que la catégorisation de toutes ces connaissances soit parfaitement alignée pour toutes les personnes impliquées, en conséquence il existe une nécessaire distance (tolérance) qui permet à deux personnes de se mettre d'accord sur un fait, sans que chacun ne partage l'exacte vision de l'autre. Cette tolérance est nécessaire pour collaborer.

Cependant, si elle est trop grande, elle peut traduire un défaut d'interprétation ou une incompréhension. À terme, des désaccords majeurs peuvent émerger d'une accumulation de ces non-alignements.

2.1.3 Sémiotique et numérique

Notions de sémiotique

Parce qu'on ne peut pas tout désigner du doigt, l'évocation d'une chose ou d'une catégorie de choses nécessite forcément de passer par un substitut, à fortiori si l'on parle d'une notion très abstraite. La sémiotique (appelée encore sémiologie) se propose d'appeler ce substitut *symbole* (plus généralement on parlera de signes). Cette science, définie comme l'étude théorique des signes et de leur articulation dans la pensée, étudie les processus de signification, la combinaison des signes et les relations entre les signes et les usagers, c'est-à-dire la production, la codification et la communication de signes [45].

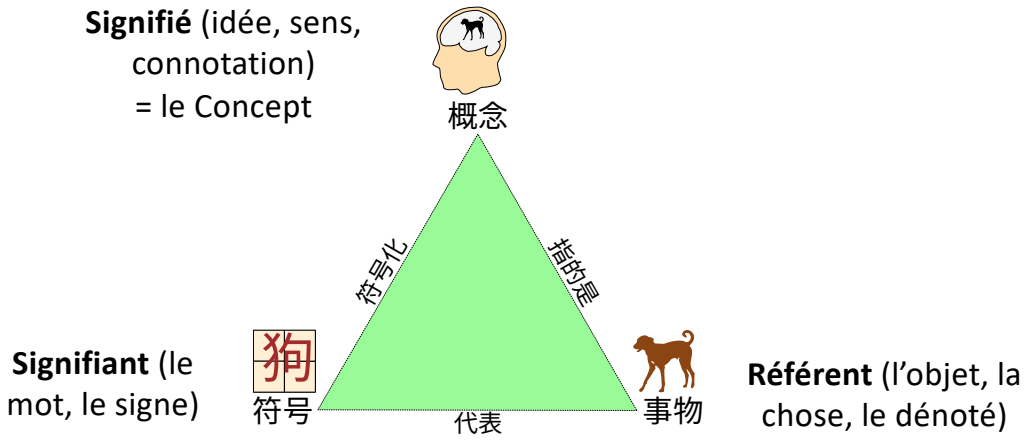

FIGURE 2.2 – Le triangle sémiotique d'Odgen et Richards

Ferdinand de Saussure, considéré comme un des deux pères de la sémiologie moderne avec Charles Sanders Peirce, définit le premier la notion de signe linguistique, et introduit la distinction entre signifié et signifiant dans son *Cours de linguistique générale* [56]. Il propose une définition du signe linguistique qui met en relation un concept (le signifié) et une forme sonore - "image acoustique" - (le signifiant).

Le triangle sémiotique d'Ogden et Richards, bien connu en linguistique, permet de distinguer le mot, la chose et le concept. La figure 2.2 illustre une version chinoise présentant l'idéogramme, le chien, et l'idée du chien.

— **Le mot** (*le symbole*) constitue le *signifiant* tel que défini par Saussure.

— **La chose** (*le référent* ou encore *le dénoté*) constitue l'objet physique ou l'action auquel il renvoit.

— **Le concept** (*la pensée*) est l'idée, le sens, la connotation, c'est-à-dire le *signifié* de Saussure.

Charles Sanders Peirce, sémiologue et philosophe américain, a quant à lui élaboré une théorie sémiotique à la fois générale, triadique et pragmatique [170, 70]. Son principe de théorie générale envisage à la fois la vie émotionnelle, pratique et intellectuelle. Sa théorie triadique du signe met en relation trois termes, le signe (ou *representamen*), l'objet et l'interprétant (une représentation mentale de la relation entre le *representamen* et l'objet). Le *representamen* est premier (une pure possibilité de signifier), l'objet est second (ce qui existe et dont on parle), mais ce processus s'effectue en vertu d'un interprétant (un troisième qui dynamise la relation de signification). L'interprétant est aussi un signe susceptible d'être à nouveau interprété, ainsi récursivement. Sa théorie triadique, pour laquelle deux éléments peuvent se substituer l'un à l'autre pour un sujet donné, est moins restrictive que celle de Ferdinand de Saussure. Elle implique une relation de substitution, elle n'est pas seulement formelle puisqu'elle implique un sujet pour saisir la relation, et elle s'élargit, au-delà du signe linguistique en une sémiotique générale.

Dans le contexte de la sémantique des langues et de leur traitement automatique, Enjalbert propose une notion abstraite et générale des mécanismes d'interprétation [69]. La question du sens se pose pour toute activité humaine (lecture d'un texte, interprétation d'une photo, etc.). Un individu appréhende et reconnaît une forme (syntaxique, langagière, visuelle), ou une configuration (reconnaissance et assemblage d'indices). Par une opération (ou calcul, ie processus cognitif), en général complexe, il se construit des représentations mentales adéquates. L'auteur propose de nommer cette activité *interprétation*, mettant en correspondance deux *systèmes*. Plus précisément, l'interprétation repère des structures ou configurations d'un système *source* (le signifiant), et construit de nouvelles configurations relevant d'un second système dit *but* (le signifié).

Enjalbert se situe dans la lignée de Pierce [170], reprise notablement par U. Eco [63], et place au centre de l'étude non pas le "sens", le "contenu" ou autre "quelque chose" associée à un signe (un signifiant), mais une activité, un processus [69]. Cette approche est à comparer avec le modèle très général de la triade sémiotique et symbolisé par le triangle sémiotique présenté figure 2.2.

La théorie du support

La théorie du support est une théorie philosophique de la connaissance qui postule que toute connaissance ne peut procéder que d'une inscription sur un support matériel : tout objet technique est l'inscription matérielle d'une connaissance et que toute connaissance est d'origine technique [11, 12]. Ainsi toute connaissance repose sur une inscription, dont elle est l'interprétation (la connaissance est l'interprétation de l'inscription tandis que l'inscription est la matérialisation de la connaissance).

Cette idée est développée en particulier autour du cas du support numérique. Une connaissance peut ainsi être écrite dans un livre, mais aussi dans un fichier vidéo ou un logiciel, sans les modéliser formellement, mais en leur offrant un support de mémorisation et de manifestation. La transmission de connaissance résulte d'une inscription intentionnelle sur un support par un auteur pour un lecteur. Dans ce contexte, le support technique est un moyen de spatialiser l'information, pour pouvoir la rejouer dans le temps.

Dans ce contexte, les propriétés du support technique conditionnent l'expressivité et l'intelligibilité de la connaissance transmise.

De la raison graphique à la raison computationnelle

Jack Goody a formulé en 1979 la notion de *«raison graphique»* [90], en montrant comment l'invention de l'écriture a profondément modifié les schèmes de représentation de la connaissance, jusque là orale. Les documents papiers ont permis en effet la représentation spatiale de l'information, en lui donnant une permanence dans le temps. Grâce à ces possibilités nouvelles d'inscription, de nouvelles connaissances ont pu naître de l'émergence de représentations qui ne peuvent être formulées oralement, par exemple :

— la *liste* permet de délinéariser le discours pour en prélever des unités que l'on ordonne ensuite dans une énumération,

— le *tableau* est le fait de représenter un ensemble de rapports entre des unités à travers leur position respective selon les deux dimensions de l'espace de l'écriture, induisant un mode de pensée spatialisé,

— la *formule* est un procédé permettant de mener des raisonnements en fonction seulement de la forme, sans avoir à prêter attention à la signification.

Le passage de l'oral à l'écrit n'est donc pas seulement un changement de support, c'est une révolution cognitive.

De même que l'écrit a permis le passage du temporel au spatial par projection de la parole, le support numérique apporte de nouvelles formes de représentation des informations, basées sur le calcul. Bachimont parle de l'émergence d'une *«raison computationnelle»*[12], reprise par Mpondo-Dicka [160].

En effet, l'ordinateur ne traite que des séquences binaires qui, par le calcul, deviennent des signes sur un support tel que l'écran. C'est cette propriété du support numérique qui est fondamentale en ceci qu'elle propose de nouvelles modalités d'inscription. Et ces nouvelles modalités induisent également la constitution de modes de représentation nouveaux.

— Le *programme* est à la raison computationnelle ce que la *liste* est à la raison graphique. Autant la liste permet de catégoriser et de classifier, d'offrir un synopsis spatial, autant le programme permet de spécifier un parcours systématique : l'exécution du programme n'est alors que le déploiement temporel de la structure spatiale symbolique qu'est le programme.

— Le *réseau* est à la raison computationnelle ce que le tableau est à la raison graphique. Alors que le *tableau* propose une structuration et une systématicité entre les contenus répartis dans les cases du *tableau*, le *réseau* propose un mode de communication et de répartition entre les cases du tableau. C'est un tableau dynamique.

— Enfin, la *couche* est à la raison computationnelle ce que la *formule* est à la raison graphique. La *formule* permet en effet de considérer la forme abstraction faite du contenu : la *couche* permet de considérer des relations calculatoires entre des unités, abstraction faite des calculs sous-jacents impliqués.

2.1.4 Codage et interprétation de données numériques

La question du sens est au cœur des problématiques liée à la généralisation du numérique dans nos sociétés. En 1994, l'observatoire de la recherche en informatique écrivait déjà : "la recherche et la représentation du sens, la mesure de la pertinence d'une réponse sont aussi des sujets qui progressent et peuvent donner lieu non seulement à de

forts intéressants développements théoriques sur les langues naturelles et l'interprétation des images mais aussi à des applications dans le cadre de la communication homme-machine" [137]. Le chemin parcouru depuis, jusqu'à l'émergence des outils d'intelligence artificielle et leur généralisation, questionne les rapports entre les données numériques et les sociétés humaines.

La manipulation de données informatiques peut également être observée sous une perspective sémiologique. C'est en 1967 que paraît la première édition du traité fondateur de Jacques Bertin : *Sémiologie graphique. Les diagrammes, les réseaux, les cartes*. Avec cet ouvrage semble s'ouvrir un nouveau champ de connaissances, la « sémiologie graphique », ou science de la représentation graphique des données [22]. Plus récemment, de nombreux travaux ont par exemple été menés autour de la sémiologie graphique dans les langages de modélisation [66].

L'usage de moyens informatiques consiste à manipuler de l'information, qui peut être encodée de multiples manières différentes sur un ordinateur (fichiers, bases de données, outils divers). Une telle unité d'information sera appelée artéfact informatique.

Définition 1 (Artéfact informatique) *Un artéfact informatique désigne une unité d'information créée, produite, modifiée ou utilisée par un homme ou une machine dans le cadre d'un « système » informatique. Un artéfact informatique peut être un programme, un script, un code source, un document, un fichier de données quelconque, un diagramme, un texte, une donnée multimédia, une page Web, etc.*

On peut par exemple considérer la représentation graphique de connaissances sous la forme d'un diagramme [66]. C'est en ce cas l'artefact informatique (le diagramme) qui joue le rôle du signe (le signifiant), pour un objet d'étude donné (le dénoté) : le sujet auquel se réfère l'artefact informatique. L'interprétation de cet artefact par un être humain ou un système constitue le signifié. Très souvent, cette interprétation comporte une grosse part d'implicite, et le sens peut être inféré du format de l'artefact (sa syntaxe textuelle ou graphique, son langage d'encodage), de l'outil utilisé pour le manipuler ou le visualiser, ainsi que du contexte général de la manipulation de l'artefact.

Le cycle de vie de cette information repose sur les 3 phases suivantes, illustrées sur la figure 2.3 :

— le **codage** (modélisation) de l'information : un être humain traduit une réalité ou une abstraction portant une sémantique dans un format de données ou un langage informatique ;

— le **traitement** de l'information et son éventuel transcodage dans une autre source d'information : ce traitement est réservé à la machine, mais peut éventuellement comprendre des phases d'interactions avec un utilisateur pour lever des ambiguïtés, valider et confirmer des décisions, etc.

— le **décodage** (interprétation) de l'information : un être humain traduit des données informatiques en les interprétant et en leur donnant un sens.

On voit que tout au long du cycle de vie de l'information, la pertinence de l'usage de l'outil informatique repose sur l'interprétation des données manipulées par l'être humain. Si un individu A encode une donnée D1, qui devient D2 après traitement et qui sera analysée et interprétée par un individu B, il y a un risque que les individus A et B ne se comprennent pas, à cause du fossé sémantique engendré par le codage de D1 par A et le décodage de D2

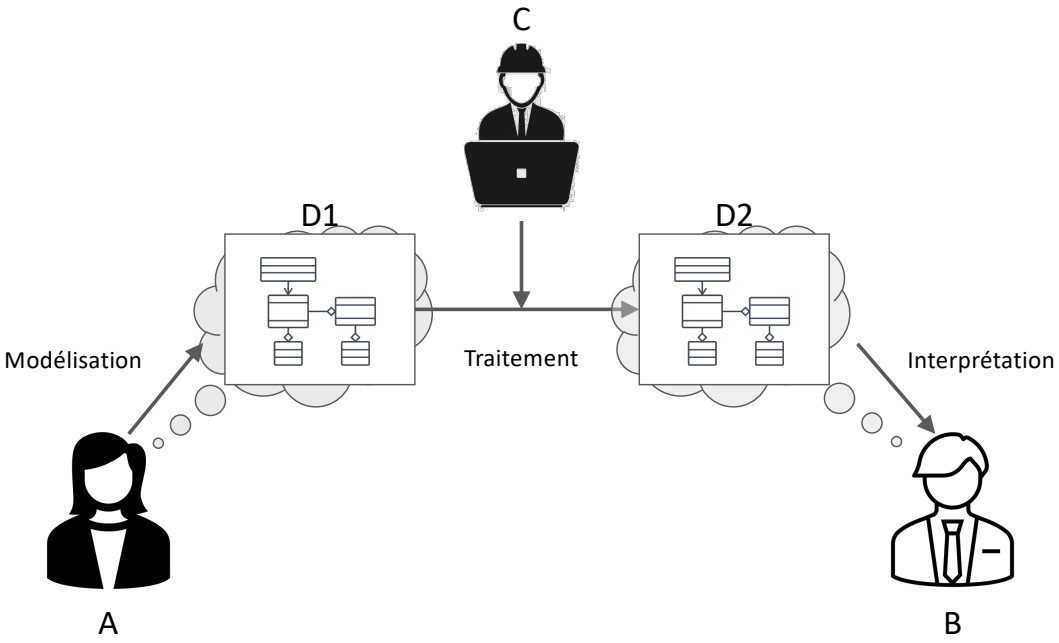

FIGURE 2.3 – Encodage, traitement et interprétation de données numériques

par B, avec éventuellement un autre fossé sémantique engendré par l'individu C, qui a codé le traitement de D1 en D2, et qui a sa propre interprétation. Ces différentes activités sont liées aux mécanismes d'interprétation par un être humain du sens de l'information manipulée [1].

Ces considérations nous amènent à définir ce que sont les modèles, à réfléchir aux mécanismes de modélisation et leur utilisation dans les processus d'ingénierie, avec une attention particulière portée aux problèmes d'alignements sémantiques [65].

2.2 Approches de modélisation

2.2.1 Notion de modèle

« Pour un observateur A, M est un modèle de l'objet O, si M aide A à répondre aux questions qu'il se pose sur O » (d'après Marvin Minsky [158])

Dans le contexte d'usage de moyens informatiques, et donc de calculs, on ne peut se dispenser de la notion de **modèle**, au sens où c'est le seul artefact manipulable par une machine.

Un modèle peut se définir comme une **représentation abstraite d'une réalité avec une préoccupation explicite**.

1. Le développement et l'utilisation de robots conversationnels nous incite d'ailleurs à réfléchir, sur la base de la figure 2.3, sur le sens de l'activité interprétative de B dans un contexte où c'est une intelligence artificielle qui remplace l'individu C, et où le traitement D1→D2 est automatique et repose sur des mécanismes de *deep learning*, sans aucun lien avec une quelconque cognition.

— La réalité adressée ici peut concerner de multiples cibles, et le modèle peut ainsi concerner un objet réel, plongé ou non dans un environnement lui même modélisé, une partie du monde physique, un procédé, une chaîne temporelle, causale ou une succession d'évènements, une autre abstraction, un problème exprimé dans un certain formalisme, etc.

— Un autre aspect important de la définition de modèle présentée plus haut est l'**intention**, qui permet le choix d'une modélisation permettant d'exprimer et de rendre calculable le modèle au regard de la tâche à accomplir.

— Un modèle est utile pour communiquer, raisonner, imaginer, comprendre, analyser, diagnostiquer.

Nous nous proposons de reprendre la définition de [125, 51], qui synthétise un large consensus d'après les travaux de l' *Object Management Group* (OMG)[2], de Muller *et al* [162], de Bézivin *et al* [33] et de Seidewitz [183].

Définition 2 (Modèle) *Un modèle est un ensemble de faits caractérisant un aspect d'un système dans un objectif donné. Un modèle représente donc un système selon un certain point de vue, à un niveau d'abstraction facilitant par exemple la conception et la validation de cet aspect particulier du système [51, 125].*

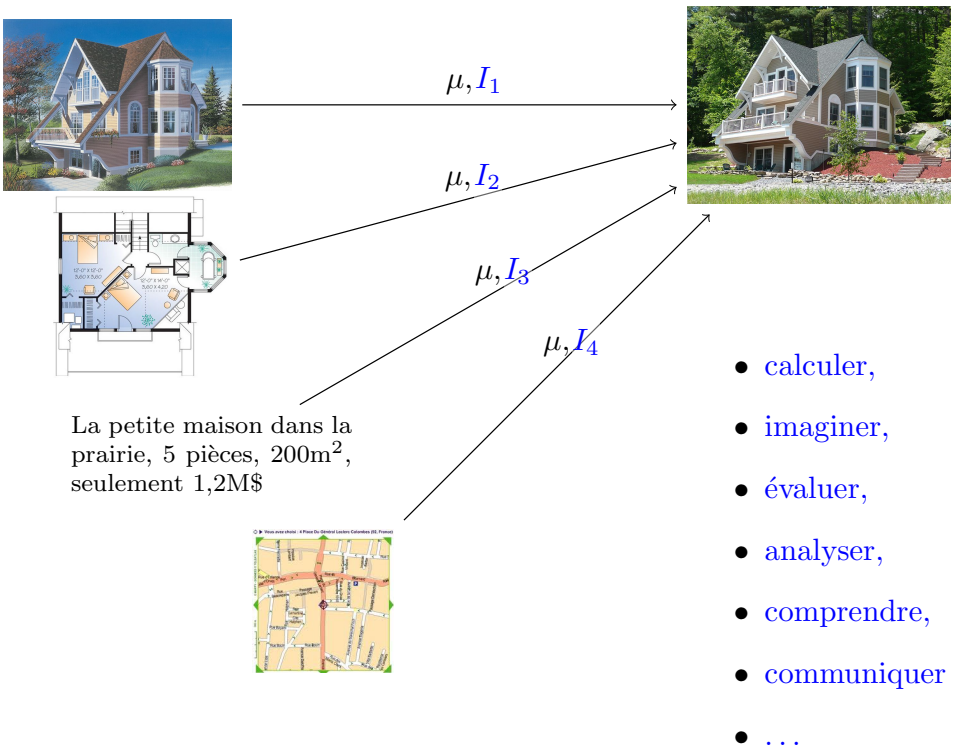

FIGURE 2.4 – Des modèles pour appréhender une réalité

La figure 2.4 illustre les multiples modèles que l'on peut manipuler autour d'un même objet d'étude. Cette figure présente en outre la relation majeure de l'IDM (Ingénierie Dirigée par les Modèles), appelée *représentationDe*, et nommée μ dans [8] et [183]. L'existence

2. *The Object Management Group* : consortium américain à but non lucratif créé en 1989 dont l'objectif est de standardiser et promouvoir le modèle objet sous toutes ses formes, cf https://www.omg.org

de multiples modèles liés à l'objet d'étude par la relation μ traduit ici la présence de plusieurs préoccupations. Chacun de ces modèles répond au principe de *substituabilité*, en ce sens qu'il est conçu pour répondre aux questions sous-jacentes posées par un point de vue particulier (lié à la préoccupation), exactement comme le système aurait répondu lui-même [3].

La modélisation, que l'on définit comme la construction et la mise en oeuvre d'un modèle, pour répondre à une préoccupation donnée, nécessite d'opérer un certain nombre de choix, notamment sur l'élicitation et la classification des concepts, sur la caractérisation de ces concepts, et des propriétés que l'on va leur associer, sur la granularité de la conceptualisation, sur la causalité et les éventuels traitements à associer au modèle, et plus généralement sur le choix du paradigme.

2.2.2 Métamodèle

Nous avons vu que le modèle constituait un artefact (1) manipulable par une machine (2) compréhensible (concevable et interprétable) par un être humain. Cette dernière assertion implique une nécessaire explicitation du langage dans lequel est exprimé le modèle, afin que (1) la machine puisse interpréter correctement le modèle, par le biais d'une sémantique attachée à ce langage, (2) plusieurs êtres humains puissent interpréter et partager de l'information de manière non ambiguë autour d'un modèle. Nous proposons de reprendre la définition proposée par Combemale, Jezequel *et al* dans [51, 125] :

Définition 3 (Métamodèle) *Un métamodèle est un modèle qui définit le langage d'expression d'un modèle [95], c'est-à-dire le langage de modélisation [51, 125].*

La figure 2.5 présente le partage d'un modèle d'un sujet d'étude par deux personnes. Cette figure met en évidence une nouvelle relation liant le modèle et le langage utilisé pour le construire (le métamodèle), appelée *conformeA* et nommée χ sur la figure. En pratique, le métamodèle joue ici le rôle d'un référentiel commun aux deux individus, et qui permet le partage explicite d'un domaine de connaissances. En outre, ce référentiel commun apporte généralement de l'outillage dédié au domaine (formats de données, éditeurs, outils, etc.)

Définition 4 (*conformeA* (χ)) *Un modèle est dit conforme à un métamodèle si chacun de ses éléments (objets et relations) est instance d'un élément du métamodèle, et s'il respecte l'ensemble des propriétés (e.g. contraintes d'invariant) exprimées sur le métamodèle [51, 125].*

Nous verrons dans la section suivante que l'IDM s'appuie principalement sur cette notion de conformance [4] et de métamodèle, et a conduit à la définition de standards. Dans ce cadre, il convient de pouvoir appréhender une grande variétés d'artefacts produits par l'industrie informatique et proposant différents paradigmes, représentations, outils.

3. peut-être pas nécessairement de manière exacte mais dans une certaine marge d'erreur tolérée en fonction de l'emploi du modèle

4. Le terme "conformance" est un néologisme emprunté à la langue anglaise, et bien que non présent dans le dictionnaire, résulte d'un consensus dans la communauté. Il se substitue au terme "conformité" qui n'aurait pas tout à fait le même sens ici.

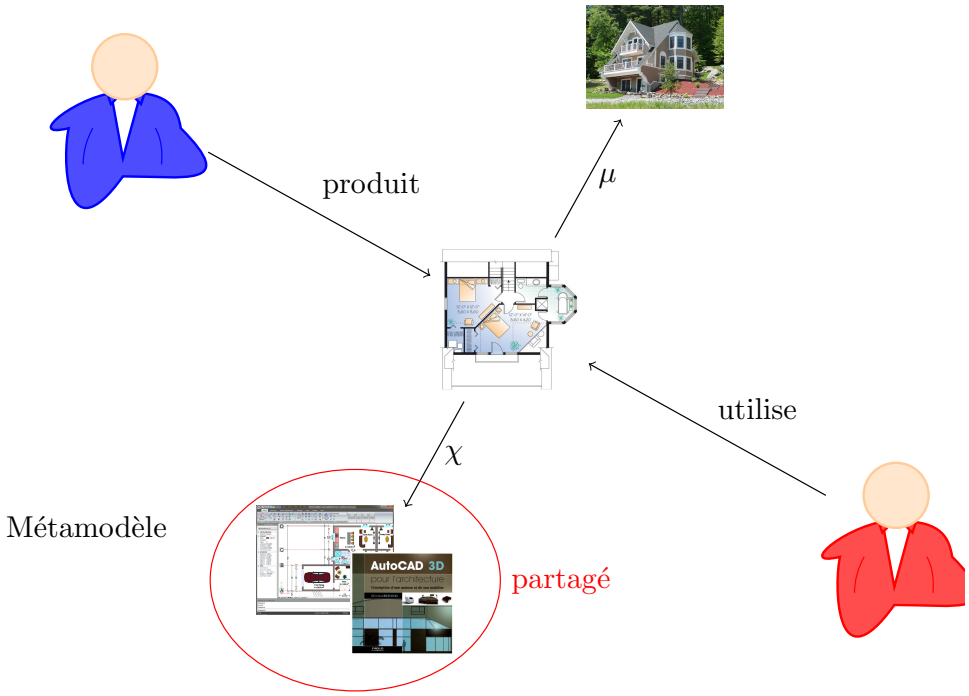

FIGURE 2.5 – Un métamodèle pour capitaliser un modèle de connaissances

2.2.3 Ingénierie Dirigée par les Modèles

Si nous avons pu constater que la modélisation est à la base de toute activité scientifique, notamment en terme de validation des modèles obtenus vis-à-vis d'expérimentations effectuées dans le monde réel, le domaine de l'ingénierie présente une spécificité dans la mesure où les ingénieurs construisent des modèles d'artefacts qui n'existent pas encore, puisqu'il s'agit en général de les construire [125].

Les pratiques actuelles en terme d'ingénierie (logicielle ou système) reposent sur les trois constats suivants :

— Les logiciels ou systèmes à construire peuvent être très gros et très complexes, notamment quand il s'agit de systèmes de systèmes, ce qui implique une décomposition structurelle en sous-systèmes et en composants, avec de nécessaires liens de communication ou interactions entre ces sous-systèmes ou composants.

— La nécessité d'aborder pour chaque partie d'un système de nombreuses préoccupations qui touchent des domaines de connaissances différents, et pour lesquelles des modèles ad hoc sont utilisés [16]. L'expression explicite de solutions efficaces pour gérer ces préoccupations permet d'établir des compromis adéquats relativement tôt dans le cycle de vie du logiciel.

— Tous les systèmes informatiques conçus aujourd'hui sont déclinés en de multiples variantes, au delà de l'aspect purement fonctionnel. Les points de variations peuvent être liés à la diversité des matériels impliqués, ou bien encore aux systèmes d'exploitation et autres intergiciels, mais également à des aspects législatifs, normatifs ou culturels. Ces points de variations sont souvent assez orthogonaux, ce qui engendre une explosion du nombre de variantes possibles d'un système. Cela se traduit par la notion de ligne de produits [172].

À ces trois constats se rajoutent, dans un contexte d'optimisation opérationnelle, la nécessaire réutilisation de logiciels, de composants ou de sous-systèmes déjà existants, qu'il faut en général configurer, ou bien modifier, afin de les intégrer.

Le métier des architectes et des ingénieurs logiciels consiste à mettre en œuvre des démarches d'analyse et de conception prenant en compte toutes ces préoccupations. Si au niveau de l'analyse, les solutions à ces préoccupations peuvent être décrites comme des aspects, le processus de conception peut alors être caractérisé comme le tissage de ces aspects dans un processus de conception détaillé [126]. Comme nous l'avons vu plus haut, cette tâche devient extrêmement complexe lorsque les systèmes ou logiciels à construire sont gros et complexes, avec des préoccupations transverses, orthogonales, et parfois contradictoires, et enfin quand on se place dans un contexte de ligne de produits.

Le défi que cherche à résoudre l'IDM est de réussir à mener à bien cette tâche, avec la contrainte supplémentaire d'un contexte où les exigences peuvent évoluer et pour lesquelles le système à construire doit être modifié en permanence, et donc d'un contexte où l'agilité des développements prime. Dans ce contexte, toute nouvelle déclinaison du système doit être obtenue rapidement, de manière fiable, et bon marché [125]. L'enjeu pour l'IDM est de proposer une mécanisation de ce processus complexe, pour reproduire automatiquement ce processus de composition manuelle de chaque aspect, que les développeurs expérimentés suivent à la main [114]. L'idée est de pouvoir produire rapidement une nouvelle variante dans une ligne de produits, en jouant automatiquement la plus grosse partie du processus de conception et en apportant juste quelques petites modifications [140].

2.2.4 L'approche *Model Driven Architecture* (MDA)

Pour toutes les raisons que nous venons de décrire, l'utilisation des modèles que requiert l'IDM (aussi appelée MDE, *Model Driven Engineering*) s'appuie nécessairement sur la notion de conformance et de métamodèle, avec en ligne de mire la nécessité pour les ingénieurs de partager un domaine d'expertise donné, avec sa masse de connaissances, explicites et implicites, des méthodologies et des processus, des formats d'encodage de la donnée, des outillages dédiés, etc.

L'essor des méthodologies de conception orientées modèles a conduit à un développement foisonnant de manières de décrire les modèles, et donc de nombreux langages et métamodèles permettant de capturer l'expertise nécessaire à traiter de nombreuses préoccupations, dans des domaines divers et variés. Au delà de la richesse de l'émergence de ces solutions est apparu un risque fort d'avoir à gérer la cohabitation de nombreux métamodèles, langages et formalismes parfois incompatibles. Il y avait une nécessité à proposer un cadre généraliste et unificateur. C'est à cette fin que le consensus sur UML (*Unified Modeling Language*) [94] fut décisif dans cette transition vers des techniques de production basées sur les modèles [51]. Une réponse logique à ce foisonnement de métamodèles fut de proposer un langage de définition de métamodèle qui prit lui-même la forme d'un modèle : ce fut le métamodèle MOF (*Meta-Object Facility*) [95]. En tant que modèle, il doit également être défini à partir d'un langage de modélisation. Pour limiter le nombre de niveaux d'abstraction, il doit alors avoir la propriété de métacircularité, c.-à-d. la capacité de se décrire lui-même.

Définition 5 (Métamétamodèle) *Un métamétamodèle est un modèle qui décrit un langage de métamodélisation, c.-à-d. les éléments de modélisation nécessaires à la définition des langages de modélisation. Il a de plus la capacité de se décrire lui-même.*

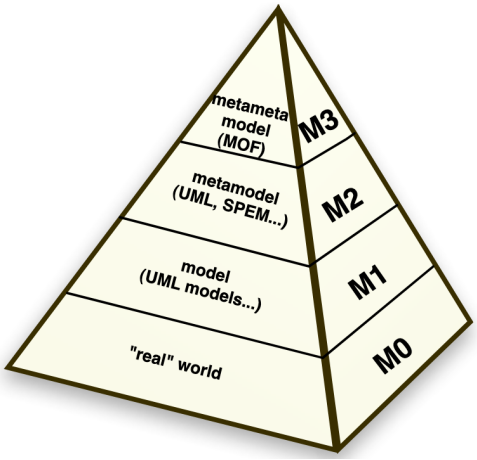

FIGURE 2.6 – Pyramide de modélisation de l'OMG, [27]

Dans le contexte de l'OMG (Object Management Group) et de la modélisation, les termes M0, M1, M2 et M3 sont souvent utilisés pour décrire les différents niveaux de modélisation. Comme illustré sur la figure 2.6, chaque niveau représente une abstraction croissante et une distance par rapport à l'implantation concrète d'un système. Le niveau M0 représente le système (le monde réel). Les modèles représentant cette réalité sont définis au niveau M1, tandis que le niveau M2 est constitué par les métamodèles. Le dernier niveau M3 est constitué du métamodèle unique et métacirculaire (le MOF). Chaque niveau correspond à une utilisation particulière des modèles, et permet de capitaliser un domaine de connaissances utilisé au niveau inférieur.

C'est sur ces bases que l'OMG a défini l'architecture MDA en 2000 [93], qui définit un ensemble de bonnes pratiques autour de l'utilisation des modèles pour concevoir, spécifier et générer des applications informatiques. L'approche MDA vise à promouvoir la réutilisation, la portabilité et l'évolutivité des applications logicielles. En utilisant des modèles abstraits, il est possible de concevoir des applications de manière plus indépendante des technologies et matériels spécifiques, ce qui facilite la maintenance, la transformation et l'adaptation des systèmes logiciels au fil du temps. L'idée centrale de l'approche MDA, qui propose la définition de plusieurs standards, notamment UML, MOF et XMI, est de séparer la logique métier d'une application de sa mise en œuvre technique. Elle favorise la création de modèles abstraits indépendants de la plateforme, qui capturent les aspects essentiels du système à développer. Plus précisément, l'approche MDA préconise l'élaboration de :

— **modèles d'exigence** (*Computation Independant Model* - CIM), qui capturent les besoins, les exigences et les objectifs métier de l'application, indépendamment de toute technologie ou plateforme spécifique ;
— **modèles d'analyse et de conception** (*Platform Independant Model* - PIM), qui représentent la logique métier de l'application de manière plus détaillée, mais restent indépendants des choix techniques spécifiques ; ils définissent les fonctionnalités, les règles métier et les flux de données de l'application ;
— **modèles d'implémentation** (*Platform Specific Model* - PSM), qui décrivent l'implémentation technique de l'application sur une plateforme spécifique, en tenant compte

des détails d'architecture, des frameworks, des bibliothèques et des technologies de la plateforme cible.

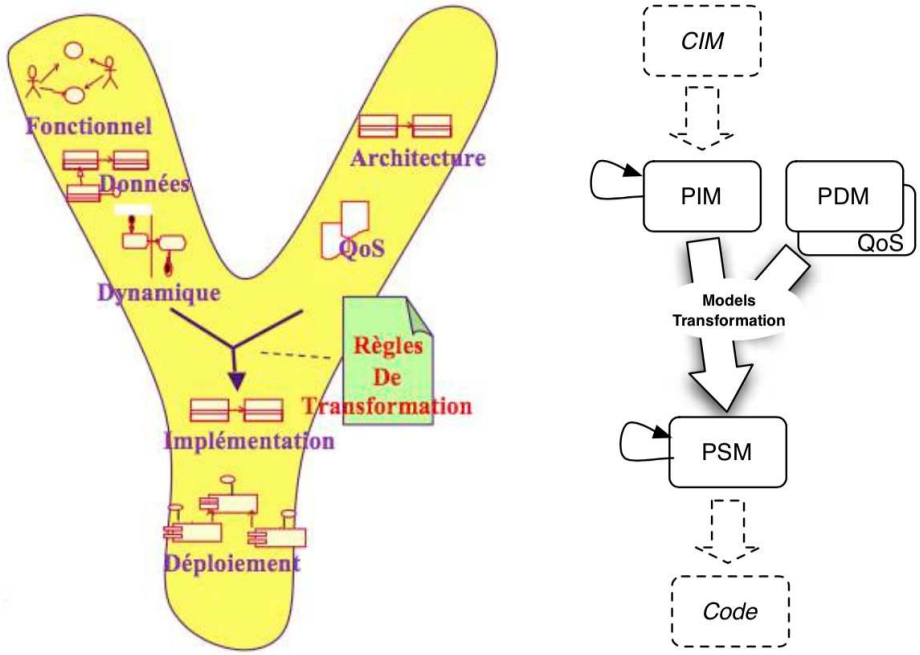

FIGURE 2.7 – MDA : Un processus en Y dirigé par les modèles, [51]

Cette démarche s'organise selon un cycle de développement "en Y" propre au MDD (Model Driven Development) [51], cf figure 2.7. Elle s'appuie sur des mécanismes de transformation de modèles, qui sont utilisées pour passer d'un modèle à un autre, par exemple, de PIM à PSM [5]. Ces transformations peuvent être manuelles ou automatisées. Enfin, l'un des avantages clés de l'approche MDA est la possibilité de générer automatiquement une grande partie du code source à partir des modèles. Cela permet de réduire le temps et les efforts nécessaires pour développer l'application.

L'approche MDA n'a cependant pas tenu toutes ses promesses initiales, pour plusieurs raisons. La première étant l'impossibilité de capturer toutes les préoccupations dans des modèles. L'automatisation totale ou partielle du processus d'ingénierie nécessite des étapes de génération de code, et des modifications du code généré pour prendre en compte les préoccupations manquantes. A terme la maintenance de tous les modèles et du code généré puis modifié - et de leur nécessaire synchronisation - devient très difficile voire insurmontable. Une deuxième raison tient également à ce que les mécanismes de transformation pour passer du PIM au PSM sont parfois très complexes et que cette expertise n'est pas capitalisable dans un modèle, mais dans la transformation en elle-même. Aucune plateforme n'a jamais réussi à proposer un mécanisme simple de composition de modèle entre le PIM et le PDM pour obtenir le PDM [124].

5. Le passage de PIM à PSM fait intervenir des mécanismes de composition et de transformation de modèles avec un modèle de description de la plate-forme (*Platform Description Model* - PDM), un modèle de description de la qualité de service (Quality of Service - QoS) et d'autres modèles (éventuellement non fonctionnels).

Transformations de modèles

La présentation de l'approche MDA nous donne l'occasion d'évoquer rapidement la notion de transformation de modèles, qui est un concept-clef de l'ingénierie dirigée par les modèles. La littérature abonde de propositions sur la définition, la classification et la mise en œuvre de transformations [27, 36, 163]. En partant des métamodèles sources et cibles de la transformation, on distingue deux types de transformations : les transformations endogènes et exogènes. Une transformation est dite endogène si les modèles impliqués sont issus du même métamodèle. Mais lorsque les modèles sources et cibles sont de différents métamodèles, la transformation est dite exogène. Les transformations exogènes peuvent être :

— (i) **Synthèse** - transformation d'un certain niveau d'abstraction vers un niveau d'abstraction moins élevé. Un exemple typique est la génération de code.

— (ii) **Rétro-ingénierie** - inverse de la synthèse

— (iii) **Migration** - transformation d'un programme écrit dans un langage vers un autre langage du même niveau d'abstraction.

2.2.5 D'autres approches de modélisation ?

Si l'approche MDA fait l'objet d'un large consensus dans la communauté IDM, il est peut être utile de se pencher sur des pratiques moins normatives. Nous nous proposons ici d'ouvrir la perspective habituellement en usage pour parler de modélisation.

"Tout est modèle !"
(à condition de savoir comment l'interpréter)

L'environnement numérique se généralisant, la donnée pré-existe souvent à la nécessité de la manipuler sous la forme d'une abstraction (peut-être sous une forme non directement exploitable) [6]. Ce peut être un ensemble de données métier liées au domaine, une base de données, une ontologie, un référentiel utilisé dans le cadre du système étudié, une base documentaire, un modèle de l'environnement (topographique, météorologique), une norme, etc. Muller et al. défendent ainsi l'idée que *tout le monde modélise sans le savoir*, comme Monsieur Jourdain fait de la prose sans le savoir [162]. Par extension, une vision moins restrictive et plus large de la notion de modélisation consiste à penser que "tout est modèle" [30, 83]. Il s'agit de ne pas se limiter aux standards et d'appréhender une grande variétés d'artefacts produits par l'industrie informatique et proposant différents paradigmes, représentations, outils.

Dans ce contexte, une approche naïve invite à se pencher sur les conditions de "modélité" (au sens décrit par Guy Caplat dans les critères de "modélité" [45] : *représentativité*, *généricité*, *subjectivité*, *finalité*), qui visent à pouvoir exploiter la donnée existante pour la considérer comme un modèle, et à exploiter des modèles déjà construits (donc avec d'autres préoccupations initiales). Pour cela, il faut pouvoir exprimer la façon d'interpréter la donnée dans le contexte dans lequel elle va être traitée (*représentativité* et *subjectivité*), et qui est peut être non parfaitement aligné ou désaligné avec le contexte original. Le contexte

6. Pour s'en rendre compte, il peut être intéressant de s'interroger sur la généralisation de pratiques de "copier-coller" dans les environnements professionnels numériques.

de réinterprétation nécessitera peut-être en outre de considérer d'autres données (d'autres modèles) qui complèteront ce premier modèle, et qu'il faudra mettre en relation. Le but de cette réinterprétation (et de cette éventuelle mise en relation) répond au principe de *finalité* (fonction du modèle qui permet de répondre à certaines questions, à certaines préoccupations).

Cette notion de "réinterprétation" est à mettre en perspective avec l'observation de certaines pratiques courantes dans des environnements de modélisation génériques tels que UML ou SysML. Il est très intéressant, en effet, de s'appuyer sur un standard de modélisation, car l'on dispose, en général, d'outils matures. En revanche, le métamodèle associé peut être trop générique, ou trop ambigu (nombreux points de variation sémantiques), pour porter seul une connaissance partagée par tous les utilisateurs des modèles produits.

La généricité et la souplesse d'un standard peu explicite peut être très utile pour exprimer des aspects qui n'ont pas été pensés a priori, et cette richesse est indispensable à la capture de la connaissance et de l'expertise métier dans certains contextes. Certaines pratiques observées mènent même parfois à modéliser des connaissances métier dans des métadonnées (ou même des propriétés graphiques), dont la sémantique d'interprétation ne peut être qu'externe à l'environnement de modélisation : on "tord" ici l'outil pour exprimer quelque chose pour lequel ce même outil n'était pas prévu. Ces pratiques nécessitent ainsi des connaissances implicites, dont il serait intéressant de pouvoir réifier l'explicitation.

Espaces technologiques

Ces réflexions sur le fait de considérer tout artefact informatique comme un modèle introduisent naturellement la problématique de l'organisation de l'hétérogénéité technologique. Favre et al définissent la notion d'*espace technique* comme l'ensemble des outils et techniques issus d'une pyramide de métamodèles dont le sommet est occupé par une famille de (méta)métamodèles similaires [72]. Ainsi la figure 2.6 représente l'espace technologique constitué par l'univers UML/MOF. Kurtev et al [136] ont proposé d'étendre cette notion pour proposer le terme *espace technologique*, qui référence un ensemble de concepts, de standards, de logiciels et d'outils liés. Un espace technologique est souvent associé à une communauté d'utilisateurs donnée avec un savoir-faire partagé. Il s'agit en même temps d'une zone d'expertise établie et d'un dépôt de ressources abstraites et concrètes. Pour illustrer cette notion, les auteurs proposent ainsi cinq espaces technologiques : l'ingénierie ontologique (OWL), les langages basés sur XML, les systèmes de gestion de base de données (SGBD), l'approche MDA et les langages de programmation. Au lieu de poser la question a priori de l'existence éventuelle d'une théorie unificatrice pour tous les espaces technologiques, les auteurs proposent d'établir des correspondances de niveau conceptuel.

Parreiras et al ont travaillé sur la convergence et la mise en relation des approches MDA et des ontologies [169], afin de pouvoir exploiter dans les modèles les connaissances liées à un domaine métier capturées dans des ontologies OWL. Ils décrivent leur approche pour l'intégration des différents espaces techniques ontologiques dans les espaces techniques de méta-modélisation. Heldal et al. [109] ont également travaillé sur la distinction et les liens entre les modèles descriptifs qui rendent compte d'une réalité (d'un domaine) et sont utilisés pour la documentation et les modèles prescriptifs, utilisés pour le développement, afin de mieux comprendre comment favoriser l'adoption de la modélisation dans l'industrie.

2.3 Gestion de modèles multiples

2.3.1 De nombreux modèles pour des préoccupations diverses

La figure 2.8 illustre une situation que l'on retrouve dans beaucoup de métiers, et notamment ceux qui ont trait à l'ingénierie. Le sujet d'étude (ici une maison, représentée au centre de la figure) est l'objet de nombreuses préoccupations métier, prises en charges par de nombreux acteurs et experts métier. Le sujet d'étude est souvent un système, voire un système de systèmes. Autour de ce sujet d'étude coexistent de nombreuses ingénieries de domaine, structurées en silos verticaux, et pilotées par des experts métiers. Ces spécialistes de leur domaine ont chacun leur culture métier, leurs outillages et leurs pratiques, et construisent et manipulent des données (des modèles et des métamodèles) illustrant leurs préoccupations. Différents modèles d'un même sujet d'étude peuvent cohabiter, se compléter, entrer en concurrence, voire se contredire les uns les autres [45].

Par ailleurs, le sujet d'étude est lui-même plongé dans un environnement complexe, parfois multi-modélisé lui aussi. La conception d'un système requiert également de pouvoir modéliser son processus de conception, son cycle de vie. La définition du cahier des charges et des spécifications en amont nécessite de définir des exigences à plonger également dans des modèles pour valider leur pertinence (mécanismes de test et de vérification).

FIGURE 2.8 – Le sujet d'étude : centre de préoccupation de nombreux modèles

De ce contexte émerge naturellement un fort besoin de pouvoir connecter les artefacts informatiques (les sources d'informations, les modèles), pour des problèmes évidents de coordination, de construction de chaînes d'outils, de synchronisation, de réconciliation de points de vues, de traçabilité, d'évolution et de co-évolution, d'identification d'inconsistances, etc.

Ce besoin répond principalement à deux caractéristiques des modèles : 1) ils ne sont pas isolés, et 2) sont deux à deux partiellement recouvrants. En effet, les bonnes pratiques des activités de modélisation (et notamment la séparation des préoccupations) exigent une

structuration des modèles qui met en évidence des relations de dépendances entre modèles, et mettent donc en jeu des références entre modèles (explicites ou implicites). Par ailleurs, peu de domaines métiers sont si verticaux qu'ils ne manipulent pas des abstractions également manipulées par d'autres domaines. Concrètement, les modèles adressés par deux domaines métiers sont forcément recouvrant sur certains concepts, même si - et c'est souvent le cas - les dits concepts ne sont pas modélisés avec le même niveau de détails.

Avant d'en donner une définition plus complète dans la section 2.3.3, nous introduisons ici la notion d'**interopérabilité sémantique de modèles** comme une réponse à la nécessité de connecter des modèles, et de gérer leur cohérence globalement.

2.3.2 Motivations

Nous nous penchons ici sur les motivations métier justifiant la pertinence de l'intérêt des approches d'interopérabilité sémantique de modèles.

Taille des sujets d'étude - Une première motivation est liée à la taille des sujets d'étude appréhendés par l'ingénierie contemporaine. Historiquement, un problème et une préoccupation pouvaient autrefois être encodés dans un unique modèle, qui permettait de raisonner et de conclure. Aujourd'hui l'industrie traite couramment des sujets d'étude beaucoup plus importants, de gros systèmes voire des systèmes de systèmes, qui mettent en jeu des milliers de modèles et d'artefacts, dont il convient d'expliciter les dépendances.

Préoccupations transverses - Une seconde motivation concerne la nécessité de gérer des préoccupations transversales sur des systèmes modélisés (par exemple la sécurité). La structuration des modèles ne peut être alors alignée à la fois sur le découpage fonctionnel et sur l'aspect transverse étudié, ce qui implique des relations de dépendances à établir entre les différents artefacts.

Un monde numérique - Un autre aspect à considérer résulte du constat d'un monde fortement et systématiquement numérisé, où les données et les modèles pré-existent, sous une forme ou une autre (artefacts globaux, locaux ou services). Il convient dans ce cadre de pouvoir se lier directement à ces sources d'information, en les réinterprétant dans un contexte donné (celui de la finalité de leur utilisation). Cette pratique permettrait d'éviter le "copier-coller" comme source d'erreur assez commune.

Une ingénierie de réutilisation - L'Ingénierie Dirigée par les Modèles (IDM) défend l'idée que pour bien connaître un sujet d'étude, il faut bien connaître son processus de construction et de développement, également modélisé [129]. Les pratiques courantes d'ingénierie mettent en œuvre des mécanismes de réutilisation de systèmes déjà construits, pour les reprendre et les adapter. Ceci implique de pouvoir appréhender tous les artefacts et les modèles d'un développement précédent, de comprendre tous les liens de dépendance, de traçabilité et des justifications afférentes, pour les requestionner à l'aune du nouveau développement à produire.

L'industrie 4.0 - L'essor de l'industrie 4.0 et les besoins émergents en terme de transition numérique, notamment portés par le jumeau numérique [128] requièrent une numérisation massive et systématique de l'ensemble des systèmes cyber-physiques, ainsi qu'une modélisation numérique des processus et la mise en réseau des machines, des produits et des individus. Aujourd'hui, cette digitalisation est un vrai enjeu pour de nombreux indus-

triels, qui doivent identifier les normes pertinentes et les ressources humaines pour traiter ce besoin, et gérer la consistance de ces modèles.

L'Intelligence Artificielle - Cette préoccupation est très liée à la précédente, et traduit également un besoin évident de pouvoir traiter et qualifier de grandes quantités de données, en leur donnant du sens.

Variabilité et lignes de produits - Dans de nombreux métiers, la gestion de la variabilité est désormais bien connue et prise en compte dans la plupart des processus de conception industriels. Ceci permet la mise en place de stratégies de lignes de produits [47], la modélisation de cette variabilité permettant de dériver automatiquement un certain nombre de produits. À ce titre, l'industrie 4.0 soutient un nouveau paradigme de personnalisation de masse où l'idée est de produire une multitude de variétés de produits personnalisés afin de répondre rapidement à la demande des consommateurs.

Traçabilité - D'une façon générale et dans de nombreux domaines métier, les besoins en terme de traçabilité sont aujourd'hui généralisés. Ces besoins sont multiples et couvrent de nombreux enjeux (normes, certification, aspects juridiques, etc.).

Ces enjeux sont au cœur des problématiques adressées par de nombreux travaux de recherche présentés dans les sections suivantes de l'état de l'art, et notamment les méga-modèles [34, 28, 73, 106], le *"model management"* [35] et l'*"inter-modelling"* [99, 100].

2.3.3 Définition de l'interopérabilité de modèles

Le sujet de l'interopérabilité de modèles a été décrit dans de nombreux contextes. Nous proposons ici les définitions du standard IEEE-100 [64] reprises par Bastien Drouot [60] :

1. La capacité de deux ou plusieurs systèmes ou éléments à échanger des informations et à utiliser les informations échangées.
2. La capacité des unités d'équipement à travailler ensemble pour effectuer des fonctions utiles.
3. La capacité, promue mais non garantie par la conformité avec un ensemble de normes, qui permet à des équipements hétérogènes, généralement construits par différents fournisseurs, de fonctionner ensemble dans un réseau.
4. La capacité de deux ou plusieurs systèmes ou composants à échanger des informations dans un réseau hétérogène et à utiliser ces informations.

La notion d'interopérabilité peut être abordée sous différents angles. Chen et al [46] identifient trois dimensions d'interopérabilité qu'ils nomment *préoccupation*, *barrière* et *approche*, comme illustrées sur la figure 2.9 (extraite de [60]).

Concernant la première dimension, les préoccupations identifiées sont les suivantes [60] :

1. *Interopérabilité de données* : approche visant à faire fonctionner ensemble différents modèles de données (hiérarchiques, relationnelles, etc.).
2. *Interopérabilité de service* : approche visant à composer et faire fonctionner ensemble diverses applications (conçues et implémentées indépendamment) en résolvant les différences syntaxiques et sémantiques ainsi qu'en assurant les connexions à des données hétérogènes.

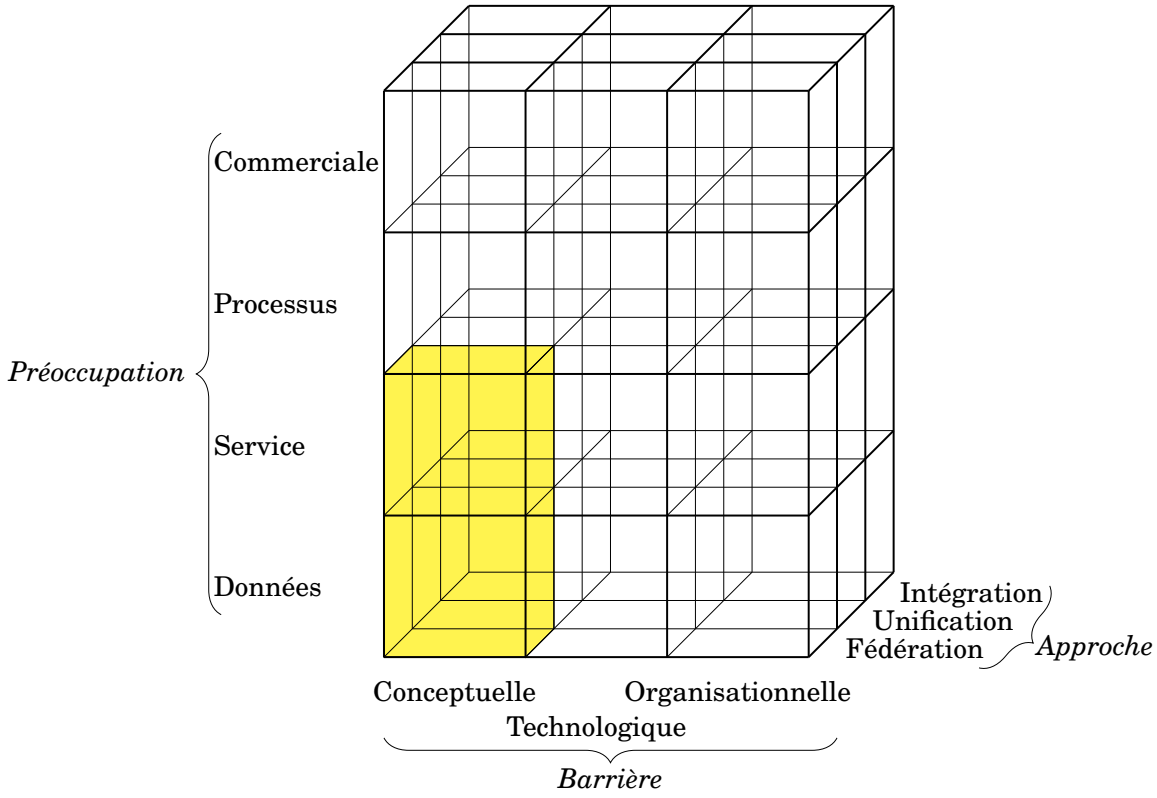

FIGURE 2.9 – Représentation tridimensionelle des approches d'interopérabilité

3. *Interopérabilité de processus* : approche visant à faire fonctionner ensemble différents processus, un processus définissant dans quel ordre les services sont fournis en fonction des besoins du système.
4. *Interopérabilité commerciale* : approche visant à faire collaborer différentes entreprises.

La deuxième dimension, appelée *barrière*, représente les différentes incompatibilités ou décalages qui font obstacle au partage et à l'échange d'informations. Elles peuvent être conceptuelles, technologiques ou organisationnelles. Les barrières conceptuelles concernent les incompatibilités syntaxiques et sémantiques des informations. Celles technologiques existent en raison de l'absence d'un ensemble de normes compatibles permettant l'utilisation de techniques informatiques hétérogènes pour partager et échanger des informations entre deux ou plusieurs systèmes. Les barrières organisationnelles représentent les incompatibilités de la structure et des techniques de gestion mises en œuvre dans différentes entreprises.

La troisième dimension représente les différents types d'approche, qui sont détaillées dans la section suivante.

2.3.4 Les différentes approches possibles

Les modèles ne sont donc pas à prendre de manière isolée, mais à considérer dans le cadre d'un réseau de dépendances avec d'autres modèles. Dans ce contexte, agir sur un modèle a un impact sur un ensemble de modèles dépendants. Par conséquent, plutôt que

de travailler sur des modèles isolés, il faut pouvoir travailler sur des ensembles de modèles, en considérant les relations de dépendance qui lient ces modèles.

Le standard ISO-14258 [116, 60] identifie trois grandes techniques pour relier des modèles :

1. l'**intégration** repose sur la construction d'un métamodèle commun à partir duquel tous les modèles sont instanciés ;
2. l'**unification** repose sur le choix d'un métamodèle pivot permettant de connecter les modèles via des transformations ;
3. la **fédération** propose de modéliser explicitement les liens sémantiques entre les modèles.

L'approche "intégration"

L'**intégration** suppose ici la construction d'un nouveau métamodèle MM′ union de tous les concepts des métamodèles utilisés (les différents formalismes ou langages intégrés) [67, 10, 156]. On parlera aussi d'approches de fusion [132, 42, 209], dans lesquelles les différents modèles conformes à un multi-modèle sont fusionnés en un seul artefact.

Cette approche est illustrée figure 2.10. Quatre "modèles" I issus de quatre espaces technologiques distincts sont figurés, avec l'outillage associé (T1, T2, T3, T4) . Trois d'entre eux sont conformes à un métamodèle MM, tandis que le quatrième n'en définit pas.

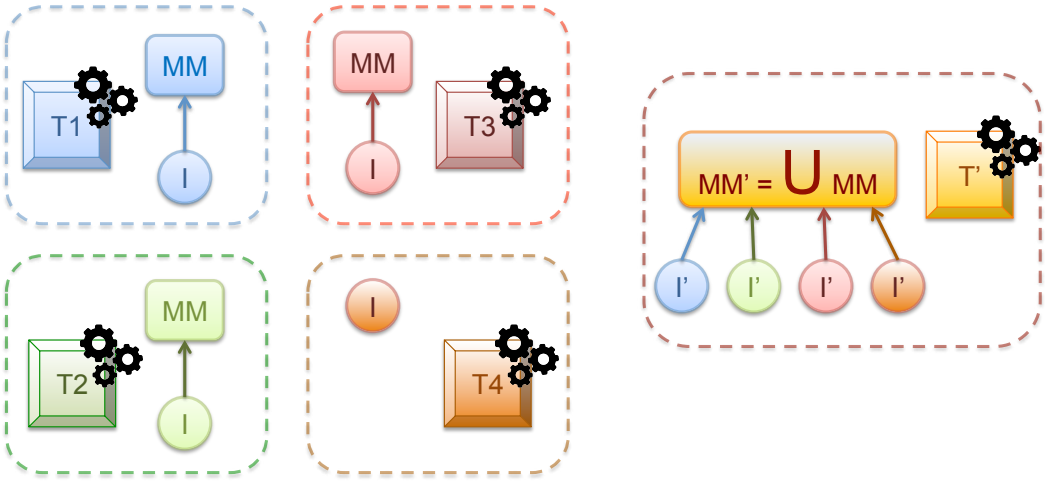

FIGURE 2.10 – L'approche "Intégration"

Cette approche requiert un travail important autour de la définition de ce nouveau format commun. Ce format peut être une norme internationale, ou bien le résultat d'un consensus entre toutes les parties pour élaborer des modèles et construire des systèmes. En pratique, la définition d'un tel métamodèle est complexe et n'est pas envisageable systématiquement. D'un point de vue technique, elle suppose une grande compatibilité des technologies connectées. Son principal inconvénient est la lourdeur du mécanisme de composition, qui suppose que les formalismes intégrés ne varient pas au cours du temps, car toute création, suppression ou modification d'un élément intégré pourrait nécessiter la redéfinition complète du métamodèle. Cette approche implique que les outils existants ne

sont plus utilisables et nécessite donc la création d'un nouvel outillage développé sur la base du nouveau standard union.

Conceptuellement, le mécanisme de composition de tous les formalismes intégrés pose aussi quelques questions sur la sémantique de la fusion. Quel type d'union ? Comment gère-t-on les recouvrements, les incompatibilités ? En terme d'organisation, ce mécanisme conduit à produire de très gros métamodèles, avec parfois des problèmes de passage à l'échelle. Enfin, l'évolutivité des métamodèles intégrés doit être limitée car suppose un nouveau calcul du métamodèle union.

L'approche "unification"

L'**unification** propose de choisir ou de construire un métamodèle spécifique (appelé métamodèle pivot) qui fournit une structure de méta-niveau commune procurant un moyen d'établir des équivalences sémantiques. Ce mécanisme est complété par la définition de transformations (birectionnelles ou non) entre tous les formalismes et le langage pivot [31].

La figure 2.11 illustre cette approche, sur la base du même cas d'utilisation original que la figure 2.10. C'est le formalisme "bleu" MM+ qui a été ici retenu pour faire office de métamodèle pivot. Le mécanisme est complété des transformations bidirectionnelles t1, t2, t3.

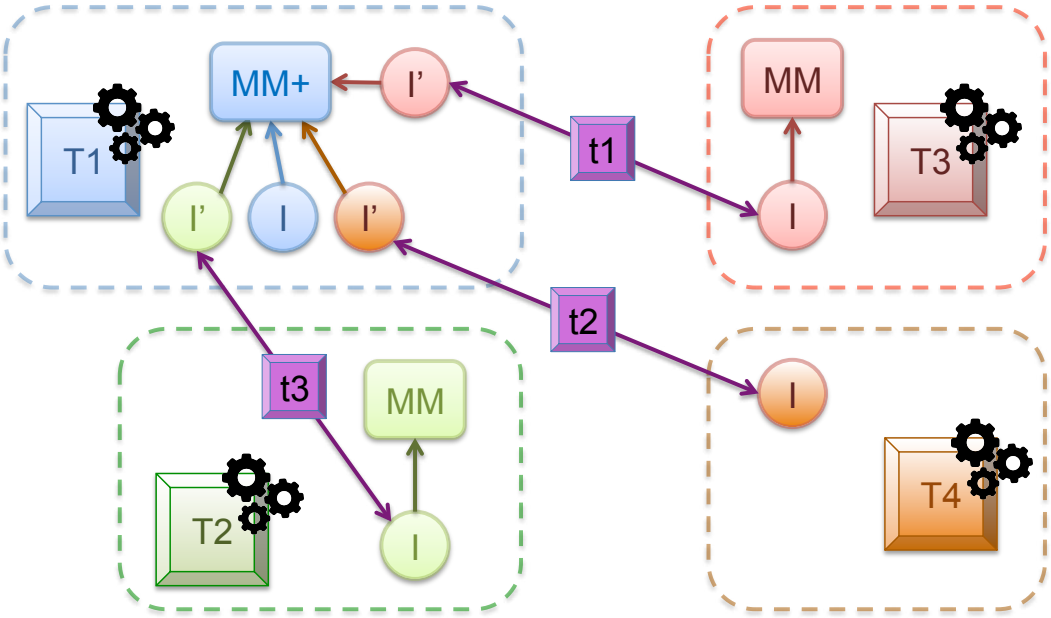

FIGURE 2.11 – L'approche "Unification"

Le principal intérêt de cette approche réside dans le fait que tous les outils existants peuvent être conservés, ainsi que le cycle de vie de tous les modèles intégrés, qui peuvent continuer à être manipulés par les utilisateurs métier avec les outils habituels. C'est l'application des transformations qui permettra de gérer l'ensemble des modèles en cohérence.

Cependant, dans cette approche la sélection ou la création du langage pivot reste une tâche difficile. Si ce dernier est trop abstrait, il en résulte une importante perte d'informa-

tion dans l'application des transformations résultantes, et inversement si l'on veut conserver toutes les informations sans pertes, il faut alors disposer d'un métamodèle pivot très large et des transformations parfaitement exhaustives, ce qui peut se révéler impossible à faire en pratique (par ailleurs, l'unification devient alors dans ce cas équivalente à l'approche d'intégration) [60].

Un inconvénient majeur de cette approche est la gestion du cycle de vie de l'ensemble des modèles en cohérence, c'est-à-dire la gestion de l'application des transformations pour un système stable et prédictible. Puisque chaque source de données (chaque modèle) est autonome dans son espace technique et espace métier d'origine, les désynchronisations entre les modèles deviennent possibles et les transformations peuvent parfois conduire à des incohérences ou inconsistences (voir même à des boucles de mises à jour). Cette approche nécessite donc une vision holistique décidant notamment de la source de l'information la plus à jour (notion de source de vérité [168, 151]). Notons enfin que des variantes cette approche permettent de travailler avec plusieurs métamodèles pivot, au prix d'une gestion plus complexe de l'ensemble des modèles.

L'approche "fédérative" ou fédération

Enfin, la dernière approche, qui consiste à **fédérer les modèles**, est illustrée sur la figure 2.12. Elle se concentre sur la modélisation de la sémantique des liens entre les concepts de langages hétérogènes, et propose **une réification des liens entre les entités fédérées**.

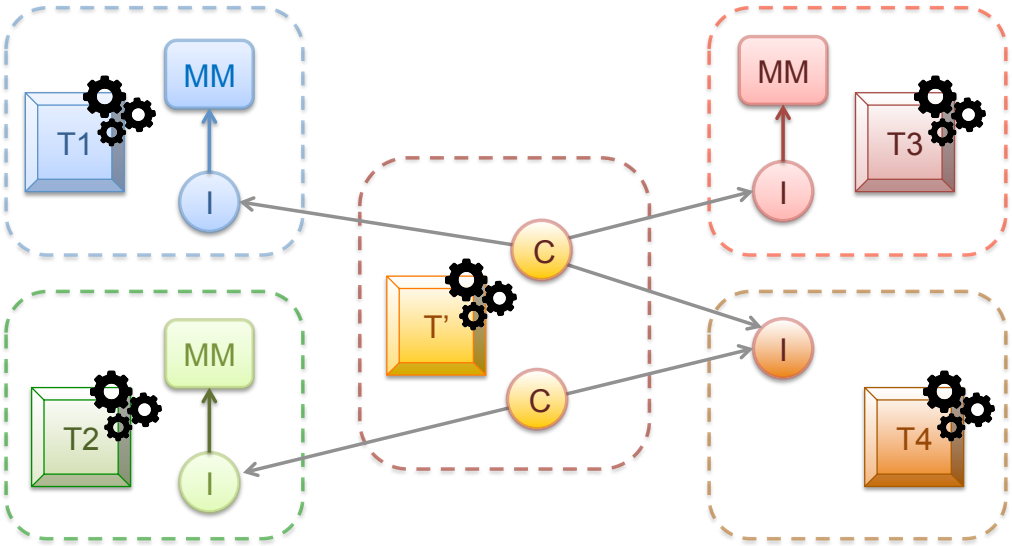

FIGURE 2.12 – L'approche "fédérative"

Cette approche se passe totalement d'un métamodèle pivot ou d'un standard mais propose une structuration et un étiquetage explicite de liens techniques. C'est le lien qui porte implicitement le sens de la désignation. Autrement dit, il existe un découplage complet entre l'élément désigné dans son environnement d'origine et sa sémantique dans le cadre de la fédération. Un même élément désigné peut intervenir plusieurs fois dans plusieurs sémantiques (au sens d'une interprétation conceptuelle), et plusieurs éléments désignés concourir à la définition d'une même sémantique.

Cette approche ne nécessite pas de transformations a priori, mais gère en cohérence l'ensemble des données à fédérer. Une approche fédérée peut mettre en jeu des transformations, mais ces dernières sont appliquées uniquement à la demande.

Cette approche présente l'avantage de conserver l'outillage existant pour l'édition des données fédérées, qui continuent à disposer de leur propre cycle de vie, mais requiert le développement d'un nouvel outillage dédié à la gestion de cette fédération.

Les approches par fédération permettent enfin d'envisager une certaine dynamicité dans la gestion de sources d'informations hétérogènes, qui peuvent être ajoutées, modifiées ou supprimées dynamiquement.

2.4 Approches "fédératives"

Nous nous concentrons dans cette section sur les approches où les liens entre les modèles sont explicitement réifiés.

2.4.1 Caractéristiques étudiées

Afin de pouvoir classer les approches "fédératives" existantes et d'établir un vocabulaire commun pour les présenter, nous proposons figure 2.13 un diagramme de caractéristiques montrant les alternatives et les propriétés existantes. Nous présentons une classification autour de la nature des **entités fédérées** (les modèles), des **liens** réifiés, sur la **fédération**, en tant que graphe d'entités fédérées, et nous considérons enfin les **aspects comportementaux** de l'ensemble des entités fédérées.

a. Entitées fédérées

Les approches fédératives permettent la manipulation de modèles hétérogènes. L'hétérogénéité des modèles peut être classée dans les quatre catégories suivantes : 1) les modèles d'une approche multimodèle peuvent correspondre à différents *formalismes* ; 2) les modèles peuvent représenter différents *domaines* ; 3) les modèles peuvent appartenir à différents *espaces technologiques* (par exemple, un modèle peut être représenté en UML, OWL, RDF ou même en feuilles de calcul Excel). Une quatrième catégorie correspond à la gestion intégrée ou non de la conformance des modèles fédérés. Si l'approche est consciente de cette conformance, les modèles peuvent alors être placés à différents niveaux d'abstraction (l'intérêt est par exemple qu'il soit possible de lier des modèles et des métamodèles au-delà de la relation de conformance).

b. Liens de fédération

Les approches "fédératives" opérationnelles se fondent sur leurs capacités à gérer des modèles multiples sur la réification de liens explicites qui peuvent avoir différentes caractéristiques.

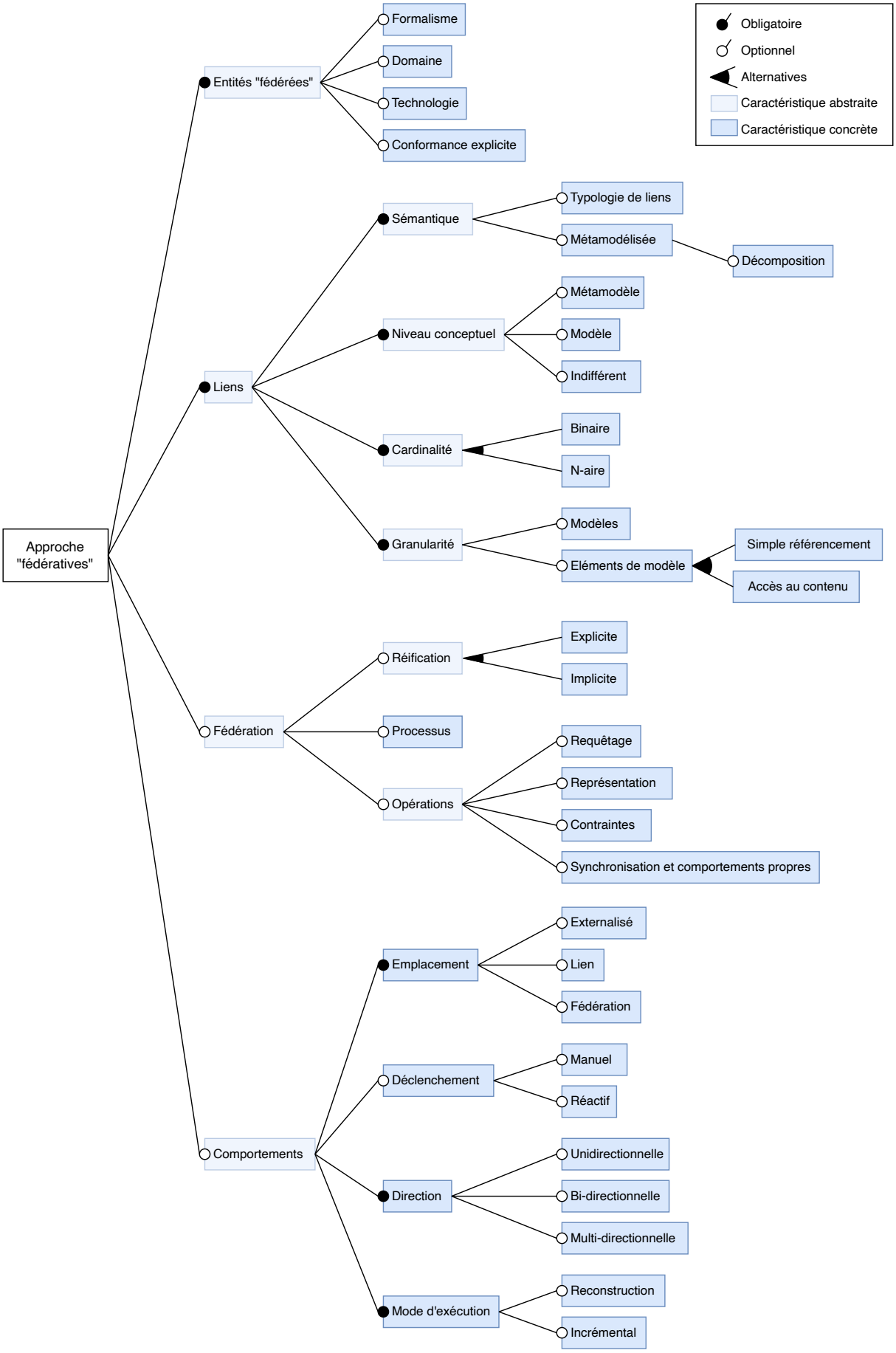

FIGURE 2.13 – Diagramme de caractéristiques des approches "fédératives"

Sémantique des liens : Les différents liens peuvent s'appuyer sur une simple caractérisation du type de lien, par exemple μ (*représentationDe*), χ (*conformeA*), τ (*transforméEn*) [73], ou bien sur une typologie ad hoc avec une sémantique prédéfinie. D'autres approches proposent une métamodélisation du type de lien, et une éventuelle structuration et raffinement du lien.

Niveau conceptuel : Les liens peuvent être définis, pour les approches qui supportent la conformance explicite, au niveau des modèles, au niveau des métamodèle ou des deux.

Cardinalité : La cardinalité minimale d'un lien est de deux, mais si l'approche peut traiter plus de deux modèles (et quelle que soit la limite supérieure), elle est n-aire.

Granularité : La granularité se réfère aux éléments qui peuvent être effectivement utilisés ou manipulés dans la définition des liens. Nous distinguons trois niveaux de granularité : 1) la granularité au niveau du modèle est la capacité de relier des modèles sans accéder à leur contenu (par exemple, les approches de gestion de modèles) ; 2) la granularité au niveau de l'élément de modèle est la capacité de relier le contenu des différents modèles en les désignant, mais sans accès à leur information contenue (par exemple, les attributs et les opérations pour les modèles UML) ; 3) enfin, le niveau de granularité qui permet d'accéder au contenu des éléments de modèle.

Cycle de vie : peut-on avoir des liens avec un cycle de vie (par exemple, des moments où ils sont valides et d'autres non) ?

c. Fédération

Nous nous intéressons ici à la fédération, en tant que graphe d'entités fédérées, avec l'ensemble des liens.

Réification : La fédération peut être réifiée ou ne pas l'être. Sa réification peut être implicite ou explicite (si elle est encodée dans un artefact dédié).

Processus : Certaines approches peuvent prescrire un processus de *"développement"* à suivre pour la *construction* du graphe de modèles, et pour son utilisation et sa manipulation ultérieures.

Opérations : Les approches "fédératives" permettent d'exécuter un certain nombre d'opérations sur l'ensemble résultant d'un ensemble de modèles connectés. Ces opérations correspondent à certaines opérations qui peuvent être effectuées dans des modèles classiques : requêtes, contraintes, vues et V&V (vérification et validation).

d. Comportements

L'une des propriétés spécifiques des approches multi-modèles est la synchronisation, c'est-à-dire la capacité de maintenir des mises à jour des entités fédérées lors de l'introduction de changements. Plus généralement, l'opérationnalisation d'une approche fédérative nécessite en général la définition et l'exécution de comportements (dont la synchronisation).

Emplacement : Dans certaines approches, les comportements sont complètement externalisés. D'autres approches permettent d'associer des comportements aux liens et/ou à la fédération (si cette dernière est réifiée).

Direction : La direction fait référence à la direction de synchronisation des changements entre une paire donnée de modèles A et B. Si une seule direction est prise en charge (soit A vers B, soit B vers A), l'approche est unidirectionnelle. Inversement, les changements peuvent être synchronisés dans les deux sens dans un schéma de synchronisation *bidirectionnel* [111, 53].

Mode d'exécution : Le mode d'exécution fait référence à la manière dont la synchronisation est réalisée. Naïvement, la synchronisation peut être réalisée en reconstruisant entièrement les liens entre les entités fédérées. C'est ce que nous appelons la synchronisation par reconstruction. Par ailleurs, les tâches de synchronisation des modèles peuvent être exécutées de manière incrémentale, de sorte que seuls les changements sont propagés sans qu'il soit nécessaire de reconstruire complètement les liens.

Déclenchement : Les tâches de synchronisation entre les modèles peuvent être déclenchées manuellement ou automatiquement (par exemple, les systèmes de transformation réactifs maintiennent les paires et/ou les réseaux de modèles automatiquement synchronisés en réagissant aux changements [21, 155]).

La présentation de ces différentes caractéristiques va dans la suite nous servir de grille de lecture pour l'analyse et le positionnement des travaux présentés ci-après, et dont une synthèse sera proposée dans les tableaux 2.1 et 2.2.

2.4.2 Gestion de la consistance multi-modèles (*Multi-Model Consistency Management*)

La problématique de la gestion d'un grand nombre de modèles dans les approches MDE est très importante, parce que les modèles ne sont pas isolés et qu'ils sont très souvent modifiés, ce qui les rend très vulnérables aux inconsistances. Cette problématique est abondamment traitée dans la littérature [152, 200, 199]. Des vocables divers partagent des préoccupations très connexes (*Model Management*, *Global Model Management*, *Multi-Model Consistency Management*, *Model Management Tools*). Ces approches abordent souvent cette problématique sous l'angle de la réparation de modèles [195], en postulant que l'émergence d'un grand nombre d'artefacts induit forcément la levée d'inconsistances, et que des outils automatiques ou semi-automatiques permettent d'aider efficacement les développeurs à résoudre ces problèmes. Ces approches font appel à différentes techniques, et notamment la synchronisation de modèles, les transformations (classiques, incrémentales, bi-directionnelles), la propagation de modifications, etc.

Au delà de la vérification de consistance et des réparations exprimés sur un seul modèle, en général au moyen d'un langage d'expression de contraintes par exemple OCL ou EOL (*Epsilon Object Language*), la plupart de ces approches se concentrent sur des couples d'artefacts (relations binaires entre modèles) pour la vérification de consistance inter-modèles. D'autres approches mettent en avant une mise en correspondance des modèles par recouvrement (souvent partielle) [59].

Il n'est pas possible ici de mentionner tous les nombreux outils dédiés à la gestion de modèles (*Model Management Tools*). Citons cependant *Epsilon* [167], qui constitue un ex-

cellent représentant d'outils permettant le requêtage et la modification de modèles, la vérification de consistance, la fusion (*merge*) de deux modèles et la transformation d'un modèle en une nouvelle représentation. Concernant ces approches, certains auteurs mettent en avant le terme d'*inter-modelling* [99, 100]. Les approches de tissage de modèles (*model weaving*) [29] proposent la construction de modèles de traces, en tant qu'(hyper)-graphes qui permettent de relier des éléments de modèles dans des modèles différents. Citons enfin l'approche déclarative *Triple Graph Grammars* (TGGs) [180, 80], où des règles de production d'une grammaire permettent de peupler un graphe de correspondances entre deux autres graphes (des modèles).

Peu de travaux s'intéressent explicitement aux défis liés à la transformation et au maintien de la synchronisation entre plus de deux sources d'information sans fusionner les métamodèles et les modèles. Certains résultats récents soulignent cette lacune [50]. Dans [193], les auteurs présentent *comprehensive systems*, un framework théorique pour représenter les relations n-aires (appelées *commonalities*) dans le contexte de la gestion de modèles multiples. Ces travaux sont fondés sur la théorie des graphes et des catégories, mais aucune mise en œuvre concrète n'est disponible à ce jour.

2.4.3 Mégamodèles

Les mégamodèles sont très souvent cités comme une réponse à la problématique de la gestion et de la mise en cohérence de modèles multiples [34, 73]. Par exemple concernant la notion de *Global Model Management* (GMM) qui a été évoquée dans la section précédente, beaucoup d'auteurs s'appuient sur la notion de mégamodèle pour y répondre [35]. Il existe une littérature très importante sur les mégamodèles et les approches liées, dont Hebig et al. proposent une vision synthétique [107]. La mise en œuvre des *approches mégamodèles* est généralement guidée par des intentions liées au maintien de la cohérence de modèles [131, 205], pour des guides de processus complexes de conception logicielle [192], à la traçabilité [182, 181], ou dans un simple but de gestion des modèles [176].

Bezivin et al. sont parmi les premiers à poser la notion de mégamodèles. Ils en donnent une première définition en 2003 : *"A megamodel is a model with other models as elements"* [32], avant de la compléter en 2007 : *"A megamodel contains relationships between models"* [17]. Favre et al. comptent également parmi les fondateurs des approches mégamodèles, et propose en 2004 cette définition : *"... the idea behind a megamodel is to define the set of entities and relations that are necessary to model some aspect about MDE"* [71], en considérant qu'un mégamodèle représente des arrangements complexes entre des artefacts sans entrer dans les détails de chacun des artefacts. Salay et al. proposent eux le terme de *macromodel* : *"A macromodel consists of elements denoting models and links denoting intended relationships between these models with their internal details abstracted away"* [177]. Bien que ces approches concernent le monde du MDE (Favre et al. plaident pour une approche ouverte et intégrative via la notion d'espace technologique, cf 2.2.5 [136, 73]), les outillages correspondants s'ancrent souvent dans le MDA, avec des artefacts conformes au MOF.

Ces différentes approches s'accordent sur une partition stricte entre le monde des modèles et celui des métamodèles. Certaines approches (par exemple [32]) excluent qu'un mégamodèle puisse être lui-même référencé dans un autre mégamodèle. D'autres approches l'autorisent cependant. Ce partitionnement impose deux espaces de modélisation différents. En effet, un mégamodèle et ses modèles se trouvent dans des espaces distincts, un

espace de gestion pour le premier et un espace de conception pour les modèles. Ces deux espaces sont séparés par un écart important en termes de processus (le mégamodèle est généralement construit après les modèles), d'organisation (ce n'est pas le même concepteur) et de technologie.

Une autre conséquence importante de ce partitionnement concerne la granularité des relations. Dans les mégamodèles, la granularité s'arrête généralement au niveau du modèle, les relations à grain fin étant encapsulées dans d'autres artefacts (par exemple, les modèles de transformation). Une exception à cette règle peut être trouvée dans [182], où les auteurs introduisent des *mégamodèles hiérarchiques* comme mécanisme de gestion de la traçabilité au niveau du modèle et de l'élément de modèle.

Ces approches offrent une classification des liens selon les quatre grandes relations du MDE : δ (*DecomposedIn*), μ (*RepresentationOf*), ϵ (*ElementOf*), χ (*ConformsTo*), plus la relation τ (*IsTransformedIn*) [73] qui modélise une transformation. Certaines approches s'appuient sur des relations prédéfinies, parfois extensibles ou redéfinissables, certaines approches proposent de définir leur propre typologie de relations. Les approches orientées mégamodèles ne considèrent pas ces relations comme des modèles eux-mêmes.

La réification de la fédération s'opère naturellement dans l'instance du mégamodèle. Malgré tout, il n'existe à notre connaissance pas de standard de métamodèle pour les mégamodèles. L'interopérabilité de modèles peut en revanche s'appuyer sur des notions de services et d'outils.

Les aspects comportementaux liés à la mise en relation de modèles multiples dans les mégamodèles ne sont pas explicitement réifiés dans les mégamodèles, mais externalisés dans les transformations et plus généralement au travers de l'ensemble de l'outillage qui porte le processus d'ingénierie sous-jacent.

Concernant des travaux plus récents, Barbowsky et al. [18] se concentrent sur une approche très opérationnelle qui vise l'efficacité et le passage à l'échelle avec une approche incrémentale qui se présente comme une extension de la technique des *Generalized Discrimination Networks* (GDNs). Leur solution s'apparente à la gestion modulaire d'un réseau d'opérations hiérarchiques qui connectent des modèles.

2.4.4 Approches multi-vues (*multi-view*)

Dans le contexte d'un système complexe modélisé dans de nombreux artefacts, et donc extrêmment fragmenté, les approches multi-vues offrent une réponse intéressante à la problématique de pouvoir manipuler des informations pertinentes pour une préoccupation donnée. L'idée est de pouvoir combiner et remanier ces différents modèles (ou des parties de ces modèles) en ce concentrant sur la question d'intérêt. C'est la notion de vue (*view*) dans un contexte de *view-based modelling*. Brunelière et al. introduisent et clarifient les notions de vues (view) et point de vue (viewpoint) dans une revue de littérature sur ce sujet [40]. La figure 2.14 présente une vision unifiée de ces différents concepts, à partir de la définition de [82]. Les approches multi-vues peuvent être *synthétiques* ou *séparées* [48], et représenter des systèmes dans lesquels chaque vue est mise en œuvre en tant que méta-modèle distinct. Le système global est obtenu en tant que synthèse des informations véhiculées par les différentes vues.

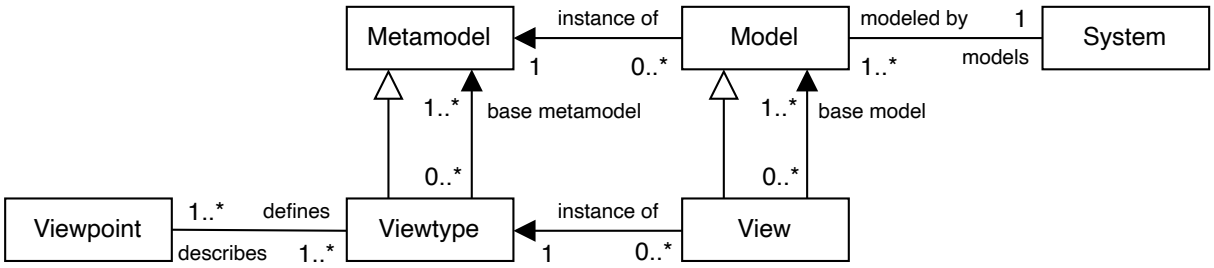

FIGURE 2.14 – Terminologie des approches multi-vues (d'après [40])

Cette approche permet de considérer des couples *View* / *Viewtype* comme des modèles/-métamodèles virtuels obtenus par composition d'autres modèles concrets, utilisés comme des *proxies* de modèles et d'éléments de modèles. En ce sens, la construction et l'instanciation de ces modèles virtuels s'apparente à une approche fédérative dont la réification des liens s'opère au sein d'une métamodélisation.

EMFViews [41] est un exemple canonique des approches de cette catégorie. *EMFViews* permet de spécifier des points de vue regroupant des éléments utilisant des concepts provenant de différents métamodèles EMF. Les vues correspondant à ces points de vue ne sont pas explicitement matérialisées, mais s'apparentent à des *modèles virtuels*, c'est-à-dire des modèles dont les éléments sont des *proxies* d'éléments contenus dans d'autres modèles et chargés à la demande. *EMFViews* ne se contente pas de filtrer et de composer des éléments provenant de différents modèles, il permet également de créer de nouveaux concepts.

Beaucoup de travaux et d'outils se revendiquent des approches multi-vues. *EMFFacet* [7] est un outillage open-source qui fournit des extensions à des modèles EMF existants pour les compléter. *EMFProfiles* [142] généralise la notion de profils pour des métamodèles EMF. *Epsilon Merge* [132], basé sur *Epsilon*, fournit un langage qui s'appuie sur des règles déclaratives avec un traitement décrit en langage impératif, et qui permet de combiner des modèles homogènes et hétérogènes pour créer de nouveaux modèles. *ModelJoin* [44] se présente comme un DSL outillé pour la création de vues construites sur la base de modèles hétérogènes (en tant qu'instances de différents métamodèles). Il propose un langage impératif avec une syntaxe textuelle telle que SQL. *ModelJoin* a été ensuite étendu pour fournir un support à la modification de données. Sur certains aspects, les approches TGGs (Triple Graph Grammar), déjà évoquées plus haut, et qui s'appuient sur les transformations de graphe et les grammaires de triple graphes pourraient également être classées dans cette catégorie [118, 108].

2.4.5 D'autres approches de "fédération de modèle"

Les termes "approches fédératives" ou "fédération de modèles" n'ont jamais été formellement définis et ne font pas l'objet d'un consensus dans la communauté. La littérature abonde donc d'un certain nombre de propositions et de travaux qui mettent en avant ces termes ou des termes similaires. Nous regroupons dans cette section d'autres approches qui se proposent de traiter de multiples modèles hétérogènes connectés par des liens explicitement réifiés.

7. https://projects.eclipse.org/projects/modeling.emft.emf-facet

Approche *Model Globalization*

Le terme de *Model Globalization* a été proposé par Combemale et al. [52], pour désigner l'emploi de multiples DSML, adapté chacun à une préoccupation donnée, pour supporter des développements coordonnés de divers aspects d'un système. Cette approche postule que les experts de domaine peuvent travailler indépendamment dans un certain formalisme et avec les outils dédiés, en même temps qu'un cadre formel régit les interactions nécessaires à la collaboration et la coordination des différentes tâches pour construire ou maintenir le système. Ces préoccupations couvrent les problématiques de communication entre les équipes, de coordination des travaux et de contrôle des résultats produits.

Concernant ces interactions, les auteurs identifient trois différents types de relations entre DSML. L'interopérabilité permet de supporter des échanges d'informations entre des modèles développés indépendamment. La collaboration désigne le développement de modèles couplés. Enfin la composition nécessite la définition explicite des relations entre modèles. Il existe de nombreux défis liés à la prise en charge de ces multiples interactions entre langages pour la conception de systèmes complexes. On citera notamment l'initiative GEMOC [8] qui vise à coordonner et diffuser des travaux sur ces thématiques.

L'approche "*Model Globalization*", telle que présentée dans ces travaux, se concentre sur des langages exécutables, disposant (ou nécessitant) d'une sémantique explicite, avec une attention particulière portée aux aspects "calculabilité" de la composition de ces langages. Les intentions portent moins sur les problématiques de gestion de modèles que sur l'interopérabilité de langages au niveau exécution.

Approche "*Glue Model*"

Dans [165], Niemöller et al. décrivent un cadre conceptuel pour la fédération de modèles et proposent l'utilisation de ce qu'ils appellent un *"glue model"*. Cette solution permet de relier de manière flexible des éléments de modèles spécifiques à différents domaines, au moyen de relations abstraites. Les points de départ et d'arrivée de ces relations constituent des références à des éléments de différents modèles au moyen d'une notion d'adaptateur. Le *"glue model"* constitue lui-même une ontologie, utilisable pour le raisonnement automatique. L'approche proposée peut être décrite sur la base d'un formalisme mathématique avec une sémantique précise ce qui permet des analyses formelles. Les auteurs utilisent cette caractéristique afin de générer un réseau bayésien pour effectuer diverses analyses. À notre connaissance, cette approche n'a été décrite qu'à un niveau purement conceptuel et aucune mise en œuvre ni aucun détail technique n'est fourni par les auteurs.

Role4All

L'approche *Role4All* [179, 178] se présente comme un langage de modélisation destiné à assurer l'interopérabilité entre plusieurs modèles hétérogènes, avec une vision orientée "rôle". La finalité est de pouvoir intégrer de multiples modèles avec différents formalismes au sein d'une modélisation "système". *Role4All* offre un formalisme dynamique et extensible pour interpréter, coupler et instancier des modèles hétérogènes.

8. http://gemoc.org

Une des grandes forces de l'approche est la prise en compte des aspects comportementaux. Concrètement, c'est le concept de rôle qui fournit une abstraction permettant de définir des comportements permettant l'adaptation des modèles externes [61]. Ces comportements traduisent opérationnellement une interprétation de la fédération au-dessus d'un ou plusieurs élément(s) de modèle(s) externe(s) et lié(s) à un domaine. Ce *framework* repose sur la théorie de la modélisation par les rôles [189, 135].

Le *framework Role4All* a été prototypé en SmallTalk et testé sur quelques cas d'étude en tant que preuve de concept. Les limitations de *Role4All* concernent une prise en charge très limitée de différents formalismes et de la définition de propriétés structurelles. Par ailleurs, le prototype actuel offre un support très restreint en terme de modularité et de composabilité.

Syndeia

Dans [15, 13], les auteurs présentent une approche pour une ingénierie MBSE (*Model Based System Engineering*) qui couvre également l'intégralité du cycle de vie du système, qu'ils appellent MBSE++. Les principes sous-tendant cette approche mettent en avant de multiples modèles, hétérogènes et décentralisés, au sein de multiples dépôts, liés par des relations à différents niveaux de granularité, et coordonnant différentes expertises de domaine.

Les auteurs présentent *Syndeia* (précédemment, SLIM [14]) en tant qu'outil de fédération de modèles basé sur SysML. Concrètement, Syndeia relie un modèle d'architecture de système décrit en SysML à de multiples modèles d'ingénierie et référentiels pour créer ce qu'ils appellent un *Total System Model*, un graphe de modèles. Peu de détails sont donnés sur les types de modèles qu'il est possible de fédérer, mais les cas d'utilisation produits incluent des modèles mathématiques (MathLab, Simulink), mécaniques ou électriques (modèles CAD, Creo, NX, CATIA), ainsi que des documents textes et tableurs. *Syndeia* propose un ensemble de types de connexions entre les modèles, en distinguant les *intra-model connexions* (dépendances/contraintes au sein d'éléments d'un même modèle) ou les *inter-model connexions* (dépendances/contraintes entre des modèles différents). Ces connexions sont binaires. Elles peuvent être de simples référencements pour des liens de traçabilité, mais aussi des transformations mono ou bi-directionnelles, des opérations de synchronisation de modèles ou encore des calculs par la définition de modèles d'exécution. La granularité de ces relations est fixée au niveau de l'élément de modèle ou même d'un attribut. Un aspect très intéressant de l'approche est le constat que l'établissement de ces relations doit s'accompagner de comportements permettant de définir des requêtes, des visualisations et des calculs divers.

Aucun détail technique n'est donné sur les aspects internes de Syndeia, si ce n'est son architecture basée sur les graphes, et le rôle central joué par le modèle SysML en tant que *Total System Model*.

Reactive Links

Dans une approche récente, Ratiu et al. présentent *Reactive Links* [173], pour la gestion de la consistance de nombreux artefacts de modèles dans un environnement multi-domaines pour les systèmes complexes, avec de nombreux outils. Ces auteurs partent du

constat qu'il existe beaucoup de solutions dans la littérature qui établissent des liens de traçabilité entre des modèles. Pour certaines approches, ces liens sont statiques et passifs, alors que d'autres approches permettent des calculs de propagation de changements, mais avec des déclenchements manuels et l'utilisation d'un nombre restreint d'outils. Peu de solutions évoquent une certaine automaticité de la synchronisation entre les modèles. Plus généralement, peu de travaux s'attaquent à la problématique de la nature comportementale de la fédération.

Dans ce contexte, la proposition de *Reactive Links* consiste à établir des *Reactive Links*, c'est-à-dire des liens dotés de comportements qui se déclenchent à la suite de certains évènements. Ces liens se caractérisent comme des relations binaires impliquant des éléments de modélisation (au niveau attribut/propriété). À l'opposé de liens traditionnels (passifs), ces liens réagissent automatiquement à des modifications des artefacts et déclenchent des opérations de mise à jour et de synchronisation qui se propagent afin d'assurer une consistance globale du système. Les artefacts considérés comme modèles hétérogènes à connecter sont des abstractions de modèles typés, accessibles via la notion d'adaptateurs (appelés *plugin*).

Reactive Links combine traçabilité, gestion de la consistance et propagation des modifications, avec une granularité fine (au niveau des éléments de modèles et des attributs). Cette solution s'appuie sur la plate-forme *Design Space [58]*.

FTG+PM++

Dans le domaine de la modélisation multi-paradigmes pour l'IDM, *FTG+PM++* constitue une proposition intéressante sous la forme d'un *framework* conceptuel qui permet la définition d'un cadre méthodologique et orienté processus pour la composition de transformations de modèles [164, 150]. Une des difficultés engendrées par la généralisation des approches multi-modèles réside en effet dans les aspects méthodologiques liés au cycle de vie du systèmes (processus, enchaînement des transformations). Concrètement *FTG+PM++* se propose de combiner un modèle de processus (*Process Model* : PM), qui définit le processus de développement du système, avec un graphe de transformations (*Formalism Transformation Graph* : FTG). Cette approche permet ainsi d'exprimer des chaînes de transformation de modèles. Des adaptateurs permettent de se connecter à différentes sources d'information.

GraphQL Federation

Dans le monde des services web (WS), *GraphQL* constitue une solution très populaire et utilisée par des acteurs majeurs. *GraphQL* est développé par *Facebook* et utilisé par d'autres entreprises telles que *Twitter*, *Airbnb*, *Github* et suscite également l'intérêt du monde universitaire. La nature déclarative de ce framework propose une vision métier (orientée modèle de domaine) des services offerts. Concrètement et plutôt que d'offrir une collection de fonctions, le serveur communique aux clients un modèle de domaine sous la forme d'un graphe, au moyen d'un langage de description de schéma (*Schema Description Language*, SDL) textuel. Ce schéma définit des "*endpoint*" [9]. Ceci permet au serveur et au

9. point de terminaison d'API en français

client d'interagir avec un protocole requête/réponse basé sur une vision commune orientée modèle. En ce sens, cette approche se conforme au paradigme MBSE [39].

Cependant, si cette approche est bien adaptée aux développements avec une interface unique, la prise en charge de multiples interfaces s'avère très compliquée. Stunkel et al. [194] présentent *GraphQL Federation* comme une solution de fédération de plusieurs interfaces *WS GraphQL*, à l'aide d'un DSL spécifiquement dédié. Ce DSL, orienté métamodélisation, fournit des abstractions appelées *correspondances*, qui permettent d'établir des relations de correspondances (n-aires) entre différents *endpoint* qui représentent différents modèles de domaine à fédérer. Les auteurs fournissent un prototype démonstratif.

Cette approche est très intéressante par les aspects conceptuels posés par la notion de correspondance. Elle est également très opérationnelle, notamment par l'aspect non-intrusif de cette technologie sur les *endpoint* fédérés. Elle souffre cependant d'une vision presque essentiellement structurelle, puisqu'elle ne propose pas la définition de comportements liés aux relations établies entre les différents modèles.

SoSoM

Nous terminons ce tour d'horizon d'approches multi-modèles par la présentation des travaux de thèse de Freddy Kamdem Simo [186] avec l'approche *SoSoM*, qui traite de la complexité des activités de modélisation pour la conception de systèmes de systèmes. Il défend une vision où les activités de modélisation sont considérées comme des systèmes (*System of Modeling*).

L'auteur présente la méthodologie *MODEF* où un système de modélisation est décrit en 3 niveaux. Le premier niveau s'adresse aux utilisateurs et définit un ordonnancement des tâches réalisées par les différents acteurs. Le deuxième niveau se compose de modèles conceptuels qui représentent une abstraction des modèles de systèmes contenus dans le troisième niveau, qui contient les modèles du système modélisé. Le développement de son approche repose sur des principes de fédération de modèles, c'est-à-dire l'autonomie et le couplage faible avec des interfaces clairement définies entre des systèmes autonomes.

Cette approche présente une vision très abstraite et méthodologique, plutôt qu'une réponse pragmatique aux problématiques de gestion de modèles multiples. Elle inclut un certain nombre d'éléments socio-techniques ayant trait aux activités de modélisation. À ce titre, elle est très illustrative des préoccupations métier, sociales et méthodologiques liées à la mise en relation de modèles.

2.4.6 Synthèse

Les tableaux 2.1 et 2.2 synthétisent les différentes approches évoquées, au regard des quatre critères primaires (entités fédérées, liens, fédération et comportements) définis dans le diagramme de fonctionnalités présenté dans la figure 2.13 page 34.

De nombreuses approches s'avèrent non-agnostiques de la technologie des entités fédérées. Beaucoup de solutions (par exemple *Epsilon*, *EMFView*) s'appuient sur la technologie *Eclipse* et des modèles EMF, d'autres solutions concernent des modèles homogènes (*GraphQL Federation*). Toutes les approches qui concernent une définition abstraite de ce qu'est

	Entités fédérées	Liens	Fédération	Comportements
Multi-Model Consistency Management				
Consistency management tools [152, 200, 199]	Modèles et métamodèles conformes au MOF (MDA) + contraintes portant sur un ou plusieurs modèles (ex. OCL)	Relations binaires encodant la correspondance de modèle	Pas de réification explicite	Déclenchement manuel, externalisés dans les transformations et l'outillage
Epsilon [167]	Modèles EMF	Relations binaires, *match traces*	Pas de réification	Déclenchement manuel, inclus dans les DSL
Model weaving [167]	Eléments de modèles	Hyper-graphes	Modèle de traces	Déclenchement manuel, externalisés
Triple Graph Grammars [180, 80]	Eléments de modèles	Correspondances (2 à 2)	Graphe de correspondances	Reconstruction, approche déclarative
Comprehensive systems [193]	Modèles vus comme des graphes	*Commonalities* (points communs)	Un modèle qui regroupe toutes les *commonalities*, pour la vérification de consistance + la réparation	Déclenchement manuel, externalisés
Mégamodèles				
Approches standard [34, 73, 107, 176, 35]	Modèles et métamodèles, modèles conformes au MOF (MDA) ou issus d'espaces technologiques [136]	Typologie de liens prédéfinis ou redéfinissables, granularité au niveau modèle, les liens portent les transformations	Réifiée dans une instance de mégamodèle, peut être éventuellement accompagnée d'un modèle de traces	Non explicitement réifiés dans les mégamodèles, mais externalisés dans les transformations et l'outillage
Mégamodèles hiérarchiques [182]	idem	Granularité au niveau élément de modèle	idem	idem
Generalized Discrimination Networks (GDNs) [18]	Modèles EMF	Opérations + query + transformations, approche modulaire et hiérarchique, granularité au niveau élément de modèle	Pas de réification explicite	Déclenchement manuel, approche incrémentale

TABLE 2.1 – Synthèse des approches fédératives 1/2

	Entités fédérées	Liens	Fédération	Comportements
Approches multi-vues				
EMFView [40, 41], *EMFFacet*, *EMFProfiles* [142], *Epsilon Merge* [132], *ModelJoin* [44]	Modèles EMF	Réifiés dans une approche de métamodélisation (concepts/instances définis dans des *ViewType/View*)	Réifiée dans des instances de vues (*virtual models*) conformes à des *ViewType* et *Viewpoint*	Limités à l'outillage lié à la mise en œuvre, accès en lecture et parfois en écriture pour certaines approches
Autres approches				
Model globalization [52]	Langages de modélisation	Métamodélisés	Non explicitement réifiée	Composition de la sémantique des langages
Glue model [165]	Modèles dans des technologies hétérogènes	Relations binaires, liens ontologiques, pas d'accès aux objets mais à des références	Un *"Glue Model"* de nature ontologique (raisonnement automatique)	Pas de comportements
Role4All [178]	Notion abstraite de modèle dans le prototype Smalltalk, avec une notion d'adaptateur	Métamodélisés par des rôles et dotés de comportements	Réifiée dans un modèle de rôles	Comportements explicitement réfiés dans les rôles
Syndeia [15]	Modèles métiers divers (mathématiques, modèles 3D, exigences, documents, tableurs)	Typologie de relations binaires basées sur les modèles, granularité niveau élément de modèle	Un *Total System Model* décrit en SysML	Comportements implicitement liés aux relations inter-modèles, mais pas explicitement réifés
Reactive Links [173]	Abstraction de modèles typés, avec une notion d'adaptateurs (plugin)	Instances de relations binaires liées à des propriétés	Modèle de liens (plateforme *DesignSpace*)	Architecture orientée évènements : les liens permettent le déclenchement d'opérations
FTG+PM++ [164, 150]	Modèles dans un langage de modélisation (avec la notion d'adaptateur)	Transformation liant deux artefacts (modèles)	"Modèle" *FTG+PM++*	Comportement intrinsèque à une transformation
GraphQL Federation [194]	Modèles de domaine vus comme des web-services *(GraphQL WS interfaces)*	Réifiés dans la notion de correspondances (métamodélisation via un DSL spécifique)	Modèle de correspondances vu comme un *EndPoint*	Pas de comportements (à l'exception des *queries*)

TABLE 2.2 – Synthèse des approches fédératives 2/2

un modèle mettent en place une notion d'adaptateur ou de *plugin* chargé de rendre accessible une source d'information au sein d'un espace technologique donné.

Selon les approches, la réification des liens est plus ou moins explicite. Ces liens concernent des relations souvent binaires pour les approches qui offrent une typologie de liens (mégamodèles, *TGGs*, *Syndeia*, *Reactive Links*, *FTG+PM++* etc.). La réification de liens n-aires s'accompagne souvent d'une vision métamodélisée (*Comprehensive Systems*, *EMF-View*, *GraphQL Federation*). La granularité des liens est très différente selon les approches, depuis les mégamodèles qui ne gèrent que des relations entre modèles [10] à *Reactive Links* qui s'appuie sur des valeurs de propriétés des éléments de modèles.

La fédération en elle-même est très peu souvent explicitement réifiée au sein d'un artefact exploitable d'un point de vue opérationnel. En général, les entités fédérées et les liens ne coexistent pas au sein du même espace de modélisation.

Très peu d'approches proposent une réification des comportements liés d'une part au cycle de vie des modèles fédérés, et d'autre part aux liens de fédération eux-même. Citons l'exception notable pour le premier cas de *Reactive Links* qui fournit une architecture orientée évènements qui permet des déclenchements de mise à jour automatisés relatifs à l'évolution des modèles.

2.5 Approches de métamodélisation

L'état de l'art des approches fédérative a montré l'intérêt de la métamodélisation dans le cadre de la réification des liens, pour l'élicitation de concepts métiers, et enfin pour la gestion de relations n-aires. Nous nous proposons ici de faire un rapide tour d'horizon de ce domaine. Au regard des fonctionnalités évoquées dans la figure 2.13, nous nous concentrerons sur des approches qui incluent un certain degré de flexibilité, ainsi que sur les approches qui prennent en charge la description de comportements.

2.5.1 Outils de métamodélisation

De nombreuses approches et outils différents pour la métamodélisation ont été proposés au cours des dernières décennies [121, 188]. Ces (méta-)outils sont parfois désignés sous le vocable de *metaCASE tools*. Les outils de type CASE (pour *Computer Aided Software Engineering*) concernent des environnements graphique permettant de modéliser. Le préfixe "*meta*" fait référence à l'outillage permettant de créer de tels outils, c'est-à-dire des métamodèles outillés. Concrètement, un atelier de métamodélisation permet de créer des langages de modélisation spécifique à un domaine (DSML), avec une syntaxe textuelle et/ou graphique, ainsi que des environnements d'édition et de traitement (génération, analyse, transformation) d'instances de ce/ces langage(s). L'objectif des outils de type *metaCASE* est de capturer la spécification de l'outil CASE requis, puis de générer cet outil à partir de cette spécification. La fonctionnalité de base de toute approche ou outil de métamodélisation comprend donc les moyens de décrire les métamodèles en utilisant un ensemble d'abstractions.

10. à l'exception notable des mégamodèles hiérarchiques

MetaEdit+ [198] est un *framework* de métamodélisation basé sur *Smalltalk*. Entre autres caractéristiques classiques (support pour la création de langages de modélisation spécifiques à un domaine, y compris les éditeurs et outils associés), *MetaEdit+* inclut une API qui peut être utilisée pour manipuler les modèles. Cette API permet la création de comportements d'exécution tels que la représentation, l'édition et l'animation de modèles. Elle permet également la définition de transformations, de générateurs ou la production d'analyses diverses. À l'instar de beaucoup d'autres solutions, cette définition de comportements est essentiellement externe au langage de modélisation produit, et s'appuie sur un outillage dédié.

$ATоM^3$ [144] est un atelier de métamodélisation similaire, avec le langage Python comme technologie sous-jacente. Il utilise des grammaires de graphes et la réécriture de graphes afin de fournir une sémantique opérationnelle aux modèles (dans un contexte de simulation). Dans la communauté des bases de données, *ConceptBase* [119] et son langage *Telos* [122] présentent des capacités de métamodélisation et la possibilité de définir des comportements via la notion de *règle active ECA* (Evènement/Condition/Action) qui permettent de décrire des contraintes d'intégrité. D'autres approches, telles que *GME* [148] ont recours à la génération de code par le biais de traducteurs afin de fournir un comportement aux modèles.

MetaDONE est un autre atelier de métamodélisation pour la création de DSML, issu des travaux de Englebert et al. [68], et qui s'appuie sur une base conceptuelle très simple et sensiblement différente du MOF où tous les concepts (*metaobjects*, *metaproperties*, *metarelations*, et *metamodels*) sont des *metaobject*. Les relations entre les métamodèles sont explicitement réifiées et l'outillage est entièrement interprété, y compris pour les aspects représentation et notation. Ainsi les langages de modélisation décrits par les utilisateurs peuvent définir plusieurs syntaxes concrètes, définies de manière déclarative.

Dans [161], les auteurs présentent *Kermeta* [11], qui se présente comme un langage de métamodélisation exécutable. Cette approche est couplée à un environnement de développements de métamodèles basé sur EMOF au sein de l'écosystème Eclipse. Comme toutes les approches de métamodélisation, *Kermeta* permet de décrire des métamodèles d'un point de vue structurel, mais il offre surtout la capacité de définir et de tisser du comportement dans les modèles EMOF. Il permet ainsi de définir et d'outiller de nouveaux DSL en améliorant la manière de spécifier, simuler et tester la sémantique opérationnelle des métamodèles. Les versions les plus récentes de *Kermeta* (Kermeta 3 - K3) [12] s'appuient sur *XTend*, un dialecte de Java qui permet également d'étendre les modèles existants avec de nouveaux comportements.

L'initiative Gemoc [13] s'intéresse aux problématiques de composition de langages exécutables, à travers une proposition scientifique appelée *Globalization of Modeling Languages* [52]. Les auteurs postulent que de nombreux langages de domaine, exécutables et hétérogènes, sont nécessaires pour capturer un système et son comportement. Ces travaux ont conduit au développement d'une infrastructure logicielle basée sur la plate-forme Eclipse (*Gemoc Studio*), et qui propose des composants logiciels orientés métamodélisation qui permettent de représenter et de manipuler des langages, au niveau définition et au niveau exécution. Ces composants permettent d'établir différents types de relations entre des langages (mise en correspondance, intégration, composition). Dans ce contexte, l'outil

11. https://kermeta.org/
12. https://diverse-project.github.io/k3/
13. https://gemoc.org/

Melange [57] s'apparente à un métalangage généraliste qui permet la spécification et la conception modulaire de DSLs . Le langage *BCool* est quant à lui défini comme un langage de coordination pour spécifier des opérateurs comportementaux afin d'automatiser la coordination de l'exécution de langages de modélisation [145]. Cet atelier est basé sur une phase générative pour définir l'ensemble des outils associés, et en particulier la coordination.

2.5.2 Approches de métamodélisation "flexibles"

Nous nous intéresserons ici plus particulièrement à la nature flexible des approches de métamodélisation. Nous désignons par *flexibilité* la capacité des approches et des outils de modélisation à relâcher (ou à assouplir) les contraintes au cours du processus de modélisation, de façon plus ou moins temporaire. Il est en effet parfois utile, pour des raisons d'expressivité, d'assouplir la contrainte de conformance en autorisant le typage des éléments du modèle a posteriori.

L'outil *FlexiSketch*, proposé par Wuëst et al. [210] illustre une première approche de modélisation qui fonctionne par des exemples. La solution s'appuie sur des outils de dessin, utilisés pour identifier des instances de modèles, tandis qu'un métamodèle est implicitement construit. Les auteurs remarquent qu'"il existe peu d'approches de métamodélisation qui s'adressent à l'utilisateur". L'une des raisons est que, du point de vue de la métamodélisation, on a longtemps cru que la métamodélisation ne devait être effectuée que par des experts en métamodélisation [130]. Ces auteurs soulignent également que la modélisation et la métamodélisation sont des activités proches, et qu'une connaissance du domaine est une compétence primordiale pour les effectuer. En ce sens, il apparaît pertinent de s'intéresser à la co-construction des modèles et de leur méta-modèle.

Dans [112], N. Hili explore cette approche en proposant *FlexiMeta*, un *framework* de métamodélisation destiné à promouvoir plus de flexibilité et de créativité en assouplissant les contraintes de vérification de conformance des modèles et des métamodèles en cours d'édition. Il préconise moins de couplage entre les modèles et les métamodèles afin de rendre possible la création par l'utilisateur de modèles et de métamodèles dans un ordre arbitraire.

La rigidité des processus de métamodélisation tient aussi dans la séparation stricte des niveaux conceptuels (instance, modèle, métamodèle, etc...). Les approches multi-niveaux proposent des alternatives à cette rigidité en préconisant par exemple l'utilisation d'un nombre flexibles de niveaux conceptuels et de relations entre ces niveaux [9], ou encore la généralisation de l'utilisation à la fois de l'instanciation linguistique et ontologique. Citons à nouveau dans cette catégorie l'outil *MetaDONE* évoqué plus haut, qui offre un support de modélisation multi-niveaux qui permet de "traverser" les niveaux conceptuels stricts (cf les niveaux M0, M1, M2 et M3 du MOF), et donc de manipuler des concepts réifiés à la fois au niveau modèle et au niveau métamodèle.

La "flexibilité" des approches de métamodélisation décrite plus haut est très liée à la notion de "typage" des modèles et des métamodèles. La flexibilité de la relation de typage est explorée dans [143], où les auteurs décrivent et formalisent un mécanisme de typage *a posteriori*. Dans leur approche, un modèle peut être *instance de* plusieurs métamodèles. Ils ne conservent qu'une partie de la règle stricte de métamodélisation : un modèle est une

instance d'un métamodèle unique de création (MM_C). Mais on peut définir aussi d'autres métamodèles (appelés *role metamodels* MM_R) dont ce modèle est instance.

2.6 Conclusions

Cet état de l'art nous a amenés à cheminer dans les mécanismes de la construction et du partage de la pensée humaine, depuis la connaissance jusqu'à la modélisation et aux mécanismes de collaboration intellectuelle autour de nombreux modèles.

L'interopérabilité sémantique de modèles multiples lève de nombreuses questions, et s'avère être au cœur d'un nombre important de travaux de recherche et de publications. *(Global) Model management*, *Inter-modelling*, *Megamodels*, *Multi-View / Viewpoint Modelling*, *Multi-Paradigm Modelling*, *Model Globalization*, *Model Alignment*, *(Bidirectional) Transformations*, *Model Mapping*, *Model Federation* : nous avons tenté de dresser un panorama de ces différents mots-clefs et domaines de recherche, de les clarifier et de les mettre en relation au regard de différentes perspectives et axes de recherche, et notamment du diagramme de caractéristiques présenté figure 2.13, page 34.

Les problématiques levées concernent tout d'abord des considérations techniques, notamment autour de l'hétérogénéité technologique des sources d'information. Un autre enjeu couramment évoqué réside dans les mécanismes d'interprétation de la donnée, et notamment dès lors qu'il s'agit d'aligner des modèles construits dans des contextes ou des intentions différentes. Dans ce contexte, nous nous sommes en particulier intéressé aux approches de métamodélisation. Un aspect important soulevé par de nombreux travaux concernent par ailleurs la nature dynamique et la gestion de consistance liée au cycle de vie de modèles qui évoluent. Des considérations méthodologiques et d'ordre sociotechniques complètent ces différentes thématiques.

Nous nous proposons dans le chapitre suivant de préciser et d'expliciter nos problématiques de recherche à la lumière des enseignements tirés de cet état de l'art.

Chapitre 3

Fédération de modèles

L'état de l'art du chapitre précédent introduit la problématique de l'interopérabilité sémantique des modèles. Nous nous proposons dans la section 3.1 de préciser cette problématique en posant trois principaux enjeux scientifiques, complétés par quatre principes et exigences supplémentaires à respecter. L'énoncé de notre problématique est illustré dans la section 3.2 par cinq scénarios de modélisation typiques qui couvrent un certain nombre de pratiques courantes de l'IDM, et qui vont servir de fil conducteur à la présentation de nos travaux. Nous nous attacherons dans la section 3.3 à définir ce qu'est la *fédération de modèles*, en proposant une base formelle et un métamodèle de l'approche. La section 3.4 présente des potentielles mises en œuvre de cette approche abstraite sur les cinq scénarios de référence, et leur intérêt au regard des trois enjeux identifiés plus haut. Nous concluons dans la section 3.5 en évoquant le framework MF^2 qui propose une sémantique opérationnelle pour cette approche.

Sommaire

3.1 Enjeux et problématique

Le tour d'horizon proposé dans l'état de l'art sur les pratiques et les usages de la modélisation nous a amené à questionner la problématique de l'interopérabilité sémantique des modèles, en tant que productions intellectuelles créées, maintenues et utilisées par des êtres humains. Au delà de simples problèmes techniques, ces modèles réifient des métiers différents, avec des concepts et des interprétations de ces concepts qui sont spécifiques à ces métiers. Ces modèles sont, en outre, partie intégrante de processus d'ingénierie différents, avec leur propre pile technologique, leurs outils dédiés, et des pratiques et des cycles de vie spécifiques.

Les modèles ne sont donc pas des éléments isolés, et doivent être considérés comme étant au centre d'un réseau de dépendances avec d'autres modèles. Dans ce contexte, agir sur un modèle a un impact sur un ensemble de modèles dépendants. La plupart des activités fondées sur le paradigme de l'Ingénierie Dirigée par les Modèles (IDM) requièrent de traiter simultanément de nombreux modèles. Nous avons vu dans l'état de l'art de nombreuses approches et de nombreuses techniques permettant cette mise en œuvre (*intermodelling*, *megamodelling*, *model management*, etc...). De nombreuses approches offrent un cadre permettant de considérer dans un même espace de modélisation à la fois les modèles (et les métamodèles) ainsi que les éléments de modèles. Cependant, à notre connaissance, il n'existe pas d'approches qui proposent un cadre permettant de définir les relations inter-modèles comme des objets ayant le même statut que les modèles et les éléments de modèle, et dont les intentions (y compris la sémantique et les comportements) peuvent être entièrement spécifiées. Un tel cadre pourrait permettre aux modélisateurs de traiter divers scénarios de modélisation de manière uniforme et flexible.

Plus particulièrement, nous nous proposons de travailler sur trois principaux enjeux que nous identifions comme primordiaux dans le contexte de l'interopérabilité de modèles :

— **Enjeu E1** : la problématique de l'**interprétation** de la donnée,
— **Enjeu E2** : la prise en compte de l'**hétérogénéité**,
— **Enjeu E3** : la nécessité de manipuler des **modèles vivants**.

Nous détaillerons dans un premier temps ces trois enjeux, avant de les illustrer et de les justifier au travers de la présentation de situations prototypiques de l'ingénierie de la modélisation.

3.1.1 Enjeu E1 : interprétation de la donnée

Ce premier enjeu, évoqué à plusieurs reprises dans l'état de l'art, découle en grande partie de la gestion d'un certain implicite dans les mécanismes d'interprétation humaine. Cet implicite découle d'un nécessaire consensus entre un groupe de personnes qui partagent un sujet de préoccupation. En revanche si cet implicite est utile et primordial aux activités humaines, il peut se révèler être un problème dans le traitement automatique de la donnée, par les points de variation sémantiques qu'il implique. Cet implicite s'incarne généralement dans des langages et des métamodèles, des normes ou des standards, des outils, des processus métiers, des méthodologies, etc. Il peut être encore moins formel et découler d'habitudes et de pratiques.

Une première approche, pragmatique, consiste à généraliser l'utilisation de ces référentiels (langages, standards, "bonnes pratiques"), et à en uniformiser leur interprétation et

leur manipulation par leurs utilisateurs (construction d'un consensus). Cependant, cette approche est peu raisonnable et "ne passe pas à l'échelle" quand il s'agit de faire interopérer des modèles, des ingénieries ou des métiers qui ne sont pas de la même culture.

Il est sans doute illusoire de vouloir donner un sens universel unique à une information. Plutôt que de viser l'alignement de sémantiques d'interprétation, il nous semble raisonnable (et intéressant) de relâcher cette contrainte (d'un sens universel) pour se pencher sur **un cadre conceptuel permettant de réifier l'interprétation de la donnée, pour un contexte spécifique**. Cette vision propose la définition d'une sémantique locale et contextuelle, et donc une certaine externalisation d'une interprétation d'une donnée (un modèle). Ce découplage permet à la fois de considérer un cadre formel [1] pour la donnée et une (ou des) interprétation(s) contextuelle(s).

Cette approche permet la mise en œuvre de la vision *"tout est modèle"*, qui consiste à pouvoir considérer n'importe quelle donnée comme un modèle, à condition de se donner les moyens de l'interpréter [2].

3.1.2 Enjeu E2 : la prise en compte de l'hétérogénéité

L'hétérogénéité dont il est fait mention dans ce second enjeu est multiple. Elle est tout d'abord **technique et syntaxique**. L'interopérabilité sémantique requiert évidemment de pouvoir connecter des technologies très diverses, qui manipulent des paradigmes syntaxiques non standardisés. Elle est également **conceptuelle et sémantique**, puisqu'il s'agit de mettre en relation des univers conceptuels et/ou techniques différents (en tout état de cause construits dans des contextes variés et avec des intentions différentes). Du point de vue de la **pragmatique**, l'hétérogénéité est enfin liée à des métiers, des pratiques, des ingénieries très différentes.

De nombreuses technologies, des piles technologiques incompatibles, des paradigmes divers, des métiers différents et des ingénieries qui n'ont pas forcément l'habitude de travailler ensemble : l'interopérabilité sémantique de sources d'information hétérogènes nécessite de pouvoir aborder ces multiples problématiques.

3.1.3 Enjeu E3 : des modèles "vivants"!

Ce troisième enjeu résulte du constat majeur que les modèles ne sont pas des artefacts figés. Un modèle est construit, édité, validé, utilisé, mis à jour, vérifié, compilé, traité, transformé, généré, etc. L'usage est parfois de considérer les modèles comme des artefacts purement structurels, destinés uniquement à sérialiser de la donnée, alors que **ces modèles sont dynamiques**. Peu de modèles embarquent leur propre sémantique et leurs comportements, alors qu'ils ont été imaginés dans le cadre de processus d'ingénierie (ou plus généralement d'activités humaines). Très souvent, ce sont les outils qui permettent de manipuler ces modèles (ou les piles technologiques liées) qui encodent implicitement les comportements permettant de faire vivre ces modèles. Il faut y voir une référence à l'implicite dont il est fait mention dans le premier enjeu.

1. et prescriptif d'un point de vue linguistique
2. ce que permet ici le cadre conceptuel de réification de l'interprétation

L'interopérabilité sémantique de ces modèles nécessite donc de considérer l'**implicite comportemental lié au cycle de vie des modèles considérés**, et donc leur place dans le processus d'ingénierie dans lequel ils sont impliqués [3]. Dans ce cadre, il nous paraît pertinent de pouvoir prendre en compte explicitement le comportement des modèles à faire interopérer.

Inversement, nous identifions un intérêt majeur à disposer de modèles qui embarquent leurs propres comportements (leur façon d'être manipulés), et donc indirectement du processus d'ingénierie qui sous-tend la nécessité de faire interopérer des modèles.

3.1.4 Principes et exigences supplémentaires

À ces trois enjeux s'ajoutent des contraintes et exigences supplémentaires, que l'on devra respecter dans tous nos travaux, et qui contraignent l'espace des solutions que nous explorons. Ces principes fondent l'acceptabilité de l'approche par des organisations humaines, pour des raisons évidentes de pragmatisme. Le contexte impose de travailler à une approche applicable à un environnement existant et pérenne (sources d'information et modèles, outils, processus, métier).

1. **Autonomie du cycle de vie des sources d'information [4] fédérées** : il s'agit ici de ne pas remettre en cause les métiers, processus et outils existants, et donc de considérer que chacun des modèles est autonome et dispose de son propre cycle de vie. Ce cycle de vie pré-existe à la mise en œuvre de la fédération et doit pouvoir être conservé [5].
2. **Connectivité intermittente** : le lien peut être temporairement "cassé" (si les sources d'information fédérées ne sont pas mutuellement accessibles). C'est à la reconnexion que les liens doivent être reconstitués (et éventuellement resynchronisés après une phase de réconciliation).
3. **Non intrusivité** : ce principe est un corrolaire technique du principe de l'autonomie du cycle de vie des sources d'information fédérées, et implique que les mécanismes d'interopérabilité ne peuvent pas reposer sur l'insertion de données non persistantes du point de vue des outils utilisés pour les manipuler [6].
4. **Gestion de la source de vérité et minimisation de la redondance conceptuelle** : cette préoccupation est d'ordre méthodologique, et concerne le nécessaire compromis à trouver entre le besoin de faire interopérer des sources d'informations multiples qui référencent les mêmes éléments conceptuels, et donc avec de potentielles redondances et contradictions, et la nécessité de gérer une "source de vérité", c'est-à-dire d'arbitrer d'éventuels conflits.

Des préoccupations telles que le passage à l'échelle et le pragmatisme de l'approche du point de vue de son acceptabilité conceptuelle, méthodologique et ergonomique par des concepteurs, des gestionnaires et des utilisateurs complètent l'énoncé de notre problématique.

3. Indirectement, c'est toute l'ingénierie sous-jacente qui doit être considérée ici.
4. On confond ici sources d'information et modèles.
5. même s'il peut être étendu ou adapté
6. Certaines technologies permettent de manipuler des métadonnées, parfois invisibles pour l'utilisateur. Si l'outillage associé persiste (stocke) ces métadonnées, ces dernières sont cependant utilisables.

3.2 Scénarios illustratifs

Nous nous proposons d'illustrer nos problématiques en présentant cinq scénarios de modélisation typiques qui couvrent une grande partie des usages de l'IDM. Ces scénarios impliquent des relations entre des modèles et une forme d'interopérabilité sémantique entre ces derniers. Pour chacun de ces scénarios, qui vont servir de fils conducteurs à la présentation de nos travaux, nous fournirons un exemple permettant de le contextualiser, et proposerons une discussion mettant en avant les 3 enjeux présentés plus haut.

3.2.1 Scénario A : génération de code à partir du modèle (avec *round-trip*)

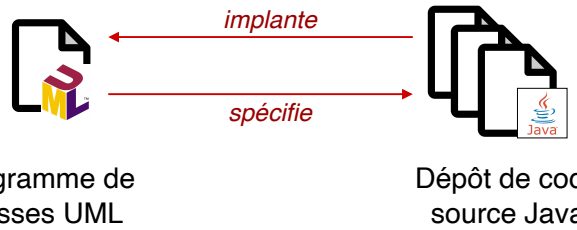

FIGURE 3.1 – Scénario A : génération de code à partir du modèle (avec *round-trip*)

a. Scénario

Ce premier scénario est extrêmement commun dans le monde de l'IDM et symbolise parfaitement une approche où un modèle pré-existe à son implantation automatiséee par un mécanisme de génération de code. Cette génération de code s'apparente à une transformation de modèle, qui spécifie la traduction d'éléments d'un modèle d'entrée en éléments d'un modèle de sortie. Il s'agit ici de raffiner et de traduire en code des éléments conceptualisés dans un (ou plusieurs) modèle(s), en faisant des choix de conception du point de vue de l'implémentation.

b. Exemple

La figure 3.1 illustre un tel scénario en montrant la relation entre un diagramme de classe UML (qui spécifie un métamodèle) et un dépôt de code source Java (qui implante cette modélisation).

Bien que très classique, ce scénario invite à s'interroger sur la problématique de la co-évolution du modèle et du code généré. En effet, dans la perspective d'une conception incrémentale, si la première itération ne pose aucun souci, les étapes suivantes du processus de conception nécessitent des aller-retours entre le code modifié par le(s) développeur(s) et le modèle maintenu par le(s) modélisateur(s).

c. Discussion

Les solutions à ces problèmes de génération de code sont souvent cachées dans des outils ou intégrées dans des processus métier et métamodèles avec peu de possibilités d'adaptation ou d'évolution (enjeu E1). D'autres solutions reposent sur des langages de contrainte et/ou des langages de transformation externes. Un concepteur doit choisir entre une solution intégrée dans l'outil de modélisation mais fermée et une approche ouverte mais externe.

Ces solutions reposent souvent sur une homogénéité technologique (par exemple dans le monde EMF), qui permet le développement d'outils dédiés, mais devient difficile à traiter dans des environnements techniques hétérogènes (enjeu E2). Enfin, ces solutions nécessitent une gestion explicite de la mise à jour de ces modèles via des processus explicites (enjeu E3).

Dans le monde de l'IDM, ce scénario est classiquement traité par des mécanismes de génération de code basé sur des transformations [197, 148, 161] et a été plus que largement exploré. La problématique du *"round trip"* fait parfois appel à des solutions ad hoc, avec par exemple la gestion de meta-données ou commentaires indiquant aux développeurs les portions de code intégralement générées et auxquelles il ne faut pas toucher (mécanismes de ségrégation). D'autres solutions de maintien de la cohérence entre modèle et code généré sont explorées par les transformations bi-directionnelles [91].

3.2.2 Scénario B : construction de modèle(s)

a. Scénario

L'une des activités clés de l'IDM est la création de modèles en tant qu'abstractions de systèmes. Les modèles ainsi construits sont valides (et utiles au-delà de la simple documentation) parce qu'ils se conforment à une certaine structure représentant la syntaxe abstraite du domaine, qui est décrite dans un métamodèle [78]. Mais les métamodèles sont eux-mêmes des modèles (ils se conforment aux métamétamodèles). La conformance peut être vue comme une relation entre les modèles.

Généralement, la relation de conformance (notée χ) entre un modèle et son métamodèle implique un processus dans lequel le métamodèle est construit en premier et devient prescriptif. Cette rigidité n'est pas toujours adaptée au processus de modélisation [7] et/ou à des situations de modélisation spécifiques [26].

Les modèles et les métamodèles peuvent être, par exemple, co-construits de manière itérative. Le processus peut même être inversé, de sorte que les métamodèles sont construits à partir d'exemples d'instances. Ainsi, des mécanismes plus sophistiqués ont été conçus pour modifier la sémantique de la relation et préconiser une plus grande *flexibilité* [98]. Cela inclut l'utilisation de plusieurs niveaux d'instanciation (parfois configurables) [5, 85] et un relâchement de contraintes de conformance [184], voire même un report de la vérification de ces contraintes [143].

7. ou limite drastiquement la capture du métier

b. Exemple

La figure 3.2 illustre ce cas de figure d'une co-construction d'un modèle et de son métamodèle. Ce cas d'utilisation s'applique ici à un service d'une entreprise dont on souhaite modéliser l'organisation. L'idée est de s'appuyer sur une instance (ici un service particulier), pour construire le métamodèle associé en identifiant les concepts organisationnels, avant de les confronter avec une nouvelle instance afin de vérifier leur pertinence[8]. Ce scénario illustre un cas où χ est défini en tant que résultat du processus de modélisation.

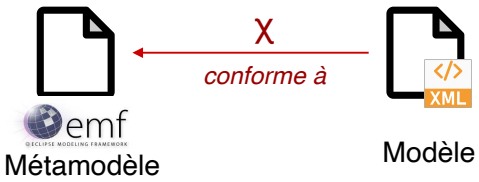

FIGURE 3.2 – Scénario B : co-construction d'un modèle et de son métamodèle

c. Discussion

Les approches classiques, basées sur la modélisation sans gestion des dépendances entre les modèles et les entités du modèle, ne peuvent pas offrir cette flexibilité. L'outillage dédié à la construction de métamodèle et celui dédié à l'édition des modèles est en général distinct. Ces derniers outils ne permettent que le développement de modèles conformes à un métamodèle, dans une technologie donnée (enjeu E2). Et généralement, ce métamodèle ne permet pas de gérer les dépendances, ni d'associer un comportement (une sémantique) à une dépendance (enjeu E3). La sémantique de la conformance est intégrée et codée en dur dans les outils (enjeu E1), et non dans le métamodèle.

La problématique de la co-évolution (étude de la dynamique de la relation de conformance pour des instances de modèles existantes dans le contexte de l'évolution du métamodèle) est plus largement étudiée [110].

3.2.3 Scénario C : composition de modèles

a. Scénario

Ce scénario implique la construction d'objets métier à partir de plusieurs sources de données ou modèles. Ce cas de figure se rencontre couramment au sein de systèmes d'information conçus comme des briques applicatives liées à un bloc de fonctionnalités. Une fonctionnalité transversale non prévue a priori, ou impliquant des composants non interopérables, peut alors nécessiter de construire des objets métier à partir de plusieurs sources d'information. Ces mécanismes sont proches de la notion de jointure des bases de données, avec la gestion d'une hétérogénéité technologique et la problématique d'identifier des clefs.

8. Nous appelons cette approche *free modelling* et nous la détaillerons dans la section 6.4

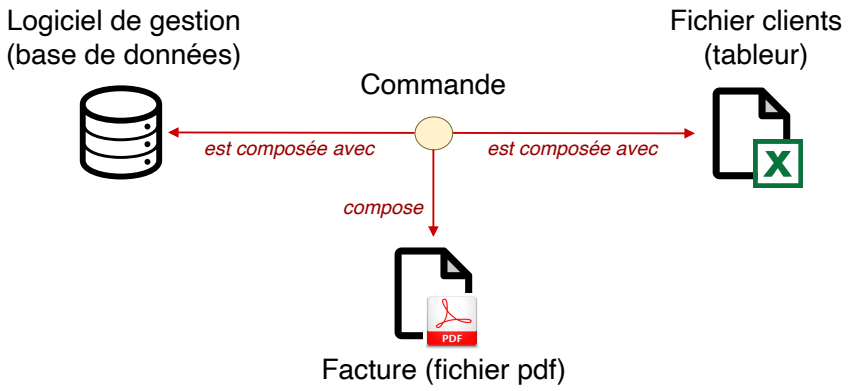

FIGURE 3.3 – Scénario C : composition de modèles

b. Exemple

La figure 3.3 présente l'exemple de mise en relation d'une base de données d'un logiciel de gestion (type ERP), avec un fichier client encodé dans un tableur pour éditer une facture (un fichier PDF). Les données dans le tableur sont ici propres à la gestion de la relation client de l'entreprise considérée, qui n'a pas fait le choix d'encoder cette relation-client dans son logiciel de gestion [9]. Le concept métier de *"Commande"* est ici construit comme une aggrégation de données issues à la fois du logiciel de gestion et du fichier client du tableur, et c'est ce concept qui sera utilisé pour construire le document de facturation.

c. Discussion

L'exemple précédent traduit des pratiques très courantes du monde de l'entreprise, où pour des raisons de simplicité, de flexibilité, d'agilité ou de formation, la bureautique se substitue souvent aux logiciels de gestion complexes.

Ce scénario est à rapprocher de la pratique du *Single Source of Truth* ou SSOT [168], qui consiste à rassembler toutes les données d'une entreprise ou d'un projet à un seul et même endroit. Ce système de collecte et de sauvegarde de l'information est nécessaire dès lors qu'il existe de multiples interfaces qui dupliquent les données au sein de l'organisation. L'idée est que tout le monde puisse avoir accès à la même information, la plus récente et la plus précise possible. Cette pratique nécessite une certaine centralisation des données et une adaptation des processus afin que ces données soient régulièrement actualisées et accessibles (enjeu E3).

La problématique d'hétérogénéité technologique se pose ici naturellement (enjeu E2). Le problème d'identifier des concepts métiers dont les données se retrouvent dans plusieurs sources d'information (modèles) se pose également (enjeu E1), avec la double difficulté de l'alignement sémantique de concepts métiers traités différemment dans plusieurs modèles, et la gestion d'identifiants permettant la recomposition de ces concepts (notamment pour la gestion de doublons). Enfin la gestion de la consistance de ces données peut être rendue délicate par l'utilisation d'outils indépendants (gestion du cycle de vie des données, enjeu E3).

9. qui a peut-être prévu cette fonctionnalité, mais dans un cadre rigide qui requiert que l'utilisateur se conforme exactement aux données et aux processus imaginés et encodés dans le logiciel.

Les solutions classiques à ce type de problème sont souvent traitées de manière ad hoc dans les outils utilisés, et via des fonctionnalités d'import ou d'export.

3.2.4 Scénario D : mise en correspondance de modèles (*model mapping*)

a. Scénario

Les pratiques de l'IDM conduisent souvent à établir des correspondances entre les éléments de deux (ou plusieurs) modèles, par exemple pour la traçabilité ou la vérification de la cohérence, ou bien dans le cadre d'activités transversales chez les industriels qui conçoivent des systèmes. Nous appelons cette activité générique "mise en correspondance de modèles" (*Model Mapping*). Le recouvrement sémantique des modèles mis en correspondance peut être large ou très partiel. Ce type de besoins émerge généralement dès lors qu'il y a une mise en commun d'ingénieries traitées en silo et notamment dans le cadre de la conception de systèmes ou de systèmes de systèmes. Cette problématique de mise en correspondance de modèles est également éprouvée par les pratiques de SSOT, dès lors que des services offrent de la redondance en terme de données, avec une nécessaire gestion de la mise en cohérence (gestion de la consistance et de la mise à jour de données).

b. Exemple

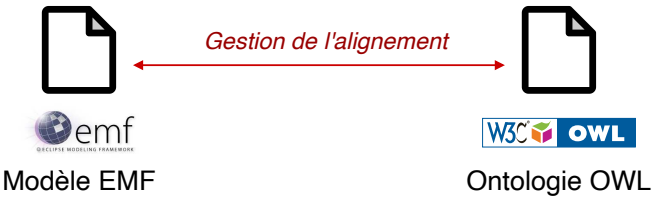

FIGURE 3.4 – Scénario D : mise en correspondance de modèles (*model mapping*)

La figure 3.4, issu d'un cas d'utilisation réel rencontré chez un systémier, présente l'exemple du développement d'une ingénierie en silo outillée dans un DSL (dans la technologie EMF), qu'il faut rapprocher d'une ontologie encodant des concepts métiers relatifs à la description du domaine considéré. Les concepts métier manipulés dans les deux modèles sont très proches, parfois totalement alignés, parfois partiellement. Un même concept métier peut ainsi requérir plusieurs instances de types différents, ou une instance particulière valuée spécifiquement, parfois encore c'est seulement un être humain qui peut établir ce rapprochement. Certains concepts métier présents d'un côté peuvent être absents de l'autre modèle.

c. Discussion

Cette problématique est souvent l'objet de travaux autour des transformations bi-directionnelles, qui est un domaine largement couvert. Le scénario présenté ici, impliquant les ontologies et les DSL EMF, fait l'objet de nombreux travaux [113, 117]. Les transformations bi-directionnelles ont été également élargies à des relations n-aires et des liens ont été établis avec la méga-modélisation [191].

Bien évidemment, l'enjeu E1 est au cœur de la problématique soulevée par ce scénario, dans l'expression de l'alignement sémantique. Il est double, et concerne (1) l'élicitation des concepts métiers à aligner, et (2) leur projection et leur définition dans chacun des modèles à considérer.

Les solutions classiquement mises en œuvre reposent souvent sur une homogénéité technologique (par exemple dans le monde EMF), qui permet le développement d'outils dédiés, mais devient difficile à traiter dans des environnements techniques hétérogènes (enjeu E2). Enfin, ces solutions nécessitent une gestion explicite de la mise à jour de ces modèles via des processus explicites (enjeu E3).

Ces solutions sont en général très efficientes dans le cas de d'alignement et de recouvrement total, mais posent le problème des concepts isolés et qui n'ont pas de pendant dans le ou les modèles opposés (comment représenter ce "pivot" sans perte d'information ?). Un autre problème classique est le recours à un être humain pour établir la mise en correspondance (enjeu E1), ce qui pose des problèmes d'outillage dans le cadre de transformations automatisées, et de la sérialisation de l'information liée à ces décisions humaines. Enfin, la gestion de la source de vérité devient également un enjeu dans la perspective de modèles "vivants" et mis à jour fréquemment.

3.2.5 Scénario E : édition de modèle(s)

a. Scénario

L'édition ou la visualisation de modèles utilise diverses syntaxes concrètes. Il est rare que les utilisateurs manipulent directement des éléments d'une syntaxe abstraite (c'est-à-dire le métamodèle). Nous nous concentrons ici sur l'édition de modèles à l'aide d'éditeurs graphiques. Dans ces outils, les représentations graphiques (par exemple dans le cas de figures ou de diagrammes, les formes et les connecteurs) correspondent à une notation donnée (syntaxe concrète) et sont présentées aux utilisateurs de manière à ce qu'ils puissent choisir, sélectionner et configurer un modèle au travers de l'édition de ces différentes représentations. Il s'agit ici de considérer les artefacts de représentations comme des modèles. En ce sens, il existe une dépendance (de représentation) entre une notation donnée et le modèle *domaine* qu'elle représente.

b. Exemple

L'IDM constitue un domaine privilégié pour les éditeurs de logiciels qui fournissent de nombreux outils d'édition graphique dédiés à la modélisation, notamment dans le cadre de la modélisation UML (classes, séquences, activités, etc...) ou SysML. La modélisation des processus d'entreprise (BPMN, BPEL) constitue également un domaine très représentatif de ces pratiques, parmi beaucoup d'autres. Les outils mis à disposition sont de bons exemple de l'utilisation de tels liens entre les formes graphiques et les concepts. La figure 3.5 illustre ce cas de figure sur l'exemple de l'édition d'un modèle UML au travers d'un diagramme de classes.

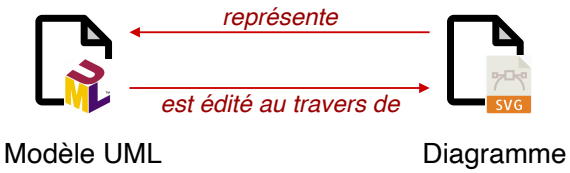

FIGURE 3.5 – Scénario E : édition de modèle(s)

c. Discussion

Les outils classiques fournissent des solutions éprouvées à l'édition de standards, mais s'avèrent plus limités pour l'édition de DSL. Parmi toutes les solutions permettant de relier les formes aux concepts, certaines sont intégrées, d'autres établissent une séparation claire entre le modèle et sa représentation (comme l'approche Sirius/GMF [92, 206] ou metaedit [198]). Cette dernière solution permet d'obtenir des solutions plus réutilisables et plus souples.

Une autre limitation courante de ces outils réside dans la capture de concepts métiers (ou d'extension de ces concepts, ou nouvelles propriétés spécifiques) pas forcément prévus à la conception du logiciel. L'expressivité est souvent limitée au cadre strict offert par le métamodèle sous-jacent, et à son interprétation implicite (enjeu E1). Les outils classiques n'offrent, en général, pas le choix des technologies utilisées (technologie du modèle édité, et technologie liée à la représentation, enjeu E2). Concernant les aspects comportementaux (enjeu E3), les outils classiques gèrent en général une nouvelle représentation suite à une mise à jour du modèle, mais ne permettent pas d'édition de la représentation hors du cadre offert par l'outil, pour un modèle donné.

3.2.6 Synthèse des motivations

Nous avons illustré comment les dépendances de modèles jouent un rôle central dans les activités les plus courantes de l'IDM. Nous avons pu mettre en évidence ces dépendances au travers de cinq situations classiques de modélisation. Sur chacun de ces scénarios, nous avons illustré les 3 enjeux présentés en introduction de ce chapitre, et analysé comment ces situations de modélisation étaient traitées dans les approches classiques.

Nous avons également examiné comment, dans ces approches, les relations entre les modèles sont soit codées en dur (par exemple, la relation de conformance), soit gérées par l'utilisation d'outils externes et/ou de solutions ad hoc (par exemple, la traçabilité des modèles), soit tout à fait implicites. Tout ceci nous amène à proposer une élicitation explicite de ces relations de dépendances, et leur réification, afin qu'elles aient le même statut que les modèles et les éléments de modèles.

Ces cinq scénarios nous ont également mené à considérer les aspects comportementaux de modèles dynamiques, mais aussi de la dynamique des dépendances entre ces modèles. Nous nous proposons donc également de travailler sur une explicitation de ces comportements.

Tout ceci fonde notre proposition sur une formalisation du paradigme de la *fédération de modèles*, en tant que cadre uniforme qui englobe ces différents scénarios et ces différentes préoccupations.

3.3 L'approche *"fédération de modèles"*

L'objectif de cette section est de présenter les bases formelles de l'approche *fédération de modèles*.

3.3.1 Le cadre posé par la fédération de modèles

La *fédération de modèles* a été identifiée dans la norme ISO-14258 [116, 133], qui énonce la définition de la fédération en regard des principes d'intégration et d'unification (cf état de l'art, section 2.3.4). L'intégration repose sur l'existence d'un modèle commun qui rassemble tous les modèles utilisés. Si les métamodèles originaux étaient différents, les modèles sont "adaptés" au métamodèle commun choisi (métamodèle original choisi, modèles adaptés). Les outils originaux peuvent être réutilisés au prix d'éventuels désalignements et adaptations sémantiques pour les modèles, par exemple lorsque certains concepts (métamodèle) sont absents ou légèrement différents. Au contraire, l'unification nécessite la création ou la réification d'un métamodèle commun afin que tous les modèles utilisés puissent être rassemblés sous le nouveau métamodèle (métamodèle adapté, modèles originaux). L'unification peut conduire à un remaniement de l'ensemble des outils.

Par rapport à ces deux approches, la fédération se concentre sur la modélisation de la sémantique des relations ou des dépendances entre les concepts dans des métamodèles hétérogènes (métamodèles originaux, modèles originaux, liens ajoutés). Dans cette approche, les outils originaux sont toujours utilisables, et un outil dédié à la fédération doit être développé.

L'objectif de la *fédération de modèles* est donc de réifier un ensemble de dépendances entre un groupe de modèles. Cette approche est illustrée dans la figure 3.6 où le disque complet représente la *fédération (de modèles)*. Il comprend plusieurs modèles décrits dans divers *espaces techniques* : les *modèles fédérés*. Le disque intérieur représente le *modèle de fédération*, un modèle conceptuel réifiant un ensemble de dépendances entre les modèles fédérés.

En utilisant la notation habituelle μ dénotant la relation de modélisation [162], nous pouvons écrire :

$$\text{modèle de fédération} \xrightarrow{\mu} \text{fédération de modèles}$$

Il convient de noter que les *modèles fédérés* peuvent évoluer indépendamment dans leur environnement technique d'origine grâce à des choix de conception qui maintiennent une forte séparation entre le modèle de fédération et les espaces techniques fédérés.

Un *modèle de fédération* n'est pas simplement descriptif (aspects statiques), il possède des comportements (aspects dynamiques). L'objectif de la définition d'une *fédération* est double. Premièrement, elle rassemble et organise les parties descriptives reflétant les dé-

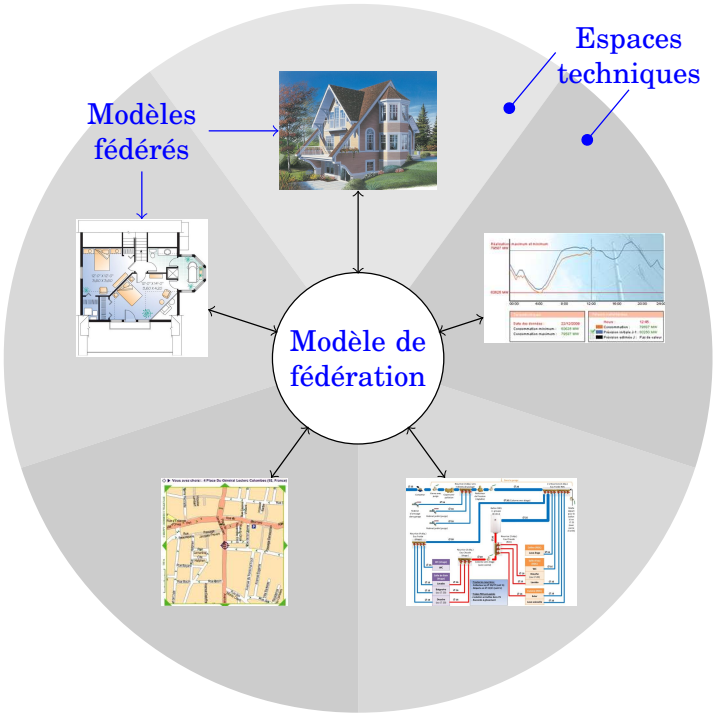

FIGURE 3.6 – L'approche "Fédération de modèles"

pendances. Deuxièmement, elle coordonne les comportements et rend possible des comportements spécifiques.

Une *fédération* sert une intention ou un but qui définit son objectif qui peut être par exemple de maintenir la synchronisation au sein d'un ensemble de modèles. Les intentions se traduisent également par la définition de comportements (actions associées aux objets et aux événements, actions d'édition ou de création d'éléments de modèle). Ces comportements peuvent être associés à n'importe quel élément (objets et/ou dépendances). Notons enfin que le modèle de fédération est lui-même un modèle et qu'il peut donc être manipulé comme tel.

L'abstraction *modèle de fédération* décrite ci-dessus ressemble à un *mégamodèle* [34]. En effet, un modèle de transformation au sein d'un mégamodèle pourrait être considéré comme une fédération des liens de correspondance d'une spécification de transformation de modèle (les *modèles fédérés* étant les modèles d'entrée et de sortie). Toutefois, dans l'approche de la mégamodélisation, la dépendance n'est généralement exprimée qu'au niveau du modèle, tandis que les dépendances au niveau de l'élément de modèle restent cachées dans le modèle de transformation. Inversement, dans l'approche de la fédération de modèles, un modèle de transformation correspondant serait représenté par une fédération de boîtes blanches contenant les dépendances de correspondance entre les éléments de modèle ainsi que les comportements qui réalisent la sémantique de la transformation. En outre, l'approche du mégamodèle maintient le mégamodèle et les transformations dans des espaces distincts, tandis que les modèles de fédération ont des comportements attachés au modèle.

3.3.2 Notion de modèle

Une instance de *modèle* est un *objet* [10] créé par un concepteur pour modéliser un autre "objet", l'objet d'étude. Être un modèle, c'est donc être en relation avec un autre "objet". Comme explicité dans l'état de l'art, cette relation de modélisation est généralement notée μ [162]. La prise en compte de multiples points de vue implique des relations μ qui relient plusieurs *modèles* à un seul *objet* étudié. À noter que les *objets* restent des abstractions, ils peuvent être des idées, des objets du monde réel et ne sont pas forcément des artefacts numériques.

Ces instances de modèle sont souvent construites en suivant certaines contraintes de construction définies dans un *modèle* (toujours un *objet*). Cette deuxième relation entre les *modèles* est souvent désignée par χ. Plusieurs variantes de cette relation de conformance existent dans la littérature, allant d'une relation stricte telle que l'approche stricte de métamodélisation [95] à une relation plus flexible telle que la métamodélisation multi-niveaux [5] ou la modélisation libre [86]. Dans ce cas, une instance de *modèle* est dite *conforme à* un *modèle*. Selon le type de conformance choisie, la cardinalité de χ côté modèle peut être de 1, 0..1 ou même $\star$ (cardinalité multiple).

Pour un *modèle*, le fait d'être en relation μ ou χ avec un autre *objet* ou *modèle* impose des contraintes sur son contenu. Par exemple, le fait de se conformer à un *modèle* contraint le type d'éléments sur lesquels le modèle est construit. Pour l'instant, les *objets* restent abstraits et les contraintes précises ne peuvent être énoncées. De plus, le fait que le contenu soit abstrait permet de couvrir différents paradigmes. Un modèle d'un autre *objet* peut être un diagramme structurel tel qu'un *diagramme de définition de bloc SysML* [105] ou il peut s'agir d'un modèle mathématique décrit par un ensemble d'équations différentielles issues de la mécanique newtonienne.

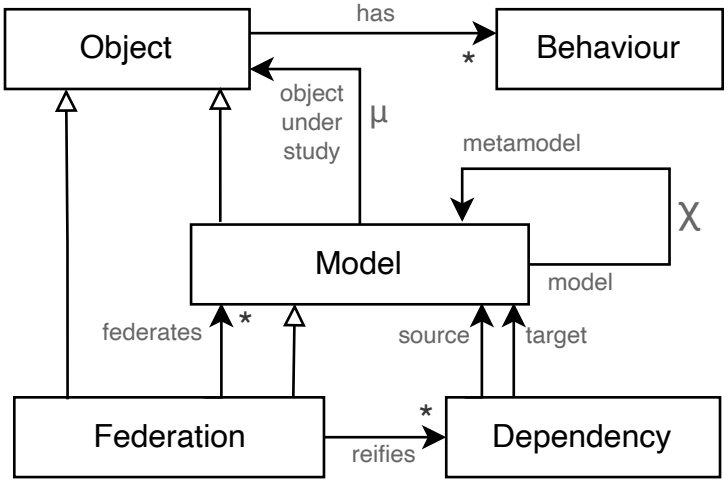

FIGURE 3.7 – Modèles et fédération : un métamodèle de l'approche

L'ensemble de tous les *objets* est noté $\mathbb{O}$. De manière abstraite, le contexte de modélisation repose sur un ensemble d'*objets*, $\mathcal{O} \subset \mathbb{O}$ et deux relations binaires sur $\mathcal{O}$, μ et χ. La figure 3.7 synthétise cette approche. On distingue ici les *modèles* (Model) des *objets* (Object) par le fait qu'ils autorisent des relations μ et χ. La cardinalité n'est pas spécifiée pour conserver le caractère générique de l'approche.

10. Nous ne parlons pas ici de technologie orientée objet, le terme "objet" est utilisé dans son sens premier.

Dans ce formalisme, le travail d'un modélisateur (un concepteur de modèles), consiste à définir un triplet $(\mathcal{O}, \mu, \chi)$ où le modèle m, le métamodèle M et l'objet étudié O sont des éléments de $\mathcal{O}$, et $(\mathsf{m}, \mathsf{M}) \in \chi$ et $(\mathsf{m}, \mathsf{O}) \in \mu$. Le modélisateur travaille avec un outil (un éditeur de *modèles*), qui permet d'exécuter des actions susceptibles de modifier $\mathcal{O}$, μ ou χ.

Il existe de nombreuses actions de modélisation. Par exemple, on peut vouloir créer une nouvelle instance de modèle m' modélisant un *objet* existant O' de $\mathcal{O}$ et se conformant à un modèle existant M'. Par la suite, il peut être nécessaire de modifier le contenu de m'. Le lecteur intéressé pourra consulter [26] pour une description plus systématique de ce que l'on appelle les *situations de modélisation*.

3.3.3 Notions de dépendances et de fédération

On définit une *dépendance* (Dependency) comme la réification d'une relation binaire entre deux *modèles* (Model), comme illustré sur la figure 3.7. Enfin on définit une *fédération de modèles* (Federation) comme un ensemble de modèles et un sous-ensemble de leurs dépendances. Les modèles regroupés dans une fédération sont appelés *modèles fédérés*.

La figure 3.8 représente un exemple d'une telle fédération. Dans cet exemple, nous fédérons un diagramme de classes UML, plusieurs diagrammes de séquence UML et un ensemble correspondant de classes Java (considérées comme des *modèles*).

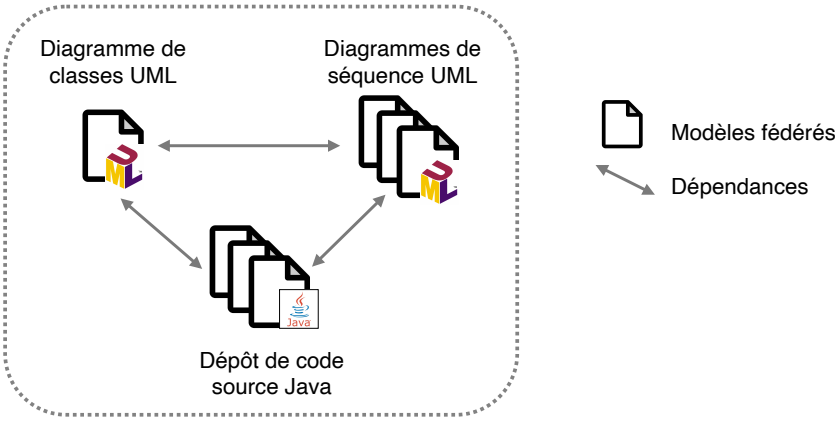

FIGURE 3.8 – Un exemple de *fédération de modèles*

Une *fédération* s'incarne dans un espace conceptuel permettant de reconnaître et de gérer un certain nombre de dépendances entre un ensemble de *modèles*. La mise en œuvre concrète d'une fédération peut prendre différentes formes. Elle peut être mise en œuvre par un logiciel, mais elle peut l'être aussi par une discipline humaine utilisant des règles et des processus, ce qui correspond à des pratiques usuelles. Par exemple, le concepteur d'un diagramme de séquence peut ne pas pouvoir créer de classe ni modifier une classe existante. Si une modification du diagramme de classes est nécessaire, une demande peut être adressée à son concepteur. La même règle s'applique aux développeurs du code Java. Lorsqu'un conflit doit être résolu, une réunion doit être organisée pour arbitrer une solution (idéalement un consensus).

Nous faisons ici le choix de réifier une *fédération* comme un *modèle* (cf le lien d'héritage entre Federation et Model sur la figure 3.7). On a défini une *fédération de modèles* comme un ensemble de *modèles* et de relations entre ces *modèles*. Formellement, une *fédération*

de modèles $\mathcal{F}$ est donc une paire composée d'un sous-ensemble $\mathcal{O}$ de $\mathbb{O}$ et d'une relation binaire $\mathcal{D} : \mathbb{O} \rightarrow \mathbb{O}$ (un sous-ensemble de $\mathbb{O}^2$). La partie inférieure de la figure 3.7 représente l'ensemble des dépendances correspondant à la relation $\mathcal{D}$.

$$\begin{gathered}\mathcal{F} := (\mathcal{D}, \mathcal{O}) \\ \mathcal{O} \subset \mathbb{O} \\ \mathcal{D} : \mathbb{O} \rightarrow \mathbb{O}, \mathcal{D} \subset \mathbb{O}^2\end{gathered}$$

Une *fédération* $\mathcal{F} := (\mathcal{D}, \mathcal{O})$ est *fermée* si et seulement si tout élément de la relation $\mathcal{D}$ est inclus dans $\mathcal{O}$ ($\mathcal{D} \subset \mathcal{O}^2$). Dans ce cas, tous les objets (*modèles*) sont connus. Au contraire, une fédération peut être *ouverte*, s'il existe au moins un *modèle* référencé par $\mathcal{D}$ qui n'est pas connu (pas dans $\mathcal{O}$). La *fermeture* d'une *fédération* $(\mathcal{D}, \mathcal{O})$ est la *fédération* $(\mathcal{D}, \mathcal{O}')$ où $\mathcal{O}'$ est $\mathcal{O}$ étendu avec les éléments référencés dans $\mathcal{D}$. Inversement, il est possible de réduire une *fédération* en retirant de $\mathcal{O}$ les *modèles* non référencés par $\mathcal{D}$. Formellement [11],

$$\begin{cases} closure(\mathcal{D}, \mathcal{O}) = (\mathcal{D}, dom(\mathcal{D}) \cup ran(\mathcal{D}) \cup \mathcal{O}) \\ trim(\mathcal{D}, \mathcal{O}) = (\mathcal{D}, dom(\mathcal{D}) \cup ran(\mathcal{D})) \end{cases}$$

Dans la suite, nous travaillerons avec des *fédérations* fermées (parfois au prix du calcul de leur fermeture). Parfois, nous ferons référence à une fédération en donnant uniquement sa relation $\mathcal{D}$, ce qui signifie que $\mathcal{O}$ est $dom(\mathcal{D}) \cup ran(\mathcal{D})$. Dans ce cas, tous les *modèles* nécessaires sont connus et aucun *modèle* n'est inutilisé.

Remarquons qu'un *modèle* peut faire partie de plusieurs *fédérations* $(\mathcal{D}_i, \mathcal{O}_i)_{i \in I}$. Lorsque ce partage peut provoquer des conflits, on peut définir la *fédération* d'union $(\cup_{i \in I}\mathcal{D}, \cup_{i \in I}\mathcal{O})$ pour y remédier. Ici, la notion de conflit reste abstraite car nous n'avons pas encore spécifié l'utilisation des *fédérations*.

Ce manuscrit ne couvre pas le partage d'objets entre *fédérations* et les conflits potentiels qui en résultent, ce qui pourra faire l'objet de travaux ultérieurs.

3.3.4 Intentions

Comme tout *modèle*, une *fédération* a une *intention* [162, 120].

Dans le contexte de la *fédération de modèles*, l'intention permet de guider le choix de l'ensemble des modèles et des dépendances correspondantes qui forment la fédération. En effet, tous les modèles d'un système donné ne doivent pas forcément faire partie d'une fédération, et toutes leurs dépendances ne doivent pas forcément être réifiées. Seuls les éléments servant l'intention de la fédération sont à considérer [12]. La préoccupation de l'encodage exhaustif et du respect de l'intégrité de la donnée n'est pas à considérer puisque ce sont les modèles fédérés qui restent responsables chacun de cet aspect. L'approche fédération permet donc de se concentrer sur les intentions liées à la mise en oeuvre de l'interopérabilité des modèles, et non aux modèles eux-mêmes.

Dans l'exemple de la figure 3.8, l'intention peut être le maintien de la cohérence entre les diagrammes de classes et de séquences et l'application d'une relation *implements* entre

11. dom et ran désignent respectivement le domaine (*domain*) et la portée (*range*) d'une relation $\mathcal{D}$
12. Notons toutefois qu'une fédération peut servir plusieurs intentions

ces diagrammes et le code. D'autres intentions peuvent inclure, sans s'y limiter, tous les scénarios décrits dans la section 3.2 (composition de modèles, mise en correspondance de modèles, édition de modèles, construction de modèles, etc.).

D'autres exemples de fédération avec une intention spécifique pourraient être :

1. Tous les modèles d'un système à l'étude peuvent être fédérés pour exprimer les diverses relations entre les modèles d'un projet (il s'agit d'une généralisation de notre exemple précédent).
2. Dans le monde MOF [95] d'UML, un modèle et son métamodèle peuvent être fédérés pour réifier la relation *instance de* entre les éléments du modèle et leurs méta-éléments.
3. Les différents modèles d'une ligne de produits pourraient être fédérés pour garantir leur conformité à un modèle de variabilité.
4. Rassembler des modèles dans une fédération (appelée mégamodèle) afin de constituer un catalogue et d'indexer des objets par exemple.

Cette approche permet un découplage sémantique entre, d'une part, les modèles fédérés, et, d'autre part, les dépendances qui sont réifiées. Chaque dépendance porte une sémantique contextuelle et locale guidée par la mise en relation des concepts impliqués. Comme nous le verrons plus loin, l'ensemble de ces dépendances et leur éventuelle structuration peut ainsi permettre de composer une interprétation de la donnée, propre à la fédération, au travers de la réification de concepts fédérés, à la fois d'un point de vue structurel et comportemental. Cette sémantique est entièrement contextuelle à la fédération et peut ne pas être alignée sur la sémantique des artefacts fédérés dans leur environnement technique d'origine. Un modèle de fédération peut alors construire une nouvelle interprétation qui peut contraindre, augmenter ou même changer l'interprétation originale des modèles fédérés pour s'adapter au (ou adopter le) point de vue fédéré.

Un dernier point important concerne la prise en compte du processus d'ingénierie sous-jacent à la fédération, dont il convient d'aligner l'intention avec l'intention de la fédération. Nous reviendrons sur ce point dans la section suivante.

3.4 Mise en œuvre de l'approche

Nous avons vu ce qu'est une fédération de modèles et quels sont les éléments qui en font partie. Dans ce qui suit, nous nous concentrons sur l'utilisation d'une fédération de modèles une fois qu'elle est mise en œuvre. En effet, l'existence d'une fédération implique qu'un certain nombre de comportements supplémentaires apparaissent suite à la réification de certaines dépendances afin de réaliser une intention donnée. En effet, une action sur un modèle contenu dans la fédération peut impliquer ou nécessiter des actions sur d'autres modèles de la fédération. Par exemple, si nous fédérons un diagramme de classes UML avec l'ensemble correspondant de classes Java pour exprimer une relation d'implémentation, la création d'une nouvelle classe Java devrait impliquer la création d'une classe dans le diagramme de classes. En outre, si la classe hérite d'une autre classe, cette dernière doit également être ajoutée au diagramme de classes, afin de garantir la cohérence de l'ensemble.

Comme le montre la figure 3.9, travailler avec une fédération peut se faire soit à travers les modèles fédérés, soit directement à travers le modèle de fédération. Dans le premier cas,

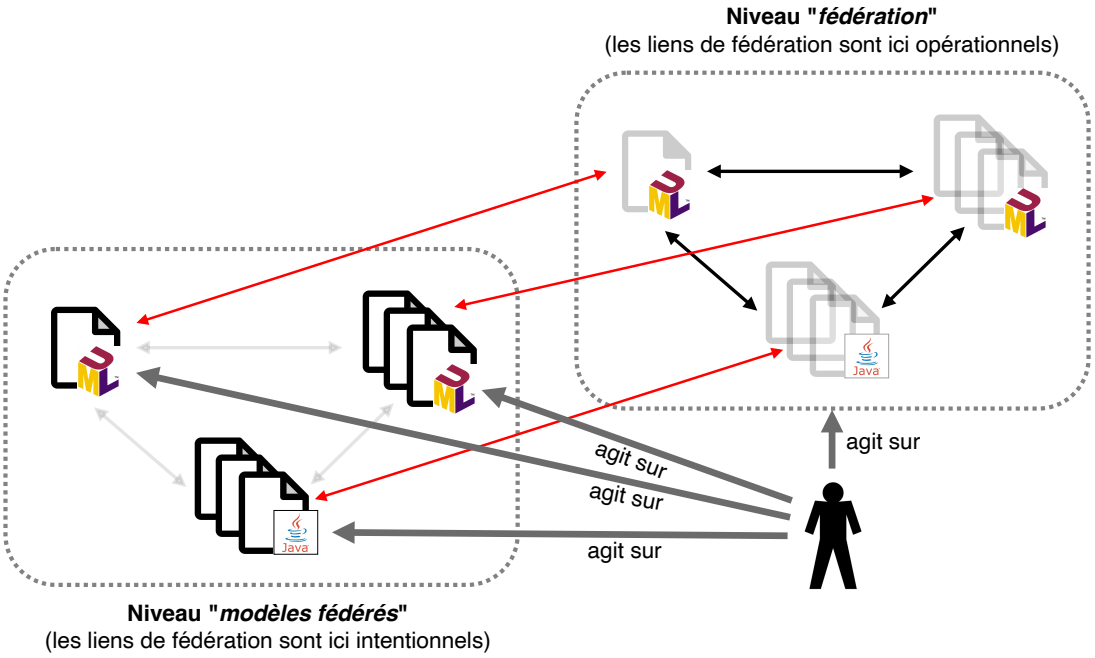

FIGURE 3.9 – Les utilisateurs peuvent interagir à deux niveaux : sur les *modèles fédérés* ou sur la *fédération de modèles*

l'utilisateur conserve ses habitudes et interagit avec son modèle (fédéré). La fédération doit réagir à cette action sur le modèle fédéré en déclenchant des actions sur les autres modèles. Dans le second cas, c'est la fédération qui devient la source de l'action. Cela peut nécessiter la création de nouveaux outils spécifiques pour interagir avec la fédération.

Supposons qu'un ensemble d'actions $\mathbb{A}$ soit défini et que l'application d'une action a de $\mathbb{A}$ à un "objet" o soit notée $a \curvearrowright o$. L'objectif d'une fédération $\mathcal{F} = (\mathcal{D}, \mathcal{O})$ est double. En premier lieu, elle doit pouvoir "transformer" une action "locale" a sur un modèle fédéré (un "objet" o de $\mathcal{O}$) en une action "globale" $a_{\mathcal{F}}$ sur l'ensemble de la fédération ($a_{\mathcal{F}} \curvearrowright (\mathcal{D}, \mathcal{O})$), et donc relativement à l'intention de la fédération. D'un autre côté, la fédération de modèles peut être dotée d'actions (de comportements) spécifiques. Nous étudierons plus tard cette deuxième possibilité. Notons cependant qu'une action $a_{\mathcal{F}}$ d'une fédération est souvent une action composite reposant sur des actions sur les modèles fédérés.

Dans l'exemple précédent, supposons que a soit l'action '*ajouter une classe Java nommée* c' sur le modèle fédéré Java. Si le processus métier lié l'autorise, une action $a_{\mathcal{F}}$ sur la fédération peut composer l'exécution de a avec l'action '*ajouter une classe nommée* c *au diagramme de classes*'. Si le diagramme de classes est en revanche considéré comme prescriptif pour le processus de développement, l'action $a_{\mathcal{F}}$ sera vide et l'action a interdite. Il est à noter que la mise en œuvre d'une telle contrainte dépend des capacités de l'outil utilisé pour effectuer l'action sur un modèle fédéré donné (ici le code source en Java).

Dans certains cas, il est possible de calculer une action fédérée et de l'automatiser. Par exemple, le refactoring de renommage des classes et des attributs dans un diagramme de classes UML peut être automatiquement appliqué au code Java correspondant [13]. Dans d'autres cas, c'est beaucoup plus complexe et la définition et l'exécution d'une action fédé-

13. Si la modification ne viole aucune des contraintes du langage de programmation Java (par exemple, nous ne pouvons pas avoir une classe nommée `class`)

rée peut nécessiter une intervention humaine et une éventuelle résolution de conflits. Par exemple, dans le contexte d'une fédération modèle-métamodèle suivant une métamodélisation stricte, l'extension d'un méta-élément E avec une nouvelle propriété p nécessite la mise à jour de toutes les instances existantes de E en définissant pour chacune la bonne valeur pour p, ce qui peut ne pas être calculable. Dans ce cas, l'action fédérée $a_{\mathcal{F}}$ doit être interactive.

Le *comportement* d'une fédération est composé d'actions issues des actions disponibles sur ses modèles fédérés et de ses actions spécifiques. Le comportement d'une fédération $\mathcal{F}$ est noté $\mathcal{B}_{\mathcal{F}}$.

Afin de mieux illustrer l'approche "fédération de modèles" et notamment le concept de "comportement" d'une fédération, nous nous proposons de la détailler sur les situations de modélisation issues de l'argumentation sur les motivations fondant l'approche, et étudiées dans la section 3.2.

3.4.1 Cas d'utilisation : scénario A (génération de code à partir du modèle avec round-trip)

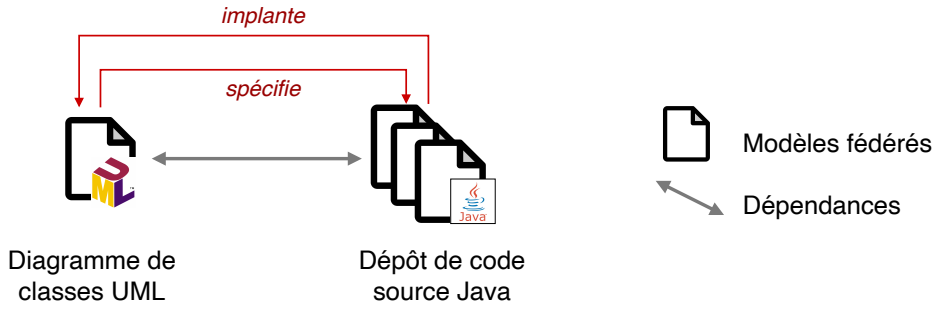

FIGURE 3.10 – Scénario A : génération de code à partir du modèle (avec *round-trip*)

Le premier exemple, rappelé dans la figure 3.10, illustre et précise le cas d'utilisation décrit dans la figure 3.1, où l'objectif était de réifier les dépendances *spécifie* (et inversement *implante*) définies entre un diagramme de classes UML et un dépôt de code source Java.

Dans cet exemple de fédération, le diagramme de classes et le dépôt de code source sont considérés tous deux comme des modèles fédérés. Ces modèles sont partiellement recouvrants (certains champs ou méthodes définis dans le code source ne sont pas définis dans le diagramme de classes, et certaines entités définies dans le diagramme de classes peuvent ne pas être directement mises en œuvre dans le code source).

Une vision plus détaillée de ce scénario est présenté sur la figure 3.11. Cet exemple met en évidence les dépendances *spécifie/implante* entre les entités Group et User définies dans le diagramme de classes, qui sont affinées par quatre dépendances plus fines reliant respectivement :

— les propriétés Group/`name` et User/`name` dans le diagramme de classes et les attributs `name` dans les classes Group.java et User.java,
— l'opération `login(password)` dans le diagramme de classes et la méthode correspondante dans la classe User.java,
— la relation `group` entre User et Group dans le diagramme de classes et l'attribut `group` dans la classe User.java.

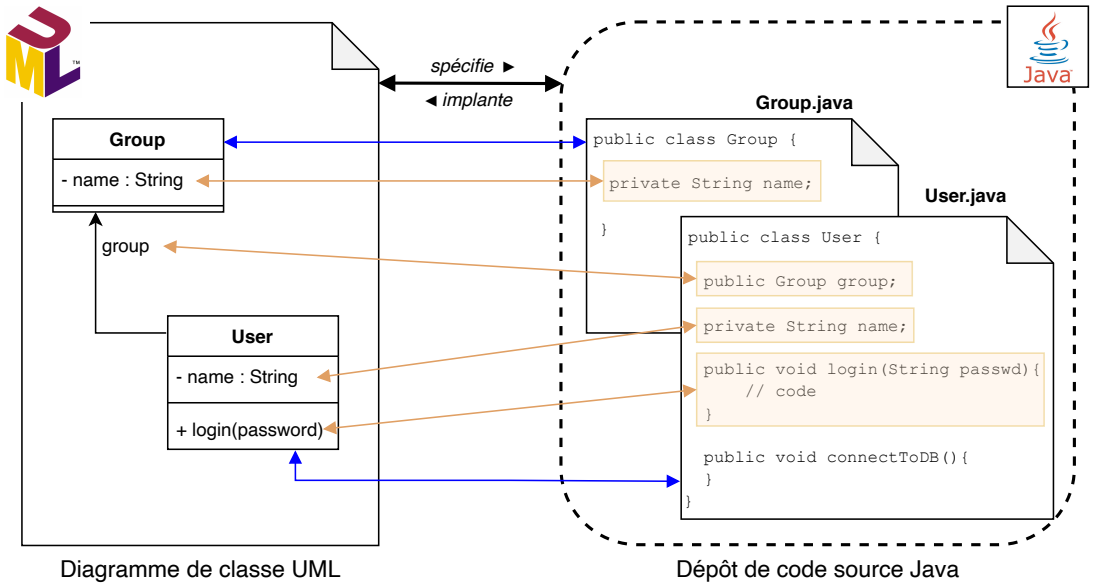

FIGURE 3.11 – Cas d'utilisation : scénario A (génération de code à partir du modèle avec round-trip)

Cette approche doit être comparée aux mécanismes de génération de code basés sur les transformations (parfois bidirectionnelles), qui impliquent la gestion d'un cycle de vie et des problèmes de round-trip [91]. L'approche *fédération de modèles* permet ici un meilleur découplage des deux modèles et des liens plus lâches, qui autorisent une évolution simultanée du diagramme de classes et du code source (au prix d'une éventuelle réconciliation). Les correspondances entre les entités du diagramme de classes et le code source peuvent être implicites et calculées (par exemple en faisant correspondre des noms exacts) ou peuvent être plus souples en demandant à l'utilisateur de définir la correspondance. Des entités peuvent être écartées de la fédération, de même que des classes Java.

Cet exemple met également en lumière le raffinement de liens de fédération. Ici les dépendances *spécifie/implante* se raffinent une première fois au sein de dépendances entre des entités UML et des classes JAVA, et se raffinent une seconde fois pour mettre en correspondance des propriétés et des attributs, ou des opérations et des méthodes. Cette qualité permet d'envisager des mécanismes de génération et de round-trip à un grain très fin, qui à leur tour se composent et se structurent dans des liens de dépendance plus généraux.

Les tables 3.1 et 3.2 présentent les aspects comportementaux de cette fédération de modèles, au travers de la description de 14 actions de base qui permettent de définir une synchronisation spécifique entre un diagramme de classes et un dépôt de code source Java.

Il n'est pas proposé ici d'actions au niveau de la fédération de modèles, mais l'outillage d'une telle fédération pourrait exposer par exemple une action de création d'une nouvelle entité, qui créerait à la fois une nouvelle entité dans le diagramme de classe et une nouvelle classe Java.

D'un point de vue méthodologique, il nous semble important de souligner ici l'importance de la prise en compte du processus d'ingénierie. Sur cet exemple, le processus d'ingénierie est à la source de l'intention de la fédération. Dans ce cadre, la fédération peut même s'envisager comme une réification du processus d'ingénierie, tant du point de vue structurel que comportemental.

Source de l'action depuis le diagramme de classes		
	Action a sur le modèle fédéré	Action $a_{\mathcal{F}}$ sur la fédération
1	Ajout d'une classe Entity	a + (éventuellement) création d'une classe Java Entity.java (ou avec un autre nom à saisir interactivement) dans le dépôt de code source, liée à l'entité Entity + (ou éventuellement) sélection interactive d'une classe Java déjà existante qui implante l'entité Entity.java
2	Suppression d'une classe Entity	a + suppression du lien existant vers la classe Java correspondante dans le dépôt de code source, (et/ou éventuellement) suppression de la classe Java correspondante, si cette dernière existe
3	Ajout d'un attribut p dans la classe Entity	a + (éventuellement) création d'un champ Java (field) p (ou avec un autre nom à saisir interactivement) dans la classe Java correspondante (ou éventuellement) sélection interactive d'un champ Java déjà existant qui implante l'attribut p
4	Suppression de l'attribut p de la classe Entity	a + suppression du lien existant vers le champ Java correspondant dans la classe Java concernée, (et/ou éventuellement) suppression du champ Java correspondant, si ce dernier existe
5	Ajout d'une relation r dans la classe Entity	Même mécanisme que (3)
6	Suppression d'une relation r de la classe Entity	Même mécanisme que (4)
7	Ajout d'une opération o dans la classe Entity	Même mécanisme que (3)
8	Suppression d'une opération o de la classe Entity	Même mécanisme que (4)

TABLE 3.1 – Aspects comportementaux d'une fédération de modèles sur le scénario A : édition du diagramme de classe

Source de l'action depuis le code source		
	Action a sur le modèle fédéré	Action $a_{\mathcal{F}}$ sur la fédération
9	Ajout d'une classe Java C.java	a + (éventuellement) création d'une classe C (ou avec autre nom) dans le diagramme de classes, liée à la classe Java C.java + (ou éventuellement) sélection d'une entité dans le diagramme de classe, à lier avec C.java
10	Suppression d'une classe Java C.java	a + suppression du lien existant vers la classe correspondante dans le diagrammme de classes, (et/ou éventuellement) suppression de la classe correspondante, si cette dernière existe
11	Ajout d'un champ f dans la classe Java C.java	Même mécanisme (avec un choix interactif qui permet de dire si le champ Java correspond à une propriété ou à une relation)
12	Suppression d'un champ f de la classe Java C.java	Même mécanisme
13	Ajout d'une méthode m dans la classe Java C.java	Même mécanisme
14	Suppression d'une méthode m de la classe Java C.java	Même mécanisme

TABLE 3.2 – Aspects comportementaux d'une fédération de modèles sur le scénario A : édition du code source

3.4.2 Cas d'utilisation : scénario B (construction de modèles)

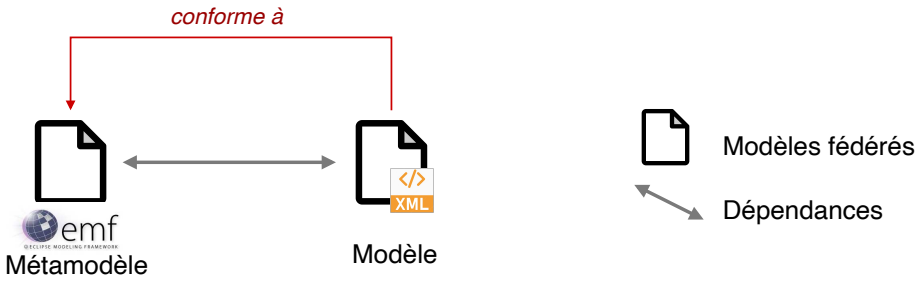

FIGURE 3.12 – Scénario B : co-construction d'un modèle et de son métamodèle

Le deuxième cas d'utilisation présenté dans la figure 3.12 illustre le scénario décrit dans la figure 3.2, où il s'agit de co-construire un modèle et son métamodèle. Ce deuxième exemple est détaillé dans la figure 3.13.

Pour cet exemple, nous proposons les comportements définis dans la table 3.3 en supposant que seules 9 actions sont possibles sur les *modèles*. La fédération contient donc un métamodèle M et un modèle m. Les actions "locales" sont définies dans la troisième colonne et regroupées par type d'"objet". Quatre actions peuvent être appliquées à M : (ligne 1-2) ajouter/supprimer un méta-élément E dans M, (ligne 3-4) ajouter/supprimer une *feature* f vers/depuis un méta-élément E dans M. Les mêmes actions peuvent être appliquées à m (lignes 5-6 et 8-9) et une action spécifique sur la ligne 7 est définie pour supporter la création d'un élément conforme à un élément de M. La quatrième colonne du tableau illustre diverses possibilités de définition de l'action globale $a_{\mathcal{F}}$, qui correspond à l'application de l'action a sur la fédération. Elle peut être simplement a comme proposé dans les lignes 1, 6-8 ; ou bien étendre a pour être composée avec d'autres actions comme dans les lignes 2-4 ou encore empêcher a de se produire dans la ligne 5.

Il convient d'examiner plus en détail la dernière action de la ligne 9. En effet, la suppression d'une *feature* d'une instance peut rompre la relation de conformance. Plusieurs stratégies sont alors possibles. On peut simplement interdire cette action ou obtenir un comportement sophistiqué préservant la conformance. Par exemple, on pourrait créer un

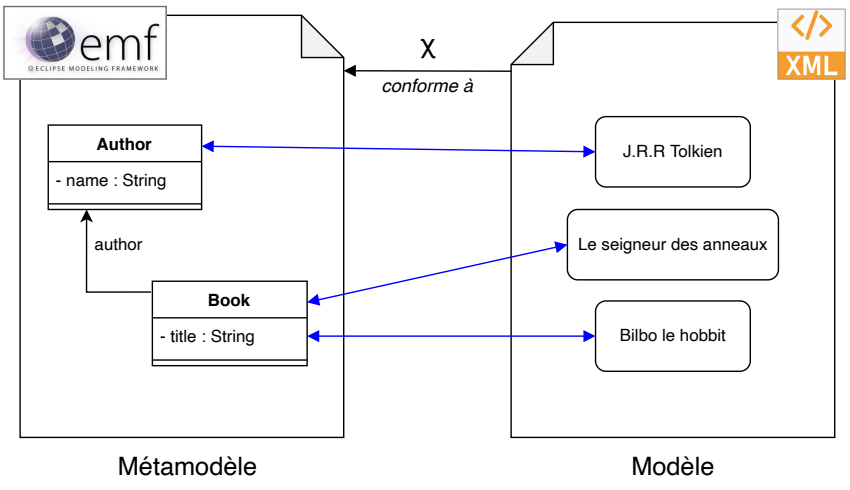

FIGURE 3.13 – Cas d'utilisation : scénario B (construction de modèles)

Source		Action a sur le modèle fédéré	Action $a_{\mathcal{F}}$ sur la fédération
Métamodèle	1	Ajout d'un méta-élément E	a
	2	Suppression d'un méta-élément E	a + suppression de toutes les instances de E
	3	Ajout d'une *feature* f à un méta-élément E	a + ajout de la caractéristique f à toutes les instances de E (en demandant la saisie des valeurs requises)
	4	Suppression d'une *feature* f d'un méta-élément E	a + suppression de f de toutes les instances de E
Modèle	5	Ajout d'un élément e	**Interdit** (nécessite un méta-élément E)
	6	Suppression d'un élément e	a
	7	Ajout d'une instance e d'un méta-élément E	a
	8	Ajout d'une *feature* f à un élément e	a (ne viole pas la conformance)
	9	Suppression d'une *feature* f à un élément e	(risque de violation de la conformance)

TABLE 3.3 – Aspects comportementaux d'une fédération modèle/métamodèle

nouveau méta-élément clonant le méta-élément actuel de e, puis changer le méta-élément associé à e en ce nouveau méta-élément et enfin, supprimer la *feature* f de ce méta-élément. Ce comportement très spécifique peut ne pas répondre à tous les besoins, mais peut être adapté à une fédération spécifique.

L'approche fédération de modèles offre la capacité à traiter la conformance χ comme une relation comme les autres qui peut être facilement personnalisée et adaptée à différentes stratégies de modélisation. Il est possible de la faire pointer vers des éléments de différents niveaux conceptuels, de réifier sa cible à un méta-niveau ou d'assouplir les exigences et relâcher les contraintes relatives à sa validation, pour gérer la conformance plus tard.

Si cette approche permet également d'envisager un outillage permettant une co-construction simultanée du modèle et de son métamodèle, elle peut également être étendue à d'autres scénarios avec plusieurs autres instances de modèles conformes à ce même métamodèle, avec une gestion explicite de la co-évolution [110]. Sur la problématique de la co-évolution, les travaux [208] ou [48] montrent qu'il est possible de définir de nombreux comportements pour une telle fédération en fonction des besoins du concepteur.

3.4.3 Cas d'utilisation : scénario C (composition de modèles)

La figure 3.14 illustre l'approche *fédération de modèles* mise en œuvre sur le scénario C présenté sur la figure 3.3. Ce scénario implique l'abstraction d'une sorte de modèle conceptuel (nous ne traitons pas ici du problème de sa réification), dans lequel est défini le concept métier "Commande", à partir de la composition de données issues à la fois du

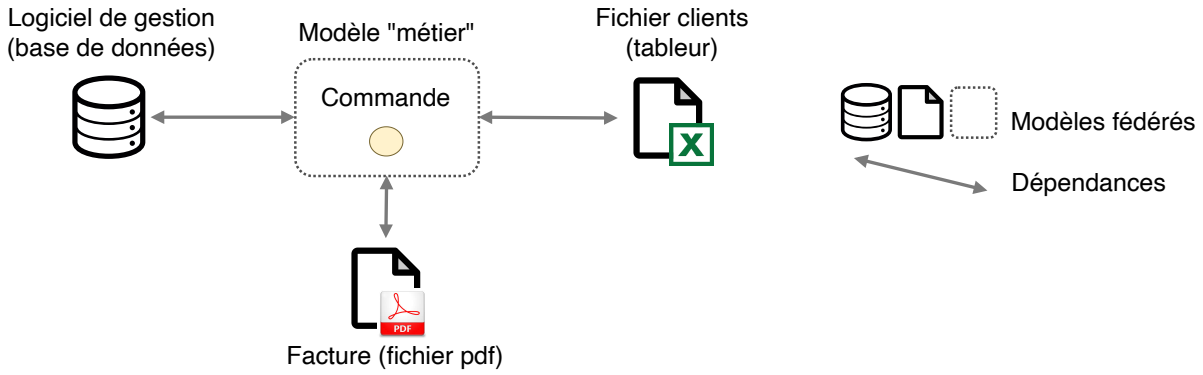

FIGURE 3.14 – Scénario C : composition de modèles

logiciel de gestion et du fichier client du tableur. Les liens de fédération sont ici établis entre les quatre modèles : la base de données, le tableur, le modèle métier conceptuel, et le fichier PDF en sortie.

Nous ne détaillerons pas cet exemple ici, mais nous y reviendrons dans la suite de ce manuscrit (section 4.3.3). Ce scénario nous permet cependant d'esquisser l'intuition d'une nécessaire structuration des liens de fédération (ici au travers du concept métier "Commande").

3.4.4 Cas d'utilisation : scénario D (mise en correspondance de modèles)

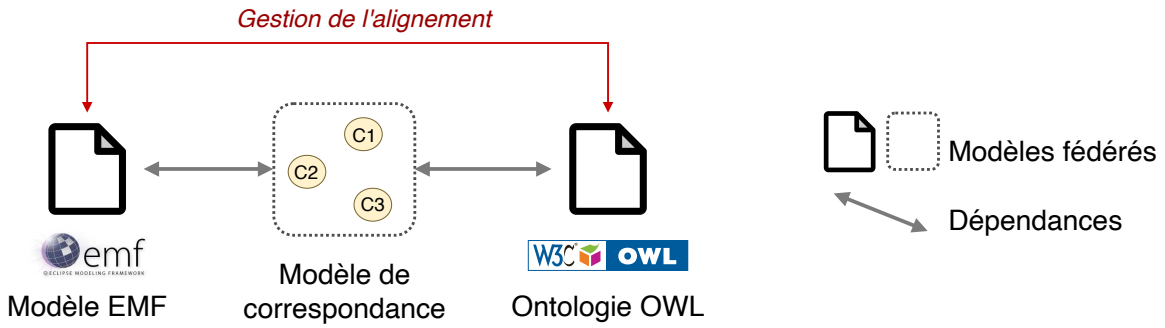

FIGURE 3.15 – Scénario D : mise en correspondance de modèles (*model mapping*)

La figure 3.15 illustre l'approche *fédération de modèles* mise en œuvre sur le scénario D présenté sur la figure 3.4.

Plusieurs types de critères de mise en correspondances sont envisageables. Ce peut être des critères purement techniques et syntaxiques, auquel cas la mise en correspondance peut être réalisée relativement simplement à l'aide d'une transformation birectionnnelle. Ce peut être également des critères liés au métier, automatisables ou devant être arbitrés par un être humain. C'est ce dernier cas de figure qui est illustré ici. Ce scénario implique également l'abstraction d'un modèle conceptuel qui contient trois concepts métier C1, C2 et C3 qui sont présents à la fois dans le modèle EMF et dans l'ontologie OWL. Ces trois concepts reflètent l'intersection sémantique des deux modèles mis en correspondance, qui peut être totale ou partielle (ce qui signifie que de nombreux concepts métier peuvent n'être

Source de l'action	Action a sur le modèle fédéré	Action $a_{\mathcal{F}}$ sur la fédération
Modèle EMF	Ajout d'un EObject ou d'une EClass	a + instantiation éventuelle d'une instance de concept C_i + instanciation éventuelle d'une Classe ou d'un Individual dans l'ontologie
	Suppression d'un EObject ou d'une EClass	a + suppression éventuelle d'une instance de concept C_i + suppression éventuelle d'une Classe ou d'un Individual dans l'ontologie
	Modification d'un EObject E ou d'une EClass C	a + déclenchement d'une mise à jour de l'ontologie si les objets E ou C sont impliqués dans une instance de concept C_i
Ontologie OWL	Ajout d'une Class ou d'un Individual	a + instantiation éventuelle d'une instance de concept C_i + instanciation éventuelle d'un EObject ou d'une EClass dans le modèle EMF
	Suppression d'une Class ou d'un Individual	a + suppression éventuelle d'une instance de concept C_i + suppression éventuelle d'un EObject ou d'une EClass dans le modèle EMF
	Modification d'une Class C ou d'un Individual I	a + déclenchement d'une mise à jour du modèle EMF si les concepts C ou I sont impliqués dans une instance de concept C_i
Modèle de correspondance	Instantiation d'un concept C_i	a + instanciation de concept(s) Class et/ou Individual dans l'ontologie + instanciation d'objet(s) EObject et/ou EClass dans le modèle EMF
	Suppression d'un concept C_i	a + suppression de concept(s) Class et/ou Individual dans l'ontologie + suppression d'objet(s) EObject et/ou EClass dans le modèle EMF
	Modification d'un concept C_i ou action explicite sur un concept C_i	a + mise à jour de concept(s) Class et/ou Individual dans l'ontologie + mise à jour d'objet(s) EObject et/ou EClass dans le modèle EMF

TABLE 3.4 – Aspects comportementaux d'une fédération de modèles sur le scénario D : mise en correspondance de modèles

présents que dans un seul modèle, et ce qui est en général un problème pour une transformation birectionnelle qui doit persister les données qu'elle ne traite pas).

La table 3.4 propose des comportements liés à cette fédération, à partir de chacun des trois modèles, les deux modèles mis en correspondance et le modèle de correspondance (équivalent ici à la fédération). Il est intéressant de voir sur cet exemple que la fédération permet ici de structurer la mise en correspondance en la raffinant avec les concepts C_i, dont la logique d'association est arbitraire et gérée au niveau métier (voire manuelle et requérant un être humain). Chaque concept C_i structure également des transformations à grain fin ($owl \xrightarrow{C_i} emf$ et $emf \xrightarrow{C_i} owl$), composables pour construire les transformations globales $owl \rightarrow emf$ et $emf \rightarrow owl$. Cet exemple illustre l'intérêt de pouvoir disposer de comportements sur les liens de fédération [14] (ici ces comportements sont les transformations), ce qui rend l'approche *fédération de modèles* immédiatement opérationnelle sur ce type de scénario.

3.4.5 Cas d'utilisation : scénario E (édition de modèles)

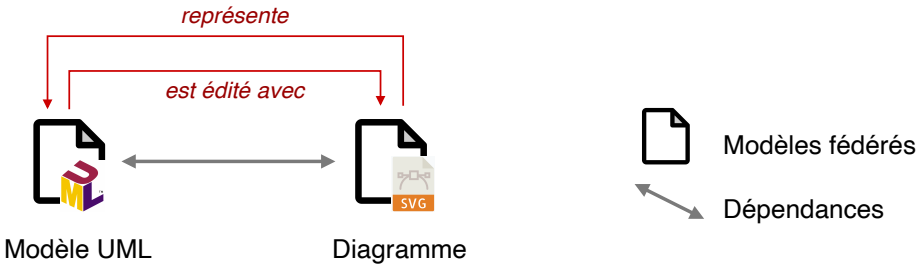

FIGURE 3.16 – Scénario E : édition de modèle(s)

Ce dernier scénario reprend le cas présenté figure 3.5 et dont l'approche *fédération de modèles* est illustrée dans la figure 3.16. Le lien de fédération est ici une dépendance qui traduit un lien de représentation, et l'on manipule un modèle implémentant une syntaxe concrète (ici graphique) comme *proxy* pour manipuler un autre modèle.

Dans cet exemple, nous décrivons la manipulation d'un modèle UML au travers d'un diagramme de classes, représenté comme un modèle de formes graphiques.

De manière très similaire à d'autres scénarios étudiés plus haut, la table 3.5 propose quelques comportements liés à cette intention [15]. Pour simplifier, nous nous concentrons ici sur huit actions de base. Les quatre premières actions permettent d'ajouter/supprimer des classes et des références à une classe vers/du modèle de diagramme de classes. Les actions fédérées correspondantes mettent à jour le modèle graphique afin que : 1) il reste synchronisé avec le modèle de diagramme de classes ; 2) il reste cohérent et consistent (par exemple, en supprimant les flèches "en l'air" comme dans la deuxième action). Les quatre actions suivantes permettent d'ajouter/supprimer une boîte/une flèche à/d'un modèle graphique. Les actions correspondantes sur la fédération ont également le double objectif de garantir la synchronisation (avec le modèle de diagramme de classes) et la cohérence.

14. encore appelés dépendances
15. Notez qu'il s'agit d'une simplification à des fins d'illustration : un éditeur graphique peut avoir besoin de modèles supplémentaires pour capturer les outils, la mise en page, etc...

Source de l'action	Action a sur le modèle fédéré	Action $a_{\mathcal{F}}$ sur la fédération
Modèle UML	Ajout d'une classe C	a + création d'une "boîte"
	Suppression d'une classe C	a + suppression de la "boîte" correspondante et de toutes les "flèches" reliées à cette "boîte"
	Ajout d'une référence r depuis une classe C vers une classe D	a + création d'une "flèche" connectant la "boîte" de C vers la "boîte" de D
	Suppression d'une référence r depuis une C vers une classe D	a + suppression de la "flèche" correspondante
Diagramme	Ajout d'une "boîte" avec le nom C	a + création d'une classe C dans le diagramme de classe
	Suppression de la "boîte" avec le nom C	a + suppression de la classe C du diagramme de classes + gestion éventuelle des références nulles
	Ajout d'une flèche entre la "boîte" C et la "boîte" D	a + création d'une référence depuis la classe C vers la classe D
	Suppression d'une flèche entre la "boîte" C et la "boîte" D	a + suppression de la référence correspondante

TABLE 3.5 – Aspects comportementaux d'une fédération de modèles sur le scénario E : édition de modèle

Dans ce contexte, l'approche *fédération de modèles* offre une séparation naturelle entre le modèle abstrait et les modèles de représentation, et apporte des mécanismes pour gérer leurs liens par le biais de dépendances explicites et outillées.

Comme pour toute activité de modélisation, les relations de conformance des notations peuvent entraver la créativité et l'expressivité. Lorsque des idées doivent être exprimées et partagées, il peut être utile de déconnecter le contrôle de conformité pour permettre l'émergence de nouvelles "choses", comme le montre l'utilisation fréquente de tableaux blancs ou d'outils de dessin. Dans un deuxième temps, la conformance peut être rétablie en étendant le modèle du domaine, sa représentation ou même les deux [16]. La *fédération de modèles* permet ce scénario en découplant le modèle de sa représentation et permet une modification de la relation de conformance.

Enfin, cette approche permet de construire des représentations graphiques indépendamment de la technologie du modèle édité, et donc de capitaliser un même outil graphique réutilisable pour une autre technologie de modélisation.

16. La conformance peut n'être aussi jamais rétablie, si l'on considère l'intérêt pour un utilisateur à exprimer quelque chose qui n'est pas prévu dans le métamodèle, mais qui est est utile à son activité et dont la communication est importante.

3.5 Synthèse et conclusion

Après une reformulation de notre problématique et la précision de nos objectifs de recherche, nous avons explicité et formalisé, à partir de cinq scénarios illustratifs des usages de l'IDM, l'approche *fédération de modèles*, comme une solution couplage souple entre modèles hétérogènes autonomes, et où les liens de fédération sont explicitement réifiés. La contrainte de fédérer des modèles autonomes vivants (enjeu E3) requiert la gestion des aspects comportementaux des modèles fédérés (transformation d'actions locales a en actions globales $a_{\mathcal{F}}$ sur l'ensemble de la fédération).

Le *framework* MF^2 (*Model Federation Framework*) constitue une proposition d'implantation formelle de l'approche *fédération de modèles*, dans laquelle les modèles sont des termes formels et disposent d'une sémantique opérationnelle [17]. Pour des raisons de concision du manuscrit, la présentation de MF^2 ne figure pas dans le corps de ce manuscrit mais est consultable dans l'annexe A. Les travaux autour d'une implantation formelle de l'approche ont été conduits pour deux principales raisons. La première raison était de définir précisément (mathématiquement) la sémantique de base de l'approche de la fédération de modèles, car contrairement à la plupart des langages de modélisation (qui sont essentiellement structurels), ce langage fait appel au calcul. Une deuxième raison est de pouvoir mener des travaux formels autour du typage de modèles (voir la conclusion de l'annexe A.6 page 189). Cette démarche s'accompagne d'un prototype simple implanté en OCaml [18], à titre de preuve de concept.

Le *framework* MF^2 n'a pas pour vocation à être utilisé dans un contexte opérationnel, car de nombreuses fonctionnalités sont manquantes, et surtout car il n'y a pas le support de technologies externes. Le chapitre 4 (et l'annexe B) présentent le langage FML comme une implantation plus complète et sophistiquée de l'approche *fédération de modèles*.

17. Fabien Dagnat, Salvador Martínez, Antoine Beugnard, Jean-Christophe Bach, Joël Champeau, Sylvain Guérin, Fahad Golra, *Working with Multiple Models through Model Federation* (en finalisation de rédaction, soumission prévue 11/2023)
18. https://github.com/openflexo-team/mf2-code

Chapitre 4

Le langage FML

Dans ce chapitre, nous proposons le langage FML (*Federation Modelling Language*), qui implante l'approche abstraite "fédération de modèles" décrite au chapitre précédent, et permet de définir et d'exécuter des fédérations. La section 4.1 présente rapidement le langage au travers des différents choix de conception qui ont été arbitrés. La section 4.2 s'attarde sur la description du langage et donne un aperçu de sa syntaxe textuelle. Les annexes B et C donnent respectivement des définitions formelles du langage et précisent sa sémantique. La section 4.3 présente la mise en œuvre du langage sur les cinq scénarios prototypiques et nous concluons dans la section 4.4.

Sommaire

4.1 Introduction au langage FML

Pour introduire aux différents choix de conception qui ont conduit à la définition du langage FML (*Federation Modelling Language*), nous nous proposons de repartir du scénario A présenté dans la section 3.2 et dont l'approche *fédération de modèles* est présentée dans la section 3.4.1.

La partie inférieure de la figure 4.1 contient une version simplifiée de l'exemple de fédération présenté dans la figure 3.11. Elle présente un diagramme de classes UML contenant une classe nommée Group avec un attribut name de type String. Il montre également un dépôt de code source Java contenant une classe Group.java définissant un champ privé nommé name de type (Java) String. Cet exemple de fédération réifie deux dépendances : l'une entre la classe UML et sa classe Java associée et l'autre entre leurs attributs.

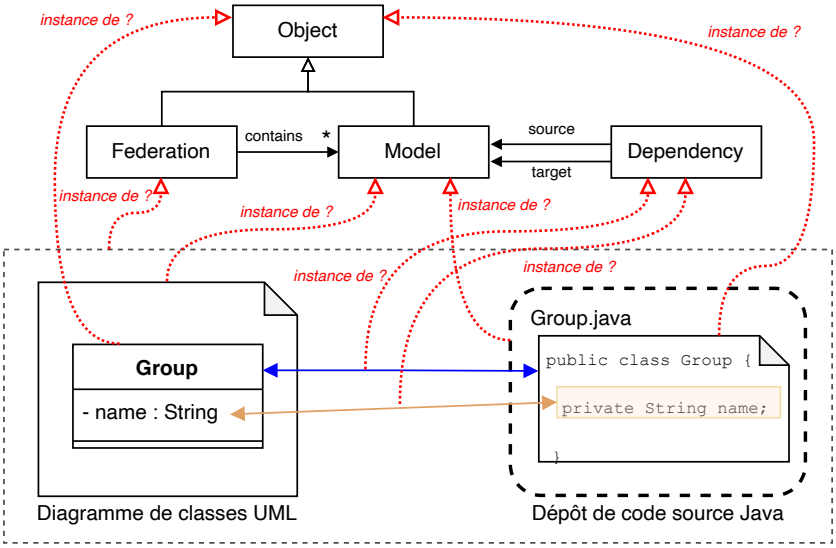

FIGURE 4.1 – Vue abstraite de l'approche *fédération de modèles* : une implémentation "naïve" (pour des raisons de lisibilité, toutes les flèches ne sont pas représentées)

L'approche *fédération de modèles*, présentée figure 3.7 et appliquée à ce scénario est présentée sur la figure 4.1. Cette approche nous invite à considérer que :

— le diagramme de classes et le dépôt du code source sont des instances de Model,

— la classe UML Group et la classe Group.java sont des instances de Objet,

— les liens entre eux sont des instances de Dependency.

D'un point de vue technique, ces relations *"instance de"* représentées par des flèches rouge en pointillé dans la figure ne sont pas réalisables. En effet, une implémentation naïve de ces relations nécessite une instanciation linguistique [3, 138] traversant les frontières technologiques (ici UML et Java), ce qui n'est pas possible dans le contexte de technologies hétérogènes.

Nous sommes ici au cœur de la problématique qui a conduit à l'élaboration du langage FML. Ce langage de modélisation vise à fournir une couche conceptuelle (enjeu E1) permettant de réifier le lien à des données externes, qui sont gérées explicitement dans leur espace technique d'origine (sérialisation et outils) (enjeu E2). La connexion à ces données

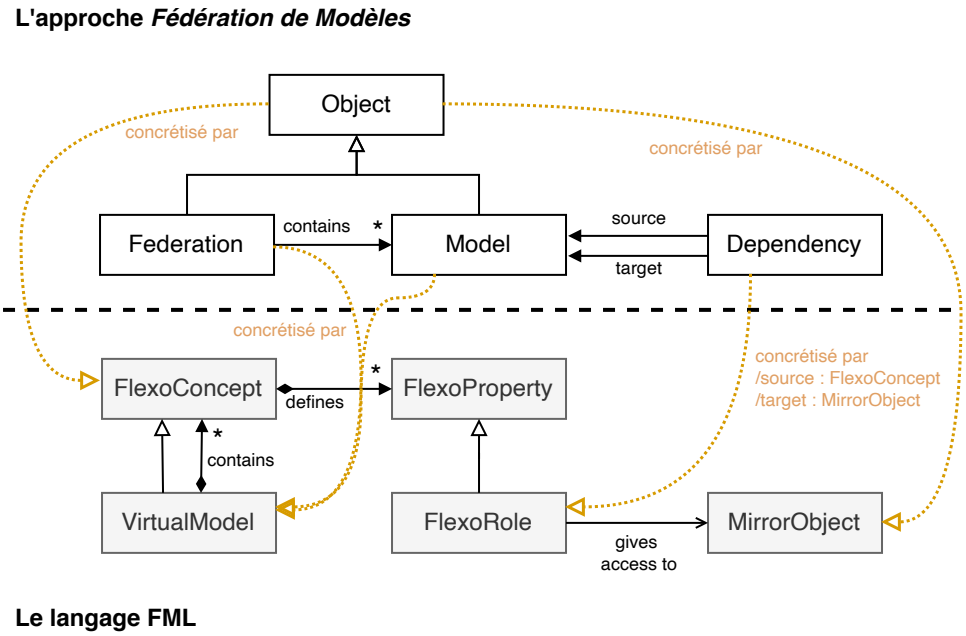

FIGURE 4.2 – Concrétisation de l'approche *fédération de modèles* dans FML

externes et hétérogènes doit en outre supporter à la fois des aspects structurels mais aussi comportementaux, à travers la manipulation de la donnée et de son cycle de vie (enjeu E3).

4.1.1 Un rapide aperçu sur FML

FML se présente comme un langage de modélisation orienté objet. Un concepteur modélisant en FML définit des *concepts* qui ont des *propriétés* (ses données) et des *comportements* (des fonctions permettant de manipuler ses données).

Un modèle FML est exécutable par un interpréteur (une machine virtuelle) qui, au moment de l'exécution, instancie des concepts du modèle pour manipuler, référencer ou stocker les valeurs concrètes des éléments du modèle. FML fournit également des constructions pour décrire le calcul, ce qui est plus rare dans les langages de modélisation. Cette partie du langage est utilisée pour définir les comportements des modèles. À l'exécution, l'utilisateur interagit avec une instance de modèle, et avec les instances de concepts qu'elle contient, via l'exécution explicite des comportements du modèle, ou par le biais des comportements des concepts contenus ou référencés.

FML repose sur une approche fortement typée. Notamment, les propriétés et les comportements ont des types et leur définition doit respecter des contraintes de typage strictes.

La figure 4.2 montre les principaux choix de concrétisation effectués lors de la mise en œuvre de l'approche de fédération dans FML, et explicite comment les concepts FML permettent de concrétiser les abstractions présentées dans l'approche.

FlexoConcept constitue le concept central de FML. Il concrétise de nombreux éléments de modélisation (Federation, Model et Object). Un FlexoConcept est déclaré au sein d'un VirtualModel et, au moment de l'exécution, les instances du FlexoConcept sont contenues dans une

instance de son VirtualModel. Un VirtualModel peut représenter un modèle ou une fédération et est également un FlexoConcept. C'est l'unité de structuration des concepts.

Chaque FlexoConcept définit un ensemble de FlexoProperty et de FlexoBehaviour [1]. La notion abstraite de Dependency est concrétisée par la notion de FlexoRole, qui permet de mettre en relation un FlexoConcept avec un autre objet. L'objet référencé peut être soit une instance de la même fédération (une instance de FlexoConcept) et l'on parle de *dépendance interne*. L'objet référencé peut aussi être ou faire partie d'un modèle fédéré. On parlera alors de *dépendance externe*, et l'objet référencé sera représenté par une instance de MirrorObject qui maintient le lien vers cet objet externe [2].

Notez à ce stade que, même s'ils ne sont pas encore décrits, les FlexoBehaviour sont essentiels pour réifier le comportement d'une dépendance.

Même si la correspondance entre FML et une approche orientée objet habituelle n'est pas exacte, elle permet de se faire une idée du fonctionnement de FML : un concept correspond à une classe, ses propriétés aux attributs de la classe et ses comportements aux méthodes de la classe.

4.1.2 Mise en œuvre de l'approche sur un exemple simple

Pour une meilleure compréhension des concepts de base de FML, nous nous proposons de poursuivre la démarche d'implantation du scénario A. La figure 4.3 présente la mise en œuvre openflexo/FML de l'approche *fédération de modèles* appliquée au cas d'utilisation présenté dans la section 3.4.1. L'approche abstraite est répétée en haut de la figure tandis que les modèles concrets fédérés de l'exemple sont en bas. Entre les deux, nous présentons les différents niveaux d'abstraction utilisés.

Nous préconisons une séparation claire entre ce que nous appelons les *espaces techniques* (ou *espace d'information* [3]), qui contiennent les modèles à fédérer dans leurs environnements techniques d'origine, et l'*espace conceptuel*, qui est l'endroit où la fédération de modèles est définie et exécutée. Au lieu de référencer directement les liens entre les objets dans les espaces techniques, nous les réifions via l'utilisation d'instances d'un concept dédié appelé FlexoRole. Ces FlexoRoles sont ensuite structurés au sein de FlexoConcepts.

Cette approche permet le découplage désiré entre la conceptualisation de la fédération de modèles et les espaces techniques fédérés. Les modèles fédérés peuvent évoluer indépendamment dans leur environnement technique d'origine. Un autre aspect intéressant à souligner est le fait que le concept réifié (un FlexoConcept) peut explicitement porter une sémantique contextuelle à la fédération (qui peut ne pas être alignée sur la sémantique des sources fédérées dans leur environnement technique d'origine).

L'espace conceptuel est quant à lui scindé en deux niveaux : un premier niveau qui contient les modèles définis à l'aide du langage FML (les types) et un autre niveau qui contient les instances de ces modèles.

1. Ils ne font pas partie de la figure et ne sont pas abordés ici car cette sous-section se concentre sur la mise en œuvre de l'approche de la fédération.
2. Un MirrorObject permet de refléter dans FML un objet externe, et se comporte comme un *proxy* de/vers cet objet.
3. Plus précisémment, on définit l'*espace d'information* comme l'union de tous les *espaces techniques*, un *espace technique* étant dévolu à une technologie donnée.

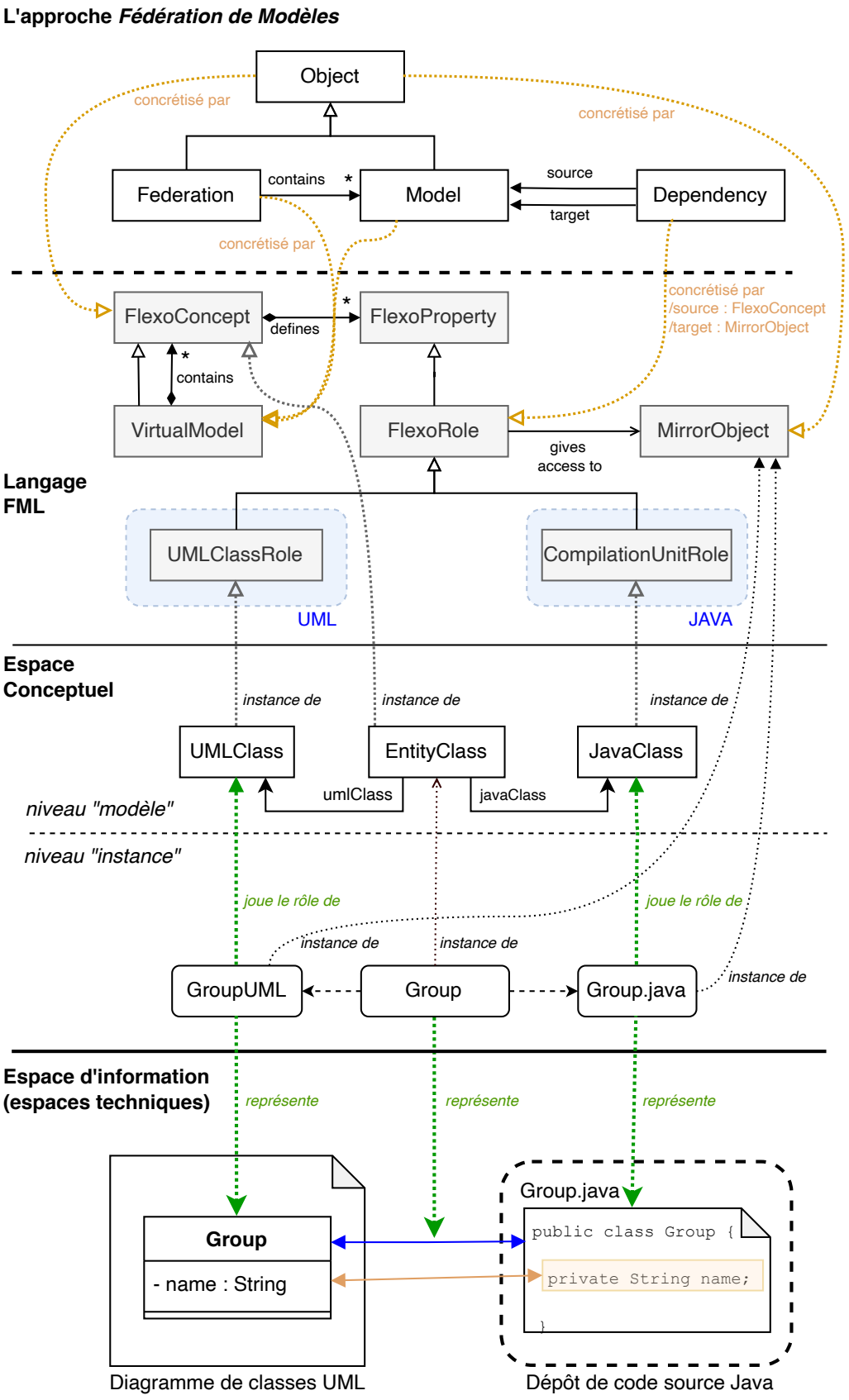

FIGURE 4.3 – Mise en œuvre du scénario A dans le langage FML

En utilisant le langage FML, nous définissons (dans le niveau "modèle" de l'espace conceptuel) un FlexoConcept appelé EntityClass pour réifier la dépendance entre une classe UML et la classe Java correspondante. Il est instancié (dans le niveau "instance" de l'espace conceptuel) pour représenter la dépendance entre la classe UML Group et la classe Group.java. EntityClass définit deux FlexoRoles (umlClass et javaClass). Ils sont respectivement des instances de deux rôles spécifiques UMLClassRole (défini dans un adaptateur technologique UML) et CompilationUnitRole (défini dans un adaptateur technologique JAVA). Ces rôles spécialisés sont chargés de la traduction technologique entre le monde Openflexo/FML et la technologie correspondante. Cette notion d'*adaptateur technologique* (*technology adapter*) est présentée plus loin dans ce document (cf section 4.2.3, page 95).

Dans le niveau "instance", deux instances d'objets de MirrorObject (GroupUML et Group.java) vont maintenir le lien avec les artefacts correspondants (un fichier XMI et un fichier Java). Ils jouent tous deux le rôle de FlexoRole : UMLClass et JavaClass.

4.1.3 Les choix de conception pour FML

Dans cette section, nous détaillons les principaux choix de conception qui ont conduit à la définition du langage FML. Certains de ces choix sont directement dérivés du *framework* MF² [4], tandis que d'autres choix affinent d'autres préoccupations dans la perspective d'une mise en œuvre efficace et pertinente du point de vue d'un utilisateur de ce langage.

1. **Partitionnement *espace conceptuel* / *espace d'information*** (ensemble des *espaces techniques*) : ce premier choix de conception a été imposé par la nature hétérogène inhérente aux modèles fédérés. La figure 4.3 l'illustre par une ligne horizontale séparant les deux espaces. L'*espace d'information* est préexistant à la fédération et contient diverses sources d'information qui sont fédérées avec leurs espaces techniques d'origine (sérialisation et outils). Quant à lui, l'espace conceptuel offre les concepts et l'expressivité pour réifier les liens de fédération, tout en permettant à l'utilisateur du langage de préciser sa propre sémantique. Ce partitionnement permet, d'une part, de ne pas polluer les modèles fédérés avec des informations liées à l'établissement de liens de fédération, et d'autre part de pouvoir définir des liens de fédération explicitement et indépendamment des couches techniques.
2. **Partionnement de l'espace conceptuel entre les types et les instances** : contrairement à MF², un choix de conception a été fait de séparer les types et les instances dans l'espace conceptuel comme le montre la figure 4.4. Alors que MF² promeut une définition de modèle générique, nous définissons dans FML deux métamodèles différents : l'un concerne la définition des types (VirtualModel et FlexoConcept ; c'est le métamodèle FML), tandis que l'autre est axé sur les instances (VirtualModelInstance et FlexoConceptInstance ; c'est le métamodèle FML@runtime).
3. **Un langage interprété** : l'exécution de FML, qui est un langage entièrement interprété, permet l'instanciation et la manipulation d'instances de *FlexoConcept*. VirtualModel et FlexoConcept sont instanciés (respectivement en tant que VirtualModelInstance et FlexoConceptInstance) et sont exécutés au niveau FML@runtime par un interpréteur FML. L'infrastructure logicielle Openflexo propose une implémentation de cet interpréteur.

4. Soulignons ici une distinction majeure entre MF² et FML : tandis que MF² se concentre uniquement sur le cœur de fédération, FML propose également la gestion et la manipulation des objets externes.

4. **Une approche orientée objet** : le paradigme de l'*orienté objet* a été choisi pour les avantages procurés en termes de modularité et de réutilisabilité. Les objets sont ici des instances de FlexoConcept et possèdent à la fois des données et des méthodes (structure et comportements). Les caractéristiques souhaitées pour FML sont l'abstraction, l'encapsulation, le polymorphisme et l'héritage (multiple). L'idée du pararadigme *orienté objet* est ici de concevoir une fédération comme un ensemble d'objets en interaction.

5. **Gestion de la contenance** : les VirtualModels et les FlexoConcepts sont structurés nativement au sein du langage FML dans un arbre de contenance. Cette propriété de contenance est explicitement exposée dans le langage. Ainsi tout FlexoConcept est explicitement déclaré dans un conteneur unique (qui est soit un FlexoConcept, soit un VirtualModel), et accessible dans le langage. Dans le monde des instances, tout FlexoConceptInstance s'exécute donc dans un autre FlexoConceptInstance ou un VirtualModelInstance [5].

6. **Typage statique et typage fort** : bien que le langage soit interprété, un compilateur FML permet d'analyser les unités de compilation FML. FML est statiquement typé et fortement typé. Tous les types doivent être explicitement décrits, même si une inférence de type est également effectuée. En tant que langage fortement typé, le compilateur FML vérifie de nombreuses règles de typage, et notamment l'assignabilité. La plupart de ces règles concernent l'affectation des variables, les valeurs de retour des comportements, les arguments des comportements et l'appel des comportements (liaison dynamique). Tous les objets sont typés, non seulement les objets du cœur de la fédération FML, mais aussi les objets référencés dans l'espace d'information (via le type des MirrorObject établissant le lien vers l'objet externe). Cela implique la gestion d'un espace de typage obtenu par composition de tous les espaces technologiques fédérés.

7. **Utilisation d'*adaptateurs technologiques* explicites** : l'objectif principal d'un adaptateur technologique est d'offrir une API claire et explicite de l'espace technique à l'espace conceptuel. Dans la pratique, un adaptateur technologique offre une définition réflexive des types et des fonctionnalités (*features*) structurelles et comportementales utilisables et qui vont être composés au sein de l'espace conceptuel FML.

8. **Une syntaxe générique** : la syntaxe FML a été conçue pour offrir des constructions syntaxiques génériques, c'est-à-dire adaptées à de mutiples paradigmes quant aux données et aux concepts manipulés. Concrètement le langage FML offre, à partir de constructions syntaxiques génériques, des points d'extension permettant de référencer des types et des fonctionnalités structurelles ou comportementales définies dans de multiples adaptateurs technologiques.

9. **Langage impératif** : ce paradigme de programmation a été choisi pour tenir compte de la diversité des comportements externes (fournis par les technologies fédérées sous-jacentes) à composer. En pratique, FML offre un style de programmation structurée avec des structures de contrôle spécifiques : séquences, sélection et itération [6]. Des primitives comportementales atomiques sont fournies par les adaptateurs technologiques utilisés dans un contexte donné et sont composées.

5. Un mécanisme de "bootstrap" permet la création d'un VirtualModelInstance racine sans conteneur.
6. en évitant les instructions de saut (*goto*) !

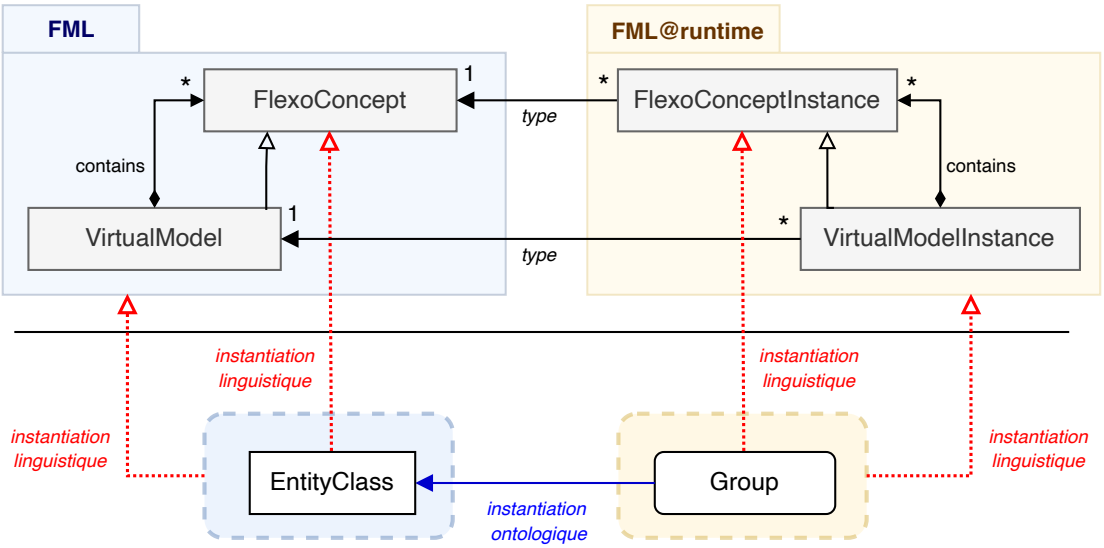

FIGURE 4.4 – Instanciation ontologique et instanciation linguistique

4.2 Description du langage FML

Nous avons choisi de ne présenter ici que les concepts fondamentaux du langage FML et leur sémantique intuitive, mais l'annexe B et le rapport scientifique [97] détaillent plus longuement le langage ainsi que sa syntaxe et sa sémantique.

4.2.1 Métamodèle FML

Comme indiqué dans la section 4.1.3 et contrairement à MF^2 qui promeut une définition de modèle générique, nous définissons dans FML deux métamodèles différents : le métamodèle FML concerne la définition des types tandis que le métamodèle FML@runtime se concentre sur les instances.

Cette partition est illustrée sur la figure 4.4 avec le scénario A présenté dans la section 4.1.2 et sur la figure 4.3. `EntityClass` représente ici une instance (linguistique) de FlexoConcept, et `Group` représente une instance (linguistique) de FlexoConceptInstance. `Group` est également une instance ontologique de `EntityClass`, conformément à la définition de la propriété `type` reliant les deux métamodèles FML et FML@runtime.

En ce qui concerne le cœur conceptuel du langage FML lui-même, nous suivons ici les règles du *strict metamodelling* [7] dans la mesure où un FlexoConceptInstance est instance d'exactement un FlexoConcept. Notons cependant que le paradigme du *strict metamodelling* ne s'applique ici qu'aux aspects syntaxiques de concepts FML. Au sein de l'espace conceptuel, il est possible de définir des références à plusieurs niveaux de modélisation, et donc *in fine* de proposer une modélisation multi-niveaux [96] (cf section 6.2).

La figure 4.5 présente la structuration des métamodèles FML et FML@runtime sous la forme d'un diagramme de classes de type UML. À gauche, le métamodèle FML définit les concepts du langage, tandis que le métamodèle FML@runtime réifie à droite les instances

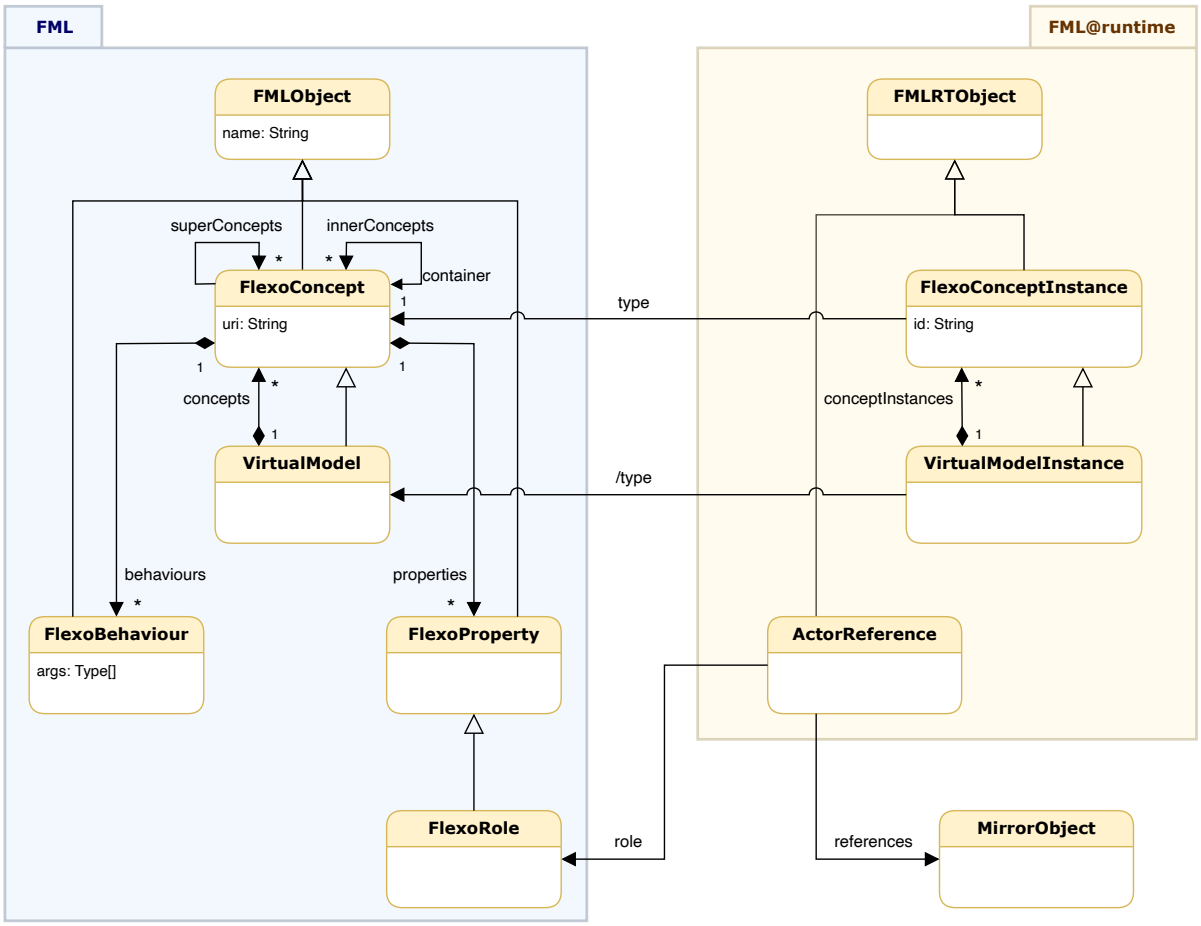

FIGURE 4.5 – Structuration des métamodèles FML et FML@runtime

ontologiques des concepts FML. Notez le concept ActorReference qui pointe sur un objet externe (spécifique à une technologie donné) par l'intermédiaire d'un MirrorObject.

Enfin, la figure 4.6 fournit plus de détails sur l'architecture (simplifiée) du métamodèle FML. Tous les concepts ne sont pas représentés pour des raisons de lisibilité.

L'élément central est FlexoConcept qui permet la réification d'un concept métier. Un FlexoConcept définit à la fois des propriétés structurelles (FlexoProperty à gauche) et des comportements (FlexoBehaviour à droite). Les FlexoConcepts se structurent au sein de VirtualModels qui les contiennent (gestion de la contenance), et s'organisent selon une sémantique d'héritage (héritage multiple : un FlexoConcept hérite d'un ensemble de FlexoConcept). On se réfèrera à la section B.2 page 191 pour une définition précise et formelle.

Les fonctionnalités structurelles (encore appelées propriétés ou FlexoProperty) permettent de définir ou de référencer une donnée (interne ou externe). Ces propriétés peuvent être simples ou calculées. Elles peuvent également être en lecture seule ou autoriser une modification de leur valeur. La figure 4.6 présente ainsi une définition de variable simple (FMLVariable) ainsi que la référence à une donnée externe (FlexoRole) ou à un modèle fédéré (ModelSlot). Une propriété peut être calculable par une expression FML (ExpressionProperty) ou même par l'exécution d'un "morceau de programme" FML (GetProperty et GetSetProperty). Ces fonctionnalités structurelles sont décrites dans la section B.11.

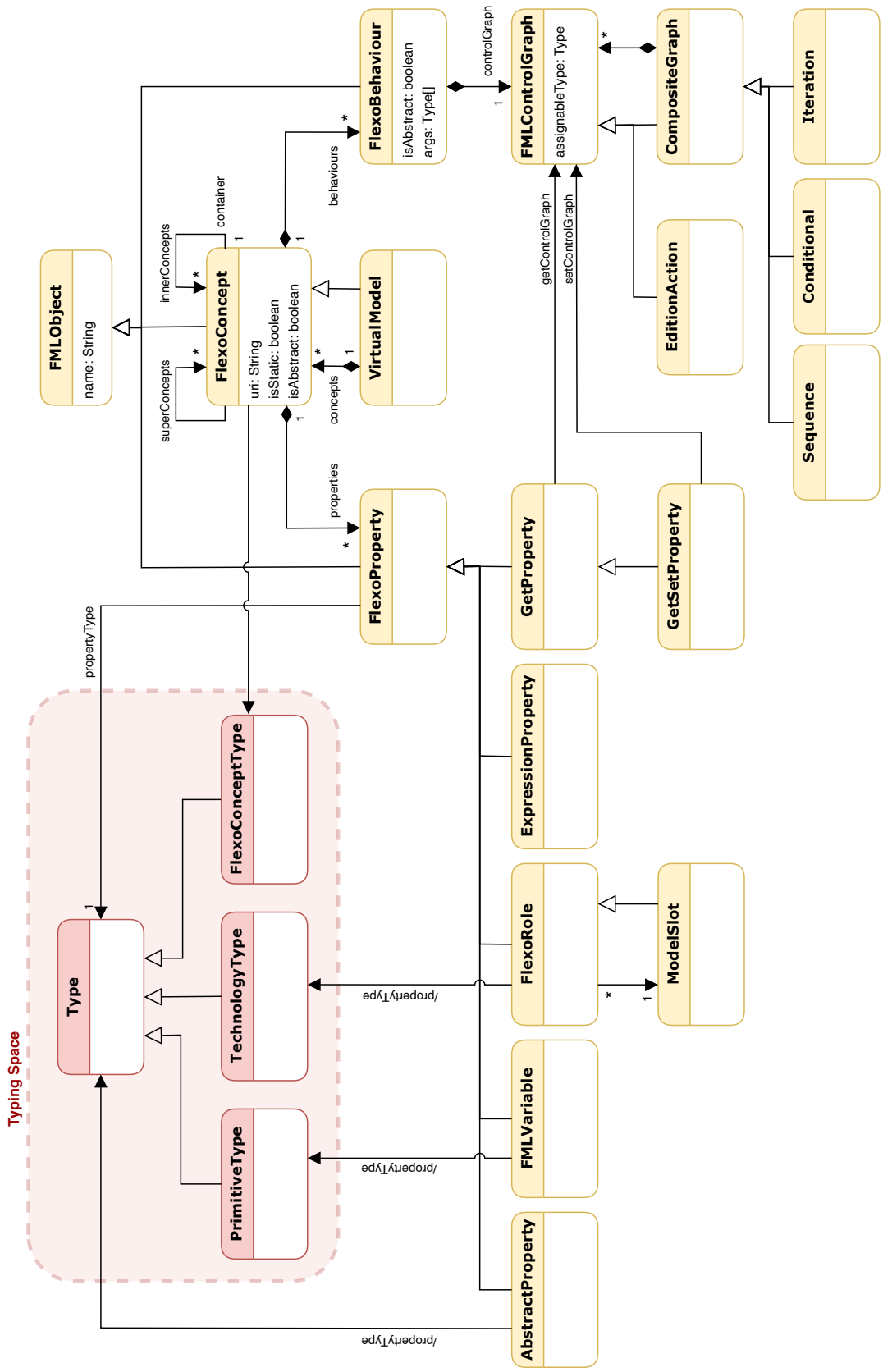

FIGURE 4.6 – Métamodèle FML (tous les concepts ne sont pas représentés)

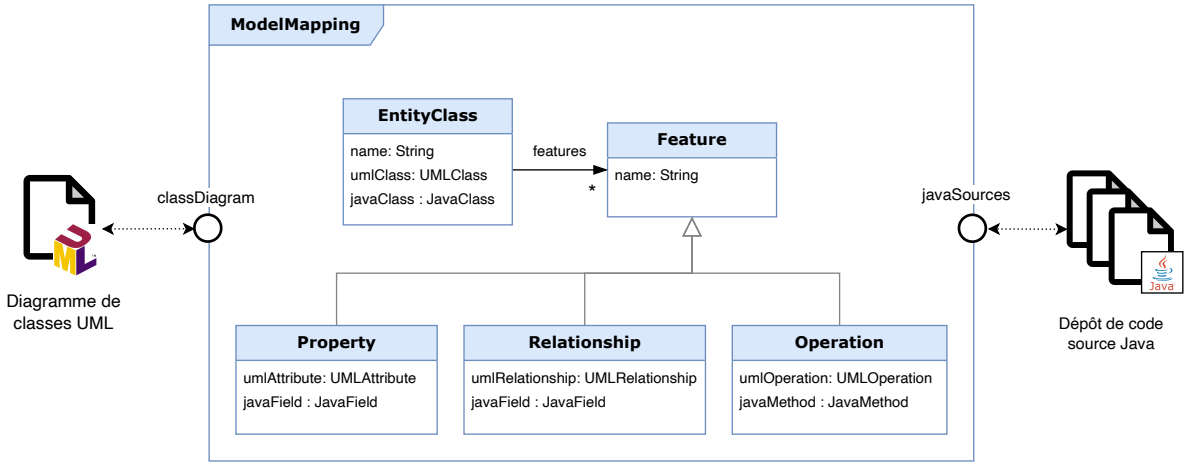

FIGURE 4.7 – Implantation (partielle) du scénario A en FML

Les fonctionnalités comportementales (FlexoBehaviour) sont très similaires à la notion de méthode d'instance de la programmation objet. Elles peuvent être liées au cycle de vie d'une instance (construction, destruction, action de base), et leur déclenchement peut être explicitement appelé, ou bien lié au cycle de vie des modèles fédérés (mécanismes réactifs). Elles sont décrites dans la section B.12. FMLControlGraph est l'abstraction générique qui permet de définir des comportements dans le paradigme de programmation impérative, en composant des structures de contrôle (Sequence, Conditional, Iteration) avec des primitives comportementales atomiques (expressions, assignations, action "de base" offerte par une technologie donnée). Tout ceci est décrit exhaustivement dans les annexes (sections B.9 et B.10).

Enfin, la figure est complétée par une hiérarchie partielle de types, définissant un espace de types qui composent le système de types FML (en haut à gauche de la figure et également décrit en annexe B.8).

4.2.2 Un exemple de code FML : le cas du scénario A

Nous poursuivons la démarche d'implantation du scénario A décrit dans la section 3.4.1. Les listings 4.8, 4.9, 4.10 et 4.11 proposent une solution de conception potentielle au problème exposé dans le cadre de ce scénario, sous la forme d'une unité de compilation FML (un fichier `ModelMapping.fml`)[7]. Ce code FML est également illustré dans la figure 4.7.

Le VirtualModel `ModelMapping` (lignes 7-95, listing 4.8) concrétise un lien conceptuel entre un diagramme de classes UML, et un dépôt de code source Java. Ce lien est raffiné en deux liens plus techniques réfiés en tant que propriétés structurelles via la notion de ModelSlot (lignes 9 et 10), et qui correspondent au référencement de ces sources de données depuis l'espace conceptuel. Cet exemple illustre la syntaxe du langage pour définir une propriété. Une propriété est définie par son nom (ici `classDiagram` et `javaSources`), son type résultant (respectivement `UMLClassDiagram` et `SourceRepository`), mais aussi par la "façon" d'accéder à la donnée, via une clause "with" qui précise la nature du ModelSlot à utiliser pour

7. Le code source de FML est organisé en unités de compilation (FMLCompilationUnit). Une unité de compilation simple contient une unique définition de VirtualModel et porte le nom de ce VirtualModel.

```
1 use org.openflexo.ta.emf.UMLModelSlot as UML;
2 use org.openflexo.ta.java.RepositoryModelSlot as JAVA_REPO;
3 use org.openflexo.ta.java.FileModelSlot as JAVA;
4 ...
5 /* Un VirtualModel dedie a la correspondance entre un diagramme de classe UML
6    et un depot de code source Java */
7 model ModelMapping {
8
9  UMLClassDiagram classDiagram with UML::UMLModelSlot();
10 SourceRepository javaSources with JAVA_REPO::RepositoryModelSlot(jdk="Java11+");
                ⋮
95 }
```

FIGURE 4.8 – Implantation du scénario A : définition du VirtualModel `ModelMapping`

```
7 model ModelMapping {
                ⋮
12  // Concept modelisant la correspondance entre une classe du diagramme UML
13  // et une classe Java dans le code source
14  concept EntityClass {
15
16    UMLClass umlClass with UML::UMLClassRole(container=classDiagram);
17    JavaClass javaClass with JAVA::JavaClassRole(container=javaSources);
18    String name values umlClass.name;
19    List<Feature> features {
20      List<Feature> get() {
21        // Retourne l'ensemble des Features dont le conteneur est l'instance EntityClass courante
22        return (select Feature from this);
23      }
24    };
25
26    // Constructeur du concept EntityClass
27    create(UMLClass umlClass,JavaClass javaClass) {
28      this.umlClass = umlClass;
29      this.javaClass = javaClass;
30    }
                ⋮
66  }
                ⋮
95 }
```

FIGURE 4.9 – Implantation du scénario A : définition du FlexoConcept `EntityClass`

```
14  concept EntityClass {
                ⋮
33    // Definition generique et abstraite d'une fonctionnalite
34    abstract concept Feature {
35       abstract String name;
36    }
37
38    // Modelise le lien entre un attribut UML et un attribut Java
39    concept Property extends Feature {
40      UMLAttribute umlAttribute with UML::UMLAttributeRole();
41      JavaField javaField with JAVA::JavaFieldRole();
42      String name values umlAttribute.name;
43      Type type values umlAttribute.type;
44      ...
45    }
46
47    // Modelise le lien entre un relation UML et un attribut Java
48    concept Relationship extends Feature {
49      UMLRelationship umlRelationship with UML::UMLRelationshipRole();
50      JavaField javaField with JAVA::JavaFieldRole();
51      String name values umlRelationship.name;
52      EntityClass source values container.getEntityClass(umlRelationship.source);
53      EntityClass destination values container.getEntityClass(umlRelationship.
            destination);
54      ...
55    }
56
57    // Modelise le lien entre une operation UML et une methode Java
58    concept Operation extends Feature {
59      UMLOperation umlOperation with UML::UMLOperationRole();
60      JavaMethod javaMethod with JAVA::JavaMethodRole();
61      String name values umlOperation.name;
62      Type returnType values umlOperation.type;
63      List<Type> argsType values umlOperation.argsType;
64      ...
65    }
66  }
                ⋮
95 }
```

FIGURE 4.10 – Implantation du scénario A : définition des features

interpréter la-dite donnée. Le langage FML présente en effet une vision "contractuelle" du lien technique, et offre la possibilité de paramétrer ce lien (par exemple ligne 10, il est précisé que la compatibilité du code source concerne les versions 11 et supérieures de la JDK).

Un autre raffinement consiste à structurer des liens entre respectivement une classe UML et une classe Java. C'est le FlexoConcept `EntityClass` déjà évoqué dans la section 4.2.1 qui réifie ce type de lien (lignes 14 à 66, listing 4.9). Les liens techniques vers les données (classe UML et classe Java) sont réalisés via la notion de FlexoRole (lignes 16 et 17), de manière très similaire aux ModelSlot évoquée plus haut (avec également une clause contractuelle). Deux propriétés calculées complètent ce concept : la propriété `name` se définit par l'expression `umlClass.name` (définition d'une ExpressionProperty, ligne 18), tandis que la propriété `features` se définit par l'exécution d'un graphe de contrôle (FMLControlGraph, lignes 19-24). Le FlexoConcept `EntityClass` définit également un constructeur, c'est-à-dire

```
7  model ModelMapping {
                ⋮
68   // A behaviour to retrieve EntityClass by name
69   EntityClass getEntityClass(String name) {
70     return select unique EntityClass from (this)
71       where (selected.name == name);
72   }
                ⋮
74   // We use here a specific behaviour defined in UMLModelSlot listening for
75   // new UMLClass creation to perform synchronization
76   newClassCreated(UMLClass umlClass) with UML::ListenClassAdded(
77       observed=classDiagram) {
78     javaSources.createCompilationUnit(umlClass.name);
79     synchronize();
80   }
                ⋮
82   // The behaviour used to synchronize mapping from UML class diagram
83   synchronize() {
84     MatchingSet<EntityClass> matchingSet = begin match EntityClass from this;
85     for (UMLClass umlClass : select UMLClass from classDiagram) {
86       // We match EntityClass in matching set from umlClass
87       // Java class (JavaClass) is retrieved from expected name
88       match EntityClass in matchingSet from this
89         where (selected.umlClass=umlClass)
90         create(umlClass,
91               javaSources.getJavaFile(umlClass.name+".java").getMainClass()));
92     }
93     end match EntityClass in matchingSet delete();
94   }
                ⋮
95 }
```

FIGURE 4.11 – Implantation du scénario A : comportements pour le VirtualModel `ModelMapping`

un comportement `create` (lignes 27-30) qui initialise les valeurs des rôles avec les arguments passés en paramètre.

Un troisième niveau de raffinement concerne l'établissement de liens au niveau des fonctionnalités de classes (attributs et méthodes). C'est le concept abstrait `Feature` qui représente cette abstraction (lignes 34-36, listing 4.10), avec la déclaration d'une propriété abstraite qui représente le nom de la fonctionnalité (ligne 34). Ce concept `Feature` se décline dans un arbre d'héritage avec les concepts `Property` (relie un attribut UML à un attribut Java, lignes 39-45), `Relationship` (relie une relations UML à un attribut Java, lignes 48-55) et `Operation` (relie une opération UML à une méthode Java, lignes 58-65). Chacun de ces concepts est construit de manière similaire avec deux `FlexoRole`, l'un pointant vers l'élément UML correspondant et l'autre vers l'élément Java. Ces concepts sont en outre complétés par des propriétés calculées à partir des éléments UML (mot-clef `values`, exemple lignes 42-43, 51-53, 61-63).

Ce code contient également trois exemples de comportements pour `ModelMapping`. Le but ici n'est pas de couvrir toute les constructions syntaxiques de FML mais de donner une idée de l'expressivité du langage [8] :

— `getEntityClass(String)` (lignes 69-72, listing 4.11) : ce premier comportement renvoie l'instance (unique) de `EntityClass` dont le nom correspond à l'argument passé en paramètre.

— `newClassCreated(UMLClass)` (lignes 76-80, listing 4.11) : c'est un comportement réactif qui est invoqué quand une nouvelle classe est ajoutée dans le diagramme de classes. Son exécution conduit à la création d'un nouveau fichier Java (ligne 78), et à la mise à jour des liens.

— `synchronize()` (lignes 83-94, listing 4.11) : ce comportement illustre les constructions utilisables dans le langage pour établir des opérations de correspondance et de synchronisation. Le principe est de synchroniser les instances de `EntityClass` avec les classes trouvées dans le diagramme de classes UML, les instances non existantes étant créées (lignes 90-91), les instances existantes ne correspondant plus à une classe UML étant supprimées (ligne 93) et le reste n'étant pas modifié. Ce comportement repose sur des opérateurs de correspondance FML spécifiques, présentés en annexe B.13.3.

4.2.3 Gestion des objets externes et des technologies tierces

La prise en compte de l'hétérogénéité technologique levée par l'enjeu E2 nécessite de pouvoir traiter des objets externes au cœur conceptuel (l'espace conceptuel), c'est-à-dire les objets de l'*espace d'information*.

L'intégration d'une technologie spécifique dans le cœur de la fédération de modèles est rendue possible par l'utilisation d'un *adaptateur technologique* dédié à cette technologie. L'objectif principal d'un TechnologyAdapter est de fournir des types et des constructions spécifiques à une technologie, pour pouvoir les utiliser au sein de l'espace conceptuel FML. Ceci permet aux éléments de modélisation qui se situent dans l'espace d'information d'être exposés (référencés) en tant que MirrorObject et d'être manipulés vers/depuis l'espace conceptuel. En pratique, un TechnologyAdapter définit un ensemble de ModelSlotKind (des types de ModelSlot), qui représentent les différentes manières possibles fournies par le TechnologyAdapter pour exposer les données d'une ressource d'une technologie donnée. Ceci correspond aux différentes façons d'interpréter une telle donnée.

La table 4.1 montre comment cette couche réflexive spécifique à une technologie est fournie au cœur FML en apportant des types et des primitives structurelles et comportementales au langage. À titre d'exemple, nous évoquons ici le TechnologyAdapter Java utilisé dans l'implantation du scénario A présenté plus haut. Ce TechnologyAdapter expose deux ModelSlotKind différents (`RepositoryModelSlot` et `FileModelSlot`), fournissant une interprétation de code source Java à deux niveaux de granularité différents (lignes 2 et 3 dans le listing 4.8). Ce TechnologyAdapter fournit également certains TechnologyTypes (des types spécifiques à cette technologie). La table 4.2 détaille ces deux ModelSlotKinds et présente les FlexoRoleKinds, EditionActionKinds et FlexoBehaviourKinds qui sont proposés pour chacun de ces types de ModelSlot.

8. On pourra toutefois se référer à l'annexe B ou à [97] pour plus de détails sur toutes les constructions du langage et leur syntaxe.

Java *TechnologyAdapter*		
TechnologyTypes	`SourceRepository`	Un dépôt de fichiers sources Java
	`JavaPackage`	Un paquetage Java
	`JavaFile`	Un fichier Java
	`JavaClass`	Une classe Java
	`JavaField`	Un attribut Java
	`JavaMethod`	Une méthode Java
ModelSlotKinds	`RepositoryModelSlot`	Accès à un `SourceRepository`
	`FileModelSlot`	Accès à un simple `JavaFile`

TABLE 4.1 – L'API du *TechnologyAdapter* Java exposée à l'espace conceptuel FML

`RepositoryModelSlot`		
FlexoRoleKinds	`JavaPackageRole`	Accès à un `JavaPackage`
	`JavaFileRole`	Accès à un `JavaFile`
	`JavaClassRole`	Accès à une `JavaClass`
EditionActionKinds	`CreateJavaPackage()`	Crée un `JavaPackage`
	`DeleteJavaPackage()`	Supprime un `JavaPackage`
	`CreateJavaFile()`	Crée un `JavaFile`
	...	...
FlexoBehaviourKinds	`ListenFileAdded()`	Un fichier Java est découvert
	...	...
`FileModelSlot`		
FlexoRoleKinds	`JavaClassRole`	Accès à un `JavaClass`
	`JavaFieldRole`	Accès à un `JavaField`
	`JavaMethodRole`	Accès à une `JavaMethod`
	...	...
EditionActionKinds	`CreateJavaField()`	Crée un `JavaField`
	`DeleteJavaField()`	Supprime un `JavaField`
	`CreateJavaMethod()`	Crée une `JavaMethod`
	...	...
FlexoBehaviourKinds	`ListenFieldAdded()`	Un attribut Java est découvert
	...	...

TABLE 4.2 – Détails des ModelSlots du *TechnologyAdapter* Java

La figure 4.12 présente comment un TechnologyAdapter donné étend le métamodèle FML de base. Un TechnologyAdapter définit en premier lieu un ensemble de TechnologyTypes (ce sont les types manipulés par le TechnologyAdapter et incorporés au système de types FML). Un TechnologyAdapter définit également un ensemble de ModelSlotKind. Chaque ModelSlotKind définit un ensemble de primitives (structurelles et comportementales) qui sont mises à la disposition du noyau FML :

— Un ensemble de FlexoRoleKind de tous les types de FlexoRole disponibles dans le contexte de ce ModelSlotKind, c'est-à-dire de tous les types de liens structurels permettant de se connecter à une donnée. Cette notion est liée à une interprétation spécifique, et s'apparente à un "contrat" paramétrable (utilisation par exemple lignes 16 et 17 du listing 4.9).

— Un ensemble de FlexoBehaviourKind de tous les types de FlexoBehaviour disponibles fournis par cette technologie (comportements spécifiques auxquels on peut se "brancher") dans le contexte de ce ModelSlotKind (le listing 4.11 présente lignes 76-80 l'exemple de l'utilisation d'un tel comportement).

— Un ensemble de EditionActionKind, qui sont tous les types de EditionAction fournis par cette technologie (primitives comportementales spécifiques) utilisables dans le contexte de ce ModelSlotKind. Ces primitives décrivent des actions de base réalisables dans l'espace technique auquel le FlexoRoleKind donne accès (utilisation par exemple ligne 78 du listing 4.11). Ces actions de base sont ensuite composées dans le langage impératif FML.

— Le TechnologyAdapter peut fournir une API de requêtage de données (d'éléments de modèle), à travers la fourniture de FetchRequest, qui sont des spécialisations de la notion d'EditionAction accessibles via une construction syntaxique spécifique (mot-clef `select`, utilisation par exemple lignes 70 ou 85 du listing 4.11). Cette construction est détaillée dans l'annexe B.13.1.

4.2.4 Gestion des resources

La fédération de modèles implique la gestion de multiples sources d'information considérées comme des modèles. La notion de Resource est une abstraction utilisée pour représenter l'accès à un "modèle", comme le montre la figure 4.13. L'idée est celle d'une indirection vers la donnée proprement dite, via une abstraction dont la préoccupation liée est celle des entrées/sorties.

Une ressource (Resource) :

— est identifiée par un identifiant (*URI*) représenté par une chaîne de caractères qui doit être unique,

— a un nom local (représenté par une chaîne de caractères),

— fournit les accès à la donnée (ResourceData),

— a son propre cycle de vie (peut être loaded ou unloaded),

— déclare (explicitement ou implicitement) ses dépendances à d'autres Resources (les dépendances cycliques sont autorisées).

ResourceData est une abstraction générique qui représente le contenu d'un modèle externe et donc une donnée dans laquelle se trouvent les MirrorObjects. MirrorObject représente un objet (externe) accessible depuis l'espace conceptuel (mais qui ne se trouve pas dedans). Un MirrorObject vise à refléter une "donnée externe" (un objet, un élément de modélisation,

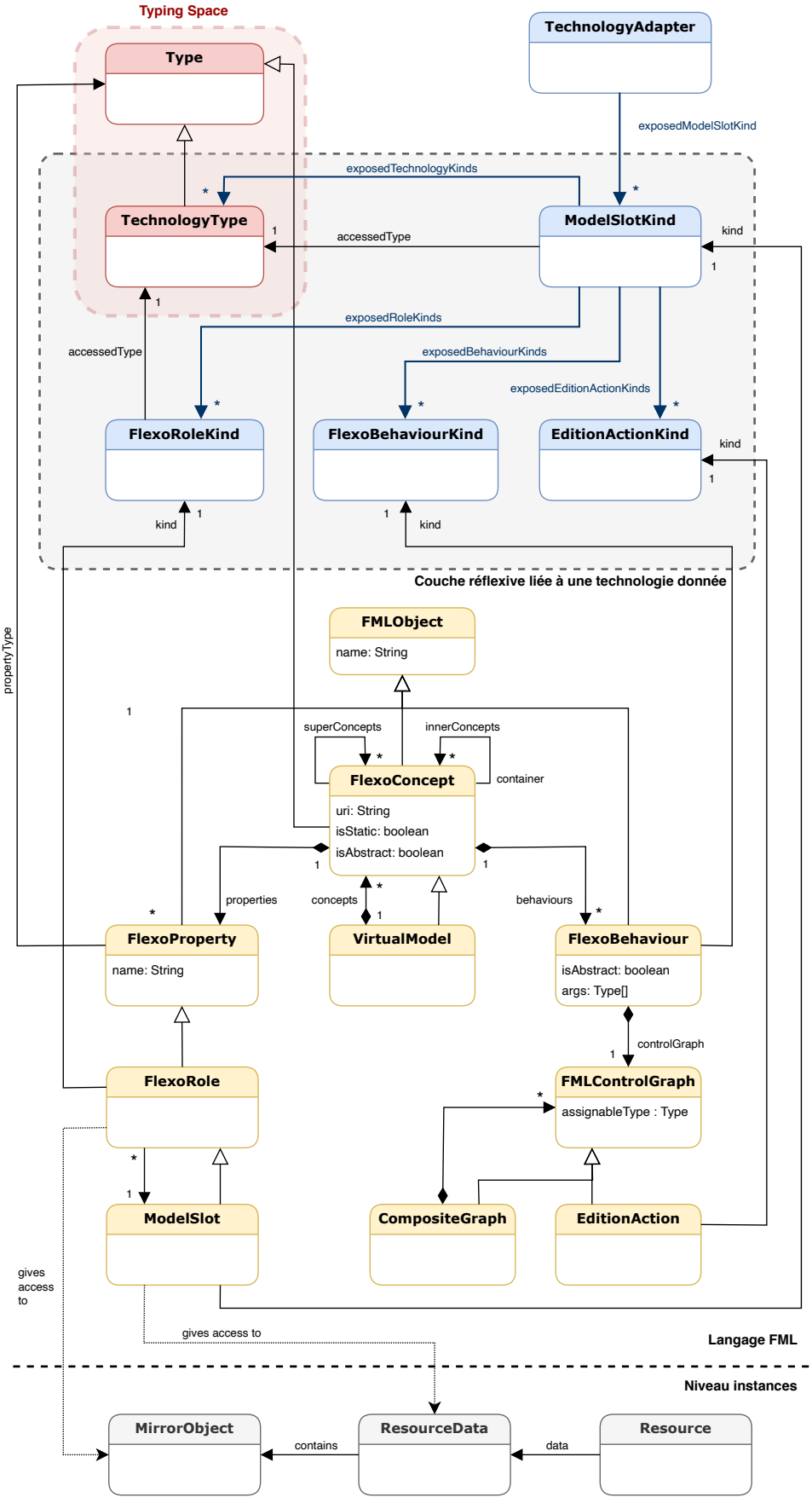

FIGURE 4.12 – Extension du modèle FML offerte par un TechnologyAdapter

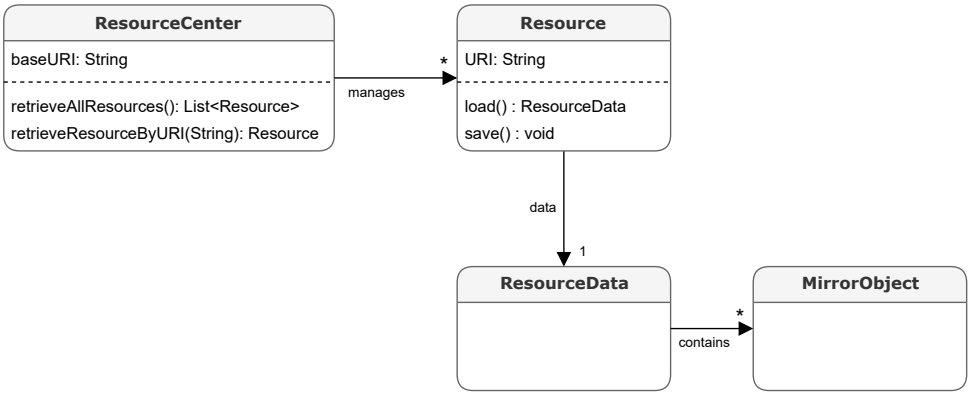

FIGURE 4.13 – Gestion des Resources

un élément d'information quel qu'il soit) qui se trouve dans l'espace d'information (plus précisemment au sein d'un espace technologique). Un MirrorObject peut être considéré comme un *proxy* et une représentation locale d'un objet spécifique externe.

L'abstraction ResourceCenter complète le mécanisme de gestion des ressources en fournissant des instances de Resource au moteur de fédération. D'un point de vue technique, un ResourceCenter fournit une couche d'accès à des Ressource à partir de différents supports (système de fichiers, réseau, bases de données, serveurs, etc.). Un ResourceCenter est organisé en dossiers (structure arborescente). Un ResourceCenter est également identifié par un identifiant (une *URI* unique, représentée sous la forme d'une chaîne de caractères). Pour une meilleure portabilité des ressources, l'*URI* des ressources contenues peut être calculée à partir de l'*URI* de base du ResourceCenter et de la structure des dossiers.

4.2.5 Système de types FML

En tant que langage à typage statique, toutes les constructions de FML sont explicitement et fortement typées (propriétés, comportements, variables locales, etc.). Ce système de typage permet de formaliser et de renforcer les règles de typage très en amont lors de la compilation (assignation de variables, calcul de l'assignabilité de types, valeurs de retour, appels de comportements).

La partie rouge de la figure 4.6 illustre une partie du système de types FML qui se définit comme l'union des ensembles de types suivants : [9]

1. **Les types représentés par les FlexoConcept** : tous les FlexoConcept sont considérés comme des types.
2. **Les types natifs FML** : le langage définit nativement un certain nombre de types, parmi lesquels on retrouve notamment :
   - les types primitifs (`int`, `long`, `short`, `byte`, `float`, `double`, `char`, `boolean`) : ce sont les mêmes types primitifs que ceux du langage Java ;
   - les types génériques : `List<Foo>` désigne par exemple une liste d'instances du FlexoConcept `Foo` ;

9. Ce système de types est décrit plus longuement dans l'annexe B.8.

- les types "*wildcard*" : également très semblables à Java, ils permettent de spécifier une bonne inférieure et/ou supérieure et permettent par exemple de construire le type `List<? extends Foo>` qui désigne une liste d'instances d'un FlexoConcept qui spécialise `Foo` ;
- des types spécifiques au langage (par exemple le type `Resource<RD>` [10], ou `MatchingSet<T>` [11]) ;
- les types réflexifs : `Concept<? extends Foo>` désigne par exemple le type FML d'une instance d'un FlexoConcept qui hérite du (ou qui est le) FlexoConcept <Foo>.

3. **Les types Java** : le langage FML étant construit au dessus du langage Java, tous les types Java sont utilisables.
4. **Les types spécifiques à une technologie** : le système de types FML comprend en outre tous les types exposés par tous les TechnologyAdapter utilisés dans un contexte donné. À noter que cet ensemble de types comprend également les types personnalisables et configurables dans la portée d'une unité de compilation. Ces types exposent au langage FML leur navigabilité, spécifique à la technologie sous-jacente [12].

Tous les types faisant partie du système de type FML implantent une fonction permettant de calculer l'assignabilité d'un type par rapport à un autre. Cette fonction permet de définir les relations transitives de sous-typage et de conformance d'une instance à son type, qui s'applique en premier lieu à tout l'espace conceptuel, mais aussi à tout l'espace d'information exposé par le biais de l'API MirrorObject [13].

4.2.6 Langage d'expression FML

Un langage d'expression fait partie de la définition de FML [14]. Ce langage d'expression est typé et est défini dans le contexte d'un espace de typage donné (définissant l'accès à un ensemble de types, par exemple une unité de compilation FML). Ce langage se définit par une combinaison d'opérateurs avec des terminaux appelés BindingPath.

Un BindingPath se présente sous la forme d'une liste chaînée d'éléments typés, telle que :

$$binding\_path ::= var\,.\,e_1\,.\,\cdots\,.\,e_n$$

Ce $binding\_path$ représente ici un "chemin" calculable où var est une variable résolvable dans un contexte donné, et où l'élément e_i représente une fonctionnalité résolvable dans le contexte de la résolution du type de l'élément e_{i-1} (ou var si $i = 1$). Cet élément e_i peut être soit une propriété simple (représentée par une chaîne de caractère), ou bien un appel de fonction paramétré avec des arguments représentés par des expressions.

Une expression FML peut être :

- un chemin (BindingPath) ;

10. où `RD` désigne un type de contenu de Resource
11. où `T` désigne un FlexoConcept et qui représente un type utilisé pour les constructions de mise en correspondance (*matching*), voir plus loin dans le document
12. Nous verrons plus loin (section 4.2.6) que FML inclut un langage d'expression qui permet de naviguer dans les modèles fédérés (et donc dans les objets externes). Dans ce cadre, les types spécifiques à une technologie permettent de définir des API spécifiques de navigation.
13. Tous les objets externes sont ainsi typés, et leur type est exposé et manipulable dans le langage FML.
14. On se référera à l'annexe B.9 pour plus de détails sur ce langage.

— une constante ;
— l'assignation d'une expression à une propriété pointée par un BindingPath ;
— un opérateur unaire appliqué à une expression ;
— un opérateur binaire appliqué à deux expressions ;
— une conditionnelle avec une expression qui exprime une condition, et deux expressions alternatives ;
— une expression dont le type est converti dans un type donné (*cast*) ;
— une expression de vérification de typage avec le prédicat typeof (*instanceof*).

Les opérateurs FML supportés sont très semblables à ceux utilisés dans le langage Java et comprennent les opérateurs unaires, binaires, logiques et arithmétiques (!,+, −, ∗, /, ++, −−, &, |, &&, ||, <, <=, >, >=, ==, !=, >>, <<) ainsi que les opérateurs d'assignation (=,+=,− =), avec des priorités bien définies [15].

Les expressions FML sont, en outre, équipées d'un système de parenthésage. On se référera à [97] pour plus de détails sur ces aspects syntaxiques ainsi que sur les opérateurs et les priorités.

Une expression FML est typée (l'exécution d'une expression bien formée garantit le retour d'une valeur conforme au type de l'expression). Les listings 4.9, 4.10 et 4.11 définissent et utilisent par exemple les expressions FML suivantes :

— `umlClass.name`, ligne 18, expression typée avec le type `String` (type FML primitif)
— `container.getEntityClass(umlRelationship.source)`, ligne 52, expression typée avec le type `EntityClass` (type FML)
— `javaSources.getJavaFile(umlClass.name+".java").getMainClass()`, ligne 91, expression typée avec le type `JavaClass` (type externe)

Le langage d'expression FML est défini sur un espace de typage qui comprend tous les types et les FlexoConcepts FML, mais aussi tous les types définis sur des espaces technologiques liés. Dans ce contexte, ce langage vise à fournir une navigation de données au sein de ces objets typés, et donc à rendre des services opérationnels (désigner et fournir des services par le biais d'un chemin). Pour ce faire, tout TechnologyAdapter fournit au langage d'expression FML des points d'extension via une BindingFactory qui définit des types spécifiques et des associations entre ces types et une liste de fonctionnalités (à la fois des propriétés simples et des appels de fonction paramétrés), ainsi que d'éventuels opérateurs unaires et binaires supplémentaires.

Ceci offre une très grande expressivité pour le langage FML. Si les notions de ModelSlot et FlexoRole permettent dans un premier temps l'accès à la donnée dans l'espace technologique, c'est le langage d'expression FML qui permet ensuite la navigation dans les espaces technologiques, à partir de ces points d'entrée, dans un paradigme adapté à la technologie visée. En pratique, cela rend le langage très opérationnel et adapté à la manipulation de données métier diversement structurées (enjeu E2, hétérogénéité).

Par exemple, la navigation dans une ontologie OWL permet de suivre des chemins à partir d'un *individual* en suivant les *object properties* et les *data properties* qui permettent de typer les données accédées (par exemple : `myDog.owner.address.city` si on imagine manipuler une ontologie avec les classes `Dog`, `Person`, `Address` et `City` et les *object properties* associées, ou encore exprimer `myCell.upper.value.asInt` sur une feuille de calcul pour désigner à partir d'une cellule la cellule immédiatement au dessus, et d'obtenir

15. et conformes aux spécifications Java

sa valeur en tant que nombre entier). Enfin, une même expression FML peut combiner différents paradigmes, pour par exemple exprimer `myDog.owner.address.city.name + myCell.upper.value.asString`.

4.2.7 Gestion des comportements FML

Nous rappelons que les modèles sont autonomes et disposent de leur propre cycle de vie, tant d'un point de vue technologique (dans leur espace technique d'origine) que méthodologique (au sein du processus métier existant pour lequel ils ont été conçus et sont édités et maintenus). Nous rappelons également le principe de non-intrusivité avec l'impossibilité de gérer des méta-données non persistées par la pile technologique d'origine.

Dans ce contexte et comme illustré sur la figure 3.9 page 69, l'enjeu de fédérer des modèles vivants (enjeu E3) requiert la double capacité pour le langage (1) de contrôler, d'agir et de modifier les modèles fédérés et (2) d'être à l'écoute des modifications des modèles fédérés. Ces deux aspects se retrouvent dans le langage FML.

FML se présente comme un langage suivant le paradigme de la programmation impérative, où les opérations se décrivent comme des graphes de contrôle qui manipulent des instructions simples, construites à partir de la notion d'expression FML typée présentée plus haut. Ces instructions sont ensuite composées avec des séquences, des affectations (assignations), des instructions conditionnelles et des boucles. Ces graphes de contrôle sont donc également typés. La logique impérative FML est décrite plus exhaustivement dans l'annexe B.10.

Les primitives comportementales spécifiques à une technologie (les actions de base offertes par les adaptateurs technologiques, cf EditionAction) complètent ce langage en offrant la capacité d'agir sur les modèles fédérés, conformément à une API offerte par le ModelSlot qui propose une interprétation de la donnée.

Ceci permet au langage de décrire des comportements sur les FlexoConcept. Le listing 4.9 présente par exemple un constructeur pour le concept `EntityClass` (lignes 27-30), dont l'implémentation est une séquence de deux affectations. La méthode `synchronize()` (listing 4.11, lignes 83-94), que l'on a décrite plus haut, est un autre exemple de comportement. On trouvera sur ce même listing ligne 78 un exemple d'exécution d'une primitive comportementale spécifique à la technologie Java (la création d'une classe Java portant un nom passé en paramètre). La logique impérative se retrouve également dans des propriétés calculables, par exemple pour le calcul de l'ensemble des fonctionnalités (*features*) du concept `EntityClass` (listing 4.9, lignes 19-24). L'annexe B.12 détaille plus longuement les différents types de comportements du langage, qui permettent d'agir et de modifier les modèles fédérés.

Un autre aspect de la gestion comportementale concerne la problématique d'être à l'écoute des modèles fédérés. Comme on l'a vu dans la section 4.2.3, une technologie peut également exposer des comportements spécifiques à l'espace conceptuel (cf ModelSlotKind). Il s'agit de comportements propres à une pile technologique donnée (et qui donc s'exécutent dans l'espace technique concerné), auxquels le langage FML permet de se "brancher" pour programmer le comportement de la fédération. Pour reprendre les notations de 3.4, c'est l'action "globale" $a_{\mathcal{F}}$ qui doit répondre à l'action "locale" a. La déclaration d'un tel comportement permet de définir un mécanisme pour la fédération qui sera déclenché par une

ressource externe fédérée. Cela permet par exemple de lier le code FML aux interacteurs des outils externes, et enfin d'implémenter des liens externes avec un comportement réactif.

Le listing 4.11 montre par exemple lignes 76-80 l'utilisation d'un comportement spécifique appelé ListenClassAdded et qui est défini dans le cadre d'un modèle UML (cf la déclaration use de UMLModelSlot, listing 4.8, ligne 1). Dans cet exemple, nous "écoutons" le diagramme de classes UML `classDiagram` déclaré à la ligne 9, et nous définissons lignes 78-79 le graphe de contrôle à exécuter lorsque ce comportement est déclenché.

Comme tous les langages objets, la liaison dynamique est la conséquence opérationnelle du support de l'héritage (cf [24]). La liaison dynamique consiste à retarder à l'exécution le choix du comportement à exécuter dans l'arbre (ou le graphe) d'héritage. Cette problématique est liée au typage dans des contextes de covariance et de contravariance. Dans ce cadre, la sémantique précise de FML a été étudiée et précisée. La signature du langage est décrite dans l'annexe B.12.

Nous avons évoqué en introduction de cette section l'enjeu E3 de la gestion de la dynamicité des modèles. Particulièrement, nous avons mis en évidence les deux aspects complémentaires de la gestion comportementale : la double capacité à agir sur les modèles fédérés et d'être à leur écoute pour éventuellement réagir. Ces deux aspects sont couverts par le langage au travers de la définition d'une logique impérative que l'on définir sur des comportements attachés aux FlexoConcept, et par la définition et l'utilisation de comportements spécifiques aux technologies mises en œuvre.

Concrètement, le langage FML nous permet de structurer les liens de fédération en les réifiant en tant que FlexoConcept, et d'attacher du comportement à ces liens. Au final, nous disposons de liens opérationnalisés, qui permettent à la fois un contrôle des modèles fédérés depuis la fédération, mais aussi de comportements réactifs aux modifications des modèles fédérés.

4.3 Mise en œuvre du langage FML

Nous nous proposons de poursuivre notre démarche méthodologique en confrontant le langage FML aux cinq scénarios illustratifs de l'approche présentés dans la section 3.2. Pour chacun de ces scénarios, nous discuterons des trois enjeux présentés dans notre problématique (section 3.1).

4.3.1 Scénario A (génération de code à partir du modèle avec round-trip)

Ce scénario a été présenté dans la section 3.4.1. Nous ne reviendrons pas sur son implantation en FML qui a servi de support à la description du langage (section 4.2).

La conceptualisation des liens entre le modèle UML et le code généré est au cœur de la problématique (enjeu E1). Ces liens ont été réfiés en tant que FlexoConcepts dans l'espace conceptuel. Ces liens ont été également raffinés en liens plus fins avec d'autres FlexoConcepts selon la sémantique de la contenance offerte par le langage (le FlexoConcept `EntityClass` est contenu dans le VirtualModel `ModelMapping`, et contient la hiérarchie de FlexoConcepts `Feature`).

L'hétérogénéité (enjeu E2) est inhérente au cas d'étude (diagramme de classes et code Java). Les adaptateurs technologiques UML et Java ont offert les types requis, les ModelSlots et FlexoRoles nécessaires, ainsi que les actions de base et enfin des comportements spécifiques indispensables à la gestion comportementale.

Les choix de conception présentés permettent de traiter la problématique de la gestion comportementale (enjeu E3) dans une perspective bi-directionnelle pour les trois cas d'utilisation suivants :

1. écoute des modifications du diagramme de classes et mise à jour du code Java,
2. écoute des modifications du code Java et mise à jour du diagramme de classes,
3. modifications depuis la fédération de modèles et mise à jour conjointe du diagramme de classes et du code Java.

4.3.2 Scénario B (construction de modèles)

La section 3.4.2 évoque le cas du scénario B qui se propose de co-construire un modèle et son méta-modèle EMF.

La technologie EMF offre un cadre de modélisation strict où la pile technologique contraint la relation de conformance χ entre un modèle et son métamodèle. Bien que cette contrainte technologique ne puisse être violée, ce scénario explore un processus de modélisation plus libre, où des étapes intermédiaires de modélisation peuvent manipuler des concepts non encore réifés en EMF, ou avec des concepts EMF (EClass et EObject) transitoirement désalignés.

Sans prétendre à une solution totalement fonctionnelle et opérationnelle, la figure 4.15 et le listing 4.14 illustrent comment le langage FML pourrait traiter ce cas d'utilisation. L'idée est d'autoriser un désalignement entre la relation de conformance technologique (d'un EObject à une EClass), et la relation de conformance conceptuelle (en cours d'élaboration par le modélisateur, cf enjeu E1). Ce dernier lien est réifié dans l'espace conceptuel au moyen de deux FlexoConcepts `Type` (lignes 19-26) et `Instance` (lignes 29-52), ces concepts étant eux-même liés par une relation de typage conceptuelle (`type`), à tout moment questionnable par le modélisateur. Ces deux concepts héritent d'une abstraction liée à une préoccupation purement descriptive (FlexoConcept `DescribableObject`, lignes 10-16). Ce choix de conception permet à une instance `Instance` d'exister sans être classifiée (constructeur par défaut, lignes 33-36), ou bien de servir de prototype à un nouveau type classifié (lignes 38-41), ou bien d'être associée a posteriori à un type existant (lignes 43-46).

Si la relation χ est homogène du point de vue de la technologie, elle concerne deux niveaux conceptuels distincts, que les outils classiques ne permettent pas de manipuler conjointement (enjeu E2). Un autre aspect intéressant apporté par cette conception réside dans la conceptualisation (réification) de la relation `Type-Instance`, qui peut se projeter dans d'autres espaces technologiques. La modélisation proposée ici pourrait s'enrichir d'une couche d'abstraction supplémentaire (avec des concepts abstraits) pour être réutilisée dans d'autres contextes, et fournir les mêmes fonctionnalités de co-constructions pour d'autres technologies.

La solution présentée n'offre pas de comportements réactifs liés à la modification externes des modèles (enjeu E3). Les choix de conception qui sont proposés permettent ce-

```
use org.openflexo.technologyadapter.emf.EMFModelSlot as EMF;
use org.openflexo.technologyadapter.emf.EMFMetaModelSlot as META_EMF;

model ConformanceModel {

    EMFMetaModel emfMetaModel with EMFMetaModelSlot();
    EMFModel emfModel with EMFModelSlot(metaModel=this.emfMetaModel);

    // Fonctionnalités de description
    abstract concept DescribableObject {
        String name;
        String description;
        create(String name) {
            name = parameters.name;
        }
    }

    // Un concept identifié comme un type dans le métamodèle
    concept Type extends DescribableObject {
        EClass eClass with EMFClassRole(container=emfMetaModel);

        create(String typeName) {
            super(parameters.typeName);
            eClass = META_EMF::CreateEMFClass(className=this.name);
        }
    }

    // Un concept identifié comme une instance dans le modèle
    concept Instance extends DescribableObject {
        Type type with ConceptInstance();
        EObject eObject with EObjectRole(container=emfModel);

        create(String instanceName) {
            super(parameters.instanceName);
            eObject = null;
        }

        declareInstanceOfNewConcept(String typeName) {
            type = new Type(parameters.typeName);
            this.createEObject();
        }

        declareInstanceOfExistingConcept(Type type) {
            type = parameters.type;
            this.createEObject();
        }

        private EObject createEObject() {
            eObject = EMF::CreateEObject(type=type.eClass);
            return eObject;
        }
    }

}
```

FIGURE 4.14 – Scénario B : extrait d'un code FML dédié à la co-construction d'un modèle et de son métamodèle

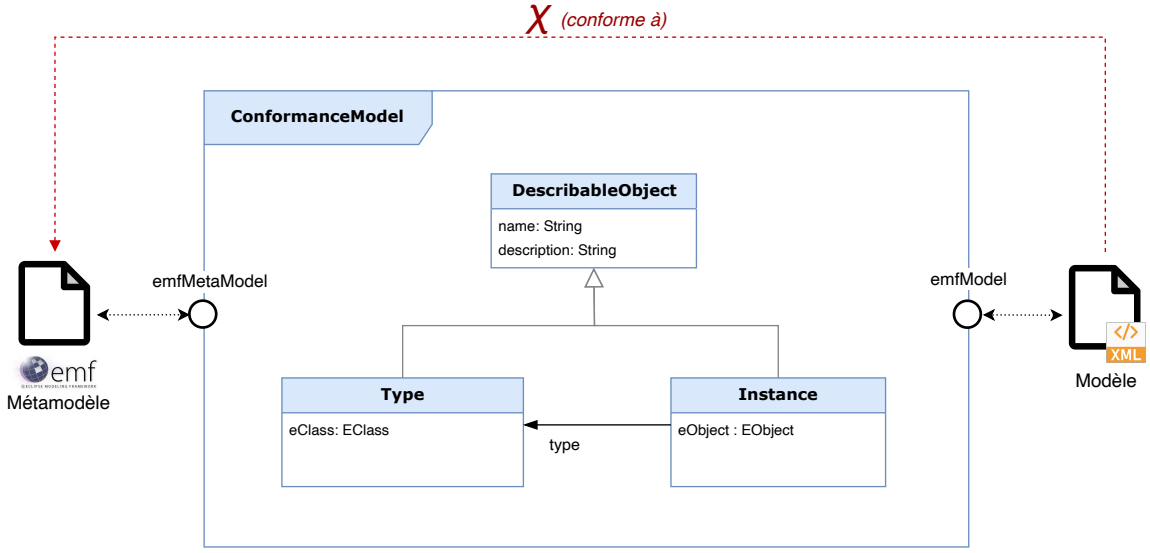

FIGURE 4.15 – Implantation (partielle) du scénario B en FML

pendant une implantation naturelle de tels comportements (écoute des deux modèles et métamodèles pour mise à jour automatique des `Type` et des `Instance`).

4.3.3 Scénario C (composition de modèles)

Ce scénario a été décrit dans la section 3.4.3. Il s'agit de conceptualiser un concept métier "Commande" à partir de deux sources de données non interopérables (une base de données qui stocke des "produits", et un fichier "clients" encodé dans un tableur).

La figure 4.16 et le listing 4.17 présentent un extrait d'une solution décrite dans le langage FML. Concernant le premier enjeu lié à l'interprétation et la conceptualisation, le concept métier "Commande" est réifié dans un FlexoConcept `Command` décrit dans le VirtualModel `CommandModel`. Deux ModelSlot sont déclarés dans le VirtualModel `CommandModel` pour encoder respectivement l'accès à la base de données "produits" et le fichier "clients".

Ce scénario est l'occasion de présenter l'exemple de ModelSlots qui permettent de refléter des données externes dans l'espace conceptuel en tant qu'instances d'un VirtualModel donné en paramètre, et qui donc spécifie l'interprétation des données pointées par le ModelSlot [16]. Le VirtualModel `ProductModel` va être utilisé par le ModelSlot `productModel` pour requêter la base de données et manipuler des instances du FlexoConcept `Product` (ligne 17) et le VirtualModel `CustomerModel` va être utilisé par le ModelSlot `customerModel` pour interpréter la feuille de calcul et manipuler des instances du FlexoConcept `Customer` (ligne 19).

L'idée est donc d'utiliser le langage FML comme un langage de spécification des données à interpréter. Ce mécanisme a montré son efficacité sur de nombreux cas d'utilisation, et s'avère indispensable pour gérer des données requêtables (par opposition aux données exhaustives où une ressource doit être entièrement parcourue pour exposer des données). C'est le cas par exemple des bases de données ou des services HTTP/REST.

16. De tels ModelSlots ont été implantés par exemple pour les technologies JDBC, Excel ainsi que HTTP/-Rest et HTTP/XML-RPC.

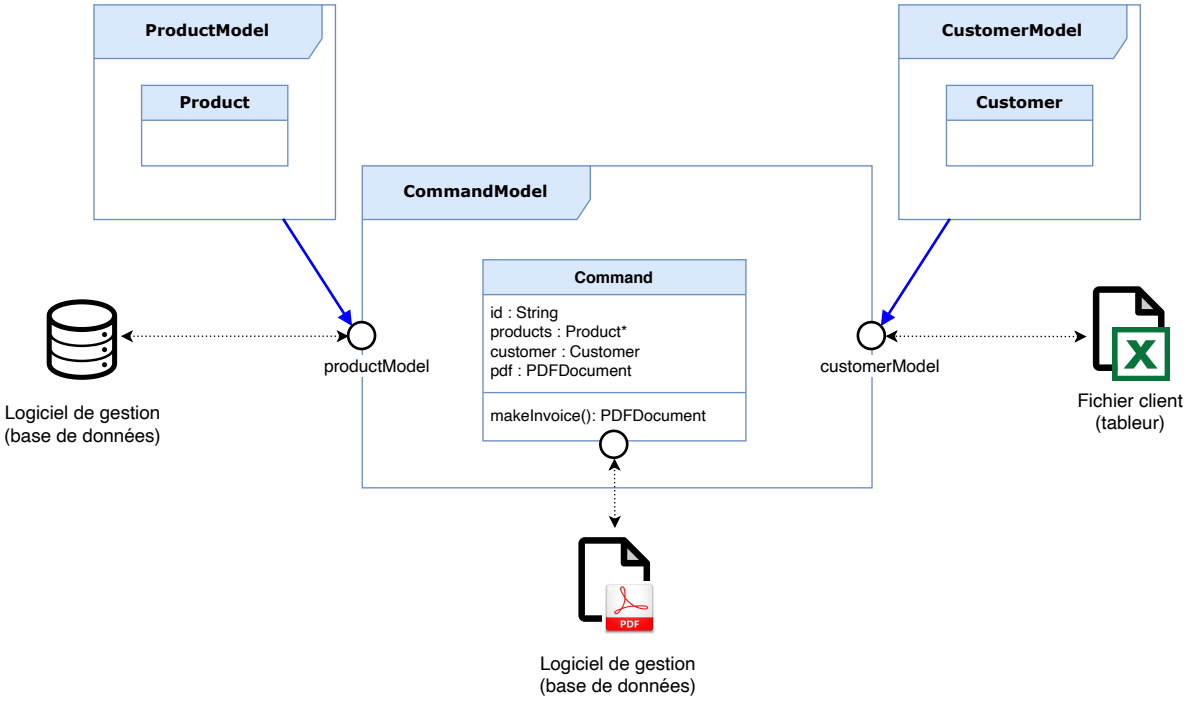

FIGURE 4.16 – Implantation (partielle) du scénario C en FML

La génération de document PDF s'opère via l'exécution d'une EditionAction spécifique, avec un "template" et les données représentées par l'instance courante du FlexoConcept `Command` (lignes 28-31).

Concernant l'hétérogénéité (enjeu E2), ce scénario illustre parfaitement l'intérêt de points d'extensions offerts par les adaptateurs technologiques. Ce cas d'utilisation fait intervenir trois technologies différentes par le biais de trois ModelSlot dédiés. Enfin, la définition des liens proposée par cette solution permet de conserver une grande liberté pour les modèles fédérés pour évoluer (enjeu E3). Il faudra cependant veiller à garantir l'intégrité des instances du FlexoConcept `Command`.

4.3.4 Scénario D (mise en correspondance de modèles)

Le scénario de mise en correspondance de modèles a été décrit dans la section 3.4.4. En général, ce cas d'utilisation constitue une pratique de l'IDM très différente du scénario A (génération de code à partir du modèle). Les outils généralement utilisés pour couvrir ces usages sont très différents. Pourtant, le traitement du scénario D de mise en correspondance de modèles s'avère être conceptuellement et techniquement très similaire au scénario A.

Selon les critères de mise en correspondance choisis, différents concepts vont être réifiés dans l'espace conceptuel (enjeu E1), en tant que FlexoConcept encodant les liens (ce sont les concepts C1, C2 et C3 de la figure 3.15, page 76). Le raffinement de ces liens peut au besoin se décomposer dans des liens plus fins, implantés en tant que FlexoConcept contenus (cf la sémantique de contenance du langage FML). Ces FlexoConcept vont encoder des comportements qui vont permettre d'exécuter la mise en correspondance, qui sera soit automatique et calculée, soit pilotée ou arbitrée par les décisions d'un être humain.

```
namespace "http://www.openflexo.org/test/ScenarioC" as SCENARIO_C;

use org.openflexo.technologyadapter.jdbc.JDBCModelSlot as JDBC;
use org.openflexo.technologyadapter.excel.SemanticsExcelModelSlot as EXCEL;
use org.openflexo.technologyadapter.pdf.PDFModelSlot as PDF;

import [SCENARIO_C + "/ProductModel.fml"] as ProductModel;
import [SCENARIO_C + "/CustomerModel.fml"] as CustomerModel;

import [SCENARIO_C+"/InvoiceTemplate.pdf"] as INVOICE_TEMPLATE;

// Représente un modèle de composition pour la manipulation de facture
@URI("http://www.openflexo.org/test/ScenarioC/CommandModel.fml")
model CommandModel {

   // La base de données est interprétée et reflétée en tant que ProductModel
   ProductModel productModel with JDBC::JDBCModelSlot(mappedModel=ProductModel);
   // Le tableur est interprété et reflété en tant que CustomerModel
   CustomerModel customerModel with EXCEL::SemanticsExcelModelSlot(mappedModel=
       CustomerModel);

   // Représente une commande
   concept Command {
      String id;
      Product[1,*] products with ConceptInstance(container=productModel);
      Customer customer with ConceptInstance(container=customerModel);
      PDFDocument invoice with PDF::PDFModelSlot(template=INVOICE_TEMPLATE);
      ...
      PDFDocument makeInvoice() {
         // Génère le document PDF à partir du "template" PDF et des données
         return invoice = PDF::GeneratePDF(template=INVOICE_TEMPLATE,data=this);
      }
   }
}
```

⋮

```
// Représente le modèle de "produits"
@URI("http://www.openflexo.org/test/ScenarioC/ProductModel.fml")
model ProductModel {
...
   concept Product {
   ...
   }
...
}
```

⋮

```
// Représente le modèle de "clients"
@URI("http://www.openflexo.org/test/ScenarioC/CustomerModel.fml")
model CustomerModel {
...
   concept Customer {
   ...
   }
...
}
```

FIGURE 4.17 – Scénario C : extrait d'un code FML dédié à la composition de modèles

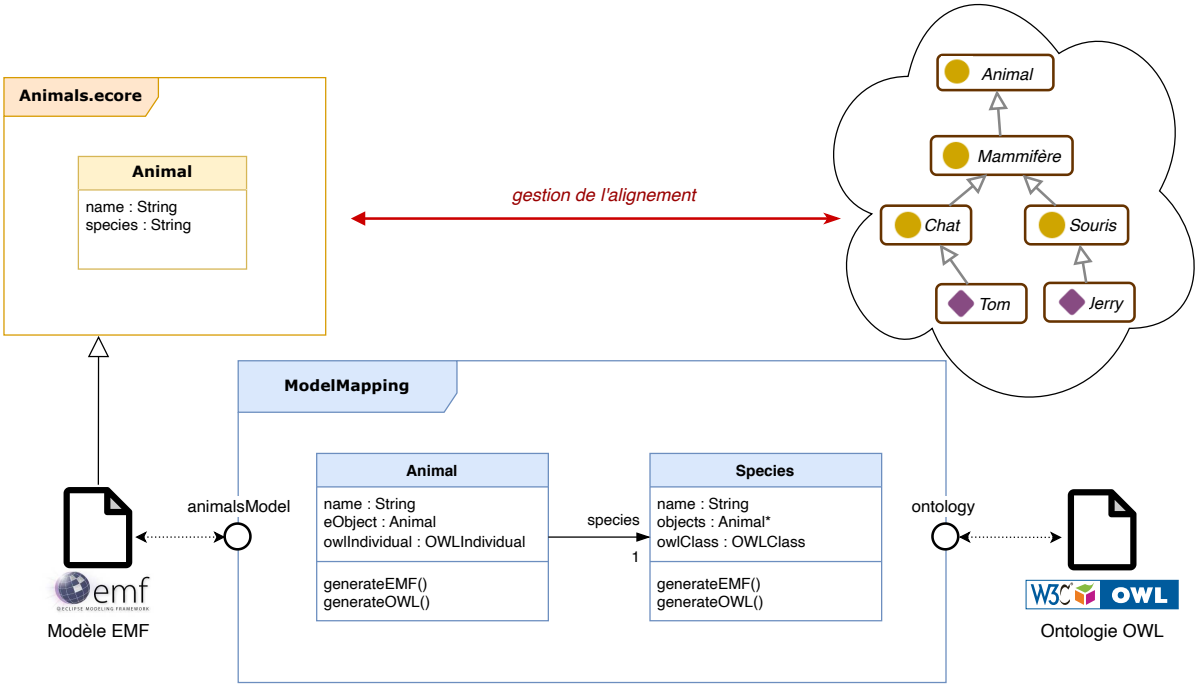

FIGURE 4.18 – Scénario D : mise en correspondance d'une ontologie avec un modèle EMF

La figure 4.18 et le listing 4.19 présentent un cas d'utilisation où il s'agit d'aligner un modèle EMF avec une ontologie OWL. Les critères de correspondance choisis requièrent la réification dans l'espace conceptuel des deux concepts `Animal` (lignes 35-47) et `Species` (lignes 49-55). Les critères de correspondance sont encodés dans le comportement appelé `synchronize()` (lignes 14-25).

Par rapport aux approches classiques, les aspects transformationnels sont répartis dans les liens de différentes granularités (lignes 27-31, 33, 43, 45), avec une logique compositionnelle dans le conteneur.

La prise en compte de l'hétérogénéité (enjeu E2) et de la dynamicité des modèles fédérés (enjeu E3) sont évidemment au cœur de la problématique de ce scénario.

4.3.5 Scénario E (édition de modèles)

Ce dernier scénario a été présenté dans la section 3.4.5 et concerne une fédération entre un modèle et sa représentation graphique (encodée en tant que modèle, ici un diagramme). L'intention est ici de construire un éditeur graphique.

Le listing 4.20 présente l'exemple d'un éditeur graphique d'ontologie OWL. Le VirtualModel `OntologyEditor` fédère une ontologie OWL (ligne 13) avec un diagramme "typé" (lignes 14-16). Ce diagramme est conforme à un "métamodèle" de diagramme qui est constitué de diagrammes d'exemple dans lesquels des formes et des connecteurs sont représentés pour être utilisés comme prototypes.

L'intention de représentation (enjeu E1) se traduit par la réification des trois concepts `OWLClassRepresentation`, `OWLIndividualRepresentation` et `IsARepresentation`, qui re-

```
use org.openflexo.technologyadapter.emf.EMFModelSlot as EMF;
use org.openflexo.technologyadapter.owl.OWLModelSlot as OWL;

import ["http://www.openflexo.org/test/Animals.ecore"] as ANIMALS;
import [ANIMALS:"Animal"] as ANIMAL;

typedef EMFObjectIndividualType(eClass=ANIMAL) as AnimalInEMF;

model ModelMapping {

   EMFModel animalsModel with EMFModelSlot(metaModel=ANIMALS);
   OWLOntology ontology with OWLModelSlot();
   ...
   synchronize() {
      MatchingSet<Animal> animalMatchingSet = begin match Animal from this;
      MatchingSet<Species> speciesMatchingSet = begin match Species from this;
      for (AnimalInEMF animalInEMF : select AnimalInEMF from animalsModel) {
         OWLIndividual individual = select unique OWLIndividual from ontology where
             (selected.name==animalInEMF.name && selected.types[0].name==animalInEMF
             .species);
         match Animal in animalMatchingSet from this where (eObject=animalInEMF,
             owlIndividual=individual) create(animalInEMF,individual);
         match Species in speciesMatchingSet from this where (selected.name==
             animalInEMF.species) create(animalInEMF,individual.types[0]);
      }
      for (OWLIndividual individual : select OWLIndividual from ontology) { ... }
      end match Animal in animalMatchingSet delete();
      end match Species in speciesMatchingSet delete();
   }

   generateOWLFromEMF() {
      synchronize();
      for (Animal animal : select Animal from this) { animal.generateOWL(); }
      for (Species species : select Species from this) { species.generateOWL(); }
   }

   generateEMFFromOWL() { ... }

   concept Animal {
      AnimalInEMF eObject with EObjectRole(container=emfModel);
      OWLIndividual owlIndividual with OWLIndividualRole(container=ontology);
      String name values (eObject != null ? eObject.name : individual.name);
      Species species;

      create(AnimalInEMF eObject,OWLIndividual owlIndividual) { ... }

      generateEMF() { ... }

      generateOWL() { ... }
      ...
   }

   concept Species {
      EObject[1,*] objects with EObjectRole(container=emfModel);
      OWLClass owlClass with OWLClassRole(container=ontology);
      String name values (owlClass != null ? owlClass.name : objects[0].species);

      ...
   }
}
```

FIGURE 4.19 – Scénario D : extrait d'un code FML pour la correspondance de modèles

```
namespace "http://openflexo.org/test/OntologyEditor" as DIAGRAM_SPECIFICATION;

use org.openflexo.technologyadapter.diagram.TypedDiagramModelSlot as DIAGRAM;
use org.openflexo.technologyadapter.owl.OWLModelSlot as OWL;

import [DIAGRAM_SPECIFICATION] as ONTOLOGY_EDITOR;
import [DIAGRAM_SPECIFICATION + "/ExampleDiagram.diagram"] as EXAMPLE_DIAGRAM;
import [EXAMPLE_DIAGRAM:"SHP-2"] as OWL_CLASS_SHAPE;
import [EXAMPLE_DIAGRAM:"CON-4"] as ISA_CONNECTOR;

// Un éditeur graphique pour une ontologie OWL
model OntologyEditor {
   OWLOntology ontology with OWLModelSlot();
   Diagram diagram with DIAGRAM::TypedDiagram(
      paletteElementBindings={ ... },
      diagramSpecification=ONTOLOGY_EDITOR);

   // C'est la représentation d'une classe OWLClass
   concept OWLClassRepresentation {
      OWLClass owlClass with OWLClassRole(container=ontology);
      DiagramShape shape with ShapeRole(label=owlClass.name,metamodelElement=
          OWL_CLASS_SHAPE);

      OWLClassRepresentation drop(String name) with DIAGRAM::DropScheme(target=
          OntologyEditor) {
         shape = DIAGRAM::AddShape(container=topLevel) in diagram;
         owlClass = OWL::AddOWLClass(className=parameters.name) in ontology;
      }

      create::representOWLClass(OWLClass anOWLClass) {
         shape = DIAGRAM::AddShape(container=diagram) in diagram;
         owlClass = parameters.anOWLClass;
      }
      ...
   }

   // C'est la représentation d'un individu OWLIndividual
   concept OWLIndividualRepresentation {
      ...
   }

   // C'est la représentation d'une relation "is a" (type)
   concept IsARepresentation {
      DiagramConnector connector with ConnectorRole(metamodelElement=ISA_CONNECTOR);
      OWLIndividual owlIndividual with OWLIndividualRole(container=ontology);
      OWLClass type with OWLClassRole(container=ontology);

      IsARepresentation associateIndividualToType() with DIAGRAM::LinkScheme(
          fromType=OWLIndividualRepresentation,toType=OWLClassRepresentation) {
         connector = DIAGRAM::AddConnector(fromShape=fromTarget.shape, toShape=
             toTarget.shape);
         ...
      }

      create::representIsA(OWLIndividual owlIndividual, OWLClass aType) {
         ...
      }

   }
   ...
}
```

FIGURE 4.20 – Scénario E : extrait d'un code FML pour l'édition de modèles

présentent respectivement une classe OWL, un individu OWL ou une relation de typage OWL entre un individu et une classe. Tous ces concepts définissent via des FlexoRole des liens vers l'élément dans l'ontologie, et sa représentation dans le diagramme. Une classe OWL est par exemple représentée (ligne 21) avec la forme prototypique `OWL_CLASS_SHAPE` (identifiée ligne 8 dans un diagramme d'exemple), avec la propriété `label` qui est redéfinie avec l'expression `owlClass.name` (calculable dans le contexte d'une instance du concept `OWLClassRepresentation`).

Deux comportements (enjeu E3) sont définis pour cette représentation : la création d'une classe OWL à partir de son nom, suite au déclenchement d'un interacteur spécifique lié au diagramme (ici un "drop" depuis la palette enchaîné avec la saisie d'un nom de classe, lignes 23-26) ; et la représentation d'une classe OWL existante dans l'ontologie, pour laquelle une nouvelle forme graphique est à créer (lignes 28-31).

Le paradigme mis en œuvre pour construire un éditeur graphique en utilisant la fédération de modèles est assez inhabituel dans le monde des outils, par le découplage qu'elle autorise entre un modèle et sa représentation. Ce découplage présente certains avantages : le "modèle" à représenter peut pré-exister, et/ou être modifié *a posteriori*, la représentation peut être incomplète et/ou n'être temporairement pas conforme (pour éventuellement faire l'objet d'une conformance plus tard), la représentation peut présenter des éléments qui ne se retrouvent pas dans le modèle. Enfin, ce paradigme permet de construire un éditeur graphique indépendamment de la technologie du modèle à éditer, et donc de capitaliser de l'outillage dans une perspective de modularité et de réutilisation.

4.4 Discussion et conclusions

L'exploration de l'espace des solutions apporté par la proposition du langage FML, au regard des cinq scénarios proposés, nous permet de discuter des trois principaux enjeux posés en problématique en tant qu'objectifs de recherche.

La figure 4.21 illustre et synthétise ces trois enjeux sur le cas d'utilisation proposé par le scénario A. Le concepteur de la fédération est illustré en bas de la figure. En tant qu'expert métier des deux domaines concernés (modélisation UML et code Java), il est responsable de la conception de la fédération et ses préoccupations requièrent des capacités d'abstraction et de conceptualisation, qu'il réifie dans l'espace conceptuel par l'écriture de code FML (enjeu E1). Les deux adaptateurs technologiques requis [17] (UML et Java) répondent à la problématique de l'hétérogénéité (enjeu E2), en fournissant les abstractions nécessaires à l'espace conceptuel. À droite et à gauche de la figure, les deux processus métiers fédérés sont chacun pilotés respectivement par le modélisateur UML et le développeur Java, au sein de leur espace technologique, et sont autonomes en terme de cycle de vie (enjeu E3). En haut de la figure, l'utilisateur de la fédération peut quant à lui piloter l'ensemble (enjeu E1, E2 et E3).

4.4.1 Enjeu E1 : interprétation

Ce premier enjeu, qui traite de la problématique de l'interprétation de la donnée, suggère de pouvoir considérer n'importe quelle source d'information comme un modèle (en par-

17. Ils peuvent être fournis ou développés sur mesure.

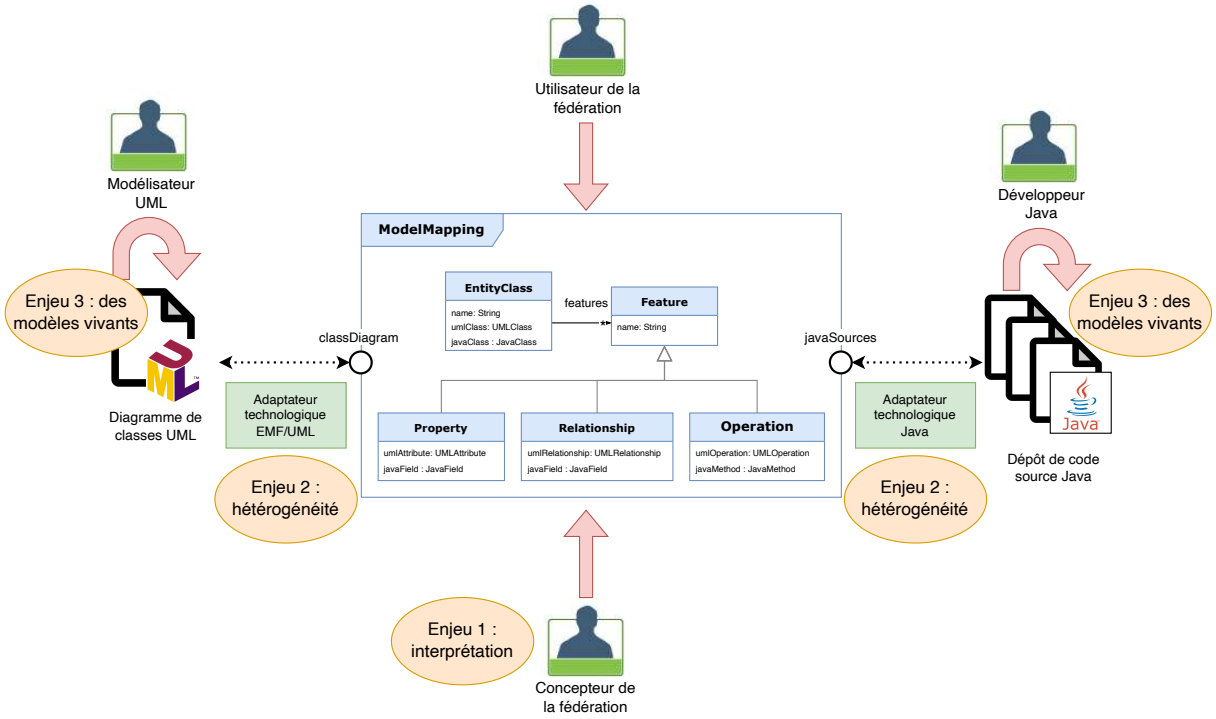

FIGURE 4.21 – Synthèse des enjeux

ticulier les données qui ne sont en général pas considérées comme telles, par exemple des documents textuels ou des tableurs, parce qu'elle n'exposent pas explicitement un métamodèle [18]). Dans ce contexte, FML se présente comme un langage de modélisation, et propose une réification de concepts métiers (FlexoConcept), au sein d'un espace conceptuel dédié dans lequel les FlexoConcepts portent une sémantique contextuelle explicite. L'interprétation de la donnée se concrétise au sein d'un FlexoConcept et se compose par l'établissement de liens techniques appelés FlexoRole ou ModelSlot (construction du lien sémantique par la composition de liens techniques), ainsi que de comportements.

La figure 4.22 propose une interprétation sémiotique du langage FML relativement au triangle d'Odgen et Richards présenté figure 2.2, page 13. Alors qu'un objet du monde réel (la "**chose**") est représenté dans un formalisme quelquonque (le "**mot**" au sein d'un "modèle" dans un espace technique), FML permet de réifier le "**sens**" (une interprétation) au sein d'un FlexoConcept dans l'espace conceptuel. Le lien "*symbolise*" est lui aussi réifié dans le langage FML par la notion de FlexoRole (ou ModelSlot). Cette relation peut éventuellement être décomposée en plusieurs FlexoRole pour accéder à plusieurs éléments de modélisation dans des espaces techniques différents.

Les cinq scénarios présentés comme des usages prototypiques de l'IDM nous ont donné l'occasion d'explorer la diversité des liens d'interopérabilité entre les modèles (μ, χ, composition, représentation, transformation, correspondance, etc...). Pour chacun de ces cas d'utilisation, nous avons vu comment la notion de FlexoConcept permet de réifier ce lien, et de porter le comportement associé à ce lien. Nous avons également traité de la structuration de cette couche conceptuelle, et débattu des apports de l'héritage (multiple) et de la gestion de la contenance au sein du cœur conceptuel. Nous avons introduit au travers

18. ou si le métamodèle associé comporte trop d'implicite, ou si le modèle doit être ré-interprété dans un autre contexte.

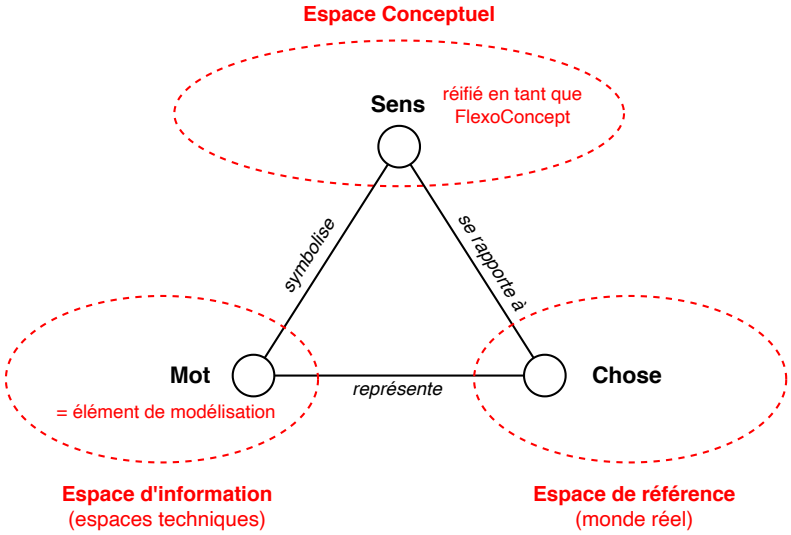

FIGURE 4.22 – Une interprétation sémiotique du langage FML

de ces cas de modélisation des aspects méthodologiques mettant en évidence l'intérêt de l'approche objet pour la modularité et les aspects réutilisation qu'elle procure.

Le langage offre en outre la possibilité de composer des éléments de plusieurs niveaux conceptuels. Un premier exemple, dans le scénario D, montre dans le VirtualModel `Model-Mapping` comment définir un type spécifique à partir d'une instance trouvée dans une ressource spécifique (listing 4.19, ligne 7). Un autre exemple, dans le scénario E, montre comment une forme graphique peut être prototypée à partir d'une autre (ligne 21 du listing 4.20). La solution présentée pour le scénario B (listing 4.14) démontre aussi les capacités du langage à faire interopérer des niveaux conceptuels différents.

Enfin, nous avons mis en évidence l'intérêt d'utiliser le langage de modélisation FML comme un support de représentation des données requêtables (cf scénario C, section 4.3.3), au sein d'adaptateurs technologiques dédiés (base de données, tableurs, services web, etc.), et qui permettent de refléter une source de données comme une instance de VirtualModel.

4.4.2 Enjeu E2 : hétérogénéité

L'enjeu de l'hétérogénéité est présent dans tous les scénarios évalués, par la manipulation de sources de données hétérogènes d'un point de vue technologique, mais aussi en terme de préoccupation initiale et de processus métier impliqué. Cet enjeu est traité naturellement par la notion d'adaptateur technologique.

Un adaptateur technologique fournit au cœur FML des abstractions opérationnelles, et notamment des connecteurs, des types, ainsi que des éléments structurels et comportementaux que le langage FML permet de composer. Les cinq scénarios nous ont permis de traiter de technologies diverses (UML, EMF, OWL, PDF, Excel, JDBC, XML) avec des paradigmes multiples (données exhaustives, données requêtables, documents textuels, code source, tableurs, diagrammes). Ces différentes primitives sont accessibles et intégrées dans le langage au moyen de constructions avec une syntaxe générique, par exemple les opéra-

tions de requêtage (via le mot-clef select, exemple listing 4.19, ligne 18) ou la définition de types paramétrables spécifiques à une technologie (cf typedef, exemple listing 4.19, ligne 7).

De par le langage d'expression fourni, FML offre une grande expressivité, avec une API générique permettant de naviguer dans les objets externes via des chemins spécifiques à la technologie (cf section 4.2.6). Ceci permet de définir des expressions spécifiques à une technologie, concises et typées "métier", tout en autorisant le mélange et la composition de paradigmes divers. Par ailleurs, le fait que ces expressions soient directement exécutables rend le langage FML efficace et opérationnel.

4.4.3 Enjeu E3 : dynamicité

FML est un langage de modélisation qui embarque son propre comportement. Ce sont les FlexoConcepts qui portent ces comportements, en suivant le paradigme OOP. Ces comportements permettent de décrire le cycle de vie des instances de ces FlexoConcepts (création, suppression, action, etc.), mais aussi les aspects réactifs liés à l'évolution de l'environnement des instances de FlexoConcepts.

En effet, la prise en compte de modèles "vivants" couvre deux aspects. Un premier aspect réside dans la capacité du langage FML à décrire un comportement dans l'espace conceptuel. Nous avons décrit les différentes primitives comportementales exposées par les adaptateurs technologiques, qu'il est possible de composer avec les actions de base et les structures de contrôle offertes au sein du langage FML, selon le paradigme de la programmation impérative. Au delà des primitives comportementales pour lesquelles une API simple est offerte au cœur conceptuel, le langage d'expression FML permet également l'évaluation de propriétés ou l'exécution de méthodes (appels procéduraux paramétrés) dans les espaces technologiques, sur la base de l'API Java.

Le deuxième aspect permet de lier la fédération à l'évolution des modèles fédérés, par des mécanismes réactifs et des mécanismes d'écoute. Le scénario A présente l'exemple d'un comportement qui se déclenche lorsque l'on détecte l'ajout d'une nouvelle classe dans un diagramme de classe UML (listing 4.11, lignes 76-80) [19]. Un adaptateur technologique peut aussi fournir des comportements liés à des interacteurs spécifiques de la technologie sous-jacente, par exemple dans le scénario E, listing 4.20, lignes 23-26, ce qui permet d'utiliser ici une technologie (un éditeur de diagramme) comme outil d'édition de modèles (en manipulant sa représentation).

Au final, le langage FML permet de définir des liens, qui portent leur propre sémantique contextuelle, et embarquent leur propre comportement.

4.4.4 Principes et exigences complémentaires

On rappelle qu'à ces trois grands enjeux se rajoutaient quatre principes ou exigences (cf section 3.1.4). Nous nous proposons ici de voir comment ces contraintes sont prises en compte dans le langage FML.

19. Si le langage permet conceptuellement ce comportement réactif, la responsabilité de la détection et de la propagation de cet évènement incombe à l'implantation de l'adaptateur technologique sous-jacent.

1. **Autonomie du cycle de vie des sources d'information fédérées** : concernant les modèles fédérés, le langage FML est construit autour de la notion de Resource, pour laquelle il n'est fait aucune hypothèse d'immutabilité. C'est la responsabilité de l'adaptateur technologique d'implanter les mécanismes de notification d'évolution des modèles.
2. **Connectivité intermittente** : le langage FML fournit des fonctionalités de persistence du modèle FML@runtime, ce qui permet de sérialiser l'état de la fédération (et donc l'établissement des liens de fédérations et des éventuelles données et/ou choix d'utilisateur). Ceci autorise les deux modes de fonctionnement (*online*/*offline*). Concernant la mise à jour "à chaud" (mode online), c'est l'adaptateur technologique requis qui assure le lien avec la ressource et gère les modifications (éventuellement concurrentes) de la ressource en dispatchant le/les évènements vers les comportements adéquats des FlexoConcepts concernés. Concernant le mode "offline" (la modification a eu lieu hors de l'instanciation du lien de fédération), la réconciliation doit être explicitement programmée en FML. Le langage est doté d'opérateurs de *matching* qui permettent de programmer de façon explicite ces stratégies de mise à jour (voir par exemple le listing 4.19, lignes 14-25).
3. **Non intrusivité** : le langage FML ne présuppose pas que toutes les technologies fédérées peuvent supporter la gestion de métadonnées utiles [20]. C'est à nouveau la responsabilité de l'adaptateur technologique concerné d'implanter une stratégie d'identification ou de référencement pérenne et persistante au regard des outils utilisés (par exemple la notion d'URI dans une ontologie qui permet de pointer un objet unique).
4. **Gestion de la source de vérité et minimisation de la redondance conceptuelle** : cet aspect méthodologique est traité au niveau conceptuel, dans le code FML, qui dispose de l'expressivité nécessaire à son implantation (par exemple l'accès aux métadonnées liées à l'édition des ressources, la définition de comportements connectés aux évolutions des données, la capacité à encoder et sérialiser des décisions humaines).

20. Quand ce support est disponible, il sera toutefois bienvenu pour encoder le lien vers la donnée à l'intérieur d'une ressource.

Chapitre 5

Implantation et outillage

Dans ce chapitre, nous décrivons une implantation Java open-source du langage FML au sein de la plate-forme Openflexo. Cette infrastructure logicielle propose entre autres un environnement de développement intégré (IDE) du langage et un interpréteur. La section 5.1 en pose les principes tandis que la section 5.2 en décrit l'architecture et certains des composants logiciels. Nous présentons dans la section 5.3 les principes de construction d'une application avec Openflexo avant d'évoquer section 5.4 le déploiement d'application web. Nous concluons dans la section 5.5.

Sommaire

5.1 Infrastructure logicielle Openflexo

Une implantation de l'approche *fédération de modèles* et du langage FML présentés respectivement dans les chapitres 3 et 4 est proposée au travers de l'infrastructure logicielle Openflexo. Plutôt qu'un logiciel monolithique, cette plate-forme se présente comme une bibliothèque de composants logiciels assemblables pour construire du logiciel traitant de la problématique de la fédération de modèles, et, au final, une usine à logiciels.

D'un point de vue fonctionnel, l'infrastructure Openflexo est illustrée sur la figure 5.1. Elle se présente comme une couche d'intermédiation entre l'espace d'information (l'ensemble des espaces technologiques, qui comprend les sources d'information à fédérer), et un outillage dédié fourni à des utilisateurs, sur la base de l'espace conceptuel géré en son sein.

La diversité des technologies concernées est évidemment importante, et le spectre fonctionnel de l'infrastructure est forcément très large. Pour ces raisons c'est le langage Java qui a été choisi, pour la richesse des bibliothèques Java disponibles et pour le dynamisme des différentes communautés de développeurs Java. Pour des raisons similaires - entre autres raisons - nous avons retenu le modèle *Open Source* avec le choix d'une double licence GNU GPL v3 et EUPL [1].

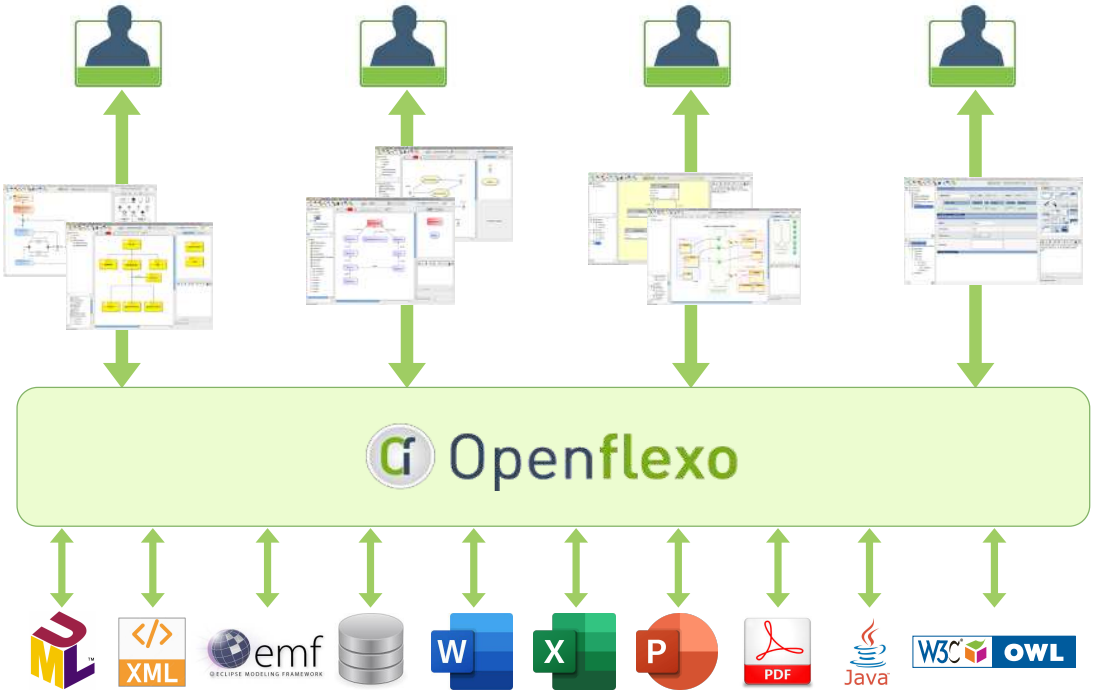

FIGURE 5.1 – Infrastructure Openflexo

1. C'est une licence dite à fort copyleft soit non permissive. Elle a été rédigée par la Commission Européenne pour répondre aux difficultés rencontrées dans la mise en œuvre des licences copyleft au niveau européen (conformité avec les différentes législations nationales) https://joinup.ec.europa.eu/sites/default/files/custom-page/attachment/eupl_v1.2_fr.pdf.

Cette infrastructure comprend notamment :

— une implantation du langage FML,
— un environnement d'édition et de construction (IDE) du langage FML,
— un environnement d'exécution du langage FML,
— des outils dédiés (langage de script, terminal, débuggeur, etc.),
— des adaptateurs technologiques,
— de l'outillage graphique (technologie Java/Swing)
— des applications packagées (clients lourds Java/Swing),
— un serveur Web,
— des bibliothèques de modèles techniques et métiers réutilisables (modèles, éditeurs, outils divers),
— différents composants logiciels génériques et réutilisables.

5.2 Architecture générale

La figure 5.2 présente l'architecture générale de l'infrastructure Openflexo. Cette architecture est modulaire et découpée en composants logiciels bien définis. Le code source de tous les dépôts liés à l'infrastructure Openflexo est disponible sur https://github.com/openflexo-team/.

L'implantation du cœur de fédération, et notamment du langage FML, est au centre de la figure 5.2. Cette implantation s'appuie sur un certain nombre de composants de plus bas niveau, et notamment sur le *framework* de modélisation PAMELA (cf section 5.2.2).

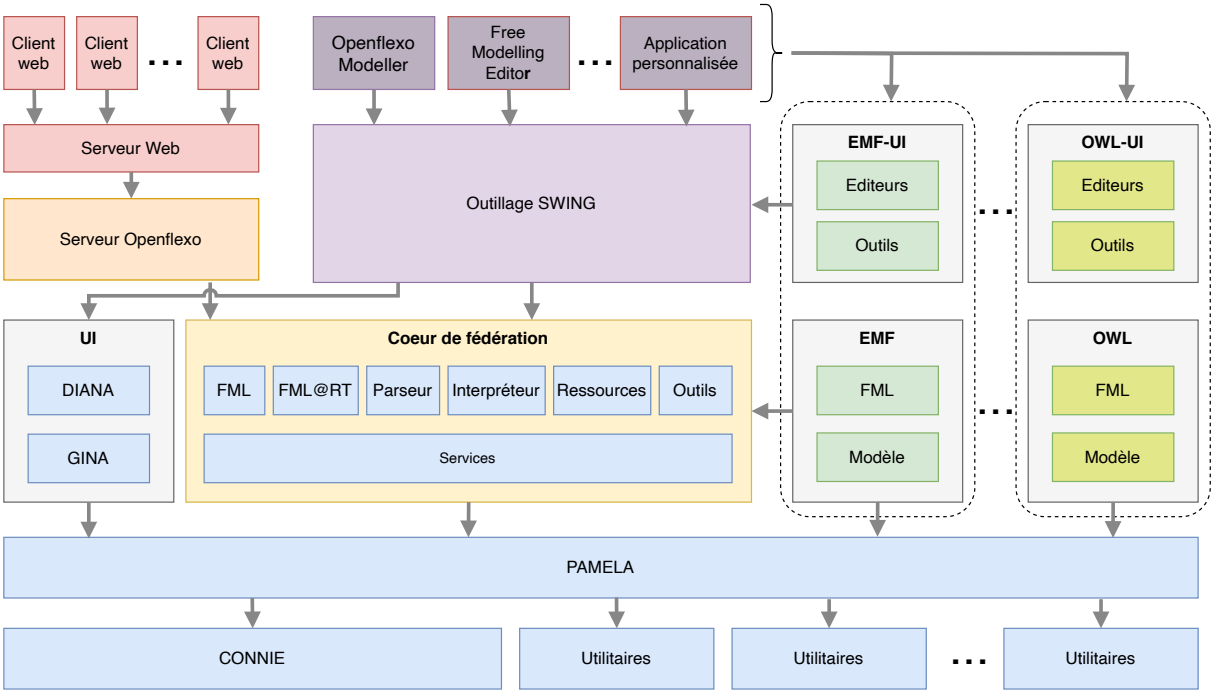

FIGURE 5.2 – Architecture de l'infrastructure logicielle Openflexo

La base de code résultante est très importante (800 000 lignes de code Java environ), notamment par la quantité de technologies intégrées dans l'infrastructure (adaptateurs technologiques). L'infrastructure logicielle fait l'objet d'une intégration continue [2] avec l'outil `Jenkins` [3], qui permet la compilation et le test de l'ensemble des composants, ainsi que l'intégration et la construction de différentes applications pour la mise en production.

5.2.1 Le coeur de fédération : Openflexo-Core

Le cœur de fédération, au centre de la figure, s'articule autour d'une architecture de micro-services qui coopèrent au sein du gestionnaire de service central. Chaque service correspond à un composant qui a ses propres responsabilités (par exemple un service pour la gestion de tous les *adaptateurs technologiques*, un service pour la gestion de tous les centres de ressources (ResourceCenters), des services pour la gestion de l'environnement d'exécution, de la console, du terminal, de l'éditeur, etc.). Le code source correspondant se trouve dans le dépôt `openflexo-core` [4].

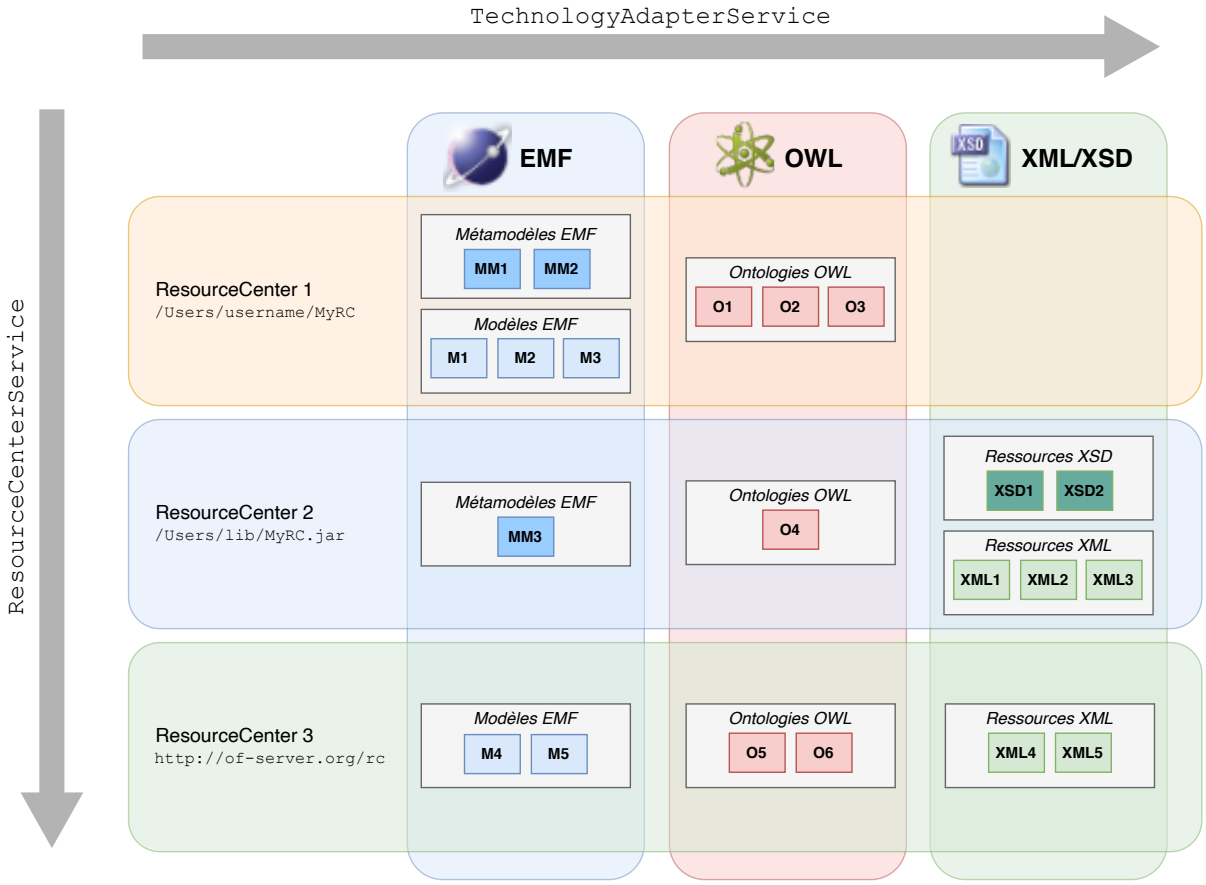

FIGURE 5.3 – Espace d'information pour l'infrastructure logicielle Openflexo

2. Cette technique de développement permet le test immédiat des modifications pour vérification de la non-régression, avec des notifications rapides en cas de code incompatible ou manquant. Les problèmes d'intégration sont détectés et réparés de façon continue, évitant les problèmes de dernière minute et une version est toujours disponible pour un test, une démonstration ou une distribution.
3. https://www.jenkins.io/
4. https://github.com/openflexo-team/openflexo-core

La figure 5.3 illustre la coopération de deux services pour fournir à l'infrastructure l'*espace d'information*, c'est-à-dire l'environnement d'accès aux ressources (et donc aux modèles fédérés). Les colonnes montrent le déploiement des trois adaptateurs technologiques EMF, OWL et XML/XSD, gérés au sein du service `TechnologyAdapterService`. Les lignes figurent, via le service `ResourceCenterService`, l'accès d'une instance de l'infrastructure à trois ResourceCenter (respectivement ici un répertoire sur un système de fichiers, une archive JAR, et un centre de ressources servies sur HTTP). Chaque adaptateur technologique permet d'accéder à certains types de ressources (par exemple des métamodèles EMF et des modèles EMF pour l'adaptateur technologique EMF). Toutes ces différentes ressources, trouvées sur les ResourceCenter et interprétables par un TechnologyAdapter, constituent l'espace d'information.

Le langage FML est implanté au sein de l'architecture au dessus d'un parseur construit avec l'outil `sablecc` [5]. La grammaire du langage (version 2.99) est disponible au sein du dépôt `openflexo-core` [6].

L'interpréteur et plus généralement l'ensemble de l'environnement d'exécution sont implantés sous la forme de composants spécifiques. L'implantation de cet environnement permet de configurer le modèle d'exécution en terme de gestion de la concurrence (distribution de l'exécution des instances de VirtualModel sur des threads). Le présent manuscrit ne traite pas de ces travaux.

Les fonctionnalités offertes par l'implantation du langage FML sont présentées au sein de deux adaptateurs technologiques FML et FML@runtime utilisables dans le langage FML. Il est donc possible de définir des ModelSlot qui permettent d'accéder, à l'exécution, à des programmes FML (y compris au programme en cours d'exécution), ou à des exécutions d'autres programmes FML. Ces mécanismes de réentrance offrent des capacités de réflexivité, et notamment d'introspection. L'intercession [7] a été peu étudiée mais la définition du langage et son interprétation au sein du même environnement la rend possible. Des expérimentations ont également été menées pour construire des comportements de mutation (modification "à chaud" du ou des types d'une instance). Un autre intérêt de cette approche réside dans la capacité d'un programme FML à fédérer différents niveaux conceptuels, et donc à traiter de programmation multi-niveaux (cf section 6.2). En terme de capacités réflexives, d'autres expérimentations nous ont mené à utiliser le langage d'expression FML comme support à l'expression d'autres langages obtenus par fédération.

Un langage de script (FML-script) est proposé au sein de l'infrastructure. Ce langage est un sur-ensemble du langage FML, qu'il complète avec des commandes et des directives permettant de gérer les services du gestionnaire de services d'un environnement (ajout ou suppression de ResourceCenter, manipulation de fichiers, activation de TechnologyAdapter). Ce langage a deux utilisations principales. Son premier intérêt est de pouvoir instancier un terminal (une invite de commandes en ligne) sur une instance de l'environnement et de piloter des exécutions de code FML, ce qui offre des capacités de débogage (*debugging*). Un deuxième intérêt réside dans la capacité à écrire des scripts de tests d'intégration automatisés qui permettent de consolider les développements au sein de la plate-forme d'intégration continue.

5. https://sablecc.org/
6. https://github.com/openflexo-team/openflexo-core/blob/2.99/fml-parser/src/main/sablecc/fml.sablecc
7. capacité d'un programme à modifier son propre état d'exécution

Tous les composants logiciels qui sont présents dans le cœur de fédération disposent également d'outillage et d'interfaces graphiques implantés en SWING (cf "Outillage SWING" sur la figure 5.2).

5.2.2 Composants de bas-niveau

Contrairement à d'autres outillages pour la modélisation qui s'appuient sur des écosystèmes existants (par exemple EMF/Eclipse), l'infrastructure Openflexo repose uniquement sur un environnement Java standard, au dessus duquel ont été développés un certain nombre de composants logiciels de plus bas niveau, réutilisables dans d'autres contextes que ceux de la fédération. Ces différents composants sont figurés sur la figure 5.2 et leur code source est disponible sur Github, sous double licence GNU GPL v3 et EUPL.

PAMELA

PAMELA [8] est un *framework* de programmation orientée-modèles en Java [104, 185]. Cette bibliothèque est utilisée massivement par tous les composants de l'infrastructure. PAMELA s'appuie sur un paradigme de tissage au run-time entre du code Java et l'interprétation d'un modèle encodé dans les métadonnées (les annotations) du code source, conservées au run-time. Ce mécanisme permet de tirer partie des avantages de l'IDM, c'est-à-dire de manipuler un modèle du code à exécuter, mais sans avoir besoin de générer ce code (source d'erreur potentielle), ni de gérer des problématiques d'aller-retour (*round-trip*) entre le code et un modèle sérialisé dans un artefact externe. Cette approche permet de disposer d'une intégration forte, typée (et donc vérifiée par le compilateur), de l'interprétation des modèles au sein d'un code écrit "à la main". Pratiquement, une grande quantité de code est décrit dans des interfaces et des classes abstraites Java, sans implantation de nombreuses méthodes.

L'utilisation de PAMELA apporte des caractéristiques intéressantes exploitées par de nombreux composants (réduction de code source et donc de potentiels bugs, héritage multiple, programmation par traits, programmation par aspects, instrumentation du code, vérification de propriétés, gestion des piles undo/redo, sérialisation, visiteurs, algorithmes de comparaison et de mise à jour de graphes d'objets, etc.)

La programmation orientée-modèle et le framework PAMELA constituent une contribution scientifique à part entière, mais en dehors du champ d'étude du présent manuscrit. Nous ne nous étendrons donc pas plus longuement sur ces aspects.

CONNIE

CONNIE [9] est le composant logiciel qui fournit les abstractions nécessaires à construire des langages d'expression typés. Il permet de naviguer dans des graphes d'objets, selon une logique de navigation donnée, et de construire des chemins. Ces chemins sont ensuite composés avec des opérateurs pour construire des expressions. Ce composant se paramètre avec un langage donné, et permet de construire des expressions dans un *environnement*

8. https://www.openflexo.org/pamela
9. https://www.openflexo.org/connie

de typage (un tel environnement définit un ensemble de variables typées). Ces expressions sont ensuite évaluables dans un *environnement d'exécution* (un environnement qui donne des valeurs aux variables de l'environnement de typage).

CONNIE est utilisé dans de nombreux composants de l'infrastructure, qui utilisent des expressions typées (et notamment GINA et DIANA). Il est également le composant à la base de l'implantation du langage d'expression FML décrit dans la section 4.2.6 et dans l'annexe B.9.

GINA

GINA [10] permet de décrire et d'exécuter des interfaces graphiques de type formulaire, dans la technologie Java/Swing. Les interfaces homme-machine (IHM) prennent la forme de modèles, sérialisés en XML, et sont interprétés à la volée. Cette technologie est utilisée par de nombreux composants pour implanter de l'outillage et des interfaces graphiques. Un aspect intéressant offert par GINA est la capacité à modifier "à chaud" (en cours d'exécution) les interfaces graphiques exécutées. L'infrastructure Openflexo bénéficie, pour les clients lourds Swing, de cette fonctionnalité qui permet le développement rapide d'applications.

DIANA

DIANA [11] permet de construire des éditeurs diagrammatiques. Le modèle de diagramme implicite est constitué de formes graphiques hiérarchiques et de connecteurs reliant ces différentes formes. Le paradigme sous-jacent au fonctionnement de ce composant réplique celui de la fédération de modèles (mis en œuvre à un grain très fin), ou une instance d'un graphe d'objets Java est "connectée" à une forme ou à un connecteur graphique. Les propriétés des formes graphiques sont exprimées par des expressions définies dans le contexte de l'objet Java représenté. Ce composant est à la base de tous les éditeurs diagrammatiques de l'infrastructure Openflexo.

5.2.3 Adaptateurs technologiques

La figure 5.2 présente sur sa partie droite comment les adaptateurs technologiques sont intégrés à l'architecture modulaire. Cette architecture de *plug-in* permet la modularisation (et la réutilisation) des fonctionnalités liées aux technologies à fédérer. Cette modularité est requise pour l'approche "fédération de modèles", considérant d'une part la variété des technologies à intégrer, et d'autre part le besoin de concevoir des ModelSlot et plus généralement des APIs parfaitement adaptées au besoin, avec notamment des critères de performance.

L'implantation d'un adaptateur technologique se structure en deux composants. Un premier composant, qui s'appuie sur le cœur de fédération, fournit une implantation de la technologie visée. Cette implantation comporte un modèle des données à manipuler, ainsi que la couche d'accès à ces données et la gestion des ressources liées. Ce composant définit également l'ensemble des extensions fonctionnelles au langage FML (ModelSlot, FlexoRole,

10. https://www.openflexo.org/gina
11. https://www.openflexo.org/diana

EditionAction, FlexoBehaviour, types spécifiques, etc.). Un deuxième composant, qui s'appuie sur la couche d'outillage Java/SWING, propose des outils, des éditeurs et des composants graphiques, utilisables pour composer des applications.

L'infrastructure Openflexo fournit à ce jour des adaptateurs technologiques pour les technologies suivantes [12] :

- **EMF** : cet adaptateur technologique permet de manipuler l'ensemble de la pile technologique EMF, et notamment les métamodèles *Ecore* accompagnés de leur implantation Java. Certains métamodèles sont fournis par l'adaptateur technologique, qui permet de gérer nativement des modèles BPMN, UML, SysML, etc. Tous les autres modèles EMF sont également interprétables si leur métamodèle correspondant est présent et identifié au sein de l'espace d'information.
- **OWL** : traite des ontologies OWL (triplets RDF)
- **XML/XSD** : permet de manipuler des resources XML, ainsi que des resources XML conformes à une resource XSD
- **JDBC** : offre un support pour des ModelSlot requêtés sur une base de données relationnelle, et exposés comme des instances d'un VirtualModel FML (cf Scénario C, section 4.3.3 page 106 et listing 4.17)
- **Outil de dessin DIANA** : permet de construire des éditeurs graphiques et des représentations diagrammatiques au sein de l'infrastructure
- **Interfaces graphiques GINA** : permet de construire des IHMs (interfaces type formulaires)
- **Microsoft Word** : offre un support pour des documents `.doc` et `.docx`
- **Microsoft Excel** : permet de manipuler des feuilles de calcul Excel (`.xls` et `.xlsx`), avec une API FML de bas niveau (feuilles, lignes, cellules), mais aussi des objets métiers reflétés avec un modèle FML
- **Microsoft Powerpoint** : traite des présentations (ressources `.ppt` et `.pptx`)
- **Services HTTP/REST** : offre un support pour des ModelSlot requêtés et qui reflètent des objets métiers spécifiés en FML
- **OSLC** : support partiel
- **PDF** : support partiel

Ces différentes implantations s'appuient généralement sur des librairies Java existantes. C'est par exemple la bibliothèque Apache POI [13] qui a été choisie pour les adaptateurs Word, Excel et Powerpoint.

L'utilisation de l'infrastructure Openflexo dans un contexte donné peut requérir le développement d'un ou plusieurs adaptateurs technologiques spécifiques. Dans ce contexte, le développement d'un adaptateur technologique consiste principalement à définir les abstractions et à construire les APIs pour le langage FML. Cette tâche correspond à des préoccupations techniques. Elle peut être guidée par exemple par des exigences de performances ou d'autres considérations purement techniques. C'est ensuite au sein de l'espace conceptuel que seront traitées les préoccupations métier.

Il est à noter que pour une même technologie, plusieurs adaptateurs technologiques peuvent être développés, avec des caractéristiques différentes, utilisant des bibliothèques

12. Le périmètre fonctionnel, la maturité et la stabilité de chacun de ces composants sont variables (septembre 2023, infrastructure 2.99.x)
13. https://poi.apache.org/

différentes, et offrant éventuellement des abstractions et des interprétations différentes. A noter également qu'un adaptateur technologique est un logiciel comme un autre, et qu'il peut être modulaire, réutilisé, etc. Un utilisateur de l'infrastructure est donc libre de choisir une implantation existante, ou de fournir sa propre implantation, ou encore de spécialiser et/ou d'adapter une implantation existante.

5.2.4 Editeurs projectionnels FML

Le langage FML tire ses origines de représentations graphiques où les liens entre les différents modèles étaient dessinés. Ce mode de conception et d'interaction avec les données est particulièrement efficace et pertinent dans les premières phases de modélisation métier. Dans un second temps, pendant la phase de raffinement des liens, et pour concevoir les objets métier, les aspects structurels du langage, et notamment la gestion de la contenance seront mieux adaptés à des représentations diagrammatiques (par exemple avec des diagrammes de classe). Enfin, lors de l'opérationalisation du programme FML et particulièrement dans les phases de programmation des comportements, une syntaxe textuelle sera beaucoup plus adaptée. En outre, selon la technologie manipulée, les outils et éditeurs peuvent être très différents, par exemple pour la sélection d'une activité dans un processus BPMN, ou pour la sélection d'un paragraphe dans un document *Word*.

La pragmatique exposée au travers de cette méthodologie de conception de la fédération montre un intérêt à pouvoir disposer de plusieurs syntaxes concrètes pour le langage. L'idée est que le concepteur de la fédération puisse disposer à tout moment de l'éditeur adéquat, dans le paradigme requis et avec la représentation et les outils adaptés.

Les problématiques d'extension, de composition de langages, et de notations multiples ont été étudiées [207]. Dans ce contexte, la notion d'éditeur projectionnel a été proposée, et a été mise en œuvre [14]. Cette technique permet de manipuler directement l'arbre syntaxique abstrait d'un programme, sans s'appuyer sur des analyseurs syntaxiques. Son potentiel réside dans sa capacité à combiner divers styles de notation - tels que le texte, les symboles, les tableaux et les graphiques - et dans la prise en charge d'un large éventail de techniques de composition.

Cette approche a inspiré l'outillage pour l'édition du langage FML dans l'infrastructure Openflexo. La figure 5.4 illustre les mécanismes d'édition de code FML offerts par l'infrastructure. Le langage est sérialisé et persisté sous la forme d'une ressource .fml (un fichier texte). La ressource est parsée pour construire la représentation intermédiaire. Concrètement cette représentation intermédiaire est une instance du métamodèle FML (cf figure 4.6). Les éditeurs sont tous branchés sur la représentation interne, suivant le schéma MVC modèle/vue/contrôleur, ce qui permet d'assurer leur synchronisation. Seul l'éditeur textuel est branché sur la représentation du code source FML.

Le principal défaut de cette approche est le mécanisme de *pretty-print* (la regénération du code FML à partir de la représentation intermédiaire, modifiée via les différents éditeurs). L'enjeu est de pouvoir conserver le formattage, la syntaxe, les commentaires et plus généralement le *layout* général [54]. C'est le rôle du composant P2PP [15] que d'offrir ces fonctionnalités de *pretty-print* à préservation de syntaxe et de *layout*. La technique utilisée

14. notamment dans le *Meta Programming System* (MPS) de JetBrains (https://www.jetbrains.com/fr-fr/mps/)
15. https://github.com/openflexo-team/openflexo-utils/tree/1.6/flexo-p2pp

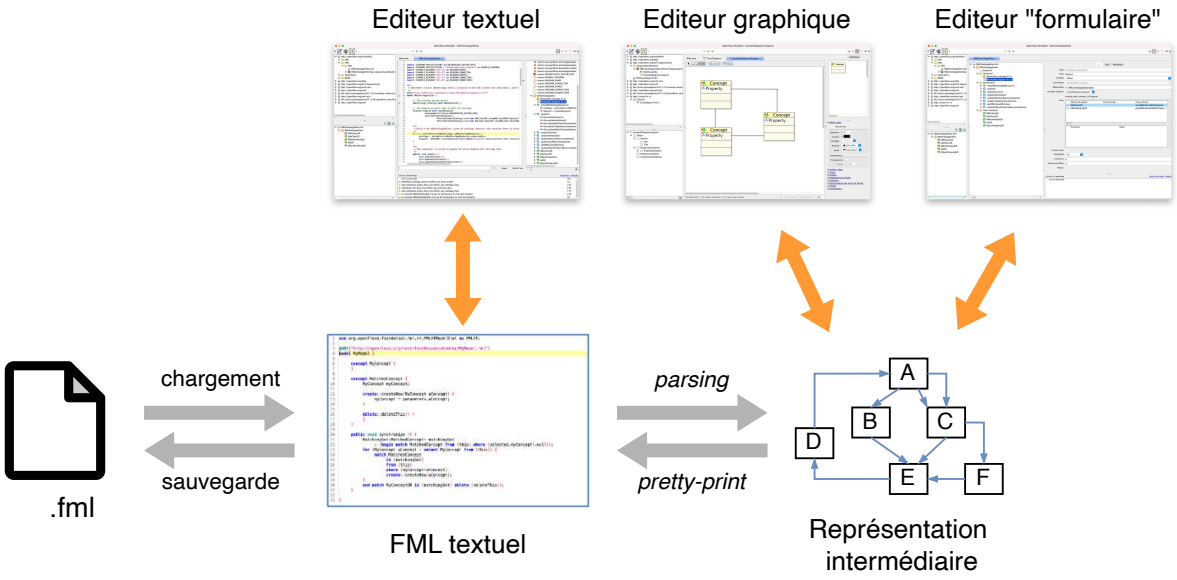

FIGURE 5.4 – Éditeurs projectionnels FML

consiste à maintenir des liens entre la représentation intermédiaire et l'AST (*Abstract Syntax Tree*) construit par le parseur, pour pouvoir recombiner à la génération les nouveaux fragments de textes et les fragments modifiés sur la représentation textuelle précédente.

Certains éditeurs tirent partie des approches d'outillage récursives. C'est le cas par exemple de l'éditeur de code FML en tant que diagramme de classes. Concrètement, il s'agit de la mise en œuvre d'une fédération d'un diagramme et d'une unité de compilation FML, que l'on instancie sur cette même unité de compilation. Cette technique a été généralisée au sein de l'infrastructure Openflexo.

Au final, l'outillage de l'environnement de développement de l'infrastructure Openflexo fournit un ensemble d'éditeurs efficaces et synchronisés, permettant la définition de modèles de fédération (code FML) au moyen de différents paradigmes et notations.

5.2.5 Les modules applicatifs standards

L'architecture présentée figure 5.2 illustre la création d'applications sous la forme de clients lourds avec la technologie Java/SWING. Ces applications font partie de l'infrastructure et correspondent à des cas d'utilisation bien identifiés.

- **OpenflexoModeller** : cette application générique offre un environnement de développement intégré (IDE) pour la conception de modèles de fédération, ainsi qu'un environnement d'exécution. Cette application se configure avec une liste (dynamique) d'adaptateurs technologiques, définis dans le `classpath`.
- **EnterpriseArchitectureEditor** : cette application est destinée à des utilisateurs experts métier. Elle fournit un environnement et des outils de modélisation (BPMN, UML, ontologies OWL, etc.)
- **FreeModellingEditor** : c'est l'application qui met en œuvre les techniques de *free modelling* dont il sera question dans la section 6.4

— **OpenflexoHeadless** : cette application fournit un environnement sans interface graphique, avec un simple terminal FML

— **OpenflexoServer** : cette application fournit un environnement servi par un serveur HTTP (cf section 5.4)

À titre d'illustration, la figure 5.5 présente une capture d'écran de l'application OpenflexoModeller (ici l'outil d'édition de code FML textuel, avec le code FML édité au centre, encadré par différents navigateurs de modèles, et en dessous une vue synthétique présentant la validation).

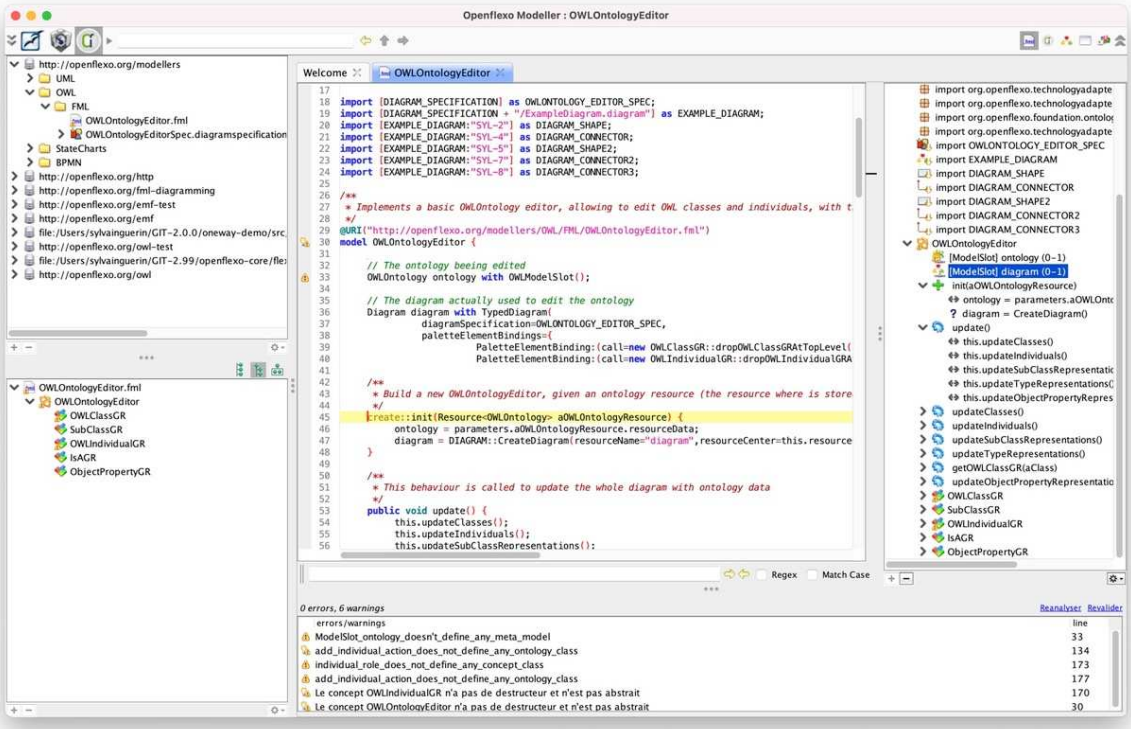

FIGURE 5.5 – Editeur FML

5.3 Openflexo : une usine à logiciel

L'infrastructure Openflexo est construite pour permettre l'assemblage de composants logiciels dans la perspective de construire des applications spécifiques. La figure 5.6 présente les principes d'assemblage d'une telle application.

Au cœur de fédération, présenté plus haut, s'ajoutent des modules applicatifs spécifiques, construits sur mesure ou choisis parmi les modules applicatifs existants dans l'infrastructure. L'application est packagée avec un ou plusieurs ResourceCenter (partie gauche de la figure), contenant les modèles de fédération à exécuter (code FML), ainsi que toutes les ressources nécessaires au fonctionnement du logiciel. Les modèles fédérés peuvent être contenus dans le ou les ResourceCenter packagés avec l'application, ou bien être trouvés dans un ou des ResourceCenter à définir au *run-time*. À droite de la figure sont représentés tous les adaptateurs technologiques nécessaires au fonctionnement de l'application (tous

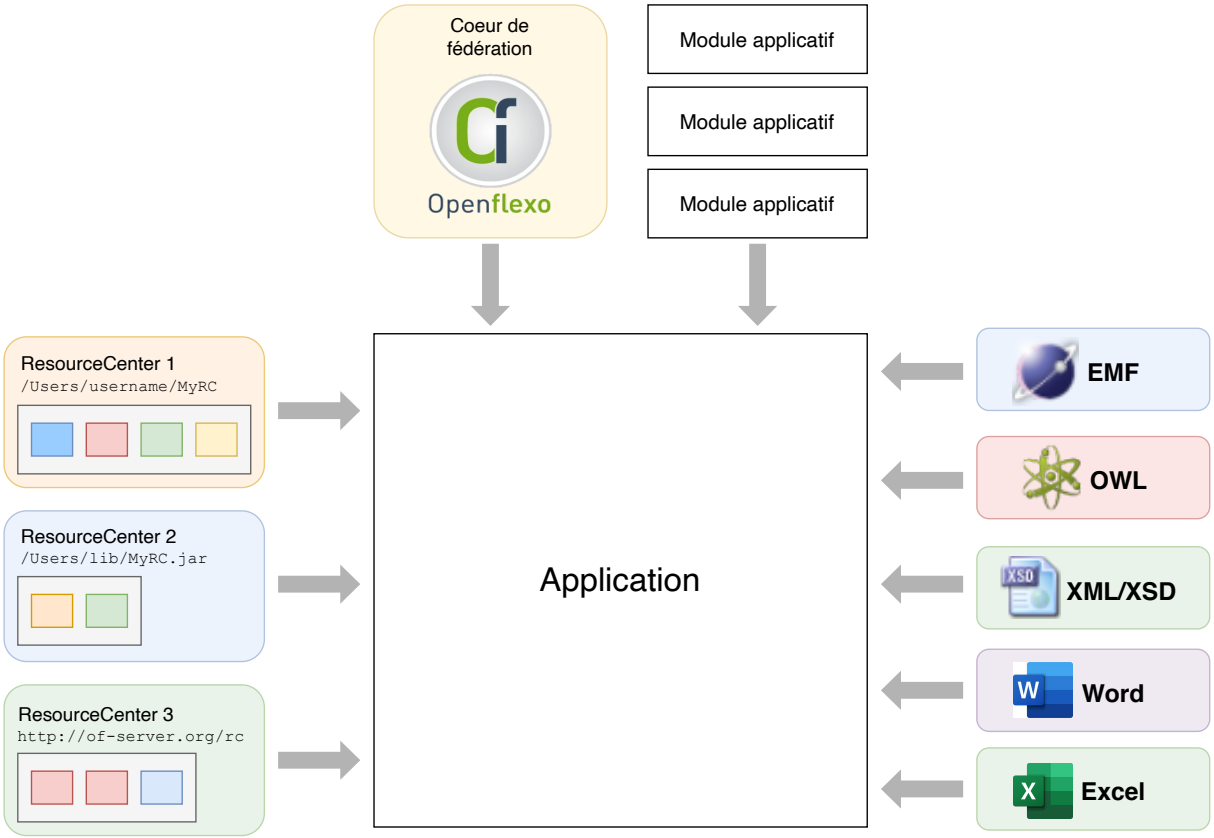

FIGURE 5.6 – Principes d'assemblage d'une application Openflexo

les adaptateurs technologiques requis par les unités de compilation FML à exécuter). D'un point de vue pratique, l'ensemble des composants qui composent une application sont requêtés dans le `classpath` via l'API `ServiceLoader` de Java [16]. L'assemblage s'opère donc naturellement par simple configuration de dépendances.

La partie gauche de la figure représente tous les composants métier de l'application à construire, tandis que les composants techniques sont représentés à droite. Ceci illustre un aspect pragmatique très important de l'implantation de l'approche fédération de modèles au sein de l'infrastructure Openflexo, qui est la séparation stricte entre les préoccupations techniques et les préoccupations métier.

5.4 Applications web

L'infrastructure Openflexo est également projetable sur le web, via le déploiement d'applications web, qui s'exécutent dans un navigateur. La figure 5.2, page 119 illustre en haut à gauche le principe des applications web Openflexo. Le serveur Openflexo est un composant logiciel qui s'exécute au dessus du cœur de fédération, et qui sert des clients web via un serveur web.

16. `java.util.ServiceLoader`

Le serveur s'appuie principalement sur une API HTTP/REST, mais propose également des canaux de communication bi-directionnels ouverts (*WebSocket*) qui permettent une plus grande réactivité et le développement d'applications de modélisation en ligne.

5.5 Discussion et conclusions

Les premiers développements autour de l'infrastructure logicielle ont été produits dans le cadre des activités d'une société commerciale [17], que nous évoquerons en conclusion de ce manuscrit (section 7.2.1, page 170). Les choix de conception pour l'architecture de cette infrastructure ont ainsi été guidés par une approche pragmatique, et validés en environnement opérationnel.

Au delà d'une réponse pertinente aux trois enjeux proposés en tant que problématique (interprétation, hétérogénéité et dynamicité), plusieurs enseignements ont été tirés de cette implantation du langage FML.

1. **Approche interprétée** : ce premier point concerne l'aspect interprétatif de l'approche. Contrairement à d'autres plate-formes qui s'appuient sur des mécanismes de transformation et de génération de code (par exemple dans les environnements construits avec EMF), l'infrastructure Openflexo bénéficie de la souplesse et de l'agilité d'une architecture construite autour d'un interpréteur. Par ailleurs, toutes les interfaces graphiques sont construites et considérées comme des modèles, et sont modifiables pendant l'exécution. Enfin, la plate-forme tire partie des fonctionnalités de réflexivité du langage. Toutes ces caractéristiques se combinent pour fournir un environnement très dynamique qui se prête au prototypage rapide.
2. **Séparation technique/métier** : la pragmatique qui émerge de la mise en œuvre du langage FML au sein de l'infrastructure Openflexo montre une réelle séparation entre les préoccupations techniques (incarnées dans le développement des adaptateurs technologiques, dont s'occupent des développeurs experts Java), et les préoccupations métiers (le développements de VirtualModels FML, tâches prises en charge par des experts métier). Ces deux types de composants logiciels sont très distincts dans l'infrastructure et sont outillés très différemment.
3. **Architecture modularisée** : l'infrastructure Openflexo permet de capitaliser un ensemble de composants logiciels réutilisables. C'est le cas de tous les composants techniques, et notamment des adaptateurs technologiques. La fédération de modèles implique de pouvoir manipuler une grande quantités de technologies. Dans ce contexte, un des intérêts du logiciel libre est de pouvoir s'appuyer sur une communauté de développeurs pour enrichir la base logicielle, et donc de tirer partie d'un écosystème riche et dynamique.
4. **Bibliothèque de modèles métier** : ce dernier point suggère l'intérêt de capitaliser un ensemble de composants métier réutilisables, en tant que VirtualModels FML. Les concepts de programmation orientée objet mis en œuvre dans le langage FML permettent les abstractions et la modularisation nécessaire à la réutilisation de modèles métier. Il est par exemple possible de conceptualiser un modèle métier abstrait, et de l'opérationnaliser dans un deuxième temps, avec un lien d'héritage, vers la technologie souhaitée ; ou encore de concevoir un éditeur graphique indépendamment de

17. la SCIC Openflexo (Société Coopérative d'Intérêt Collectif)

la technologie du modèle sous-jacent. La notion de ResourceCenter permet la mise à disposition de telles bibliothèques de modèles métier réutilisables, ce qui reste un point-clef dans les approches de modélisation.

Chapitre 6

Expérimentations et cas d'utilisation

Les contributions sur la fédération de modèles présentés dans le cadre de ce manuscrit s'appuient sur de nombreux travaux antérieurs, menés sur une période de plus de dix ans, à la fois dans l'équipe de recherche P4S du LabSTICC (ENSTA Bretagne, IMT Atlantique, Université de Bretagne Ouest) et dans un contexte industriel (SCIC Openflexo). C'est la raison pour laquelle certains des projets évoqués ici sont relativement anciens. Le choix de les inclure dans nos travaux résulte à la fois d'une contribution directe, et de leur intérêt au regard des problématiques de cette thèse, parce que tous ces projets ont participé à la consolidation du langage FML et de la plateforme Openflexo.

Sommaire

6.1 Projet Formose

Nous présentons ici un premier cas d'application de l'approche FML/Openflexo. Le contexte est le projet ANR Formose (ANR-14-CE28-0009) [1] qui vise à proposer une démarche d'ingénierie des exigences (IE), dans le contexte de la modélisation et des méthodes formelles, pour les systèmes complexes critiques. Le projet a débuté en novembre 2014, avec différents partenaires industriels et académiques (ClearSy, LACL [2], Institut Mines-Telecom, Thales).

L'ingénierie des exigences implique de nombreuses parties prenantes, ce qui conduit à la production de nombreux types de modèles en fonction de leur point de vue. L'un des objectifs du projet était de démontrer l'intérêt de la fédération de modèles dans une démarche d'intégration continue de l'ingénierie des exigences. Cet objectif nécessite de pouvoir manipuler et relier des modèles de différents paradigmes (documentation textuelle, modèles structurés semi-formels et modèles formels), tout en permettant aux modèles de rester éditables et analysables avec les outils habituels. Cette fédération permet l'analyse d'inconsistances pour aider à détecter les causes ou déterminer les impacts des incohérences. Elle permet également de fournir une méthodologie conduisant à des spécifications formelles qui permettent la vérification, la validation, jusqu'à la certification. Pour cela, Formose a défini un langage formel étendant les langages d'IE existants pour prendre en compte l'architecture abstraite des systèmes complexes et les exigences spécifiques en matière de sécurité, de performance et de modes de reconfiguration.

Dans ce contexte, notre contribution porte sur la définition d'un processus méthodologique et d'un outil développé avec l'infrastructure Openflexo, qui démontre la faisabilité de cette approche collaborative.

6.1.1 Contexte et problématique

Les systèmes critiques sont présents dans de nombreux domaines tels que l'avionique ou les chemins de fer, où la vie humaine est en jeu. La sûreté, la sécurité, les performances et les propriétés fonctionnelles sont donc vitales pour ces systèmes. Par exemple, un système critique typique pour les chemins de fer doit généralement respecter le niveau SIL4 (*Safety Integrity Level 4*, moins de 10^{-9} de défaillances par heure) [115]. La complexité croissante des systèmes génère naturellement une grande variabilité en terme de besoins utilisateurs et de fonctionnalités, et donc en terme d'exigences.

On trouve plusieurs définitions de l'ingénierie des exigences (IE), dont l'une des plus anciennes a été proposée par Ross et Schoman [175]. Ces auteurs ont notamment jeté les bases d'une discipline qui doit aider à comprendre et à expliquer les raisons pour lesquelles le futur système doit être construit et à quoi il peut servir dans l'organisation dans laquelle il est utilisé. L'IE utilise des modèles à la fois pour exprimer la dimension du POURQUOI et pour décrire le QUOI et recommande d'établir le lien entre les éléments du premier modèle et ceux du second.

L'importance qu'un problème soit identifié le plus tôt possible dans un processus de développement est désormais bien connu, et ce pour des raisons de coûts, de délai de livraison

1. http://formose.lacl.fr
2. Laboratoire d'Algorithmique, Complexité et Logique, Université Paris Est Créteil

et de taux de défaillances résiduel. C'est pourquoi l'IE est cruciale dans le développement de systèmes critiques. L'IE doit capturer correctement les objectifs initiaux définis dans un langage naturel et doit produire une spécification claire, complète et non ambiguë des exigences pour les phases de conception et de mise en œuvre. Au cours de la dernière décennie, l'IE a gagné en maturité et est désormais intégrée dans les processus de modélisation.

Cependant, comme indiqué dans [20], de nombreuses questions restent en suspens. La multidisciplinarité et la collaboration sont notamment des fonctionnalités peu intégrées dans les outils et processus d'ingénierie des exigences. Un des défis majeurs de l'IE est de proposer des outils pour créer, utiliser et gérer les liens de traçabilité entre les exigences pertinentes, les différents acteurs et les artefacts logiciels à travers les projets.

Par ailleurs, les autorités de certification envisagent désormais l'utilisation de méthodes formelles dans les nouveaux processus de développement des systèmes critiques [81]. Ces méthodes permettent d'identifier très tôt les problèmes, les risques et les conflits même à un niveau élevé de spécification (non exécutable), établissant ainsi un processus de raffinement qui gère la complexité du système et aide à comprendre ses tenants et aboutissants. En outre, l'obtention de preuves sur les modèles donne une meilleure confiance dans le système, en particulier dans des conditions bien connues, où les tests de scénario peuvent manquer de possibles défaillances.

Toutefois, les pratiques d'ingénierie des exigences dans l'industrie reposent souvent sur des processus et des documents internes. Les outils de gestion des exigences (traitement de texte, outils de traçabilité ou bases de données d'exigences) ne sont pas considérés comme suffisamment efficients et matures par l'industrie elle-même, en particulier lorsque des processus agiles sont mis en œuvre [79]. En outre, comme les activités d'ingénierie des exigences (y compris la vérification et la validation des exigences) sont essentiellement effectuées par des êtres humains et donc sujettes à des erreurs (car nécessitant une vérification croisée coûteuse entre les documents fournis) elles sont considérées comme un point faible en terme de sûreté.

La validation des systèmes complexes repose sur des pratiques et des outils d'ingénierie spécifiques. Par conséquent, les exigences liées à la conception de ces systèmes proviennent d'acteurs multiples et impliquent de nombreuses ingénieries et donc de nombreux artefacts (théorie du contrôle, facteurs humains, physique, etc.). Dans ce contexte, le projet Formose se propose d'explorer l'utilisation d'un formalisme et d'une méthodologie automatisée pour relier ces différents artefacts, au sein d'une approche outillée qui augmente considérablement le niveau de confiance dans le processus d'ingénierie des exigences. Cette méthode doit offrir des points de vue à différents niveaux d'abstraction en utilisant de nombreux langages de modélisation et fournir des moyens d'assurer une cohérence globale en permanence. En outre, l'approche proposée suggère de ne plus considérer l'ingénierie des exigences comme une étape initiale à la conception d'un système, mais itérative, dans le cadre du cycle de vie d'un développement continu, comme pour le logiciel.

Pour atteindre ces objectifs, nous formalisons les quatre objectifs de recherche suivants :

1. **RG1** : proposition d'un cadre pour établir des liens entre des artefacts (modèles) de différents paradigmes (documentation textuelle (.doc), modèles structurés semi-formels (KAOS, ontologies) et modèles formels (Event-B)) ;

2. **RG2** : permettre aux modèles de rester éditables avec leurs outils habituels, et donc de préserver les ingénieries existantes ;

3. **RG3** : utilisation des liens établis entre les différents artefacts pour analyser les impacts et détecter des inconsistances des exigences ;
4. **RG4** : fournir une méthodologie pour conduire à la production de spécifications formelles qui permettent la vérification, la validation et la certification.

6.1.2 Réalisations

C'est le langage SysML/Kaos [141] qui a été utilisé dans le cadre du projet Formose. Ce langage est basé sur SysML [105] et KAOS [204]. Il met l'accent sur un raisonnement formel pour le raffinement des buts, et présente des caractéristiques intéressantes en terme de traçabilité. Le langage de spécification formelle choisi est Event-B [1], une méthode formelle pour la modélisation de systèmes, utilisée depuis plus de 25 ans dans des domaines critiques tels que les chemins de fer ou l'aéronautique[147]. Il s'agit d'une méthode basée sur le raffinement qui prend en charge la preuve par construction et aide à construire itérativement la spécification d'un système complexe tout en garantissant sa correction en prouvant des propriétés. Le support de Event-B s'articule autour d'un outillage mature, et notamment sur des prouveurs tels que Atelier B [49] ou Rodin [174], et sur le *model checker* ProB [149] qui peut être utilisé pour valider les spécifications formelles.

Au sein du consortium formé pour le projet, les travaux de notre équipe se sont principalement centrés sur deux préoccupations :

1. la proposition et la définition d'une approche méthodologique agnostique des processus d'IE,
2. le développement d'un prototype démonstratif de l'approche.

Concernant le premier point, nous avons travaillé sur l'hypothèse de processus métier relatifs à l'ingénierie des exigences propres à une organisation donnée. Plutôt que de concevoir une méthodologie et un processus figé, nous nous sommes intéressés à la conception d'une approche générique dans laquelle il est possible de mettre en œuvre des processus métier spécifiques, qui reposent sur des modèles et des outils dédiés.

Concernant le développement d'un démonstrateur, appelé FORMOD, notre contribution a porté sur l'utilisation de la fédération de modèles pour une mise en œuvre d'un prototype basé sur FML/Openflexo. L'application fournie repose sur l'exécution d'un ensemble de programmes FML au sein d'un environnement d'exécution, dont les interfaces graphiques ont été redéfinies et packagées.

Le démonstrateur est organisé autour de quatre vues principales, illustrant la diversité des différentes préoccupations liées à l'IE :

1. une *perspective documentaire* qui présente un outil intégré de traçabilité documentaire, avec une collection de documents textuels au format Word où les utilisateurs peuvent identifier des fragments de texte comme des exigences (ou des morceaux d'exigences) ou des éléments, associer des figures ou des illustrations à des exigences ou à des éléments, justifier une exigence existante ou faire des commentaires sur des exigences existantes, etc.
2. une *perspective orientée "buts"* où les utilisateurs peuvent structurer et raffiner des exigences, et les allouer à des agents,
3. une perspective *modèle de domaine* qui fournit les connaissances liées au futur contexte opérationnel du système étudié, sous la forme d'ontologies OWL,

4. une perspective *"preuve"* où tous les niveaux de raffinement sont traduits en Event-B, et où les preuves peuvent être traitées.

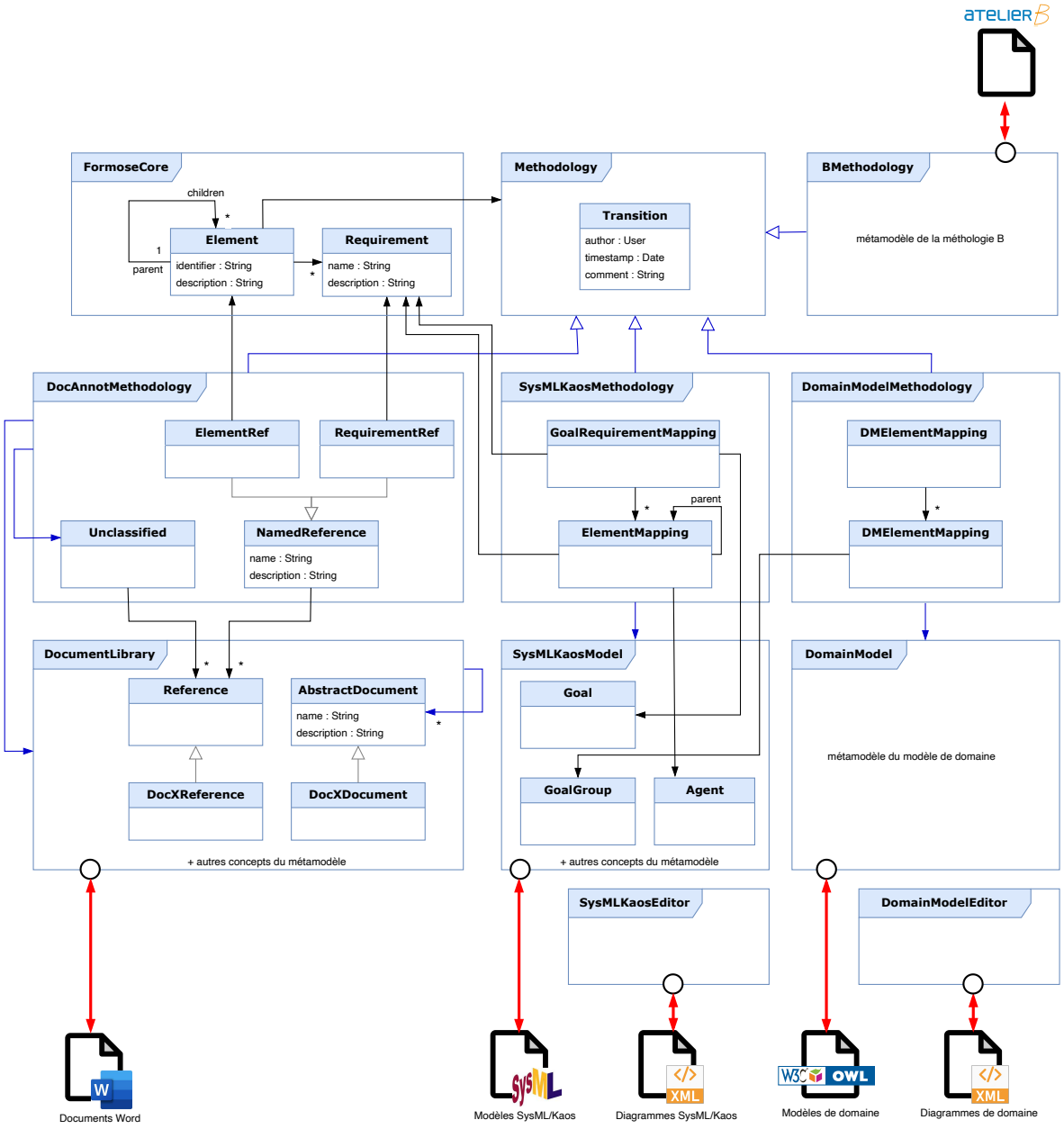

FIGURE 6.1 – Architecture (partielle) du prototype FORMOD

La figure 6.1 présente les principes sous-tendant l'architecture générale du prototype FORMOD. Le cœur de la méthode repose sur la définition d'un métamodèle générique trivial réifié sous la forme d'un VirtualModel `FormoseCore` (en haut à gauche de la figure), qui a pour vocation de supporter n'importe quelle méthodologie de capture et de décomposition d'exigences. Ce métamodèle contient les deux concepts `Element` et `Requirement` qui traduisent une structuration des exigences arborescente autour de la notion agnostique d'*élément*. Un `Element` n'est pas une partie du système à construire, mais un élément de décomposition quelquonque, lié à une méthodologie d'IE.

Cette structuration des exigences est accompagnée d'un ensemble de méthodologies, qui permettent de piloter le cycle de vie de ces exigences. Ces méthodologies s'apparentent à des processus métier spécifiques, implantés pour un cas d'utilisation donné, au sein d'une ingénierie d'exigences propre à une organisation donnée. Cette notion de méthodologie est réifiée dans le VirtualModel abstrait `Methodology`, qui définit notamment le concept de `Transition`, qui correspond à un choix méthodologique opéré par un acteur à un instant donné d'un processus. Une `Transition` est en général liée à une décision menant à une décomposition des exigences. Cette décision est traçable, apparaît dans l'historique, et permet de produire des justifications dans les mécanismes d'analyse pour la détection d'inconsistances d'exigences. Une `Methodology` est liée à un `Element`, ce qui permet de décomposer un projet en de multiples arbres et d'appliquer sur des sous-arbres des méthodologies diverses.

Afin d'illustrer l'approche, 6 méthodologies ont été développées dans le cadre du projet Formose, et 4 méthodologies sont présentées sur la figure 6.1.

— La méthodologie `DocAnnotMethodology` étend `Methodology` et couvre les aspects élicitations et justification des éléments et des exigences présentés plus haut. Ce composant logiciel s'appuie sur le VirtualModel `DocumentLibrary`, qui représente une bibliothèque de documents Word, et qui gère des références à des fragments de texte. Les concepts `ElementRef` et `RequirementRef` réifient les liens entre les éléments/exigences et les références textuelles.

— La méthodologie `SysMLKaosMethodology` implante le langage et la méthode SysML/-Kaos décrite dans [141]. Le raffinement des buts pour un agent est explicitement encodé comme une `Transition` dont le comportement est décrit en FML dans la méthodologie. Un aperçu de l'outillage lié à cette méthologie est illustré figure 6.2.

— La méthodologie `DomainModelMethodology` correspond à la modélisation du modèle de domaine associé à un niveau de raffinement du modèle de buts. Cette modélisation s'appuie sur une description ontologique décrivant les entités du système et de son environnement. C'est ce modèle qui capture les connaissances liées au futur contexte opérationnel du système étudié.

— La méthodologie `BMethodology` combine les deux méthodologies précédente pour traduire en Event-B les différents niveaux de raffinement. L'approche fédération de modèles permet la génération iterative des systèmes Event-B sur la base des diagrammes de buts et modèles de domaine liés à un niveau de raffinement, mais permet aussi à des experts du langage B de continuer à travailler sur les artefacts Event-B, dans l'outil Atelier B, synchronisé avec FORMOD. Un aperçu de l'outillage lié à cette méthologie est illustré figure 6.3 : l'écran représente un niveau de raffinement : un niveau du diagramme de buts figurant en haut à gauche, un modèle de domaine en haut à droite, et deux systèmes Event-B en bas qui représentent le système et son environnement. La conception de l'outil FORMOD a nécessité le développement d'un adaptateur technologique spécifique pour Event-B et l'Atelier B.

La description précise de ces différentes méthodologies est consultable sur [75], ainsi que dans les publications qui ont été produites dans le cadre de ce projet, et dont une liste figure ci-après.

- Fahad Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guérin, Christophe Guychard, *Addressing modularity for heterogeneous multi-model systems using model fe-*

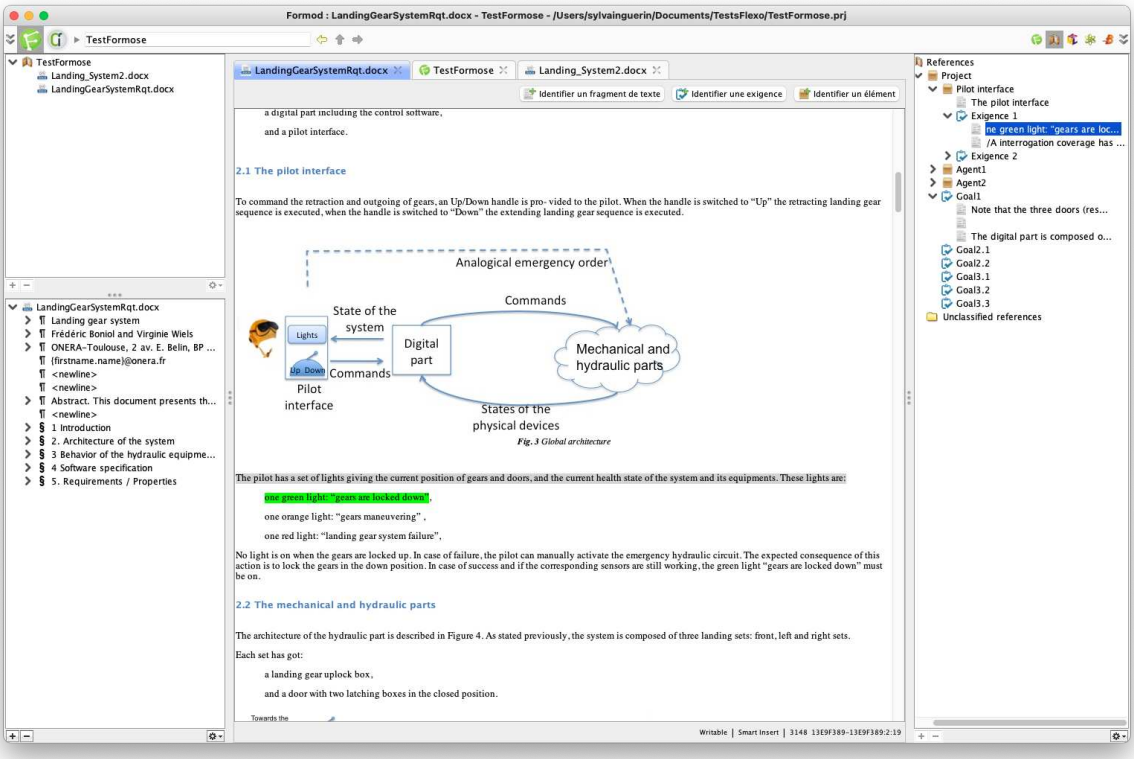

FIGURE 6.2 – Capture d'écran de FORMOD présentant la perspective documentaire (élicitation et justification des exigences)

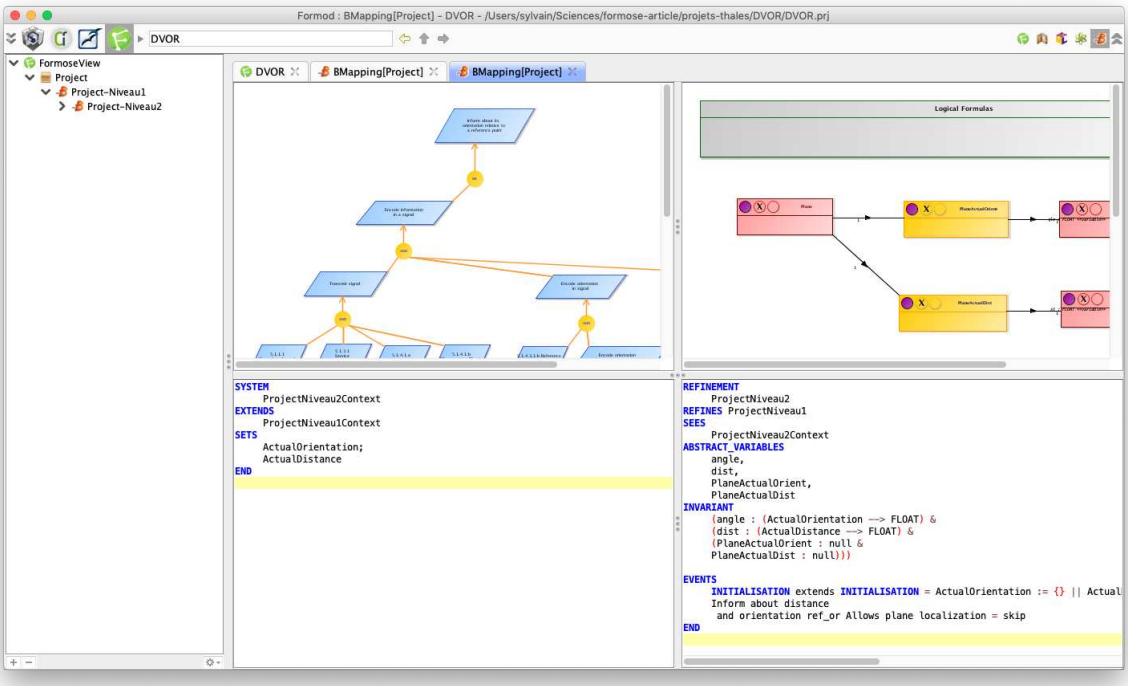

FIGURE 6.3 – Capture d'écran de FORMOD présentant la perspective *"preuve"*

deration, in Companion Proceedings of the 15th International Conference on Modularity, pages 206-211, 2016 [84].

- Fahad Rafique Golra, Fabien Dagnat, Jeanine Souquières, Imen Sayar, Sylvain Guérin, *Bridging the Gap Between Informal Requirements and Formal Specifications Using Model Federation*, International Conference on Software Engineering and Formal Methods, janvier 2018 [88].
- Fahad Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guérin, Christophe Guychard, *Continuous Requirements Engineering Using Model Federation*, In : 24th International Requirements Engineering Conference (RE). (Sept 2016) 347–352 [83].
- Steve Tueno, Régine Laleau, Amel Mammar, and Marc Frappier. Towards Using Ontologies for Domain Modeling within the SysML/KAOS Approach. IEEE pro- ceedings of MoDRE workshop, 25th IEEE International Requirements Engineering Conference, pages 1–5, 2017 [202].
- Steve Tueno, Marc Frappier, Régine Laleau, Amel Mammar, and Hector Ruiz Barradas. The Generic SysML/KAOS Domain Metamodel. ArXiv e-prints, 1811.04732, November 2018 [201].
- Steve Jeffrey Tueno Fotso, Amel Mammar, Régine Laleau, and Marc Frappier. Event-B expression and verification of translation rules between SysML/KAOS domain models and B System specifications. In ABZ, volume 10817 of LNCS, pages 55–70. Springer, 2018 [77].
- Steve Jeffrey Tueno Fotso, Marc Frappier, Régine Laleau, and Amel Mammar. Back propagating B system updates on SysML/KAOS domain models. In Proceedings of the 23rd International Conference on Engineering of Complex Computer Systems, ICECCS 2018, Melbourne, Australia, December 12-14, pages 160–169. IEEE Computer Society, 2018 [76].

L'outil FORMOD est distribué en open source. Son code source est disponible en ligne [3] et repose principalement sur l'exécution d'un ensemble de VirtualModels FML. En outre, une version de FORMOD et l'étude de cas peuvent être trouvées dans la section de téléchargement de la page web du projet Formose [4].

6.1.3 Conclusions et enseignements

Les fonctionnalités de FORMOD ont été validées sur plusieurs études de cas. Toutes ces expérimentations ont montré la pertinence de l'approche pour faire collaborer différents experts métier de l'IE. Les diagrammes de buts (SysML/KAOS) permettent de structurer un niveau intermédiaire entre les documentations textuelles et les spécifications formelles, qui constituent sinon un écart conceptuel très significatif. Toutefois, les expériences restent relativement limitées. Des cas d'utilisation plus importants seraient nécessaires pour conclure à une réelle utilisabilité de l'approche.

Nous avons atteint nos quatre objectifs de recherche avec le démonstrateur FORMOD. Les cas d'utilisation ont montré comment les modèles ont été fédérés (RG1) en gardant les modèles éditables avec leur outil d'origine (RG2). Les différents cas d'utilisation étudiés nous ont permis de révéler des cas d'inconsistances que la définition de liens conceptuels

3. https://github.com/openflexo-team/formod
4. http://formose.lacl.fr/Documentation.html

ont aidé à résoudre (RG3&4). La méthode que nous proposons et l'outil qui lui est associé sont destinés à être utilisés pour des systèmes critiques, mais ils pourraient être utilisés dans d'autres contextes. La principale limitation de son application aux systèmes non-critiques est le coût de l'utilisation des méthodes formelles. En effet, l'utilisation de Event-B nécessite des ingénieurs à très haut niveau d'expertise et du temps pour construire et prouver les spécifications.

Le projet Formose est illustratif des trois enjeux présentés section 3.1, page 53. Le langage FML s'est révélé parfaitement adapté à la modélisation de la réification des liens entre les différents artefacts manipulés et à la définition du modèle métier sous-jacent (enjeu E1). L'hétérogénéité (enjeu E2) était naturellement présente dans le cadre de la fédération de documents textuels, d'ontologies, de modèles semi-formels et formels. L'enjeu lié à la dynamicité des modèles (enjeu E3) était également présent, par le fait que ces différents artefacts devaient être éditables avec les outils habituels (RG2).

Un premier enseignement, et un aspect particulièrement intéressant de cette expérimentation, a été de construire un outil générique et paramétrique du point de vue des processus et cycles de vie des concepts manipulés (les méthodologies). Le langage FML s'est révélé parfaitement adapté pour implanter des aspects méthodologiques au sein d'un processus (par exemple la décomposition SysML/KAOS avec les agents), en tant que comportements FML. Ceci revient à pouvoir construire des modèles qui embarquent leur propres comportements, décrivant leur cycle de vie et comment ces modèles doivent être manipulés, *a contrario* des approches classiques où les comportements sont implantés "en dur" dans les outils associés.

Un deuxième enseignement du projet Formose concerne des aspects de conception liés au langage FML. Le développement du prototype FORMOD a nécessité le développement de 26 VirtualModels, et un grand nombre de FlexoConcepts. La conception et la maintenance d'une telle architecture logicielle a requis la mise en œuvre de la modularisation et de principes de la conception orientée objet, pour lesquels le langage FML a montré son efficacité. En effet, le prototype FORMOD a utilisé des composants logiciels (des VirtualModel) externes, et certains VirtualModel développés dans le cadre du projet Formose ont pu faire l'objet de réutilisation dans d'autres projets.

6.2 MULTI Process Challenge

6.2.1 Introduction

L'ingénierie dirigée par les modèles (IDM) a traditionnellement adopté une approche hiérarchique stricte à deux niveaux conceptuels, où les métamodèles résident dans un certain méta-niveau, et les modèles sont créés un méta-niveau plus bas et utilisent les types du métamodèle. EMF [190] ou MOF [95, 166] illustrent classiquement cette approche.

Cette approche stricte montre ses limites lorsqu'il s'agit de modéliser des domaines complexes nécessitant plus d'un niveau conceptuel. En effet, l'approche à deux niveaux ne reconnaît pas et ne prend pas en charge 1) l'existence de différentes formes de classification et/ou d'instanciation (par exemple, instanciation ontologique ou instanciation linguistique); et 2) la dualité type-objet pour les éléments du modèle. Dans un contexte de

conceptualisation multi-niveaux, l'approche de métamodélisation stricte devient complexe et peu pratique à utiliser.

Pour résoudre ce problème, le paradigme de la modélisation multi-niveaux a été introduit. Contrairement à l'approche stricte, la modélisation multi-niveaux préconise l'utilisation d'un nombre flexible de niveaux et de relations entre ces niveaux. Ce paradigme est de plus en plus étudié et utilisé, comme en témoigne la contribution de nombreux nouveaux outils et approches de modélisation multi-niveaux tels que les *powertypes* [89], Melanee [4], LIMM [87], MetaDepth [55], MultEcore [153], DeepTelos [123] ou DMLA [203]. C'est dans ce contexte, et profitant du dynamisme de la communauté, qu'est né le challenge de modélisation multi-niveaux au sein du *workshop* MULTI de la conférence MODELS.

Le cas d'utilisation présenté ici est une réponse au challenge 2021 [2] qui consiste à fournir une solution à un problème de spécification et de mise en œuvre de processus métier.

6.2.2 Analyse

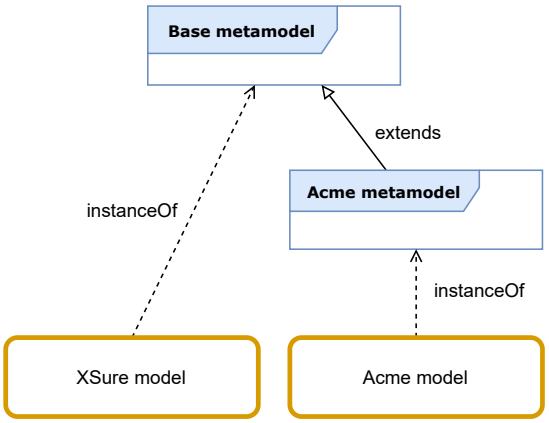

FIGURE 6.4 – Architecture de notre solution

Le challenge MULTI propose la modélisation de la définition d'un processus métier via la définition d'un certain nombre d'assertions P_1–P_{19} qui définissent des spécifications [2]. Ce processus doit également pouvoir être instancié, ce qui implique un niveau de conceptualisation supplémentaire, comme illustré par l'exemple de l'instanciation d'un processus pour une compagnie d'assurance appelée XSure. En outre, le challenge propose une spécialisation de cette modélisation au travers de la modélisation d'un processus d'ingénierie logicielle pour une entreprise fictive appelée *Acme*, qui implique de nouvelles contraintes portant sur des entités des différents niveaux conceptuels (S_1–S_{13}).

Notre analyse a bien sûr été guidée par l'approche fédération de modèles, pour 1) modéliser les relations entre plusieurs modèles, indépendamment de leurs niveaux d'abstraction et de l'architecture de modélisation, 2) prendre en compte la réutilisabilité des relations en identifiant leur sémantique et en les réifiant comme des concepts, et enfin 3) organiser et structurer ces concepts pour améliorer la réutilisation et l'extensibilité de la solution.

La figure 6.4 présente l'architecture générale de la solution proposée[5]. Nous avons défini deux VirtualModels `Base metamodel` et `Acme metamodel`, ainsi que deux VirtualModelInstances `XSure model` et `Acme model` qui instancient respectivement ces deux VirtualModels. Les liens `instanceOf` traduisent le lien de conformance χ entre un modèle et son métamodèle tandis que les VirtualModels `Base metamodel` et `Acme metamodel` sont liés par une relation d'héritage.

Ceci nous conduit à identifier les deux axes de modélisation que sont l'instanciation ontologique et l'instanciation linguistique, comme illustré sur la figure 6.5 et avec la même syntaxe graphique.

— L'axe horizontal caractérise l'instanciation ontologique, c'est-à-dire que l'instanciation est portée par une relation explicitement réifiée dans le métamodèle correspondant (par exemple le concept `ProcessType` qui est référencé depuis le concept `Process`).

— L'axe vertical caractérise quant à lui l'instanciation linguistique : le métamodèle (un VirtualModel) peut être instancié en tant que VirtualModelInstance grâce au mécanisme d'instanciation de FML.

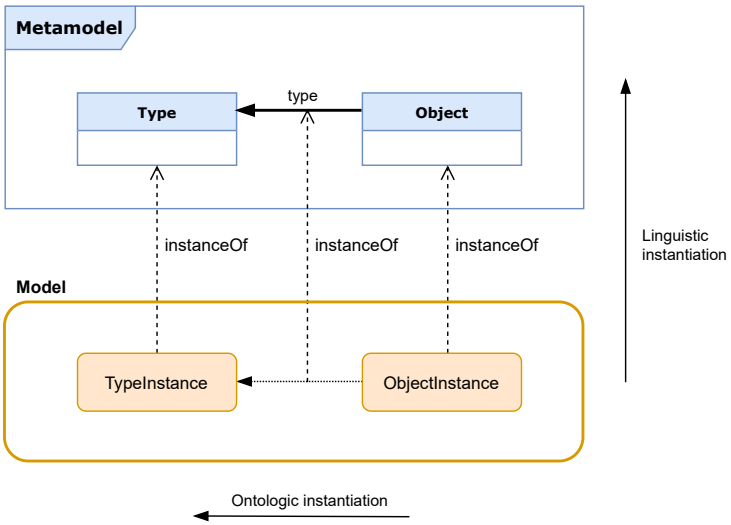

FIGURE 6.5 – Instanciation ontologique et instanciation linguistique

6.2.3 Solution

La figure 6.4 capture les deux cas d'utilisation *XSure* et *Acme* présentés en illustration du challenge.

Modélisation du processus de base

Le métamodèle de base (le VirtualModel `Metamodel`) est présenté sur la figure 6.6, avec un formalisme proche de UML.

5. Nous conserverons dorénavant la syntaxe graphique proposée ici, avec les éléments VirtualModel/FlexoConcept en bleu, et les instances linguistiques (VirtualModelInstance/FlexoConceptInstance) représentées en brun.

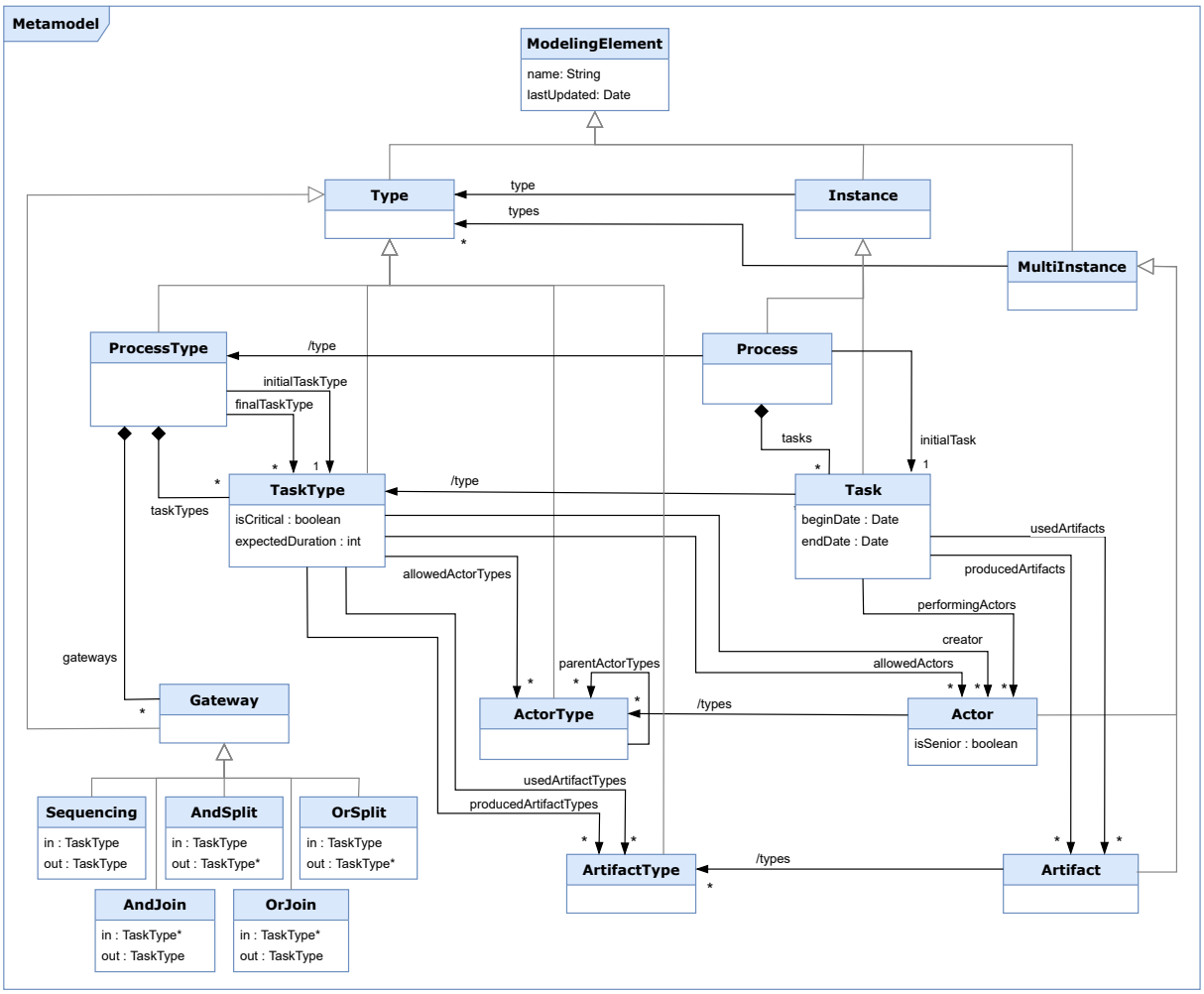

FIGURE 6.6 – Le métamodèle de base pour la modélisation de processus métier

Notre proposition repose sur le mécanisme d'instanciation ontologique présenté figure 4.4, avec un concept générique `ModelingElement`. Deux sortes d'instanciation ontologiques sont requises pour répondre aux spécifications du challenge. L'instanciation simple suppose qu'une instance ne soit conforme qu'à un et un seul type (`Process` et `Task` sont respectivement conformes à `ProcessType` et `TaskType`). L'instanciation multiple permet à des entités de définir leur conformance avec plusieurs types (`Actor` et `Artifact` sont respectivement conformes à plusieurs `ActorTypes` et `ArtifactTypes`). Ces instanciations sont exprimées à l'aide des relations `type` and `types` entre les FlexoConcept `Type`, `Instance` et `MultiInstance`. Sur le diagramme, ces spécialisations sont indiquées avec des relations dérivées `/type` ou `/types`.

Le métamodèle de définition de processus est représenté à gauche de la figure, et est également partiellement représenté sur le listing 6.7. `ProcessType` hérite de `Type` et référence une collection de `TaskTypes` (concept contenu dans `ProcessType`) ainsi qu'une collection de `Gateway`, spécialisés par types. `ProcessType` référence également une unique `TaskType` en tant que tâche initiale, et une collection de `TaskType` en tant que tâches finales. Les concepts d'`ActorType` et `ArtifactType` complètent ce métamodèle.

```
model MetaModel {

 concept ModelingElement { ... }

 concept Type extends ModelingElement { ... }

 concept ProcessType extends Type {
   TaskType[0..*] taskTypes;
   TaskType initialTaskType;
   TaskType[0..*] finalTaskTypes;
   Gateway[0..*] gateways;

   concept TaskType extends Type {
     Actor creator;
     Actor[0..*] allowedActors;
     ActorType[0..*] allowedActorTypes;
     ...
   }

   abstract concept Gateway extends Type {
     abstract void execute(Process process);
     ...
   }

   concept Sequencing extends Gateway {
     TaskType in;
     TaskType out;
   }
   ...
 }
}
```

FIGURE 6.7 – Extrait FML du métamodèle de base

Le métamodèle d'instance de processus figure en regard à droite, et présente les concepts `Process`, `Task`, `Actor`, et `Artifact`, instances ontologiques des concepts respectifs `ProcessType`, `TaskType`, `ActorType`, et `ArtifactType`. Le fait que ces deux "morceaux" de métamodèles soient définis au même niveau conceptuel linguistique permet de pouvoir

faire des références de l'un à l'autre, tel que par exemple le concept `TaskType` qui référence une collection d'`Actor` à travers la relation `allowedActors`.

Des explications détaillées de la façon dont sont implantées toutes les spécifications du challenge figurent dans notre article [96].

Modélisation du processus spécialisé (*Acme*)

La figure 6.8 présente notre solution à la spécialisation de la définition d'un processus de base pour métamodéliser le processus d'ingénierie logicielle *Acme*. Les contraintes supplémentaires - multi-niveaux -, associées à la spécialisation de cette définition et instance de processus, sont encodées dans un VirtualModel intermédiaire, qui étend (héritage) le métamodèle de base, et tire partie des mêmes mécanismes d'instanciation linguistiques et ontologiques. Nous utilisons en plus une fonctionnalité du langage FML non encore présentée dans ce manuscrit, qui consiste à définir des invariants sur les concepts. Nous nous référerons également à [96] pour plus de détails sur les différents choix de conception permettant de satisfaire les exigences du challenge.

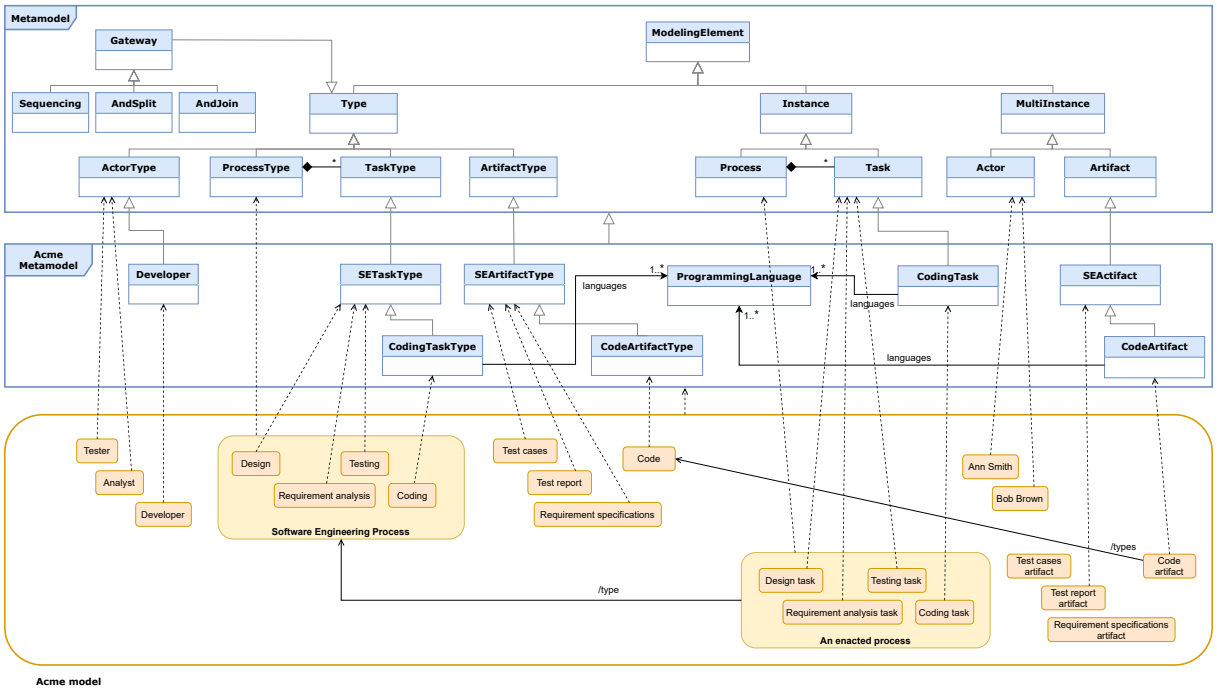

FIGURE 6.8 – Architecture de la spécialisation pour le processus *Acme*

Outillage de la solution

La solution a été implantée en FML avec Openflexo. Les deux cas d'utilisation *XSure* et *Acme* ont été modélisés avec l'environnement de conception interactif et sont exécutables dans l'environnement d'exécution FML/Openflexo.

Nous avons tiré parti de la fédération de modèles et des fonctionnalités d'édition de diagrammes apportées par l'adaptateur technologique DIANA pour mettre en œuvre deux

outils graphiques interactifs construits sur les niveaux conceptuels détaillés dans les sections précédentes. La figure 6.9 présente l'architecture de l'outillage. Le premier outil permet l'édition (graphique) d'un processus, et le deuxième outil permet d'instancier ce processus, de faire les assignations nécessaires (acteurs) et de l'exécuter. L'outil offre une représentation graphique au travers d'un éditeur interactif permettant de faire "avancer" le processus.

Le métamodèle de base est complété par des comportements mettant en œuvre une sémantique d'exécution pour les processus exécutés. Toutes les tâches gèrent un état et sont instanciées à partir de `TaskType` pour un processus donné. Cet état est soit `Not startable` (lorsqu'il n'est pas attribué à un acteur ou lorsque les artefacts d'entrée requis ne sont pas disponibles), soit `Startable`, soit `Started`, soit `Completable` (lorsque tous les artefacts de sortie sont prêts), soit `Completed`. Une `Task` gère également un ensemble d'acteurs, une date de début et de fin, certains artefacts utilisés et produits. La logique métier de `Gateway` a également été mise en œuvre par le biais d'un comportement abstrait `execute(Process)`.

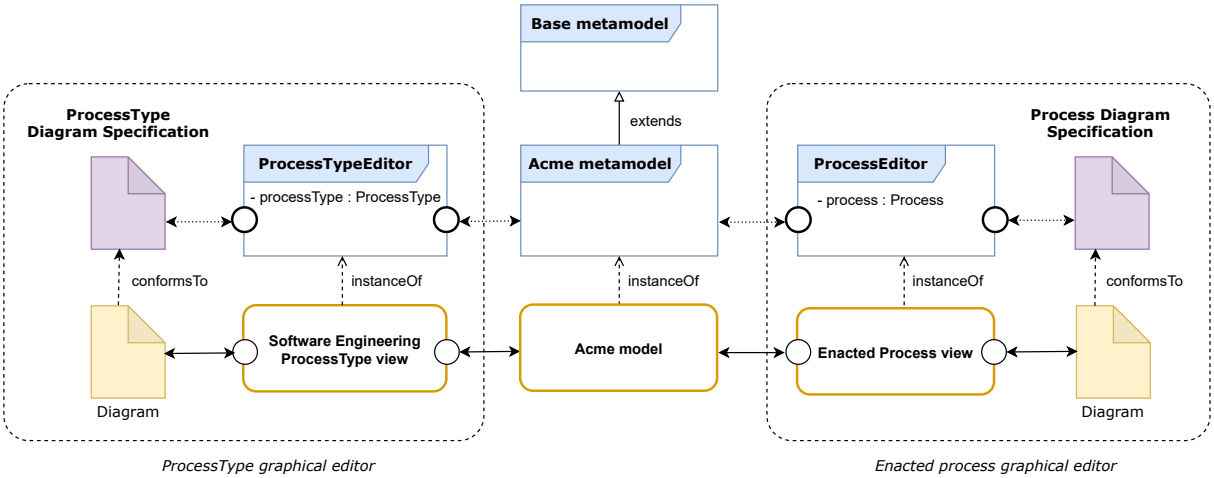

FIGURE 6.9 – ToolingArchitecture

Les figures 6.10 et 6.11 montrent des captures d'écran, respectivement de l'outil d'édition du code FML et de l'outil d'exécution de processus métier du challenge MULTI. Une vidéo de démonstration de l'outil développé est disponible sur le site web Openflexo [6], accompagnée des instructions de téléchargement et d'installation.

6.2.4 Evaluation

La solution proposée satisfait toutes les exigences exprimées dans le challenge. Les fonctionnalités et l'expressivité du langage FML permettent en effet la mise en œuvre de différentes techniques, que l'on peut classer en trois catégories :

— *Conformance syntaxique* : les exigences sont satisfaites si le modèle est bien conforme à son métamodèle

— *Vérification de contraintes* : les exigences sont satisfaites si à la fois la conformance syntaxique et l'évaluation des invariants des concepts sont vérifiées

— *Outillage* : les exigences sont mises en œuvre dans l'implantation des outils, et en complément des comportements décrits dans les modèles.

6. https://research.openflexo.org/MLMChallenge.html

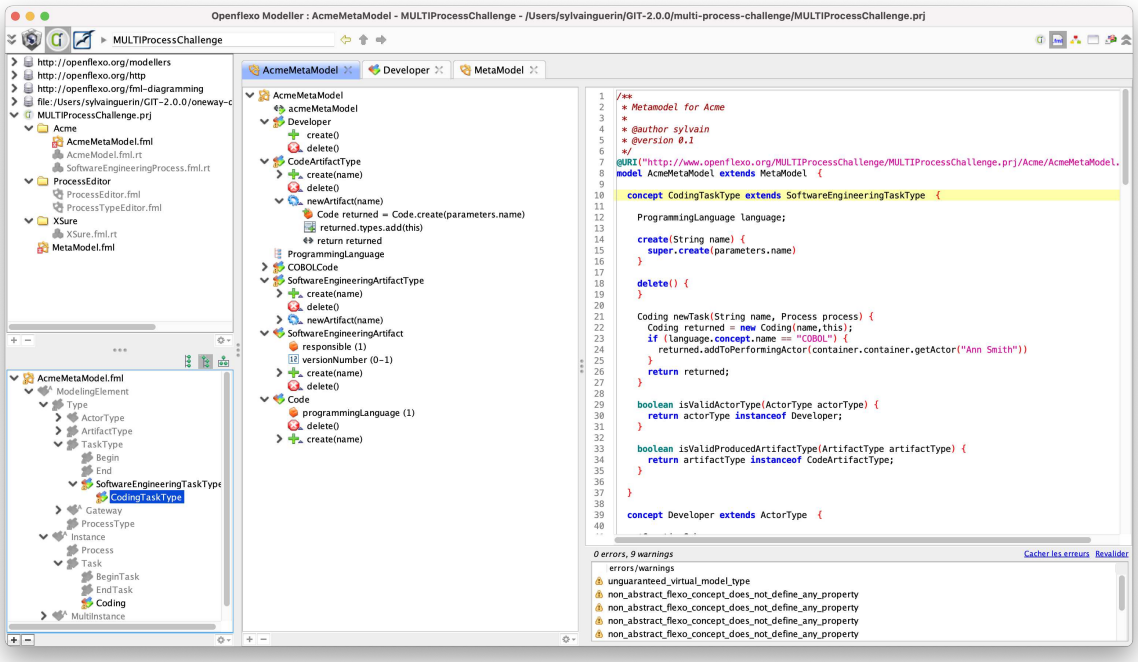

FIGURE 6.10 – Capture d'écran de l'outil d'édition du code FML du challenge MULTI

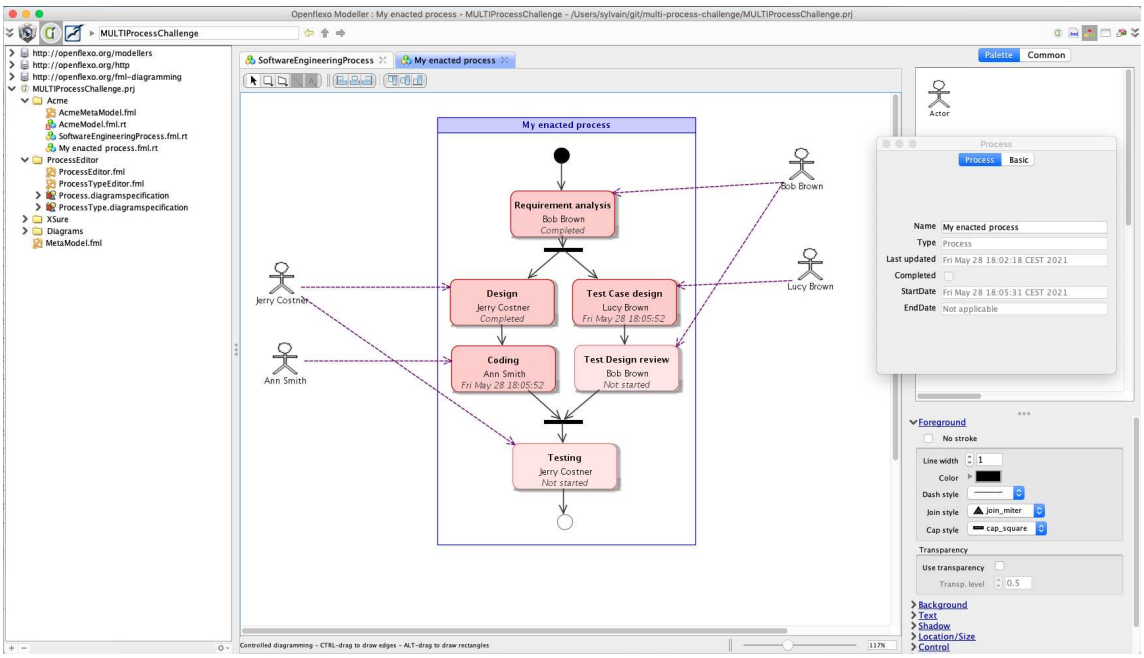

FIGURE 6.11 – Capture d'écran de l'outil d'exécution de processus métier du challenge MULTI

La table 6.1 montre la couverture des exigences pour le métamodèle de base illustré par le cas d'utilisation *XSure* tandis que la table 6.2 montre la satisfaction des exigences pour le processus d'ingénierie logicielle *Acme*. La *conformité syntaxique* est plus précisément classée en six sous-catégories :

— *Conceptualisation* (*Conc.*) : un concept est créé par satisfaire une exigence (par exemple pour P_1 : *process type* est conceptualisé comme FlexoConcept `ProcessType`)
— *Spécialisation* (*Spec.*) : un concept est spécialisé (au sens de la modélisation objet) (par exemple pour P_2 : `Gateway` définit un comportement `execute()` et `Sequencing`, `AndSplit`, `AndJoin`, `OrSplit` et `OrJoin` sont définis comme des concepts hérités implémentant une logique métier spécifique)
— *Composition* (*Comp.*) : plusieurs concepts sont mis en relation ou bien des attributs spécifiques sont ajoutés à un/des concepts (par exemple pour P_3 : un *process type* a exactement une *initial task type*).
— *Instantiation linguistique* (*Inst.L.*) : une exigence est satisfaite par l'instanciation d'un ou plusieurs concepts (par exemple pour S_1 : 'Requirement analysis' est définie comme instance de `SETaskType`)
— *Instantiation ontologique* (*Inst.O.*) : le respect d'une exigence est obtenu par l'établissement d'une relation de typage ontologique (par exemple la notion de *developer* de S_3 est implémentée par à la fois un concept de *Acme metamodel* et une instance de *Acme model* (cf en bas à gauche de la figure 6.8)
— *Modélisation comportementale* (*M.Comp.*) : une exigence est satisfaite par la définition d'un ou plusieurs comportements, et peut être combinée avec des définitions de contraintes (invariants) et des outils (par exemple P_{17}).

6.2.5 Conclusions et enseignements

Le langage FML couplé à l'infrastructure Openflexo nous ont permis de répondre à toutes les exigences posées dans le cadre du challenge MULTI. Nous avons également proposé un outil qui démontre l'usabilité de la solution. Nous avons développé une solution ad hoc, incluant des modèles et des métamodèles, grâce à la flexibilité offerte par l'infrastructure de métamodélisation proposée par Openflexo.

Nous avons cherché à explorer comment dépasser les limites conceptuelles posées par le paradigme du *strict metamodeling* avec le langage FML. Une première réponse réside dans le couplage de l'instanciation ontologique et de l'instanciation linguistique. Un deuxième aspect concerne le fait que le langage FML est *level agnostic*, c'est-à-dire que la fédération de modèles permet de fédérer dans un niveau conceptuel donné des "modèles" de différents niveaux. Les caractéristiques du langage FML permettent notamment ce couplage, et le rendent efficient.

Au final, même s'il n'a pas été conçu dans une perspective de répondre à des problématiques de métamodélisation multi-niveau, le langage FML se révèle disposer de la flexibilité et de l'expressivité nécessaire. Des travaux futurs pourraient être conduits pour implanter en FML un support multi-niveaux explicite, par exemple un FlexoConcept `Clabject` [5] avec deux FlexoRole qui référencent deux niveaux (ou plus) conceptuels différents.

Au regard des enjeux posés dans la section 3.1, page 53, ce cas d'utilisation a permis de valider la pertinence du langage FML pour la métamodélisation d'une problématique multi-

	Conformance syntaxique						Vérif. contraintes	Outils
	Conc.	Spec.	Comp.	Inst.L.	Inst.O.	M.Comp.		
P_1	✓		✓					
P_2	✓	✓	✓					
P_3			✓					
P_4	✓				✓			
P_5	✓		✓					
P_6	✓		✓					
P_7	✓		✓					
P_8			✓					
P_9			✓				✓	
P_{10}	✓				✓			
P_{11}	✓		✓		✓			
P_{12}			✓					
P_{13}	✓		✓		✓			
P_{14}	✓				✓			
P_{15}	✓				✓			
P_{16}	✓				✓			
P_{17}	✓					✓	✓	✓
P_{18}	✓					✓		
P_{19}		✓	✓			✓		

TABLE 6.1 – Satisfaction des exigences pour le métamodèle de base

	Conformance syntaxique						Vérif. contraintes	Outils
	Conc.	Spec.	Comp.	Inst.L.	Inst.O.	M.Comp.		
S_1				✓				
S_2				✓				
S_3	✓	✓	✓	✓				✓
S_4	✓	✓	✓	✓				
S_5		✓		✓		✓		
S_6		✓		✓		✓	✓	
S_7				✓		✓		
S_8				✓				
S_9						✓	✓	
S_{10}	✓	✓	✓					
S_{11}				✓				✓
S_{12}				✓				✓
S_{13}				✓				

TABLE 6.2 – Satisfaction des exigences pour le métamodèle *Acme*

niveaux (enjeu E1). Les aspects hétérogénéités (enjeu E2) ont été moins évalués puisque l'on se plaçait dans un environnement homogène du point de vue de la technologie. Cependant, nous avons tiré parti de l'adaptateur technologique DIANA pour construire l'outillage graphique de ce cas d'utilisation. Enfin, le langage FML s'est révélé très adapté à décrire et embarquer le comportement des modèles, notamment pour la conception des deux outils permettant la manipulation des modèles ainsi construits (enjeu E3).

Ce cas d'utilisation a donné lieu à la publication de l'article suivant, dans la revue *EMISA Journal*.

- Sylvain Guérin, Joel Champeau, Jean-Christophe Bach, Antoine Beugnard, Fabien Dagnat, Salvador Martínez, *Multi-level modeling with Openflexo/FML- A contribution to the MULTI process challenge*, in EMISA Journal, 2022 [96]

6.3 Projet SSE4Space

Ce troisième cas d'utilisation concerne une validation industrielle de l'approche. Il s'inscrit dans le cadre de la réponse à un appel d'offres de l'Agence spatiale européenne (ESA), sur le programme *General Support Technology Programme* (GSTP), et plus précisément dans le cadre d'une collaboration entre la société Telindus (Proximus Luxembourg) et l'IMT Atlantique. Le contexte général est la sécurité des systèmes spatiaux, aujourd'hui vitaux pour une société de plus en plus dépendante à ces systèmes, aujourd'hui une cible privilégiée des cyberattaquants. Le projet concerne un *framework* appelé *Secure Systems Engineering for Space* (SSE4Space), qui vise à rapprocher la gouvernance de la sécurité au plus haut niveau et les pratiques d'ingénierie des systèmes spatiaux sécurisés.

6.3.1 Introduction

Il existe aujourd'hui de nombreuses normes et méthodologies liées à la sécurité pour le développement de systèmes d'information terrestres, mais peu dans le domaine du spatial. Or la cybersécurité fait l'objet d'une attention croissante dans le spatial. Les missions spatiales civiles ont parfois négligé de manière cruciale l'importance de l'ingénierie de la sécurité, en raison notamment de la complexité et de la spécificité de ces systèmes, méconnus des cyberattaquants. Dans ce contexte, l'Agence Spatiale Européenne (ESA) a la responsabilité de protéger et de développer les intérêts et les capacités de ses États membres dans le domaine spatial civil et a répondu à cette demande accrue de sécurité spatiale par un certain nombre d'initiatives et de règlements et directives de sécurité de haut niveau, tels que l'obligation d'un cadre de sécurité de l'ESA et d'une politique de sécurité de l'information pour tous les projets et missions de l'ESA. Parmi les initiatives récentes en matière de sécurité, on peut citer la création du centre d'excellence en cybersécurité (*Cyber Security Centre of Excellence*, SCCoE), d'un centre opérationnel de cybersécurité (*Cyber Safety and Security Operation Centre*, CSOC), d'un certain nombre de solutions automatisées de tests de cybersécurité et enfin du *framework* SSE4Space.

Le *framework* SSE4Space, actuellement en cours de développement par l'ESA, vise à combler le fossé entre la gouvernance de la sécurité au plus haut niveau et les pratiques en terme d'ingénierie des systèmes sécurisés. Il s'agit notamment d'identifier et d'étendre les méthodologies applicables en matière d'évaluation des menaces et des risques pour

l'ingénierie des systèmes sécurisés ; de définir un vocabulaire cohérent à utiliser ; de créer des outils logiciels pour guider les utilisateurs tout au long du processus d'ingénierie ; et de définir et de générer des artefacts réutilisables tels que des catalogues d'exigences de sécurité, des rapports de tests et des rapports d'évaluation des risques, entre autres.

Notre contribution porte sur la conception d'une plate-forme logicielle dédiée à l'ingénierie de la sécurité pour les systèmes spatiaux. Elle se présente comme un outil d'analyse et de suivi des exigences de sécurité dans le cadre de la conception d'engins spatiaux. Ce logiciel permet la mise en œuvre de *workflow* (flux de travail) dédiés à l'évaluation des menaces et des risques, à l'ingénierie des exigences de sécurité, à la définition et à la mise en œuvre de tests de sécurité et des processus de certification de la sécurité. Une attention particulière est portée au support de l'échange de flux de données entre différents outils externes.

6.3.2 Exigences et contraintes

La conception de la solution implique de considérer un certain nombre d'exigences et de contraintes :

— La sécurité d'un système est une propriété émergente du système et résulte de la composition et de l'intégration de multiples composants, fonctions, interfaces et processus. Elle dérive des comportements et des interactions entre ces divers éléments (données, logiciels, matériels, équipements, personnes, processus, etc.). Il convient d'envisager la sécurité au niveau global du système.

— La conception doit être modulaire pour intégrer des données provenant de différents outils et technologies utilisés dans divers domaines. Le passage d'un outil à un autre doit se faire simplement et de façon "transparente" pour l'utilisateur. Chaque module ou sous-système doit pouvoir faire l'objet d'ingénieries de sécurité indépendantes et être réintégrées et consolidées au niveau du système global.

— Les utilisateurs sont des experts de domaines très divers, et le logiciel doit être aussi simple et facile à utiliser que possible pour tous ces acteurs. L'outil doit être suffisamment souple pour tenir compte des écarts par rapport au processus initialement défini.

— La traçabilité doit être maintenue tout au long du cycle de vie du processus - toutes les modifications apportées à un élément du système doivent pouvoir être retracées jusqu'à la source du changement, dans la perspective d'élaborer une méthodologie de certification. Il convient de vérifier dans quelle mesure le processus a été suivi et dans quelle mesure le niveau de sécurité souhaité a été assuré..

D'autres contraintes liées aux spécificités du domaine spatial sont à prendre en compte :

— Les attaques peuvent impliquer des canaux spécifiques au spatial (brouillage des signaux de radiofréquence, *spoofing* des signaux émis ou reçus par un engin spatial, interception et la modification de l'intégrité du signal et/ou de la navigation). Il convient de considérer les standards de sécurité des télécommunications.

— Les systèmes spatiaux sont généralement construits à base de matériel sobre en puissance de traitement. Le logiciel utilisé pour faire fonctionner ce matériel est souvent

écrit dans des langages de programmation de bas niveau tels que C/C++, Ada et Fortran. Ces langages sont souvent sujets à des vulnérabilités dues à de mauvaises pratiques de codage.

— La durée de vie des systèmes spatiaux est généralement longue (souvent entre 5 et 20 ans) et l'obsolescence technologique est fréquente.

6.3.3 La solution proposée

Le processus SSE4Space

La solution repose sur la définition d'un processus de conception dédié à la sécurité. Ce processus a été construit dans un contexte d'agrégation de multiples standards ECSS (*European Cooperation for Space Standardisation*), et dans la perspective d'être aussi générique que possible. Un projet peut ainsi choisir sa ou ses méthodologies d'analyse de risques et les outils qui correspondent le mieux au projet. Ce processus se fonde sur la définition de rôles, et le principe RACI [7] pour définir le contrôle d'accès. Une particularité du processus SS4Space tient à sa nature flexible. Plutôt que la définition d'un graphe de contrôle statique tel que BPMN, où le processus se définit comme un enchaînement de tâches liées par des décisions, le processus SSE4Space s'apparente à une approche décentralisée où chaque tâche définit ses entrées et ses contraintes (par exemple les éléments nécessaires pour terminer la tâche) et les résultats produits (les éléments modifiés et les éléments créés). L'état du processus global est donc calculé en permanence. Une autre forme de flexibilité consiste à pouvoir diviser le processus en plusieurs branches, puis à les fusionner. Le fractionnement peut se faire sur une seule tâche ou sur un ensemble de tâches choisies par un utilisateur ayant le rôle approprié. Chaque branche de la division peut se voir attribuer des droits d'accès différents en fonction du contexte. Cela reflète une situation réelle où plusieurs organisations, par exemple, travaillent simultanément sur différentes parties d'un artefact donné et où chacune d'entre elles a des droits d'accès différents sur l'artefact.

La plate-forme logicielle

La conception de la plate-forme SSE4Space est basée sur une architecture de microservices avec une interface web. Elle repose sur des composants ou modules logiciels, dédiés à des tâches spécifiques, qui communiquent entre eux. Les entrées gérées par SSE4Space sont très hétérogènes : interactions des utilisateurs, fichiers ou données extraits de divers outils, etc. Les sorties sont également très diverses, car la plate-forme peut mettre à jour des outils, pousser des fichiers ou afficher des résultats sur l'interface graphique de l'utilisateur. La plate-forme s'appuie sur architecture de modules connectés par des flux de données.

La figure 6.12 présente l'architecture de la plate-forme SSE4Space, et montre les différents composants et technologies utilisés, et les flux d'information et d'authentification entre ces derniers.

— **Outils externes** : ce sont les outils avec lesquels la plate-forme est en interaction et échange des données. Ces différents outils concernent la conception du processus, la

7. Responsible, Accountable, Consulted et Informed

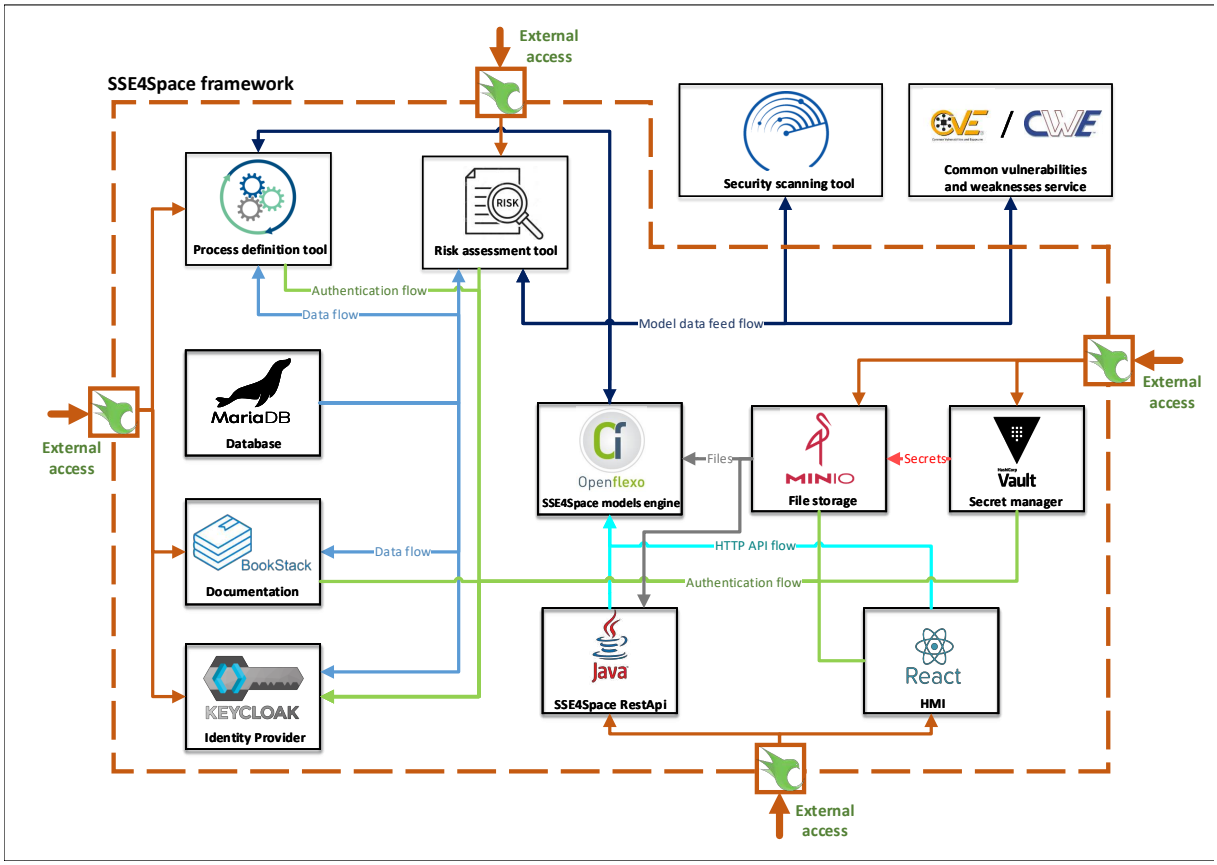

FIGURE 6.12 – Architecture logicielle SS4Space

spécification des exigences, l'évaluation des risques, le traitement des vulnérabilités et l'analyse de la sécurité.

— **Interface utilisateur** : l'interface web repose principalement sur un tableau de bord (cf figure 6.13), qui est alimenté par la liste des tâches calculées du processus. Pour garantir le niveau de sécurité visé, le logiciel doit guider les utilisateurs dans le déroulement d'un projet en veillant à ce que les tâches soient exécutées correctement et dans le bon ordre, en intégrant les tâches spécifiques à la sécurité dans le processus. Ce tableau de bord est complété d'une gestion documentaire très complète des éléments du processus SSE4Space et d'indicateurs de métriques et de performances

— **Accès et authentification** : l'authentification est centralisée dans l'application, et permet l'accès à tous les modules applicatifs.

— **Stockage** : les mécanismes de stockage reposent sur le stockage de fichiers (S3 API), structurés par projets, mais aussi sur des bases de données relationnelles sur lesquelles fonctionnent certains outils externes intégrés à la solution.

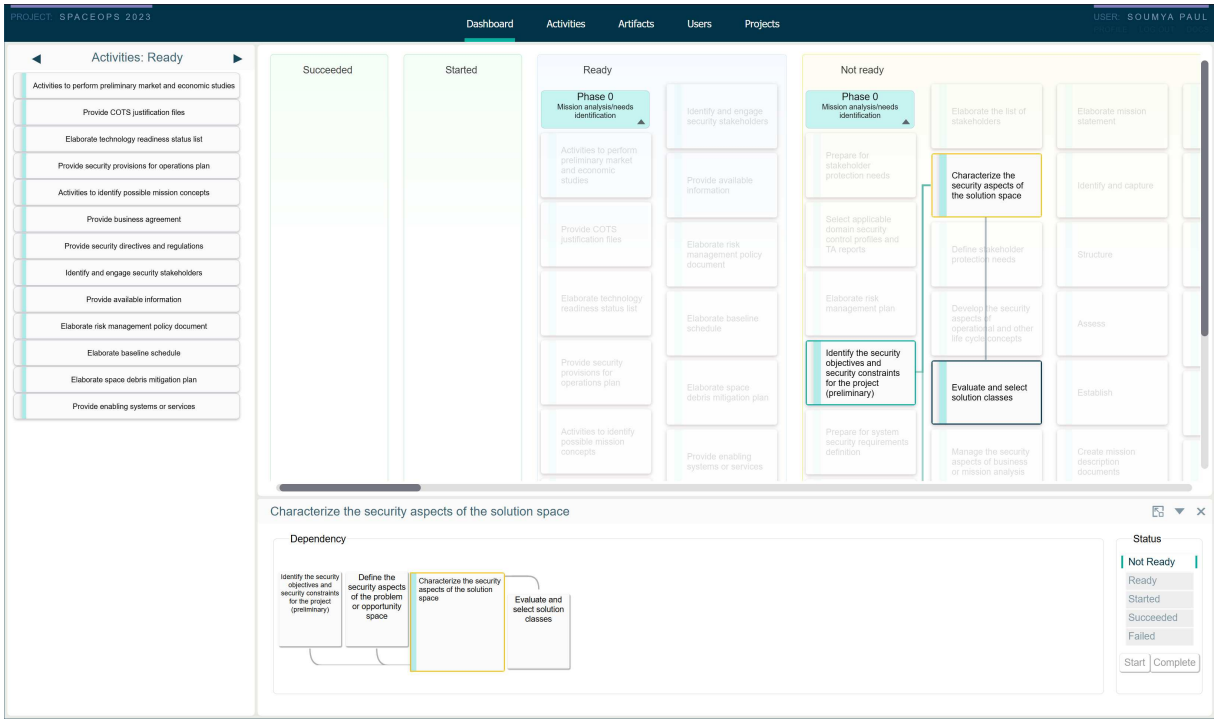

FIGURE 6.13 – Capture d'écran du logiciel *SS4Space*

L'approche fédération de modèle est très adaptée à ce contexte multidisciplinaire de composant métier à assembler et à faire interopérer. Plutôt que de réfléchir en terme de flux de données que s'échangent les différents outils, les artefacts sont considérés comme des modèles. La conception de l'application repose sur une vision globale avec un modèle métier qui regroupe et qui exploite de multiples modèles de domaines. Plutôt que d'échanger des données dans des documents, ces modèles de domaines constituent le principal moyen d'échange d'informations entre les ingénieurs, avec leurs données issues des différents outils, et leurs propres comportements.

La plate-forme repose sur un certain nombre d'outils propriétaires, pour lesquels des adaptateurs technologiques ont été développés, permettant de fournir les abstractions nécessaires à la construction des modèles de domaine. La figure 6.14 présente l'architecture

simplifiée du modèle SSE4Space, avec les différents modèles de domaines construits à partir des données exposées par les différents outils externes. Cette modélisation se structure autour de trois modèles principaux :

— **Workflow** : ce modèle se décompose en trois sous-modèles Artifact, Task et Workflow. Les modèles Artifact et Task sont construits en utilisant les données de l'outil de définition du processus. Ces données comprennent des propriétés telles que `name`, `description`, `id`, `tags`, etc. de l'artefact/de la tâche ainsi que le comportement des modèles eux-mêmes. Par exemple, le modèle de tâche a deux comportements appelés `startTask()` et `finishTask()` qui contiennent la logique permettant de maintenir le statut de la tâche. Le modèle de *workflow* orchestre les tâches et les artefacts (modèles) principalement à l'aide du comportement `computeWorkflow()` qui calcule le processus à partir des dépendances entre les tâches et les artefacts et met à jour dynamiquement l'état des tâches. Le comportement `synchronize()` est utilisé pour synchroniser les informations du modèle avec les outils externes. Le *frontend* utilise les données de ce modèle pour mettre à jour le tableau de bord.

— **RiskAnalysis** : ce modèle comprend tous les sous-modèles pertinents pour l'analyse des risques du projet. Les sous-modèles immédiats sont appelés Assets et Vulnerabilities. Le modèle Assets contient les données relatives aux actifs (Asset) pertinents pour le projet, qui se décomposent en PrimaryAsset et SupportAsset, et qui définissent un comportement `analyzeThreatImpact()` qui calcule l'impact d'une menace. Le modèle des vulnérabilités (Vulnerabilities) comprend des sous-modèles pour les menaces, les faiblesses, les scénarios et les adversaires, ainsi que pour les CVE et CWE qui constituent les données d'entrée de ces mécanismes d'analyse. Ce modèle d'analyse des risques est complété par la notion de risque Risk, liée aux actifs à protéger, aux objectifs en terme de sécurité, et aux mesures mises en place.

— **Requirements** : ce modèle est construit à partir des données de l'outil de spécification des exigences pour le projet. Les modèles d'exigences et de risques sont fédérés au sein du modèle SSE4SpaceCore. L'analyse de risque ne sera alimentée que par des exigences de sécurité ou des exigences qui ont un impact sur la sécurité. Cette logique est exécutée par le comportement `updateRiskFromRequirements()`.

6.3.4 Conclusions et enseignements

Ce cas d'utilisation a permis une validation industrielle par la société Telindus du langage FML et de l'infrastructure Openflexo, et montre l'adéquation de l'approche à la réalisation d'applications opérationnelles pour faire interopérer des sources d'informations hétérogènes. Il témoigne également de la capacité d'Openflexo à être pris en main par un industriel "externe" et étendu selon ses besoins, ce qui constitue un premier enseignement.

Un autre enseignement tient à la validation des API proposées pour implanter des adaptateurs technologiques. En effet, de nombreux outils externes utilisés dans le cadre de ce projet sont propriétaires et des adaptateurs technologiques ad hoc ont dû être développés.

Un troisième enseignement concerne la validation de l'adéquation du langage FML à construire des modèles métier à partir de modèles de plus bas niveau, et à les composer pour fournir des abstractions et des services à d'autres modèles métiers plus généraux.

Par ailleurs, et d'un point de vue technique, ce cas d'utilisation industriel a démontré la capacité de l'infrastructure à être utilisée comme *backend* en dessous d'une application web, via l'API HTTP/REST offerte par le serveur Openflexo.

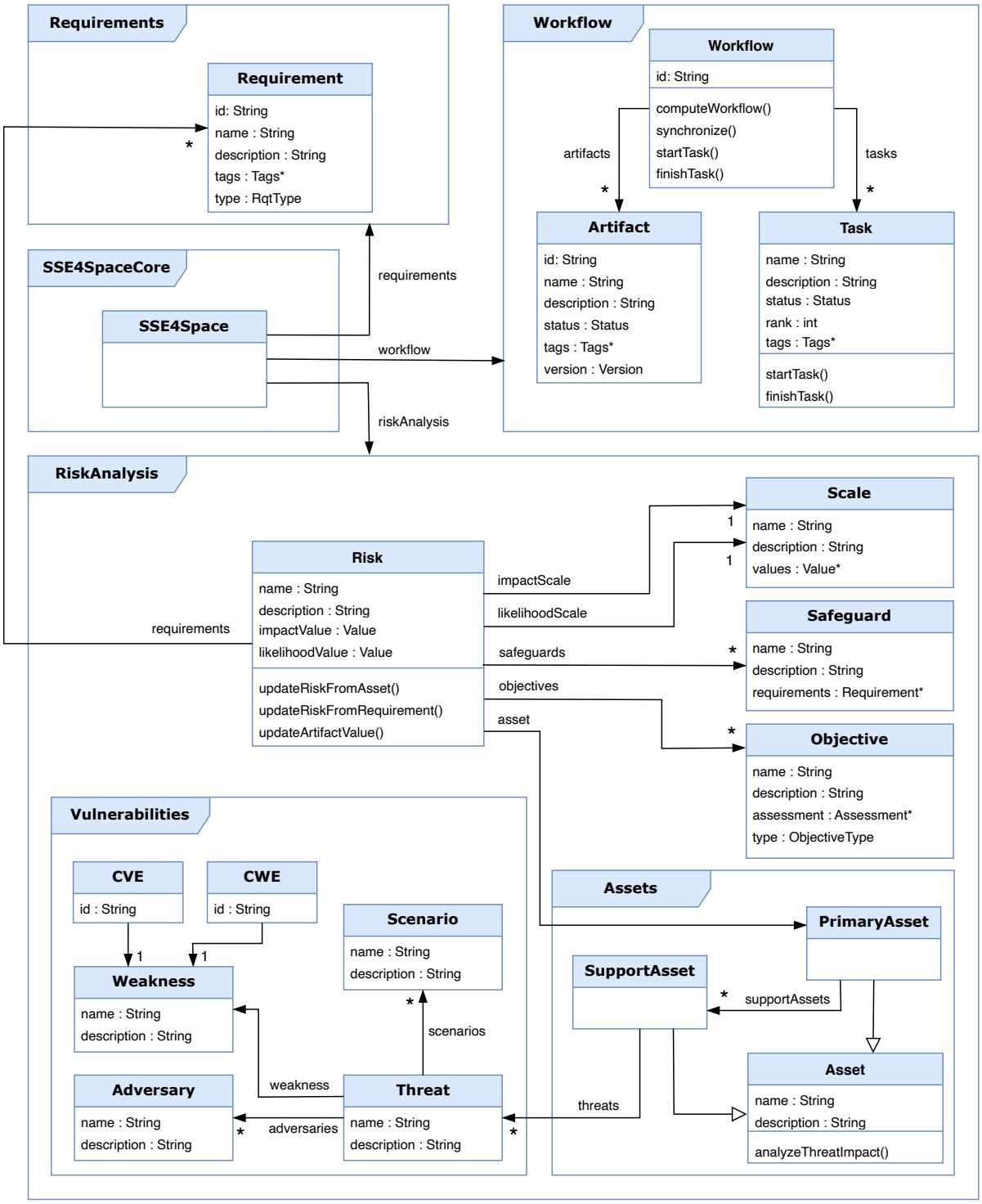

FIGURE 6.14 – Architecture (simplifiée) de la modélisation SSE4Space

Un dernier enseignement de ce cas d'utilisation tient à la méthodologie avec laquelle le projet a été mené. L'infrastructure Openflexo a en effet été utilisée dès le début du projet, dans le cadre des premières étapes de capture des exigences et de modélisation. Ces travaux ont été menés collectivement avec l'ensemble des parties prenantes (Telindus/Proximus et l'IMT Atlantique), sur la base de modèles métiers décrits en FML. Ces modèles de domaine étaient dans un premier temps très abstraits, puis se sont concrétisés au fur et à mesure des itérations successives lorsque les adaptateurs technologiques nécessaires se sont révélés disponibles, et que la connexion aux données réelles des outils externes est devenue possible. Au final, le langage FML permet de considérer un *continuum de modélisation* en conservant les mêmes artefacts [8] depuis les phases initiales de conception jusqu'aux étapes d'opérationalisation et d'exécution de ces mêmes modèles dans l'infrastructure Openflexo.

Nous concluons enfin sur les enjeux posés dans la section 3.1, page 53. Cette application industrielle a permis de valider la pertinence du langage FML pour la métamodélisation de modèles de domaines encodés dans des outils divers (enjeu E1), avec différents formalismes (enjeu E2), et leur fédération. Enfin, le langage FML et l'infrastructure Openflexo se sont révélés très adaptés à définir et exécuter des comportements sur la fédération, à partir de comportement réactifs décrits pour les outils externes (enjeu E3).

Ce cas d'utilisation a donné lieu à la publication de l'article suivant, dans le cadre de la 17^{th} *Conference on Space Operations*, Dubai, du 6 au 10 mars 2023.

- Tom Leclerc, Soumya Paul, Jussi Roberts, Fabien Dagnat, Florian Ledoux, Jean-Christophe Bach, Marcus Wallum, Nicky Mezzina, Daniel Fischer, Sylvain Guérin, Ihab Benamer, and Pierre JeanJean. *A flexible and robust framework for the secure systems engineering of space missions*. In 17th International Conference on Space Operations, Dubai, United Arab Emirates, 6-10 march 2023, 2023, [146]

6.4 Modélisation libre (*free modelling*)

Nous introduisons dans ce dernier cas d'utilisation les principes et l'outillage de techniques de capture de modèles métier dans le cadre d'une audience non-familière avec la modélisation. Après un rapide exposé sur des situations de modélisation, nous introduisons à la problématique de la capture de l'expertise métier comme modèle de domaine dans un formalisme donné. Nous proposons des techniques d'interaction avec les modèles (la « modélisation libre ») que nous avons expérimentées dans un outil que nous avons appelé le *Free Modelling Editor*. Cet outil est construit et assemblé avec les techniques de la fédération de modèles.

6.4.1 Introduction

Nous avons longuement évoqué, dans l'état de l'art, la modélisation comme procédé intellectuel à la base de la plupart des stratégies de résolution de problèmes, en particulier de la démarche scientifique et de l'ingénierie [139]. On modélise en s'appuyant sur des

8. au prix d'éventuelles étapes de refactoring ou d'abstraction

écrits [9] sous forme de phrases ou de dessins. Ces phrases et dessins, artefacts issus de la modélisation, doivent respecter des règles de construction plus ou moins explicites ou formalisées.

Dans « *On the art of modeling* » [159], W. Morris propose de passer d'un processus de modélisation intuitif à une approche explicite. Il illustre l'article par un problème de planification de transport. Au-delà des étapes de réflexion, il fait apparaître deux étapes qu'on retrouve dans tout processus de modélisation :

1. *Consider a specific (...) instance of the problem* : identifier des exemples ;
2. *Establish some symbols* ; déterminer leur généralisation ou abstraction, et en définir des représentations, à travers des variables mathématiques par exemple.

Il note que la production de ces artefacts (les exemples et les symboles) suit un processus d'élaboration par enrichissement :

« *The process of model development may be usefully viewed as a process of enrichment or elaboration. One begins with very simple models, quite distinct from reality, and attempts to move in evolutionary fashion toward more elaborate models which more nearly reflect the complexity of the actual management situation.* »

Enfin, il met également en évidence le besoin de liens (implicites ou explicites) entre ces artefacts :

« *Analogy or association with previously well developed logical structures plays an important role in the determination of the starting point of this process of elaboration or enrichment.* »

Dans ce contexte de la modélisation libre, nous appelons modèle une représentation explicite d'un système ou d'un problème et métamodèle, l'ensemble des règles explicites - quel que soit leur mode d'explicitation - qu'un modèle doit respecter. Nous nous proposons de généraliser l'approche présentée à l'occasion du scénario B, étudié section 3.4.2, page 74, et nous considérerons que la relation de conformance χ entre un modèle et un métamodèle peut être connue a priori, mais est parfois élaborée a posteriori, comme le résultat d'un choix de modélisation.

Nous considérerons donc deux niveaux de modélisation : le modèle (exemple, instance, concrétisation) et le métamodèle (généralisation, abstraction) et nous nous intéressons aux relations entre modèles et métamodèles sans prendre en considération leur structure ou leur contenu. Enfin, pour simplifier le discours, nous ne nous intéresserons pas à l'intérieur des artefacts de modélisation.

6.4.2 Des situations de modélisation

L'identification des usages de ces deux niveaux de modélisation peut servir de base pour réfléchir aux mécanismes de modélisation, en tant qu'activité intellectuelle. Nous présentons ici une classification identifiant des situations (élémentaires) de modélisation. Ces situations constituent une grille de lecture intéressante pour évaluer des outils de modélisation [23]. Cette classification nous sera ensuite utile pour nous positionner dans le

9. Même si les premiers philosophes discutaient sur l'Agora sans laisser de traces écrites et que nos ancêtres ont dû résoudre bien des problèmes avant l'invention de l'écriture...

contexte de la capture de modèles de domaine par des utilisateurs non formés aux techniques de modélisation ou par des modélisateurs non-experts.

Pour décrire les situations de modélisation, nous avons deux types d'artefacts à considérer : les modèles et les métamodèles. Les situations varient selon l'ordre dans lequel ces artefacts apparaissent ou sont reliés dans la démarche. Nous simplifions la description en ne considérant qu'un seul acteur dans chaque situation. Une analyse plus fine de ces situations pourraient être intéressante en prenant en compte différents acteurs et donc différentes intentions dans le processus. La figure 6.15 présente les 11 situations considérées (le niveau instance figure en bas et niveau méta en haut).

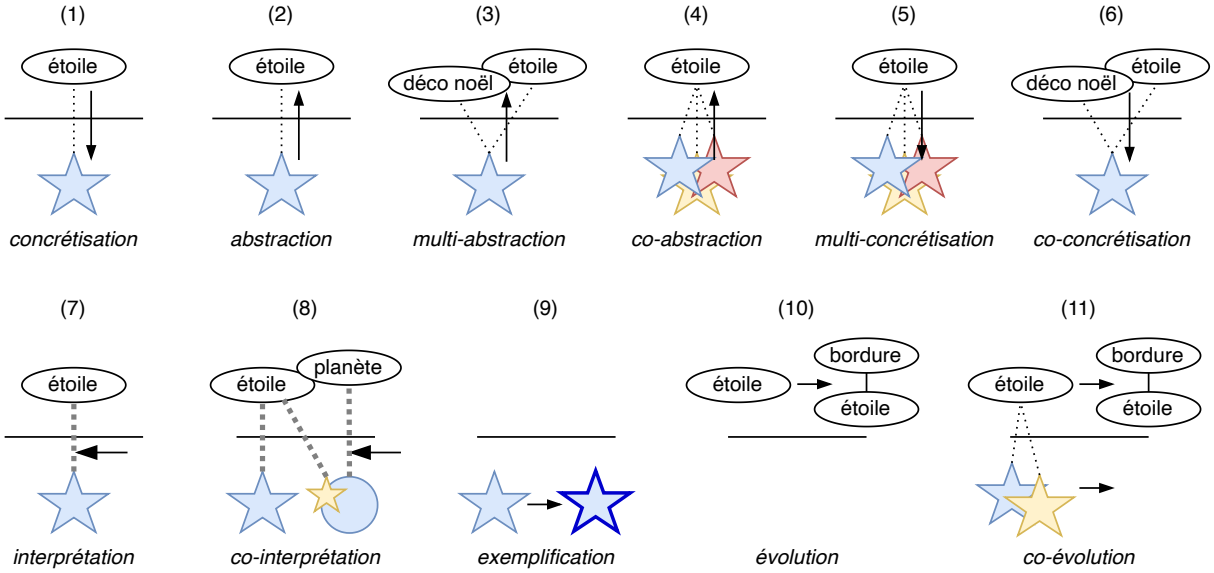

FIGURE 6.15 – Des situations de modélisation

1. Un métamodèle existe, on cherche produire un modèle *[concrétisation]*.
2. Un modèle existe, le travail consiste à trouver un métamodèle *[abstraction]*.
3. Un modèle existe, il faut trouver plusieurs métamodèles *[multi-abstraction]*.
4. Des modèles existent, il faut élaborer un métamodèle *[co-abstraction]*.
5. Un métamodèle existe, le travail consiste à construire plusieurs modèles *[multi-concrétisation]*.
6. Des métamodèles existent, le travail consiste à construire un modèle *[co-concrétisation]*.
7. Un modèle et un métamodèle existent, il faut les relier *[interprétation]*.
8. Des modèles existent, des métamodèles existent, le travail consiste à les relier *[co-interprétation]*.
9. Un modèle existe, le travail consiste à construire un autre modèle (sans aucun métamodèle) *[exemplification / extension]*.
10. Un métamodèle existe, le travail consiste à construire un autre métamodèle (sans aucun modèle) *[évolution / extension]*.

11. Un métamodèle existe avec plusieurs de ses modèles conformes, le travail consiste à faire évoluer le métamodèle (cas précédent) en adaptant (ou non) ses modèles *[co-évolution]*.

Les cas (9) et (10) sont probablement équivalents dans le cadre d'une interprétation multi-niveaux de la modélisation. En effet, à un niveau donné d'abstraction, l'absence de référence à un autre niveau, plus concret ou plus abstrait, rend le travail équivalent. Nous les différencions tout de même car la plupart des outils proposent des moyens de manipulations de ces deux niveaux très différents, liés à leur mode de représentation.

Une fois ces situations de modélisation identifiées, la question de leur composition se pose naturellement. Nous ne chercherons pas ici à être exhaustifs, mais nous pouvons illustrer quelques exemples de composition :

— *Composition comme séquence de situations* : la situation de multi-concrétisation (respectivement multi-abstraction) peut être vue comme une composition de plusieurs concrétisation (resp. abstraction). Ce n'est pas nécessairement le cas, car le fait de regrouper plusieurs actions dans la même situation ne représente pas exactement la même situation que de reproduire indépendamment plusieurs fois l'action.

— *Composition comme empilement* : un modèle peut parfois être interprété comme un métamodèle et conduire à la création de modèles de niveaux conceptuels différents. Inversement, un métamodèle peut aussi être considéré comme la concrétisation d'un méta-métamodèle. Il est donc possible d'empiler les situations de concrétisation, abstraction ou interprétation.

— Composition de modèles : l'approche que nous avons choisie ne prend pas en compte la structure des modèles et des métamodèles, qui peuvent contenir des éléments de modèles (ou de métamodèles) qui peuvent être eux-même considérés comme des modèles (ou des métamodèles). Cette relation de composition peut amener à de nouvelles situations que nous ne considérons pas ici.

Notons enfin que ces trois formes de composition se composent. Nous ne présentons ici qu'une ébauche de l'étude des situations de modélisation et de leur composition qu'il conviendrait d'approfondir.

6.4.3 Principes de la modélisation libre

Nous nous plaçons maintenant dans le contexte de la capture de la connaissance en tant que modèle de domaine (ou modèle métier), dans un cadre générique qui n'est pas celui d'un modélisateur expert, mais celui d'un expert de domaine.

Nous nous proposons de partir du scénario classique d'une organisation socio-technique (entreprise, administration, etc.), qui articule ses activités autour d'un modèle métier non explicité, mais qui manipule un certain nombre d'artefacts numériques. La figure 6.16 illustre par exemple le cas d'une organisation manipulant une présentation *powerpoint*, un fichier *excel* et un document PDF). Ces différentes sources d'informations "encodent" implicitement le modèle de domaine de l'organisation, informel mais pourtant bien connu des différents acteurs qui manipulent ces données, au travers de processus métier qui peuvent être eux-mêmes explicites ou non.

La problématique posée ici consiste à proposer une méthode outillée pour assister des utilisateurs non-modélisateurs dans l'émergence du modèle de domaine sous-jacent, en

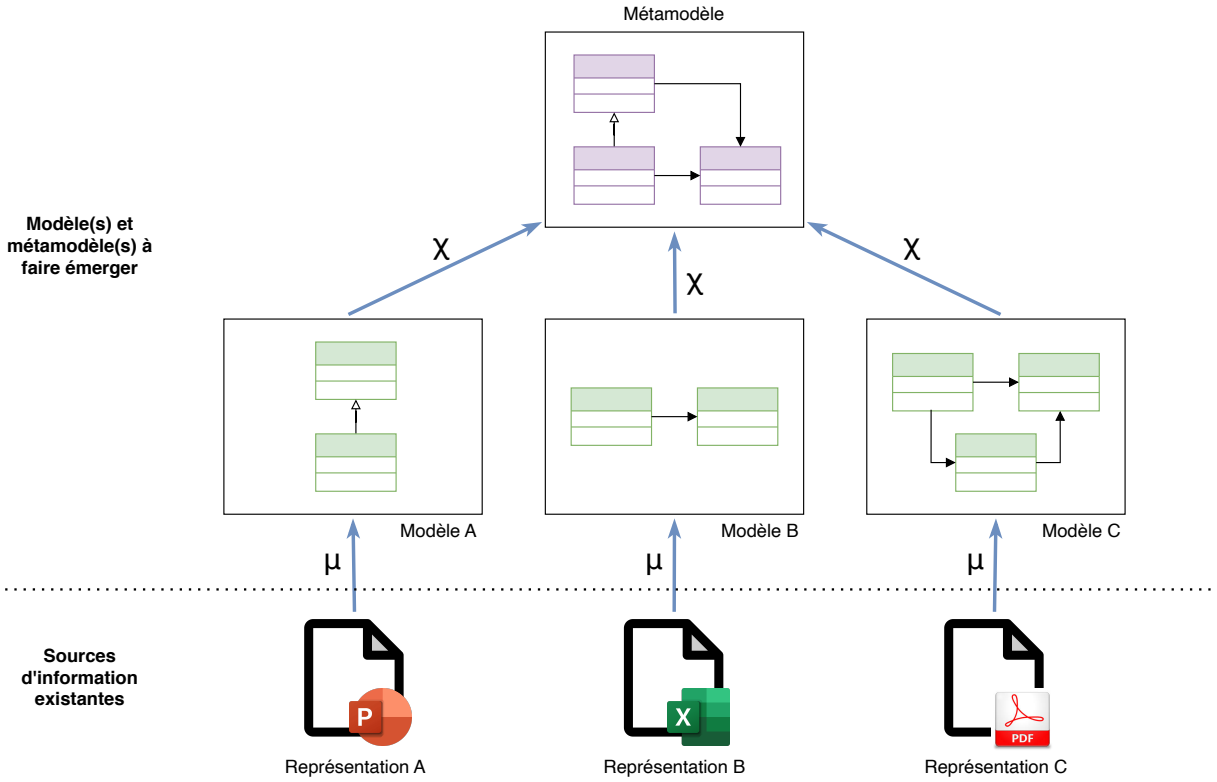

FIGURE 6.16 – Principes de la modélisation libre

tant que modèle conceptuel du domaine. Ce modèle de domaine se définit comme un système d'abstractions (concepts) significatifs du monde réel et qui concernent le domaine à modéliser. L'enjeu est par exemple de concevoir un logiciel dédié à l'outillage de certaines tâches de l'organisation, mais peut être aussi à des fins de documentation, de formation, de communication, de certification, etc. En ingénierie ontologique, ce modèle serait par exemple une représentation formelle d'un domaine de connaissance avec ses concepts, ses rôles, ses types de données, ses acteurs et ses règles, fondée sur une logique de description et mise en œuvre en langage OWL. A contrario des approches ontologiques, nous cherchons ici à capturer aussi le comportement. Les concepts incluent donc les données concernées par l'activité mais aussi les règles métier en rapport avec ces données. Un modèle de domaine utilise généralement le vocabulaire du métier pour que ses représentations puissent être utilisées pour communiquer avec des parties prenantes non techniques.

La figure 6.16 présente une vision (simplifiée) de ce scénario où l'on suppose que le modèle de domaine est encodé sous la forme d'un métamodèle (en haut de la figure), que les modèles A, B et C (au centre de la figure) encodent implicitement des instances de ce métamodèle (des exemples), au travers de la représentation qui encodée dans les sources d'informations A, B et C (en bas de la figure). On retrouve les deux étapes explicitées par W. Morris, et que l'on retrouve dans tout processus de modélisation [159] : l'identification d'exemples, et la définition de représentations.

Nous définissons la **modélisation libre** (*free modelling*) comme un ensemble d'activités de modélisation visant à faire émerger des concepts (ou métaconcepts) et des instances de ces concepts à partir d'une syntaxe graphique (ou de données existantes dans un outil quelconque, ce qui est équivalent). La présentation de ce scénario implique des sources d'information existantes, mais on peut le généraliser en imaginant des cycles d'itération

rapides, où l'on alterne l'édition de sources d'informations, et la conceptualisation des données représentées.

Outils	Situations de modélisation											Commentaires
	1	2	3	4	5	6	7	8	9	10	11	
Dessin [a]	-	-	-	-	-	-	-	-	+	+	-	sans métamodèle
Dédiés [b]	+	(1)	(1)	(1)	+	-	-	-	+	(1)	-	avec métamodèle
Méta-éditeurs [c]	+	+	(2)	+	+	(2)	(2)	(2)	+	+	+/-	interpr. stricte

(1) : pour UML et SysML, les profils s'en rapprochent peut-être...
(2) : on peut probablement le faire en le programmant...

a. Powerpoint, Libreoffice, Visio, draw.io, etc.
b. Editeurs UML, SysML, BPMN, etc.
c. MetaEdit+ par exemple

TABLE 6.3 – Couverture des situations de modélisation par les outils de dessin

Nous nous proposons dans un premier temps de nous concentrer sur la capture de connaissances à partir de représentations diagrammatiques. Un rapide tour d'horizon des outils existants (démarche très préliminaire et non-exhaustive) est synthétisé sur la table 6.3. Nous considérons comme outil de modélisation des outils de dessin ou de présentation (par exemple *PowerPoint*, *Libreoffice*, *Visio*, *draw.io*, etc.), des outils de modélisation dédiés, avec un métamodèle, tels que des éditeurs UML, et des ateliers de modélisation comme *MetaEdit+*. Les situations présentées précédemment servent de critère de comparaison pour chacun des types d'outils. Il s'agit d'une esquisse de comparaison ; des ateliers ou des outils pourraient offrir des capacités spécifiques non étudiées ici. Les outils de dessins ne permettent pas de travail de méta-modélisation ; ils restent au niveau de la syntaxe concrète. Les outils dédiés sont spécialisés pour traiter les modèles prévus, mais n'offrent pas d'outils pour changer ou élargir le point de vue. Les méta-éditeurs permettent d'enrichir les points de vue mais dans une approche méta vers instance et sans offrir d'outils de composition.

La mise en œuvre de la modélisation libre suppose l'enchaînement de l'ensemble des situations de modélisation présentées dans la section 6.4.2. Or, il n'existe à notre connaissance aucun outil standard qui couvre l'intégralité des situations de modélisation décrites.

6.4.4 L'outil *Free Modelling Editor*

Assemblé selon les principes de la fédération de modèles au sein de l'infrastructure Openflexo, le *Free Modeling Editor* (FME) est un outil destiné à l'expérimentation de nouveaux modes d'interaction avec les modèles. Il est utilisé pour faciliter l'émergence d'une syntaxe graphique simultanément au modèle conceptuel associé. Son interface, illustrée par la figure 6.17, est découpée en une partie outils de dessin sur la droite et une partie modèle conceptuel sur la gauche. L'utilisateur peut librement dessiner (situation d'exemplification) dans l'espace au centre en plaçant sur la page de travail des formes graphiques libres sélectionnées depuis une des palettes « métier » ou en ré-utilisant des formes définies dans ce contexte (palette « contextuelle »). Lorsqu'une forme est sélectionnée, il est possible (via un menu contextuel) de l'associer à un élément du modèle conceptuel présenté sur la gauche (interprétation). Si le concept correspondant n'existe pas encore, l'utilisateur peut

en définir un nouveau (abstraction). Il est facile d'expérimenter les différentes situations de modélisation décrites plus haut avec le *Free Modeling Editor*. La séparation des espaces de travail et des responsabilités conduit à un fonctionnement de l'atelier qui permet de pratiquer toutes les situations présentées.

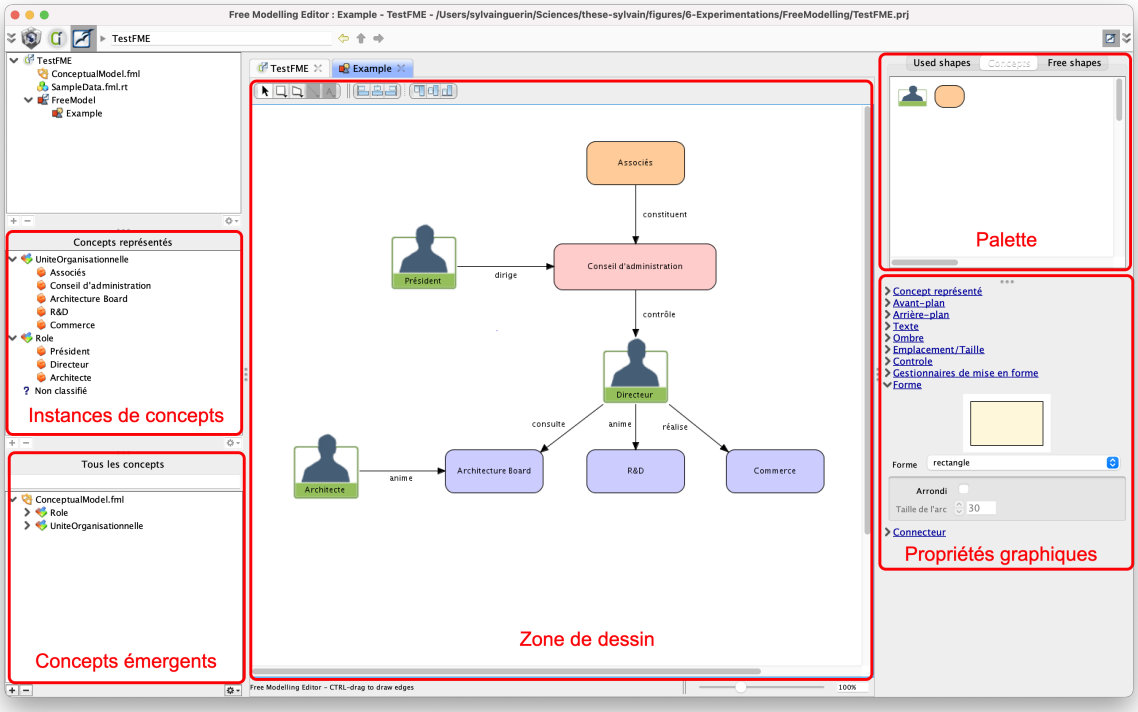

FIGURE 6.17 – Capture d'écran du *Free Modelling Editor*

L'architecture de l'outil FME est illustrée figure 6.18, et l'implantation repose sur la définition et l'exécution d'un VirtualModel `FreeModel`, dont la responsabilité est l'établissement de liens entre la représentation des concepts à droite (ici des formes graphiques dans un diagramme), et un modèle conceptuel, lui-même encodé en FML (à gauche). Cette architecture est très similaire au scénario B présenté section 3.4.2 et étudié section 4.3.2. L'outil s'appuie sur les deux niveaux conceptuels (modèle/méta et instance), et permet de manipuler des concepts et des instances de ces concepts (des exemples). Une forme graphique sur le dessin peut ainsi être promue à tout instant comme une instance d'un concept existant (dessin + concrétisation), comme une instance d'un nouveau concept (dessin + abstraction + concrétisation), ou comme une instance d'une évolution de concept (dessin + évolution + concrétisation), etc. L'outil permet également de travailler sur les propriétés des concepts émergents (propriétés simples ou liens entre concepts), ainsi que sur les valeurs de ces propriétés pour les exemples. L'outil gère la structuration de concepts par des relations de contenance, et gère enfin la réification de liens entre concepts.

6.4.5 Expérimentations

Au delà des premiers prototypes, le *Free Modelling Editor* a pu être expérimenté et validé dans un environnement industriel, et notamment par la société commerciale Openflexo SCIC qui l'a utilisé dans le cadre de projets commerciaux, pour la capture de modèles métier.

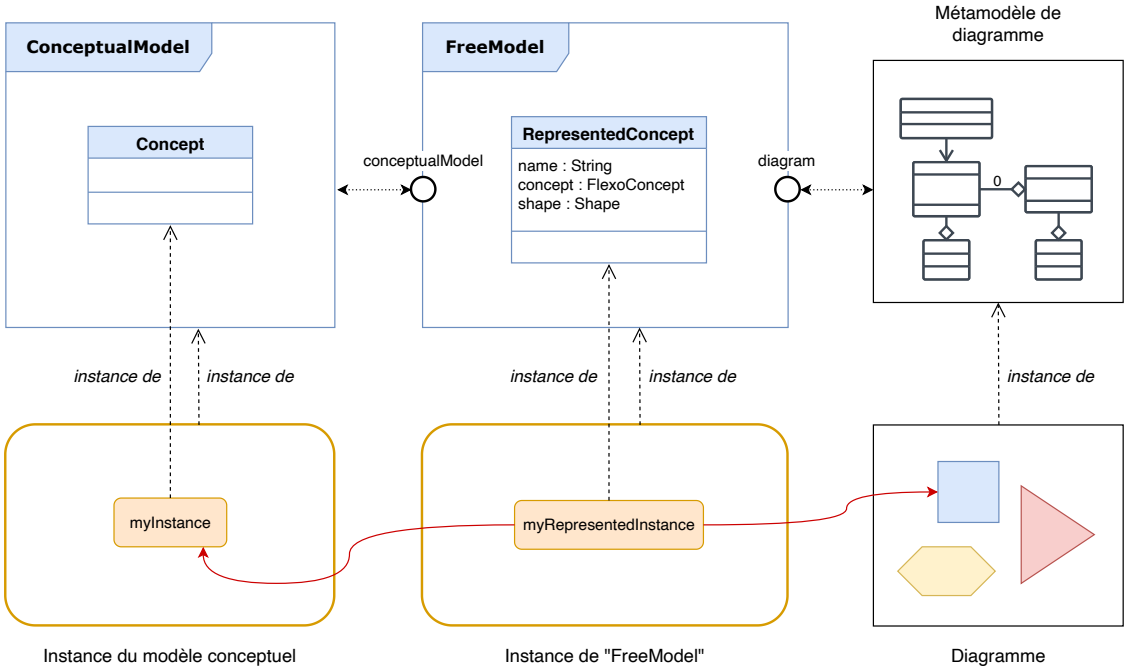

FIGURE 6.18 – Architecture de l'outil *Free Modelling Editor*

Le *Free Modelling Editor* a notamment été enrichi d'une fonctionnalité supplémentaire qui permet de fédérer une présentation powerpoint. L'exercice consistant à faire émerger un modèle métier à partir de "dessins" powerpoint existants s'est révélé très instructif. La méthodologie s'est structurée autour de séances de travail (d'une heure environ), avec un groupe d'experts de domaine (les "clients") mais non familiers du monde de la modélisation, et un expert modélisateur (expert FME), sur la base d'une ou plusieurs présentations *powerpoint*. L'exercice consiste à discuter des différents concepts métiers impliqués, informellement connus de tous les membres du groupe d'experts métiers, et de les désigner dans les représentations graphiques, en les classant comme concepts, ou comme instances de concepts (des exemples), en raffinant ou en spécialisant les concepts, en qualifiant les concepts avec des propriétés, etc. À l'issue de ces séances, l'outil FME permet de disposer des livrables suivants : un (ou plusieurs) métamodèle(s) (VirtualModel FML), une (ou plusieurs) instance(s) (exemples) de ce(s) métamodèle(s) et les représentations graphiques associées, et un éditeur graphique du domaine métier modélisé, qui permet d'instancier de nouveaux exemples du DSL sous-jacent (ce qui permet de se rendre compte d'éventuelles inconsistances métier et d'itérer à nouveau).

Ces expérimentations se sont révélées fructueuses en enseignements, et notamment :

1. la pertinence d'une approche graphique par le dessin pour l'émergence de DSL ;
2. la pertinence des méthodes agiles et l'intérêt de multiples itérations avec les experts métiers ;
3. l'importance de disposer d'outils de dessin/modélisation flexibles pour la liberté d'expression qu'ils offrent ;
4. la nécessité d'un équilibre entre consistance et flexibilité ;
5. la nécessité de la prise en considération des outils, des pratiques, des données et des modèles existants ;
6. l'intérêt de pouvoir mettre en œuvre des approches *top-down* et *bottom-up*.

La modélisation s'avère en effet une démarche complexe, avant tout intellectuelle, mais qui peut être outillée. Le papier et le crayon (ou leur équivalent) restent des outils de base grâce à leur flexibilité et à la liberté qu'ils offrent. L'outillage informatique apporte des fonctionnalités supplémentaires qui permettent de vérifier des propriétés telles que la conformance. Pourtant, cet apport se fait au détriment de la liberté. Comment concilier vérification et liberté ? Le *Free Modelling Editor* nous a permis d'explorer des pratiques différentes de celles offertes par la plupart des outils standards.

Un dernier enseignement concerne l'adéquation de l'outil à concevoir dans un premier temps un modèle de domaine qui peut dans un second temps être opérationnalisé (par exemple pour se connecter à des données réelles, ou pour construire une application dédiée au métier modélisé), tout en conservant les mêmes artefacts (un/des modèle(s) FML). Cette idée de *continuum de modélisation* a également été évoquée dans le cas d'utilisation SSE4Space (section 6.3.4, page 154).

6.4.6 Conclusions et perspectives

Le cas d'utilisation offert par l'implantation de la *modélisation libre* illustre pleinement les trois enjeux constitutifs de notre problématique.

La capture d'un modèle de domaine à partir de représentations diagrammatiques est le cœur même de l'enjeu lié à l'interprétation (enjeu E1). La solution présentée plus haut exploite le langage FML à deux niveaux : la conceptualisation (en FML) proprement dite (la partie gauche de la figure 6.18, et la conceptualisation de la représentation (au centre de la figure) au sein d'un outillage dédié (un éditeur graphique). Ce premier point évoque les ontologies, en tant qu'approches descriptives permettant de capturer un ensemble de connaissances. L'inspiration ontologique est d'ailleurs à la base de la conception du langage FML, en tant que langage de description (ou de ré-interprétation) de données/modèles existants pour construire de la connaissance, à la différence notable que le langage FML permet de modéliser également du comportement, et s'étend à de nombreux espaces technologiques.

Les aspects hétérogénéité et dynamicité (enjeux E2 et E3) sont présents également avec la capacité à se lier à des dessins et des présentations powerpoint existantes, et qui peuvent continuer à évoluer.

Si l'approche de la *modélisation libre* semble prometteuse, l'outil *Free Modelling Editor* souffre de certaines limitations :

1. Pour le moment, l'outil ne permet pas de manipuler d'autres représentations que des représentations diagrammatiques. L'architecture conceptuelle du FME permettrait d'offrir ces mêmes fonctionnalités de conceptualisation sur tout type de données, notamment grâce à l'ensemble des adaptateurs technologiques disponibles, et permettant donc d'autres formes de médiation entre la donnée, la représentation et la conceptualisation.
2. Dans sa forme actuelle, l'outil ne permet la conceptualisation que dans le formalisme FML (un modèle FML et des instances de ce même modèle). La séparation des préoccupations reflétée par l'architecture du FME permettrait de pouvoir capturer concepts et instances de concepts dans d'autres formalismes (par exemple un couple métamodèle/modèle EMF, ou une ontologie OWL).

3. Il serait bon d'étendre l'outil à la capture de multiples modèles sur la base de multiples sources de données.
4. Une limitation conceptuelle importante réside dans le choix implicite que l'utilisateur a à effectuer pour classifier une donnée comme un type ou une instance (un concept ou une instance de concept). Cette limitation est particulièrement pénalisante dans des contextes de modélisation à plusieurs niveaux conceptuels (modélisation multi-niveaux). Dans ce contexte, le langage FML offre une solution potentielle dans la définition d'une couche conceptuelle décrite en FML qui fournirait un langage intermédiaire qui gère explicitement ces aspects multi-niveaux (implantation par exemple des approches *clabjects* ou *potency* [127, 153, 6]).

Ces différents verrous offrent des perspectives intéressantes pour faire évoluer une approche outillée de la *modélisation libre*, en tant qu'objet de recherche à part entière. Des travaux futurs concerneront chacun de ces quatre aspects.

Ce cas d'utilisation a donné lieu à la publication des articles suivants :

- Fahad Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guérin, Christophe Guychard, *Using Free Modeling as an Agile Method for Developing Domain Specific Modeling Languages*, In : Proc. of the ACM/IEEE 19th International Conf. on Model Driven Engineering Languages and Systems. MODELS '16, New York, USA, ACM (2016) 24–34 [86].

- Antoine Beugnard, Fabien Dagnat, Sylvain Guérin, Christophe Guychard, *Des situations de modélisation pour évaluer les outils de modélisation*, INFORSID, Lyon, France, 2014, pp. 181-196 [23].

- Antoine Beugnard, Fabien Dagnat, Sylvain Guérin, and Christophe Guychard, *Des situations de modélisation pour décrire un processus de modélisation*, in Ingénierie des systèmes d'information, volume 20/2, 41-66, 2015 [26]

6.5 Conclusion sur les expérimentations

Nous avons eu l'occasion de conclure sur chacun des quatre cas d'utilisation évoqués plus haut, notamment sur les différents enjeux de notre problématique, et sur les limitations et les perspectives de chacune des expérimentations.

Ces quatre différents cas d'utilisation nous ont permis d'expérimenter et de valider à la fois l'approche, mais aussi la plate-forme, dans des contextes très différents. Couche applicative d'aggrégation et d'intégration de travaux de recherche, prototype de démonstration, logiciel d'infrastructure déployé en contexte industriel, preuve de concept et appui à la recherche : ces quatre exemples ont été choisis parmi de nombreux autres, sur une période qui couvre plus de dix ans d'une collaboration entre chercheurs et industriels, parce qu'ils illustrent chacun un aspect particulier et donnent chacun un éclairage différent. Tous ces projets ont permis de réorienter, valider ou consolider les choix scientifiques, conceptuels et techniques.

D'un point de vue conceptuel, tous ces exemples ont permis de converger et ont contribué à la maturité du langage FML, de sa sémantique et de sa syntaxe textuelle. Ces exemples ont notamment montré l'adéquation du langage FML à supporter une conception générique

et modulaire, qui facilite la réutilisation. Ils ont validé également l'expressivité du langage et son adéquation à la modélisation de domaines métiers divers.

D'un point de vue technique et en terme de génie logiciel, ces différents projets ont permis une certaine validation de l'architecture logicielle Openflexo, notamment en terme de séparation des préoccupations, de découpage des composants logiciels, de définition des APIs, et de déploiement d'une application fonctionnelle.

D'un point de vue méthologique, la plateforme Openflexo a montré son adéquation pour traiter différents cas d'utilisation, mais a montré également aussi ses limites en terme d'utilisabilité et d'expérience utilisateur. Le logiciel, dans sa version actuelle (2.0 pour la dernière version stable, et 2.99 pour la version de développement), s'avère très compliqué à utiliser et la courbe d'apprentissage est assez élevée. Il souffre notamment d'un manque d'ergonomie et de l'utilisation de technologies d'IHM assez anciennes (*Java Swing*). Enfin, la complexité des aspects conceptuels portés par la fédération de modèles ne facilite pas une prise en main rapide. Dans ce contexte, les développements récents, d'une part autour de la réification de la syntaxe textuelle de FML et d'autre part autour d'interfaces web qui permettent de construire des clients légers qui cachent la complexité, s'avèrent prometteurs.

Chapitre 7

Conclusion générale et perspectives

7.1 Conclusion générale

7.1.1 Objectifs de recherche et méthodologie

Les travaux présentés dans ce manuscrit ont permis de détourer certaines problématiques liées à la généralisation de la pratique de la modélisation. Même si très peu d'activités humaines revendiquent mener des tâches de métamodélisation explicites, et même si seule une petite partie de l'humanité en prend conscience, la modélisation est une pratique universelle, à l'origine de la pensée humaine et de la coopération entre être humains. Si, à l'évidence, le monde de l'ingénierie est concerné en premier chef (cf Ingénierie Dirigée par les Modèles), force est de constater qu'aujourd'hui presque toutes les sociétés manipulent des artefacts informatiques, sous une forme ou une autre, ne serait-ce qu'à travers des logiciels de traitement de textes, de tableurs, et de bases de données cachées derrière des formulaires. Toutes ces sources d'information sont des modèles (à la condition parfois de donner les conditions de leur interprétation si le métamodèle associé n'est pas explicite). Par ailleurs, tous ces modèles, formels ou moins formels, décrits dans des formalismes plus ou moins métamodélisés, sont intrinsèquement dynamiques et évoluent selon leur propre cycle de vie. La problématique d'une gestion coordonnée et consistante d'un ensemble de sources d'information se pose tout naturellement.

Après ce premier constat et un état de l'art des approches qui s'intéressent à cette gestion multi-modèles d'une part, et des approches de métamodélisation d'autre part, nous avons précisé nos objectifs de recherche en explicitant trois enjeux : l'interprétation de la donnée (E1), la prise en compte de l'hétérogénéité (E2), et la prise en compte de la dynamicité des modèles à fédérer (E3). Un certain nombre d'exigences et de contraintes techniques complètent l'exposé de ces objectifs de recherche.

D'un point de vue méthodologique, nous avons cherché à motiver et expliquer nos propositions en nous appuyant sur cinq scénarios illustratifs des usages du monde de la modélisation, qui ont été utilisés en tant que fil conducteur de notre discours. Nous avons également cherché à valider nos propositions tant sur le plan scientifique que sur des aspects plus opérationnels de mise en œuvre (preuves de concept, démonstrateurs, pro-

totypes, voire applications déployées), en nous concentrant sur l'utilisabilité, l'ergonomie, l'expérience utilisateur, les performances et le passage à l'échelle.

7.1.2 Contributions et résultats

Une première contribution a permis l'explicitation et la formalisation de l'approche **fédération de modèles** (chapitre 3, page 51). Nous en avons précisé les concepts abstraits et proposé une terminologie et un métamodèle (figure 3.7 page 65). Nous avons également positionné cette approche par rapport à de multiples vocables et domaines de recherche, notamment par la proposition d'un diagramme de caractéristiques des approches "fédératives". Nous avons passé en revue les cinq scénarios prototypiques en détaillant une mise en œuvre abstraite de la fédération de modèles et en justifiant l'apport de cette technique au regard des trois enjeux et des contraintes techniques explicitées dans nos objectifs de recherche. Cette contribution est complétée par la proposition du *framework* MF^2 qui précise la sémantique opérationnelle de l'approche.

La deuxième contribution et le cœur scientifique de cette thèse repose sur la proposition du langage FML (chapitre 4, page 81), en tant que langage de métamodélisation permettant de réifier des liens de fédération et de leur associer un comportement. FML est à la fois un langage permettant la **conceptualisation** et la réification de l'interprétation, mais il est également doté d'un **mécanisme de désignation** qui permet l'établissement de liens de fédération vers des sources de données hétérogènes. C'est en outre un langage de modélisation qui offre à la fois la capacité à réagir aux comportements des modèles fédérés mais qui permet aussi de programmer des comportements agissant sur ces modèles. Nous avons décrit rapidement le langage et donné un aperçu de sa syntaxe textuelle et de sa sémantique opérationnelle. Nous avons également montré comment cette proposition constituait une réponse uniforme aux cinq scénarios prototypiques présentés plus haut.

Une troisième contribution s'appuie sur la proposition d'une infrastructure logicielle (appelée Openflexo, chapitre 5, page 117), qui se présente comme une bibliothèque de composants logiciels qui permet la mise en œuvre du langage FML et de la fédération de modèles. Cette infrastructure implante un environnement de développement intégré de FML, ainsi qu'un moteur d'exécution et un grand nombre d'adaptateurs technologiques. Ce *framework* s'accompagne d'une méthodologie outillée permettant son utilisation pour produire et instancier des logiciels métiers.

La présentation de quatre cas d'étude complète les contributions présentées dans le cadre de cette thèse. Le premier cas d'utilisation (le projet *Formose*, section 6.1, page 132) porte sur un projet de recherche en ingénierie des exigences, impliquant un consortium de partenaires industriels et académiques. Dans ce contexte, l'approche FML/Openflexo a été utilisée comme solution d'intégration des différentes contributions scientifiques, et a permis le développement d'un outil paramétrique du point de vue des méthodologies "utilisateur" mises en œuvre. Nous avons validé sur ce projet les aspects abstraction, modularité et réutilisation du langage, ainsi que le passage à l'échelle sur un démonstrateur représentatif d'un cas d'utilisation industriel. Le deuxième exemple (*MULTI Process Challenge*, section 6.2, page 139) a pour cadre un *challenge* scientifique. Il a permis de confronter le langage FML à la modélisation multi-niveaux et au développement d'un prototype outillant la solution. Le troisième cas d'utilisation (*SSE4Space*, section 6.3, page 149) a permis la validation effective de l'approche dans un contexte industriel, puisque l'infrastructure Openflexo est utilisée en *backend* d'une application déployée pour l'ESA .

Le dernier cas d'utilisation (section 6.4, page 156) traite d'une proposition scientifique à part entière, que nous avons appelée la modélisation libre (ou *free modelling*), qui se propose de traiter le typage de modèles d'un point de vue ontologique plutôt que linguistique. Le démonstrateur développé permet d'assister des utilisateurs dans la capture et l'émergence d'un modèle métier à partir d'exemples et de diagrammes.

Evaluation La majeure partie de ces contributions ont été évaluées par la communauté scientifique (les quatre cas d'utilisation décrits dans ce manuscrit ont donné lieu à des articles scientifiques). Certains projets impliquant des industriels ont pu également se confronter à une forme d'évaluation relative à un milieu plus professionnel. *SSE4Space* (section 6.3, page 149) est sans doute le projet qui reflète le plus une certaine adéquation de la solution Openflexo à des attentes industrielles. Cependant, la validation de propositions scientifiques ayant trait au génie logiciel n'est jamais aisée, et il serait souhaitable d'envisager une réelle évaluation de FML/Openflexo en la comparant avec d'autres approches. À notre connaissance, il n'existe cependant pas de *challenge* scientifique offrant un cadre permettant cette comparaison, pour des raisons évidentes de complexité de mise en œuvre.

Limitations Les différentes expérimentations menées dans le cadre de ces travaux ont cependant mis à jour certaines limitations de l'approche et/ou de l'infrastructure logicielle implantée.

Une seconde limitation tient dans des problèmes d'ergonomie de l'outillage développé, et de technologies d'IHM vieillissantes, comme évoqué dans la section 6.5. Les développements en cours de versions client/serveur avec des technologies web devraient améliorer sensiblement les expériences utilisateurs et favoriser l'adoption des logiciels produits.

Une troisième limitation identifiée réside dans la grande complexité conceptuelle exposée aux utilisateurs dans la mise en œuvre de la fédération de modèles, et qui peut générer une grande surcharge cognitive chez certains utilisateurs dans certains contextes. La prise en charge conceptuelle de nombreux modèles, connectés à leurs propres métamodèles, nécessite la maîtrise simultanée de toutes les expertises métiers fédérées. Cette complexité a également été observée dans des contextes de conceptualisation multi-niveaux. Ces observations ont nourri les réflexions sur le langage FML (notamment sur les aspects modularité et généricité) dont un usage adéquat et une méthodologie adaptée tendent à réduire cette surcharge cognitive.

7.2 Impact de ces travaux

Les travaux présentés dans le cadre de cette thèse s'appuient sur une période de plus de dix années de collaborations entre des partenaires industriels, des équipes de recherche, et une société privée qui a un temps exploité commercialement l'infrastructure logicielle présentée ici.

7.2.1 La société Openflexo

La SCIC Openflexo est née en 2014. Cette entreprise a pu bénéficier d'un statut juridique original, à mi-chemin entre le monde de l'entreprise et celui des associations. De forme privée et d'intérêt public, la SCIC (Société Coopérative d'Intérêt Collectif) associe des personnes physiques ou morales autour d'un projet commun alliant efficacité économique, développement local et utilité sociale.

L'intérêt public identifié ici est d'une part la conception et l'exploitation de la plate-forme logicielle Openflexo alors en construction, en tant que logiciel libre, mais aussi et surtout le développement d'un modèle économique innovant dans les valeurs de l'ESS (Economie Sociale et Solidaire). Ce modèle économique repose sur un grand nombre d'acteurs de la chaîne de valeur autour du logiciel, et notamment les salariés, mais aussi les partenaires privés, fournisseurs ou clients de solutions informatiques, des partenaires académiques issus du monde de la recherche, ainsi que des individuels et des collectivités qui peuvent prendre part à la gouvernance de la structure. De par sa nature, ce modèle économique ne prévoit pas d'acteurs financiers, dans la mesure où il se situe en dehors du champ usuel où interviennent des investisseurs (car ne générant pas de revenus liés au capital).

Le modèle économique promu se propose de co-gérer une base de composants logiciels open-source. Schématiquement, plutôt que de vendre de la prestation logicielle, l'idée est d'identifier différents acteurs qui ont les mêmes besoins et de détourer chaque besoin pour l'abstraire et pour concevoir un composant logiciel dont les coûts de développement seront mutualisés. Le composant logiciel retourne ensuite dans une base commune et devient accessible et utilisable par d'autres acteurs, en combinaison avec d'autres composants. Indirectement, ce modèle repose sur la valorisation du temps passé sur un développement, et non sur la valeur du logiciel produit. L'esprit de ce modèle économique est de susciter la coopération entre toutes les parties prenantes autour du logiciel.

D'un point de vue technique et du point de vue des entreprises bénéficiaires, ce modèle suppose une séparation stricte entre les aspects techniques et les aspects métier. Alors que les problématiques techniques doivent être ouvertes et partagées pour être mutualisées, les aspects métiers doivent au contraire pouvoir être protégés puisqu'ils y encodent la valeur ajoutée d'une entreprise (d'un client) qui n'a pas intérêt à partager son savoir-faire. En ce sens, le langage FML et l'infrastructure Openflexo répondent parfaitement à ce besoin, en proposant du logiciel complètement configurable qui résulte de l'interprétation et de l'exécution de modèles. Schématiquement, le modèle reste propriété de l'entreprise mais s'exécute sur une infrastructure logicielle ouverte construite en coopération. Les composants logiciels partagés incluent par exemple des adaptateurs technologiques, des éditeurs, des interfaces graphiques, des bibliothèques diverses, etc.

Du point de vue de l'organisation de la coopération, le statut juridique de SCIC permet à toute personne morale ou physique de participer à la gouvernance de l'entreprise (y compris les établissements d'enseignement et de recherche, et même les collectivités). La société Openflexo s'est construite autour d'un consortium composé de salariés, de bénévoles, d'entreprises bénéficiaires et d'organismes de recherche (principalement au sein de Télécom Bretagne, aujourd'hui IMT Atlantique, et de l'ENSTA Bretagne). Le cadre offert a permis l'établissement d'une coopération très étroite entre les travaux des chercheurs et les équipes de développeurs de l'entreprise, et la construction d'une gouvernance partagée. Du point de vue de la propriété intellectuelle, la SCIC Openflexo a travaillé sur la quali-

fication juridique du travail réalisé en coopération comme une œuvre collective (cf article L113-2 du code de la propriété intellectuelle [1]).

La recherche est source de connaissances, mais aussi source d'évolutions sociétales. Les stratégies de valorisation couramment encouragées s'appuient sur un transfert unidirectionnel de l'innovation et de la recherche vers le secteur privé, au service d'un modèle économique visant à augmenter la compétitivité des entreprises. Bien que l'entreprise Openflexo ait été liquidée en 2019, cette expérience témoigne de l'existence de formes de valorisation de la recherche alternatives. En effet, la SCIC Openflexo a expérimenté une autre forme de coopération étroite entre la recherche académique et le monde des entreprises, allant jusqu'au partage de la gouvernance. Un des intérêts de cette forme de partenariat étroite est pour les scientifiques d'avoir un accès direct aux problématiques des industriels. Cette coopération n'a été rendue possible que par la nature même de l'approche FML/Openflexo qui permet explicitement la séparation de la technique et du métier.

Enfin, une structure de l'ESS telle que la SCIC Openflexo propose un *pont* entre des partenaires aux objectifs souvent éloignés comme les industriels privés, à la recherche de profits, et les académiques publics, aux objectifs de publication, pour produire de la recherche et de l'innovation couplées, avec l'espoir d'améliorer la coopération entre ces partenaires.

7.2.2 Une plate-forme d'intégration continue de la recherche

Les différents scénarios illustratifs des usages de modélisation qui ont servi de fil conducteur à notre discours suggèrent une grande transversalité de l'approche *fédération de modèles* par rapport à de nombreuses thématiques de recherche en génie logiciel et plus particulièrement en IDM. Si la fédération de modèles est un domaine de recherche à part entière, elle peut également servir de support et de technologie intégratrice dans d'autres contextes.

C'est une des raisons de l'adoption de l'infrastructure Openflexo en tant que plate-forme d'intégration continue d'une partie de la recherche de l'équipe P4S [2] du laboratoire LabSTICC [3]. Cette équipe comprend des chercheurs issus de l'ENSTA Bretagne, de l'IMT Atlantique et de l'Université de Bretagne Occidentale (UBO).

Sur une période d'une dizaines d'années, de nombreux projets de recherche impliquant cette équipe ont pu s'appuyer sur la plate-forme Openflexo, à la fois au sein des équipes IMT Atlantique (par exemple le projet ANR *Formose* ou le projet *SSE4Space* pour l'ESA avec Telindus) et ENSTA Bretagne (par exemple le projet *OneWay* avec Airbus, et d'autres projets nationaux pour la DGA).

Au fil des années, la plate-forme Openflexo a permis d'agréger de nombreux composants logiciels génériques et réutilisables, et de fournir un environnement de prototypage rapide, au service de la recherche.

1. https://www.legifrance.gouv.fr/codes/article_lc/LEGIARTI000006278882
2. *Processes for Safe and Secure Software and Systems*, cf https://p4s.enstb.org/
3. UMR 6285, https://labsticc.fr/fr

7.3 Perspectives

7.3.1 La fédération de modèles pour différents usages

La fédération de modèles montre une certaine pertinence dans le contexte de différents usages et activités identifiés dans ce manuscrit, et notamment la conception et la gestion de systèmes (construction de modèles composites, synchronisation de modèles, alignement de modèles, représentation de modèles, éditeurs de modèles, de générateurs de code, etc.). Cependant, certains usages n'ont pas été beaucoup explorés, pour lesquels la fédération de modèles aurait peut-être un certain intérêt. Citons par exemple :

— la **fédération documentaire**, définie comme la gestion fédérée de la documentation liée à un logiciel ou un système, éventuellement couplée à un générateur de documentation ;

— la **rétro-ingénierie** d'un logiciel ou d'un système [62], qui pourrait être opérée selon les principes de la modélisation libre sur la base des artefacts qui le représentent (code source, fichiers de configuration, documentation, etc.) ;

— le **prototypage rapide** d'applications métier liées à la capture d'une expertise de domaine ;

— la conception de **jumeaux numériques** [25] qui s'appuient sur la composition de composants logiciels et d'artefacts existants ou reflétant un composant du monde réel. Des travaux sont en cours au sein de l'équipe P4S pour prototyper une architecture générique de *jumeau numérique* avec l'infrastructure Openflexo.

A noter que les deux premiers points évoquent l'idée du *continuum de modélisation* suggéré dans les cas d'utilisation SSE4Space (section 6.3.4, page 154) et la modélisation libre (section 6.4.5, page 162).

Plus généralement, les approches MBSE (*Model Based System Engineering*) longtemps préconisées voient leur déploiement s'intensifier dans différents domaines de l'industrie, notamment propulsées par la digitalisation de l'industrie. Dans ce contexte, les outils MBSE isolés ne peuvent plus suffire sans une infrastructure destinée à prendre en charge l'intégration sémantique et technologique des différents outils [154]. Des besoins en fédération de modèles pourraient se concrétiser avec le déploiement de ces méthodes MBSE.

7.3.2 Capture du métier et représentation de la connaissance

La capture de modèles de domaine par des informaticiens constitue un enjeu majeur dans la perspective de la conception de logiciels métiers. Plus généralement, cette problématique peut être mise en perspective dans le contexte de la sémiotique de données numériques (cf section 2.1.3). Comment donner du sens à la donnée ? Comment capturer une expertise métier de personnes qui ne maîtrisent pas les concepts de la modélisation ? Ces questions se posent avec la perspective d'exploiter le sens de cette donnée pour en faire quelque chose (par exemple construire du logiciel).

Dans ce cadre, la fédération de modèles et l'approche portée par la modélisation libre s'avèrent prometteurs, à la condition de s'intéresser aux quatre verrous majeurs évoqués dans les conclusions sur la modélisation libre (section 6.4.6) : manipuler toutes les représentations de données existantes, pouvoir conceptualiser dans d'autres formalismes que

FML, manipuler plusieurs sources de données, sortir du cadre type/instance pour aller vers une conceptualisation multi-niveaux.

7.3.3 Autres perspectives

Concernant des aspects techniques d'évolution de l'infrastructure, les travaux envisagées s'orientent dans différentes directions :

— le **déploiement de solutions "*headless*"** qui s'appuient sur un serveur Openflexo,

— le développement d'**interfaces web** qui permettent de construire des clients légers qui cachent la complexité,

— l'**édition collaborative** et le versionnement,

— des expérimentations autour des problématiques de **passage à l'échelle** avec la manipulation de gros volumes de données.

D'un point de vue plus théorique, les travaux présentés dans le cadre de ce manuscrit pourraient être approfondis sur différents points :

— **Multiples fédérations** : ce document ne couvre pas le partage d'objets entre différentes fédérations et les conflits potentiels qui en résultent, ce qui pourra faire l'objet de travaux ultérieurs.

— **Sémantique opérationnelle du langage FML** : quelques éléments de sémantique opérationnelle du langage FML sont décrits dans l'annexe C, sans que suffisamment de détails ne soient donnés pour la définition d'un oracle. Nous n'avons pas traité dans ce manuscrit la problématique liée au modèle d'exécution des VirtualModelInstance, notamment en terme de concurrence. Cette problématique est complexe dans un contexte de composition de modèles qui s'exécutent, puisque le modèle d'exécution résultant est lui-même une composition de plusieurs modèles d'exécution. Même si ces aspects de sémantique ont été rapidement évoqués dans les annexes B et C, de futurs travaux pourront concerner une exploration et une évaluation de différentes stratégies.

— **Typage de modèles** : nous avons longuement développé dans la proposition du langage FML l'idée que les modèles puissent embarquer leur propre comportement. Parallèlement, nous défendons l'idée de modèles typés. Des travaux sont en cours sur ce sujet. Dans ce contexte, il serait intéressant d'explorer des mécanismes de détection d'erreurs et d'inconsistances liées à l'exécution de comportements sur des modèles (à la manière d'un compilateur, il deviendrait possible de calculer les comportements qui rendent un modèle inconsistant au regard d'un système de types).

Liste de publications

Revues internationales

- Fabien Dagnat, Salvador Martínez, Antoine Beugnard, Jean-Christophe Bach, Joël Champeau, Sylvain Guérin, Fahad Golra, *Working with Multiple Models through Model Federation* (en finalisation de rédaction, soumission prévue 11/2023)

- Sylvain Guérin, Joel Champeau, Jean-Christophe Bach, Antoine Beugnard, Fabien Dagnat, Salvador Martínez, *Multi-level modeling with Openflexo/FML- A contribution to the MULTI process challenge*, in EMISA Journal, 2022 [96]

- Sylvain Guérin, Guillaume Polet, Caine Silva, Joel Champeau, Jean-Christophe Bach, Salvador Martínez, Fabien Dagnat, Antoine Beugnard, *PAMELA : an annotation based Java Modeling Framework*, in Science of Computer Programming, volume 210, 102668, 2021 [104]

- Antoine Beugnard, Fabien Dagnat, Sylvain Guérin, and Christophe Guychard, *Des situations de modélisation pour décrire un processus de modélisation*, in Ingénierie des systèmes d'information, volume 20/2, 41-66, 2015 [26]

Conférences internationales et workshops

- Tom Leclerc, Soumya Paul, Jussi Roberts, Fabien Dagnat, Florian Ledoux, Jean-Christophe Bach, Marcus Wallum, Nicky Mezzina, Daniel Fischer, Sylvain Guérin, Ihab Benamer, and Pierre JeanJean. *A flexible and robust framework for the secure systems engineering of space missions*. In 17th International Conference on Space Operations, Dubai, United Arab Emirates, 6-10 march 2023, 2023 [146]

- Sylvain Guérin, Joel Champeau, Antoine Beugnard, Salvador Martínez, *Association Constraints in Model-Oriented Programming*, in MLE'2023 workshop, MODELS'23, Vasteras 2023 [103]

- Bastien Drouot, Valéry Monthe, Sylvain Guérin, Joël Champeau, *Security Analysis : from model to system analysis*, CRiSiS 2022 : International Conference on Risks and Security of Internet and Systems, 2022 [19].

- Caine Silva, Sylvain Guérin, Raùl Mazo, Joël Champeau, *Contract-based design patterns : a design by contract approach to specify security patterns*, Proceedings of the 15th International Conference on Availability, Reliability and Security, ARES '20, 2020, [185].

- Tithnara Sun, Bastien Drouot, Fahad Golra, Joël Champeau, Sylvain Guérin, Luka Le Roux, Raul Mazo, Ciprian Teodorov, Lionel Aertryck, Bernard Hostis, *Domain-specific Modeling Framework for Attack Surface Modeling*, International Conference on Information Systems Security and Privacy, 2020 [196].

- Fahad Rafique Golra, Fabien Dagnat, Jeanine Souquières, Imen Sayar, Sylvain Guérin, *Bridging the Gap Between Informal Requirements and Formal Specifications Using Model Federation*, International Conference on Software Engineering and Formal Methods, janvier 2018 [88].

- Fahad Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guérin, 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, 2016 [84].

- Fahad Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guérin, Christophe Guychard, *Using Free Modeling as an Agile Method for Developing Domain Specific Modeling Languages*, In : Proc. of the ACM/IEEE 19th International Conf. on Model Driven Engineering Languages and Systems. MODELS '16, New York, USA, ACM (2016) 24–34 [86].

- Fahad Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guérin, Christophe Guychard, *Continuous Requirements Engineering Using Model Federation*, In : 24th International Requirements Engineering Conference (RE). (Sept 2016) 347–352 [83].

- Antoine Beugnard, Fabien Dagnat, Sylvain Guérin, Christophe Guychard, *Des situations de modélisation pour évaluer les outils de modélisation*, INFORSID, Lyon, France, 2014, pp. 181-196 [23].

- Christophe Guychard, Sylvain Guérin, Ali Koudri, Antoine Beugnard, Fabien Dagnat, *Conceptual interoperability through models federation*, SIMF Workshop / 16th International Conference, MODELS 2013 [101].

- Ali Koudri, Christophe Guychard, Sylvain Guérin, Antoine Beugnard, Joël Champeau, Fabien Dagnat, *De la nécessite de fédérer des modèles dans une chaîne d'outils*, GENIE LOGICIEL, numéro 105. Juin 2013 [134].

Rapport scientifique

- Sylvain Guérin, Joel Champeau, Jean-Christophe Bach, Antoine Beugnard, Fabien Dagnat, Salvador Martínez. *FML language : formal definition, syntax and semantics*. Technical and scientific report, ENSTA Bretagne, IMT Atlantique, 2023 [97]

Communications par poster et tutoriels

- Sylvain Guérin, Joël Champeau, Antoine Beugnard, Jean-Christophe Bach *FML : un langage d'assemblage de modèles pour l'interopérabilité sémantique de sources d'information hétérogènes*, Tutoriel présenté lors des journées GDR-GPL, Rennes 2023

- Sylvain Guérin, Joël Champeau, Fabien Dagnat, Salvador Martínez, Antoine Beugnard, Jean-Christophe Bach *Fédération de modèles, une solution d'assemblage de modèles pour l'interopérabilité de sources d'information hétérogènes : l'approche FML / Openflexo*, Poster de présentation lors des journées GDR-GPL, Vannes 2022

- Christophe Guychard, Sylvain Guérin, Ali Koudri, Antoine Beugnard, Fabien Dagnat, *Free The Modeling !*, Poster de présentation de la plateforme Openflexo de fédération de modèles, ICSE 2013, [102].

Annexe A

Le framework MF^2

Le *framework* MF^2 (*Model Federation Framework*) constitue une proposition d'implantation formelle de l'approche *fédération de modèles* présentée dans le chapitre 3. Dans MF^2, les modèles sont des termes formels et disposent d'une sémantique opérationnelle. Cette démarche s'accompagne d'un prototype simple implanté en OCaml [1], à titre de preuve de concept. La présentation du langage sous-tendant MF^2 est ici volontairement limitée et ne couvre pas toutes les fonctionnalités mises en œuvre.

Le chapitre 4 et l'annexe B présentent le langage FML comme une implantation plus complète et sophistiquée de l'approche *fédération de modèles*. Le principal intérêt du *framework* MF^2 est de définir précisemment (mathématiquement) la sémantique de base de l'approche de la fédération de modèles. Nous estimons que cela est nécessaire car, contrairement à la plupart des langages de modélisation (qui sont essentiellement structurels), notre langage fait appel au calcul.

L'objectif de cette annexe est de présenter le framework MF^2 en commençant par l'exposé des concepts de base avant de poursuivre sur une présentation plus exhaustive du modèle formel. Nous nous concentrerons sur les aspects de sémantiques opérationnelles, et nous n'aborderons pas les problématiques de typage de modèles, qui feront l'objet de travaux ultérieurs.

1. https://github.com/openflexo-team/mf2-code

A.1 Syntaxe du langage : modèles et fédérations

La syntaxe MF² est décrite dans la figure A.3 mais plutôt que de commencer par la décrire complètement, cette section présente les différents éléments syntaxiques en les expliquant progressivement au fur et à mesure de leur définition. Chaque ligne est numérotée afin de pouvoir s'y référer dans la suite de la description. Nous commençons par les entités structurelles, puis nous décrivons les expressions appelées ici *contenu* (Content).

A.2 Aperçu de MF²

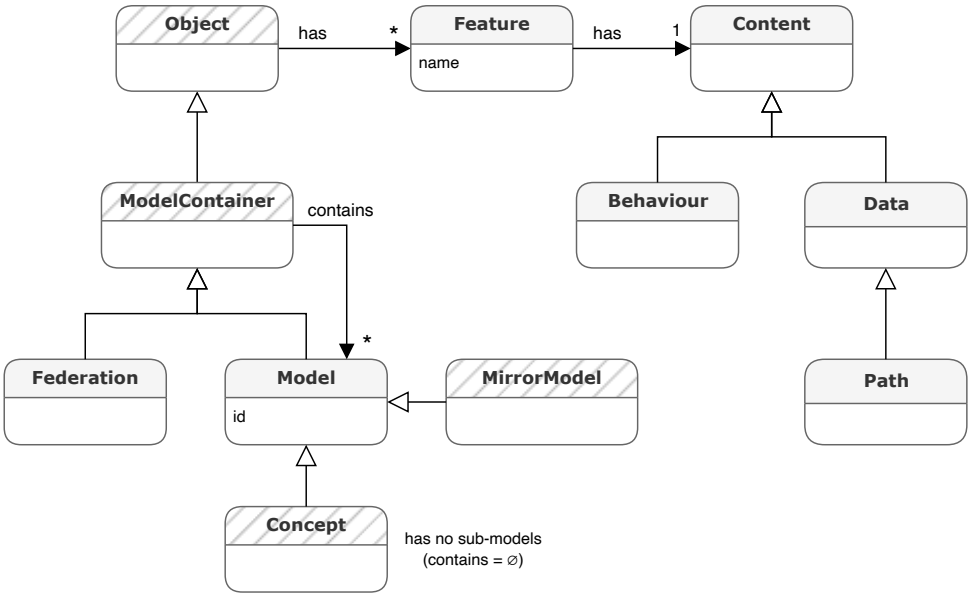

FIGURE A.1 – Concepts de base du *framework* MF² (les concepts hachurés n'ont pas de représentation syntaxique)

Pour faciliter la présentation informelle, la figure A.1 présente un métamodèle simplifié des concepts présents dans MF². Ils sont présentés ci-dessous. Les concepts hachurés n'ont pas de représentation syntaxique spécifique dans le langage, mais permettent la mise en correspondance avec l'approche abstraite.

L'approche MF² repose sur les *modèles* en tant qu'éléments de base. Ces modèles sont construits à partir d'informations de manière structurée. Par conséquent, un modèle (Model sur la figure) est un *objet* (Object) et, en tant que tel, possède un ensemble de *features* (Feature) nommées associées à un *contenu* (Content). À ce stade, le contenu est encore abstrait, mais il peut être classé en deux sous-classes : des *données* (Data) et des *comportements* (Behaviour). Les objets contiennent des contenus et ont un comportement qui exprime leur sémantique. Parmi les données, nous utilisons la notion de *chemin* (Path) qui désigne un autre objet, ce qui permet de relier les objets entre eux.

Il est à noter que les données peuvent être concrètes ou abstraites, car les modèles doivent pouvoir contenir des éléments concrets (*e.g.*, un concept avec une *feature 'name'* contenant une valeur de chaîne) et des éléments plus abstraits, soit pour définir des con-

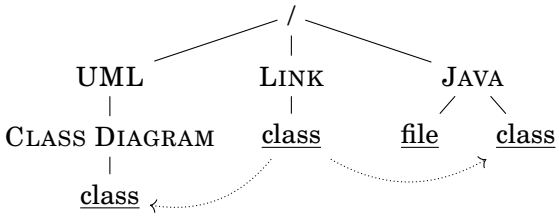

FIGURE A.2 – Une fédération est la racine d'un arbre de modèles

trats, soit pour représenter des éléments non encore connus. Par exemple, un concept peut avoir une *feature 'name'* contenant des valeurs de type `String` (chaîne de caractères).

Plutôt qu'une organisation en un univers "plat" de modèles, nous proposons de structurer les modèles comme des conteneurs (de modèles), suivant [43] et [38]. Ainsi, un modèle peut contenir d'autres modèles qui, à leur tour, peuvent contenir d'autres modèles (héritage entre Model et ModelContainer). Une fédération (Federation) est la racine d'une telle hiérarchie de modèles et son contenu est l'ensemble des modèles contenus dans la hiérarchie. Les feuilles d'un tel arbre sont appelées *concepts* (Concept).

La hiérarchie formée par les modèles fournit : (1) un moyen de structurer le contenu de la fédération et (2) une gestion facile de l'identité globale. Chaque modèle est contenu dans un conteneur unique (un modèle ou une fédération) et possède une identité qui doit être unique au sein de ce conteneur. Combiné à la hiérarchie de contenance, cela permet un mécanisme de nommage global avec la notion de *chemin* (Path). Un chemin commence à la racine (la fédération) et suit les conteneurs de modèles jusqu'au modèle ciblé. Le système de fichiers de Microsoft Windows offre une bonne métaphore de ce modèle, dans lequel la fédération est un disque et chaque modèle un répertoire. La figure A.2 illustre une telle fédération. Comme ce chapitre se concentre sur des fédérations isolées, la fédération sur laquelle on travaille n'est pas nommée (d'où le /). Elle contient trois modèles. Le premier, UML, est un modèle (partiel) du langage de modélisation UML, il contient un modèle CLASS DIAGRAM qui contient à son tour un concept class. À droite, on trouve un modèle objet JAVA contenant des fichiers et des classes (concepts file et class). Le modèle intermédiaire LINK est habituel dans les fédérations et est chargé de maintenir le lien entre un diagramme de classes et son implémentation Java (ici, il n'y a qu'un seul concept de lien : class). Les liens sont représentés par des flèches en pointillé dans la figure.

Les modèles fédérés ne sont pas directement réifiés dans MF². Certains modèles au sein de la fédération représentent les modèles fédérés externes. Nous les appelons MirrorModel. Nous supposons que chaque action sur un modèle fédéré est reflétée par son MirrorModel. C'est le mécanisme à la base de la notion d'*adaptateur technologique* détaillée dans la présentation du langage FML, section 4.2.3. Nous ignorerons largement cet aspect dans la présentation du *framework* MF² pour nous concentrer sur la sémantique opérationnelle d'une fédération.

A.3 Syntaxe du langage : modèles et fédérations

La syntaxe MF² est décrite dans la figure A.3 mais plutôt que de commencer par la décrire complètement, cette section présente les différents éléments syntaxiques en les expliquant progressivement au fur et à mesure de leur définition. Chaque ligne est numé-

rotée afin de pouvoir s'y référer dans la suite de la description. Nous commençons par les entités structurelles, puis nous décrivons les expressions appelées ici *contenu* (Content).

A.3.1 Espaces de nommage

Le *framework* MF² repose sur la définition de trois espaces de nommage : $\mathbb{I}$ qui permet l'identification des modèles, $\mathbb{N}$ pour le nommage des *features* et $\mathcal{X}$ pour les variables (lignes N1 à N3 de la figure A.3 (a)).

A.3.2 Modèles

Un modèle (Model) définit un ensemble de *features*, contient d'autres modèles et est identifié par un élément de $\mathbb{I}$. Une *feature* (Feature) est nommée avec un élément de l'ensemble $\mathbb{N}$ et définit un contenu (Content), élément de l'ensemble $\mathbb{C}$. Les éléments de $\mathbb{C}$ restent abstraits pour le moment, mais des valeurs brutes concrètes telles que des nombres ou des chaînes de caractères et des types concrets pour ces valeurs sont des exemples de ces éléments. Les exemples utilisent des valeurs spécifiques chaque fois qu'ils en ont besoin.

Un modèle est un terme fini généré par la règle G2 de la grammaire de la figure A.3 (b). L'ensemble des modèles est dénoté $\mathbb{M}$. Par exemple, le modèle JAVA de la fédération de la figure A.2 n'a aucune *feature* et contient deux concepts file et class. Tous deux ne contiennent pas de sous-éléments mais possèdent des *features* (une seule est présentée ci-dessous). Ces *features* ("path" et "name") ont un contenu str qui représente le type `String` (chaîne de caractères) :

$$\text{JAVA} \triangleright \left\langle \underline{\text{file}} \triangleright \langle path \mapsto \text{str}, ... \rangle, \underline{\text{class}} \triangleright \langle name \mapsto \text{str}, ... \rangle \right\rangle$$

Le domaine de JAVA est ici $\{\underline{\text{file}}, \underline{\text{class}}\}$, conformément à la règle F4.

Tous les termes générés par cette grammaire ne sont pas des modèles valides. Chaque modèle fournit l'espace de noms pour ses *features*, qui doivent avoir des noms uniques, et pour ses sous-modèles, qui doivent avoir des identifiants différents. Le prédicat wf défini par la règle F5 identifie les termes "bien formés" qui définissent les modèles valides.

A.3.3 Fédérations

Une *fédération* est définie comme un modèle racine sans identifiant tel que défini dans la règle G1 de la grammaire. Elle est soumise aux mêmes conditions de "bonne formation" qu'un modèle. Lorsque nous voulons parler d'une fédération ou d'un modèle indistinctement, nous utiliserons le terme *conteneur de modèle* (ModelContainer).

A.4 Syntaxe du langage : contenus

Dans MF², les features définissent des expressions appelées Content. L'ensemble des contenus possibles $\mathbb{C}$ présentés ici se limite aux éléments nécessaires pour les exemples. Il peut être facilement étendu. Il existe deux classes de contenus : le contenu de base pour

N1 : Identifiants de modèles	$i \in \mathbb{I}$	dénombrable et doté de la relation d'égalité $\overset{\mathbb{I}}{=}$
N2 : Nom des *features*	$n \in \mathbb{N}$	dénombrable et doté de la relation d'égalité $\overset{\mathbb{N}}{=}$
N3 : Variables	$x \in \mathcal{X}$	dénombrable

(a) Espaces de nommage

G1 : Fédérations $f \in \mathbb{F}$ [a]	f	$::= \langle n \mapsto c, ..., n \mapsto c \mid m, ..., m \rangle$
G2 : Modèles $m \in \mathbb{M}$ [b]	m	$::= i \triangleright \langle n \mapsto c, ..., n \mapsto c \mid m, ..., m \rangle$
Chemins $\pi \in \Pi$	π	$::=$
G3 : *chemins absolus* $\pi \in \Pi^A$		$\mid /i/.../i$
G4 : *chemins relatifs* [c]		$\mid \cdot \mid \cdot\cdot/.../\cdot\cdot/i/.../i$
Contenus de *features* $c \in \mathbb{C}$	c	$::=$
G5 : *chemins, contenus indéterminé*		$\mid \pi \mid ?$
G6 : *type et littéraux "chaîne"* [d]*, "unit"*		$\mid \mathsf{str} \mid \texttt{"..."} \mid \texttt{()}$
G7 : *features : accès et affectation*		$\mid c.n \mid c.n \leftarrow c$
G8 : *comportements : définition et appel* [e]		$\mid \texttt{fun}\ x \Rightarrow c \mid \texttt{fun}\ \_ \Rightarrow c \mid c\ c \mid x$
G9 : *création de modèles*		$\mid \texttt{new}\ id\ \texttt{in}\ c$

(sucre syntaxique) : G10 : $\texttt{let}\ x = c_1\ \texttt{in}\ c_2 \overset{\text{def}}{=} \texttt{fun}\ x \Rightarrow c_2\ c_1$
G11 : $c_1\texttt{;}\ c_2 \overset{\text{def}}{=} \texttt{fun}\ \_ \Rightarrow c_2\ c_1$
G12 : $(c) \overset{\text{def}}{=} c$

a. ensemble des *features* dénoté F, ensemble des sous-modèles dénoté M
b. séparateur omis si une partie est manquante
c. les $\cdot\cdot$ et les descentes ne peuvent être simultanément vides
d. l'ensemble des *chaînes* (`String`) est dénoté $\mathbb{S}$
e. λ-calcul

(b) Grammaire du langage

F1 : Identifiant d'un modèle	$\mathsf{id}: \mathbb{M} \to \mathbb{I}$	$\mathsf{id}(i: \pi \triangleright \langle ... \rangle) \overset{\text{def}}{=} i$
F2 : Nom d'une *feature*	$\mathsf{n}: \mathbb{N} \times \mathbb{C} \to \mathbb{N}$	$\mathsf{n}(n \mapsto c) \overset{\text{def}}{=} n$
F3 : Contenu d'une *feature*	$\mathsf{c}: \mathbb{N} \times \mathbb{C} \to \mathbb{C}$	$\mathsf{c}(n \mapsto c) \overset{\text{def}}{=} c$
F4 : Domaine d'un conteneur	$\mathsf{dom}: \mathbb{F} \cup \mathbb{M} \to \wp(\mathbb{I})$	$\mathsf{dom}(i: \pi \triangleright \langle F \mid M \rangle) \overset{\text{def}}{=} \{\mathsf{id}(m) \mid m \in M\}$
F5 : Objets bien formés	$\mathsf{wf}: \mathbb{F} \cup \mathbb{M} \to \{\mathbf{T}, \mathbf{F}\}$	$\mathsf{wf}(i: \pi \triangleright \langle F \mid M \rangle) \overset{\text{def}}{=} \bigwedge_{m \in M} \mathsf{wf}(m)$
		$\wedge (\forall f_1, f_2 \in F,\ \mathsf{n}(f_1) \overset{\mathbb{N}}{=} \mathsf{n}(f_2) \Rightarrow f_1 = f_2)$
		$\wedge (\forall m_1, m_2 \in M,\ \mathsf{id}(m_1) \overset{\mathbb{I}}{=} \mathsf{id}(m_2) \Rightarrow m_1 = m_2)$

(c) Utilitaires

FIGURE A.3 – Syntaxe du langage MF²

modéliser les propriétés (classe Data sur la figure A.1) et le contenu fonctionnel pour modéliser le comportement (classe Behaviour).

A.4.1 Contenus élémentaires

Un contenu peut être une chaîne de caractères qui est une séquence de caractères entre " ou la valeur "unit" dénotée `()`. Il est également possible d'utiliser la valeur str qui est le type des chaînes de caractère (règle G6). $\mathbb{C}$ contient également un élément ? pour représenter une feature qui n'est pas encore connue (règle G5). Il est à noter qu'un tel contenu inconnu doit être redéfini avant qu'un calcul n'y accède, sinon une erreur se produit.

A.4.2 Liens entre les modèles

L'objectif du *framework* MF² étant de fournir un cadre formel pour la fédération de modèles, nous devons *lier* les modèles entre eux. Pour référencer un autre modèle, nous utilisons son contenu *chemin* (Path) dans la structure de la fédération pour l'identifier. Ce chemin peut être relatif ou absolu (règles G3 et G4 de la grammaire).

Au sein d'une fédération, un chemin absolu est soit inconnu, soit correspond à un modèle unique. Il est défini par la règle G3 de la grammaire. La résolution des chemins fait partie de la sémantique de MF² et est expliquée dans la section suivante.

La longueur d'un chemin π est notée $|\pi|$ et vaut 0 pour $/$ et n pour $/i_1/.../i_n$.

Un chemin relatif dépend du conteneur de modèle actuel qui le contient. Pour un tel chemin, le premier élément du chemin est recherché dans le conteneur de modèle actuel. Le symbole $\cdot\cdot$ permet de remplacer le conteneur de modèle actuel (seuls les modèles peuvent avoir un parent) par son parent. Par exemple, le sous-modèle class de JAVA dans la figure A.2 peut contenir une *feature* "file" contenant un chemin relatif vers le modèle de même niveau hiérarchique file :

$$\text{JAVA} \triangleright \left\langle \underline{\text{file}} \triangleright \langle ... \rangle, \underline{\text{class}} \triangleright \langle \mathit{name} \mapsto \text{str}, \mathit{file} \mapsto \cdot\cdot/\underline{\text{file}}, ... \rangle \right\rangle$$

Une référence au conteneur de modèle actuel peut être explicitée par le chemin relatif $\cdot$ (règle G4).

A.4.3 Utilisation des liens

Les liens permettent à un modèle de réutiliser une *feature* d'un autre modèle. Dans ce cas, un terme Content contient un accès à une *feature* d'un autre modèle (règle G7). En général, tout contenu dont le calcul produit un chemin peut être utilisé. Par exemple, un contenu du modèle de classe de notre exemple en cours pourrait accéder à sa feature "file" pour obtenir le *path* de son fichier associé : $\cdot\,.\,\mathit{file}\,.\,\mathit{path}$.

A.4.4 Affectation du contenu d'une *feature*

En outre, il est possible d'insérer un nouveau contenu ou de modifier le contenu d'une *feature* (accesseur en modification). Pour ce faire, un opérateur $\leftarrow$ est défini (règle G7), l'opérande gauche définit l'élément à créer ou à mettre à jour (selon qu'il existe déjà ou non) et l'opérande droit définit son nouveau contenu.

A.4.5 Comportements

L'un des aspects les plus importants du *framework* MF^2 est la possibilité de définir des comportements pour les modèles. Un terme Content peut être une fonction ou un appel à une fonction. Pour que la syntaxe (et la sémantique) reste simple, nous nous en tenons à une approche simple de type λ-calcul (règle G8 de la grammaire). Une fonction $\texttt{fun}\ ; x \Rightarrow c$ ou $\texttt{fun}\ ; \_ \Rightarrow c$ a un corps c et attend un argument, la première stocke la valeur de cet argument dans la variable x tandis que la seconde la rejette. Lorsqu'une fonction est appliquée, deux expressions sont fournies, l'une pour la fonction et l'autre pour l'argument. Un sucre syntaxique permet de simplifier la lecture et l'écriture du contenus (règles G10 à G12).

Par exemple, le concept class peut être étendu pour inclure une fonctionnalité "rename" qui, lorsqu'elle est appelée, reçoit une valeur et l'affecte à la *feature* "name" :

$$\underline{\text{class}} \triangleright \langle name \mapsto \mathsf{str}, rename \mapsto \texttt{fun}\ x \Rightarrow \cdot.name \leftarrow x \rangle$$

A noter qu'il est possible de définir une fonction récursive f comme une feature d'un modèle i : $i \triangleright \langle f \mapsto \texttt{fun}\ x \Rightarrow ... \rangle$ en utilisant $\cdot.f$ dans son corps. En pratique, comme ce n'est pas très efficace, on peut étendre le langage pour qu'il prenne directement en charge les fonctions récursives.

A.4.6 Création de modèles

Enfin, un contenu peut être une expression de création (règle G9). La création consiste à créer un nouveau modèle vide. Pour cela, elle requiert un identifiant id et d'un emplacement. Intuitivement, le résultat est la création d'un nouveau modèle avec l'identifiant id à l'emplacement considéré. L'emplacement est évalué par un chemin et l'identifiant ne doit pas être déjà défini au sein du conteneur de modèle.

A.5 Sémantique formelle

La description de la sémantique s'appuie sur les mécanismes de réduction standard, à petits pas, et notamment sur la β-réduction (appel par valeur) du λ-calcul telle qu'on la trouve par exemple dans [171]. Le principe est de spécifier pour chaque terme comment il peut se réduire, en une étape, à un autre terme. Cette réduction (relation) n'agit pas seulement sur un terme mais aussi sur la fédération courante. En effet, un terme peut modifier la fédération en cours de calcul lors de la création de nouveaux modèles. La réduction doit connaître le chemin courant du terme évalué pour interpréter les chemins relatifs. Un

C1 : Contexte	$C ::=$	
C2 : *trou*		$\bullet$
C3 : *features : accès et mise à jour* [a]		$C\,.\,n \mid C\,.\,n \leftarrow c \mid v\,.\,n \leftarrow C$
C4 : *appels* [b]		$C\ c \mid v\ C$
C5 : *création de modèle*		$\texttt{new}\ id\ \texttt{in}\ C$
C6 : valeur de $\mathbb{V}$	$v ::=$	$\pi \mid ? \mid \texttt{str} \mid \texttt{()} \mid \texttt{"..."} \mid \texttt{fun}\ x \Rightarrow c \mid \texttt{fun}\ \_ \Rightarrow c$

a. le modèle cible est calculé avant la valeur d'affectation
b. la fonction est calculée avant le paramètre

FIGURE A.4 – Grammaire de description de contextes

terme c au chemin π_c dans la fédération f qui se réduit à c' et change la fédération en f' est désigné comme la règle SN. Si elle conduit à une erreur, elle est désignée par la règle SE.

$$SN : \pi_c \vdash f, c \longrightarrow f', c' \qquad SE : \pi_c \vdash f, c \longrightarrow \bot$$

Toutes les règles de réduction sont données sur la figure A.5. Pour définir les règles, nous nous appuyons sur la notion usuelle de contexte $C[\bullet]$ qui est un terme avec un *trou* [74] (un emplacement) unique. Tout terme peut être écrit $C[c]$ pour un contexte $C[\bullet]$ (parfois appelé contexte de réduction) et un terme c (parfois appelé *redex*). La dénotation $C[c]$ décrit l'emplacement de calcul courant où le terme actuellement calculé est c. La règle habituelle de réduction du contexte peut alors exprimer le calcul à n'importe quel endroit d'un terme, en décrivant la réduction de $C[c]$ en termes de $C[c']$ lorsque c se réduit à c'. La règle exacte est donnée dans la sémantique (règle CTX). Lorsqu'une erreur survient dans un contexte, elle entraîne une erreur (règle CTXERR).

$\mathbb{C}$ possède un sous-ensemble $\mathbb{V}$ dont les éléments sont des valeurs. Elles sont inertes du point de vue du calcul, c'est-à-dire qu'elles ne se réduisent pas. Elles sont définies par la grammaire de la règle C6 dans la figure A.4. Comme notre objectif est de fournir un langage de modélisation, ces valeurs couvrent à la fois le cas des données telles qu'une chaîne de caractères mais aussi des types tels que str. Tous les autres éléments de $\mathbb{C}$ peuvent se réduire (c'est-à-dire qu'ils produisent des calculs).

Les règles APP1, APP2 et APPERR expriment l'usuel appel de fonction, selon une approche d'appel par valeur, définie par la grammaire de la figure A.4. La fonction est d'abord réduite. Si la réduction conduit à une valeur, ce doit être une fonction ou alors cela conduit à une erreur. Si c'est une fonction, l'argument est réduit. Si la fonction nomme un argument (règle APP1), cette valeur remplace toute occurence de l'argument nommé au sein du corps de la fonction réduite. Si l'argument n'est pas nommé (règle APP2), la valeur de l'argument est écartée et seul le corps de la fonction est réduit.

L'accès à une *feature* d'un modèle nécessite tout d'abord qu'elle soit strictement appliquée à un chemin (règle FEATERR1). Ce chemin doit être valide relativement au chemin courant (règle FEATERR2). L'opérateur $\pi \oslash \pi_c$ est utilisé pour résoudre le chemin π relativement au chemin courant π_c. Cet opérateur est défini par les règles de figure A.6 (c). Une fois le chemin résolu, il doit correspondre à un modèle au sein de la fédération (règle FEATERR3). Ce calcul repose sur les règles de la figure A.6 (b) qui utilise le calcul de l'ensemble des modèles indexés par leurs chemins définis dans la figure A.6 (a). Enfin, le

$$\textsc{Ctx}\ \frac{\pi_c \vdash f, c \longrightarrow f', c'}{\pi_c \vdash f, C[c] \longrightarrow f', C[c']} \qquad \textsc{CtxErr}\ \frac{\pi_c \vdash f, c \longrightarrow \bot}{\pi_c \vdash f, C[c] \longrightarrow \bot}$$

$$\textsc{App1}\ \pi_c \vdash f, \lambda x.c\ v \longrightarrow f, [x \mapsto v]c \qquad \textsc{App2}\ \pi_c \vdash f, \lambda\_.c\ v \longrightarrow f, c$$

$$\textsc{AppErr}\ \frac{v \text{ is neither } \lambda x.c' \text{ nor } \lambda\_.c'}{\pi_c \vdash f, v\ c \longrightarrow \bot}$$

$$\textsc{Feat}\ \frac{\pi_c \oslash \pi \neq \bot \qquad \mathsf{model}(\pi_c \oslash \pi, f) = \_ \triangleright \langle F \mid \_ \rangle \qquad n \mapsto c \in F}{\pi_c \vdash f, \pi\,.\,n \longrightarrow f, \pi_c, c}$$

$$\textsc{FeatErr1}\ \frac{v \notin \Pi}{\pi_c \vdash f, v\,.\,n \longrightarrow \bot} \qquad \textsc{FeatErr2}\ \frac{\pi_c \oslash \pi = \bot}{\pi_c \vdash f, \pi\,.\,n \longrightarrow \bot}$$

$$\textsc{FeatErr3}\ \frac{\pi_c \oslash \pi \neq \bot \qquad \mathsf{model}(\pi_c \oslash \pi, f) = ?}{\pi_c \vdash f, \pi\,.\,n \longrightarrow \bot}$$

$$\textsc{FeatErr4}\ \frac{\pi_c \oslash \pi \neq \bot \qquad \mathsf{model}(\pi_c \oslash \pi, f) = \_ \triangleright \langle F \mid \_ \rangle \qquad n \notin dom(F)}{\pi_c \vdash f, \pi\,.\,n \longrightarrow \bot}$$

$$\textsc{SetFeat}\ \frac{\pi_c \oslash \pi \neq \bot \qquad \mathsf{model}(\pi_c \oslash \pi, f) \neq ? \qquad f' = \mathsf{upd}(f, \pi_c \oslash \pi, n, v)}{\pi_c \vdash f, \pi\,.\,n \leftarrow v \longrightarrow f', \pi_c, v}$$

$$\textsc{SetFeatErr1}\ \frac{v \notin \Pi}{\pi_c \vdash f, v\,.\,n \leftarrow v \longrightarrow \bot} \qquad \textsc{SetFeatErr2}\ \frac{\pi_c \oslash \pi = \bot}{\pi_c \vdash f, \pi\,.\,n \leftarrow v \longrightarrow \bot}$$

$$\textsc{SetFeatErr3}\ \frac{\pi_c \oslash \pi \neq \bot \qquad \mathsf{model}(\pi_c \oslash \pi, f) = ?}{\pi_c \vdash f, \pi\,.\,n \leftarrow v \longrightarrow \bot}$$

$$\textsc{New}\ \frac{\pi_c \oslash \pi \neq \bot \qquad \mathsf{model}(\pi_c \oslash \pi, f) = \_ \triangleright \langle \_ \mid M \rangle \qquad i \notin dom(M) \qquad f' = \mathsf{new}(f, \pi_c \oslash \pi, i)}{\pi_c \vdash f, \nu i\ \texttt{in}\ \pi \longrightarrow f', \pi_c, \pi/i}$$

$$\textsc{NewErr1}\ \frac{v \notin \Pi}{\pi_c \vdash f, \nu i\ \texttt{in}\ v \longrightarrow \bot} \qquad \textsc{NewErr2}\ \frac{\pi_c \oslash \pi = \bot}{\pi_c \vdash f, \nu i\ \texttt{in}\ \pi \longrightarrow \bot}$$

$$\textsc{NewErr3}\ \frac{\pi_c \oslash \pi \neq \bot \qquad \mathsf{model}(\pi_c \oslash \pi, f) = \_ \triangleright \langle \_ \mid M \rangle \qquad i \in dom(M)}{\pi_c \vdash f, \nu i\ \texttt{in}\ \pi \longrightarrow \bot}$$

FIGURE A.5 – Sémantique d'exécution de MF^2 : règles de réduction

$$\mathsf{models}: \mathbb{F} \to \wp(\Pi^A \times \mathbb{M})$$
$$\mathsf{models}(\langle \_ \,|\, M\rangle) \stackrel{\text{def}}{=} \bigcup_{m \in M} \{/\pi \mapsto m' \mid \pi \mapsto m' \in \mathsf{models}(m)\}$$
$$\mathsf{models}(\langle F \,|\, M\rangle) \stackrel{\text{def}}{=} \{/ \mapsto \langle F \,|\, M\rangle\} \cup \bigcup_{m \in M} \mathsf{submodels}(/, m)$$
$$\mathsf{models}: \mathbb{M} \to \wp(\Pi^S \times \mathbb{M}) \quad \text{où } \Pi^S \text{est l'ensemble des chemins relatifs de la forme } i_1/\ldots/i_n$$
$$\mathsf{models}(i \triangleright \langle F \,|\, M\rangle) \stackrel{\text{def}}{=} \{i \mapsto i \triangleright \langle F \,|\, M\rangle\} \cup \bigcup_{m \in M} \{i/\pi \mapsto m' \mid \pi \mapsto m' \in \mathsf{models}(m)\}$$

(a) Modèles d'une fédération

$$\mathsf{model}: \Pi^A \times \mathbb{F} \to \{?\} \cup \mathbb{F} \cup \mathbb{M}$$
$$\mathsf{model}(/, \langle F \,|\, M\rangle) \stackrel{\text{def}}{=} \langle F \,|\, M\rangle$$
$$\mathsf{model}(\pi, \langle F \,|\, M\rangle) \stackrel{\text{def}}{=} m \qquad \text{si } \pi \mapsto m \in \mathsf{models}(\langle F \,|\, M\rangle)$$
$$\mathsf{model}(\pi, \langle F \,|\, M\rangle) \stackrel{\text{def}}{=} ? \qquad \text{sinon}$$

(b) Résolution de chemin absolu au sein d'une fédération

$$\_ \oslash \_ : \Pi^A \times \Pi \to \{\bot\} \cup \Pi^A$$
$$\pi \oslash \cdot \stackrel{\text{def}}{=} \pi \qquad / \oslash \cdot\cdot/\_ \stackrel{\text{def}}{=} \bot$$
$$/ \oslash i_1/\ldots/i_n \stackrel{\text{def}}{=} /i_1/\ldots/i_n \qquad /i \oslash \cdot\cdot/\pi \stackrel{\text{def}}{=} / \oslash \pi$$
$$/i_1/\ldots/i_n \oslash i'_1/\ldots/i'_m \stackrel{\text{def}}{=} /i_1/\ldots/i_n/i'_1/\ldots/i'_m \qquad /i_1/\ldots/i_n \oslash \cdot\cdot/\pi \stackrel{\text{def}}{=} /i_1/\ldots/i_{n-1} \oslash \pi$$

(c) Résolution de chemin au sein d'une fédération relativement au chemin (absolu) courant

FIGURE A.6 – Résolution de chemin

modèle pointé par le chemin doit contenir la *feature* considérée (règle FEATERR4). Lorsque toutes ces conditions sont remplies, le contenu associé à l'élément est renvoyé (règle FEAT).

De manière similaire, l'affectation d'une valeur à une *feature* doit s'appliquer à un chemin (règle SETFEATERR1), ce chemin doit être valide relativement au chemin courant (règle SETFEATERR2), et le modèle référencé par ce chemin doit exister (règle SETFEATERR3). Lorsque toutes ces conditions sont remplies, la fédération est mise à jour avec l'affectation de la valeur spécifiée à la *feature* considérée (règle SETFEAT), à l'aide de l'opérateur upd défini figure A.7. Si la feature n'existe pas déjà, elle est créée.

$$\mathsf{upd}: \mathbb{F} \times \Pi^A \times \mathbb{N} \times \mathbb{V} \to \mathbb{F}$$
$$\mathsf{upd}(\langle F \,|\, M\rangle, /, n, v) \stackrel{\text{def}}{=} \langle \{n' \mapsto c' \mid n' \mapsto c' \in F \wedge n' \neq n\} \cup \{n \mapsto v\} \,|\, M\rangle$$
$$\mathsf{upd}(\langle F \,|\, M\rangle, /i, n, v) \stackrel{\text{def}}{=} \langle F \,|\, \mathsf{upd}(M, i, /, n, v)\rangle$$
$$\mathsf{upd}(\langle F \,|\, M\rangle, /i_1/\ldots/i_k, n, v) \stackrel{\text{def}}{=} \langle F \,|\, \{m \mid m \in M \wedge \mathsf{id}(m) \neq i_1\} \cup \{\mathsf{upd}(m', /i_2/\ldots/i_k, n, v) \mid m' \in M \wedge \mathsf{id}(m') = i\}\rangle$$

$$\mathsf{upd}: \mathbb{M} \times \Pi^A \times \mathbb{N} \times \mathbb{V} \to \mathbb{M}$$
$$\mathsf{upd}(i_0 \triangleright \langle F \,|\, M\rangle, /, n, v) \stackrel{\text{def}}{=} i_0 \triangleright \langle \{n' \mapsto c' \mid n' \mapsto c' \in F \wedge n' \neq n\} \cup \{n \mapsto v\} \,|\, M\rangle$$
$$\mathsf{upd}(i_0 \triangleright \langle F \,|\, M\rangle, /i, n, v) \stackrel{\text{def}}{=} i_0 \triangleright \langle F \,|\, \mathsf{upd}(M, i, /, n, v)\rangle$$
$$\mathsf{upd}(i_0 \triangleright \langle F \,|\, M\rangle, /i_1/\ldots/i_k, n, v) \stackrel{\text{def}}{=} i_0 \triangleright \langle F \,|\, \mathsf{upd}(M, i, /i_2/\ldots/i_k, n, v)\rangle$$

$$\mathsf{upd}: \wp(\mathbb{M}) \times \mathbb{I} \times \Pi^A \times \mathbb{N} \times \mathbb{V} \to \wp(\mathbb{M})$$
$$\mathsf{upd}(M, i, \pi, n, v) \stackrel{\text{def}}{=} \{m \mid m \in M \wedge \mathsf{id}(m) \neq i\} \cup \{\mathsf{upd}(m', \pi, n, v) \mid m' \in M \wedge \mathsf{id}(m') = i\}\rangle$$

FIGURE A.7 – Mise à jour de la valeur d'une *feature* dans une fédération

Comme pour la construction précédente la création de modèle doit s'appliquer à un chemin valide (règles NEWERR1 et NEWERR2). Il faut par ailleurs que l'identifiant i ne soit pas déjà utilisé dans le modèle-cible qui va accueillir le nouveau modèle (règle NEWERR3). Lorsque ces conditions sont remplies, la fédération est mise à jour avec l'ajout d'un nouveau modèle vide identifié par i au sein du modèle-cible, à l'aide de l'opérateur new défini dans la figure A.8.

$$
\begin{array}{l}
\mathsf{new}\colon \mathbb{F} \times \Pi^A \times \mathbb{I} \to \mathbb{F} \\
\mathsf{new}(\langle F \,|\, M\rangle, /, i) \stackrel{\text{def}}{=} \langle F \,|\, \{i \triangleright \langle\rangle\} \cup M\rangle \\
\mathsf{new}(\langle F \,|\, M\rangle, /i', i) \stackrel{\text{def}}{=} \langle F \,|\, \mathsf{new}(M, i', /, i)\rangle \\
\mathsf{new}(\langle F \,|\, M\rangle, /i_1/\ldots/i_k, i) \stackrel{\text{def}}{=} \langle F \,|\, \{m \mid m \in M \wedge \mathsf{id}(m) \neq i_1\} \\
\qquad\qquad\qquad\qquad \cup \{\mathsf{new}(m', /i_2/\ldots/i_k, i) \mid m' \in M \wedge \mathsf{id}(m') = i'\}\rangle \\
\\
\mathsf{new}\colon \mathbb{M} \times \Pi^A \times \mathbb{I} \to \mathbb{M} \\
\mathsf{new}(i_0 \triangleright \langle F \,|\, M\rangle, /, i) \stackrel{\text{def}}{=} i_0 \triangleright \langle F \,|\, \{i \triangleright \langle\rangle\} \cup M\rangle \\
\mathsf{new}(i_0 \triangleright \langle F \,|\, M\rangle, /i', i) \stackrel{\text{def}}{=} i_0 \triangleright \langle F \,|\, \mathsf{new}(M, i', /, i)\rangle \\
\mathsf{new}(i_0 \triangleright \langle F \,|\, M\rangle, /i_1/\ldots/i_k, i) \stackrel{\text{def}}{=} i_0 \triangleright \langle F \,|\, \mathsf{new}(M, i, /i_2/\ldots/i_k, i)\rangle \\
\\
\mathsf{new}\colon \wp(\mathbb{M}) \times \mathbb{I} \times \Pi^A \times \mathbb{I} \to \wp(\mathbb{M}) \\
\mathsf{new}(M, i', \pi, i) \stackrel{\text{def}}{=} \{m \mid m \in M \wedge \mathsf{id}(m) \neq i'\} \cup \{\mathsf{new}(m', \pi, i) \mid m' \in M \wedge \mathsf{id}(m') = i'\}\rangle
\end{array}
$$

FIGURE A.8 – Création d'un nouveau modèle dans une fédération

A.6 Conclusion

L'étude de la sémantique opérationnelle concerne au premier chef les langages exécutables, c'est-à-dire les langages qui sont construits explicitement avec l'intention de faire du calcul. Dans ces langages, toutes les constructions structurelles sont explicitement définies, car absolument nécessaires pour spécifier le calcul à exécuter. *A contrario*, les langages de modélisation sont conçus dans le but de servir de support à la pensée humaine. Ainsi par exemple, beaucoup de choses peuvent ne pas être définies (peut-être le seront-elles plus tard).

Dans ce contexte, et parce qu'on s'intéresse aux aspects calculs d'un langage de modélisation, le *framework* MF^2 se distingue des approches classiques (fonctionnelles) en ceci qu'il ajoute des aspects modélisation (notamment la manipulation de choses inconnues, des chemins qui peuvent ne pas être résolus, des valeurs qui sont des types, etc.). L'intérêt de MF^2 réside dans la description d'une sémantique opérationnelle qui décrit précisément le comportement.

À noter que ce *framework* est facile à étendre par des propriétés fonctionnelles classiques. L'ajout de nouvelles structures et d'opérateurs, par exemple, est trivial par l'extension de $\mathbb{C}$. Par exemple, si $\mathbb{C}$ contient un constructeur de paires $(,)$, C doit être étendu avec (C, c) et (c, C) tandis que (v_1, v_2) est une nouvelle valeur. Évidemment, il faut aussi étendre la sémantique en décrivant les nouvelles règles.

MF^2 n'a pas pour vocation à être réellement utilisé dans un contexte opérationnel. Il a plus été conçu comme un démonstrateur qui permet d'explorer les aspects formels de constructions de fédération (et notamment leur sémantique). L'implantation du prototype est seulement partielle au regard de toutes les constructions nécessaires à un cas d'uti-

lisation réel, et des fonctionnalités ne sont pas couvertes. Il manque par exemple des opérations d'instanciation qui permettent de créer un modèle dans un autre. Il manque également le support d'adaptateurs technologiques.

MF² fait également l'objet de travaux en cours sur le typage. La manipulation d'abstraction (des "choses") pas très bien définies dans un langage de modélisation renforce l'intérêt à gérer du typage (de modèle), et donc la capacité à pouvoir calculer et dire des choses sur les modèles produits (par exemple pour empêcher les erreurs). Une première approche propose l'ajout de types prescriptifs qui peuvent contraindre les évolutions d'un modèle, tandis qu'une autre approche consisterait à travailler avec des types inférés (le type d'un modèle serait par exemple défini par l'ensemble des *features* qu'il contient). L'intérêt du typage est lié au fait que les modèles définissent leur comportement et qu'il est intéressant d'explorer des mécanismes de détection d'erreurs et d'inconsistances liées à l'exécution de ces comportements sur des modèles (à la manière d'un compilateur, il deviendrait possible de calculer les comportements qui rendent un modèle inconsistant au regard d'un système de types).

Annexe B

Le langage FML : définitions formelles

Pour un manque de place évident dans ce manuscrit, nous n'avons présenté dans le chapitre 4 qu'un aperçu du langage FML au regard de nos objectifs de recherche. Nous nous proposons dans cette annexe de présenter plus en détail le langage au travers de définitions formelles. Nous détaillerons également certaines constructions du langage et de leur syntaxe textuelle. On se référera au rapport scientifique [97] pour plus de détails sur ces différents aspects.

B.1 Espace conceptuel

Nous désignons par $\mathbb{O}_{\text{FML}}$ l'ensemble des objets du cœur conceptuel FML (espace conceptuel), qui se définit comme une partition disjonctive de tous les objets de $\mathbb{O}_{\text{FML}}$ entre *types* et *instances*. Nous définissons $\mathcal{M}$ comme l'ensemble de tous les types (FlexoConcept et VirtualModel) et $\mathcal{I}$ comme l'ensemble de toutes les instances (FlexoConceptInstance et VirtualModelInstance).

$$\begin{cases} \mathbb{O}_{\text{FML}} = \mathcal{M} \cup \mathcal{I} \\ \mathcal{M} \cap \mathcal{I} = \emptyset \end{cases}$$

B.2 FlexoConcept

C'est le concept central du langage FML : il permet la conceptualisation d'un concept métier. Un FlexoConcept est très similaire à la notion de *classe*. Il définit un ensemble de propriété structurelles (FlexoProperties) et comportementales (FlexoBehaviours).

Un FlexoConcept :

— est identifié par un identifiant (*Uniform Resource Identifier* ou URI),
— peut être contenu dans un autre FlexoConcept, ou dans un VirtualModel
— peut contenir des FlexoConcepts,

— hérite d'un ensemble de FlexoConcept,
— définit des propriétés structurelles et comportementales, conformément à la sémantique du polymorphisme en orienté-object (OOP),
— peut être abstrait,
— peut être statique (en ce cas, il n'est pas déclaré au sein d'un conteneur, et ne permet donc pas de référence vers ce dernier),
— est un type du langage FML, et fait donc partie de son système de types.

Propriétés : $n \mapsto c_p$, $n \in \mathbb{N}$, $c_p \in \mathbb{P}_{\text{FML}}$ représente une FlexoProperty. Une propriété FML définit un accès direct à une donnée, ou bien une indirection ou encore le résultat d'un calcul (avec ou sans effet de bord) à une donnée typée. On se référera à la section B.11 pour plus de précisions sur la définition des propriétés.

Comportements : $n \mapsto c_b$, $n \in \mathbb{N}$, $c_b \in \mathbb{B}_{\text{FML}}$ représente un FlexoBehaviour. Cette notion est très similaire à la notion de méthode dans les langages objets. Un comportement FML s'apparente à un appel de procédure avec des arguments typés. Ces types forment la signature de la méthode, et c'est cette signature qui sera utilisée pour le calcul de la liaison dynamique. Un comportement FML peut également retourner une valeur dont le type est spécifié dans la déclaration du comportement. On se référera à la section B.12 pour plus de précisions sur la définition des comportements.

$$\begin{cases} \mathbb{C}_{\text{FML}} \subset \mathbb{C} \\ \mathbb{C}_{\text{FML}} = \mathbb{P}_{\text{FML}} \cup \mathbb{B}_{\text{FML}} \\ \mathbb{P}_{\text{FML}} \cap \mathbb{B}_{\text{FML}} = \emptyset \end{cases}$$

Héritage multiple : FML supporte l'héritage multiple et la programmation par traits : un FlexoConcept peut hériter d'un ensemble de FlexoConcept. Cela implique des vérifications de contraintes sur les superconcepts afin d'éviter une hiérarchie de concepts inconsistente.

Gestion de la contenance : FML supporte nativement et implicitement la notion de conteneur pour tout FlexoConcept. A moins d'être déclaré comme `static`, un FlexoConcept est toujours déclaré dans un conteneur unique, et reflète l'accès à son conteneur dans son contexte. Cela implique également certaines vérifications de contraintes sur les superconcepts afin d'éviter certaines inconsistences, étant donné que toutes les clauses d'héritage doivent être compatibles avec les clauses implicites de déclaration de contenance.

Le listing B.1 illustre la définition de FlexoConcept sur un exemple de code FML très simple présentant cinq concepts :

— `Person` est un FlexoConcept qui définit le concept de "personne" (avec ses attributs) ;
— `Address` représente une adresse avec trois attributs ; ce FlexoConcept est déclaré au sein du FlexoConcept `Person`, donc implicitement considéré comme conteneur ;
— `Employee` spécialise `Person` ;
— `Student` spécialise `Person` ;
— `PhDStudent` spécialise à la fois `Employee` et `Student` ;
— Comme on le verra plus loin, `MyModel` est le VirtualModel qui déclare ces cinq concepts, et qui sera considéré comme conteneur de `Person`, `Employee`, `Student` et `PhDStudent`.

```
model MyModel {
  concept Person {
    String firstName;
    String lastName;
    Address address;
    ...
    concept Address {
       String streetName;
       String city;
       int zipCode;
    }
  }
  concept Employee extends Person {
    ...
  }
  concept Student extends Person {
    ...
  }
  concept PhDStudent extends Student,Employee {
    ...
  }
}
```

FIGURE B.1 – Extrait de code FML présentant des définitions de FlexoConcept

Notations formelles

Formellement, et pour rappeler la notation MF², un FlexoConcept fc avec le nom local $i \in \mathbb{I}$, héritant des superconcepts $fc_1 ... fc_m$ et qui contient les FlexoConcepts $fc_{e_1} ... fc_{e_k}$ se définit ainsi :

$$fc ::= i[: fc_1 ... : fc_m] \triangleright \langle n_1 \mapsto c_1, \ ... \ , n_n \mapsto c_n \mid fc_{e_1}, ..., fc_{e_k} \rangle$$
$$fc_i \in \mathcal{M}, fc_{e_i} \in \mathcal{M}, n_i \in \mathbb{N}, c_i \in \mathbb{C}_{\text{FML}}$$

La contenance d'un FlexoConcept fc_e dans un conteneur FlexoConcept fc découle de la définition de son conteneur :

$$fc ::= \_ \triangleright \langle \_ \mid ..., fc_e, ... \rangle$$
$$fc, fc_e \in \mathcal{M}$$

Une fonction id extrait l'identifiant d'un FlexoConcept et deux fonctions name et content extraient respectivement le nom et le contenu d'une fonctionnalité ou *feature* (propriété ou comportement), tandis que feat_set fait référence à l'ensemble des fonctionnalités. types_closure représente la fermeture de types d'un FlexoConcept (tous les types étant directement ou indirectement référencés). Les supertypes d'un FlexoConcept sont définis comme l'ensemble transitif de ses types directs et indirects. La fonction t_feat_set (ensemble de fonctionnalités transitives) complète la fonction feat_set en incluant les fonctionnalités héritées.

id	Identifiant d'une fonctionnalité	$\mathsf{id}(i \triangleright \langle ... \rangle) \stackrel{\text{def}}{=} i$
name	Nom d'une fonctionnalité	$\mathsf{name}(n \mapsto c) \stackrel{\text{def}}{=} n$
content	Contenu d'une fonctionnalité	$\mathsf{content}(n \mapsto c) \stackrel{\text{def}}{=} c$
types_closure	Fermeture de types	$\mathsf{types\_closure}(fc) \stackrel{\text{def}}{=} \mathsf{closure}(fc) \cap \mathcal{M}$
supertypes	Supertypes directs	$\mathsf{supertypes}(\_ : fc_1 ... : fc_m \triangleright \langle ... \rangle) \stackrel{\text{def}}{=} \cup_{i \in [1..m]} \{fc_i\}$
t_supertypes	Supertypes transitifs	$\mathsf{t\_supertypes}(fc) \stackrel{\text{def}}{=} \mathsf{supertypes}(fc) \cup \cup_{t \in \mathsf{supertypes}(fc)} \{\mathsf{t\_supertypes}(t)\}$
contained	Concepts contenus	$\mathsf{contained}(\_ \triangleright \langle \_ \mid fc_1, ..., fc_k \rangle) \stackrel{\text{def}}{=} \cup_{i \in [1..k]} \{fc_i\}$
container	Conteneur	$\mathsf{container}(fc) = fc_p \stackrel{\text{def}}{=} \exists fc_p \in \mathcal{M}, fc \in \mathsf{contained}(fc_p)$
feat_set	Ensemble des fonctionnalités	$\mathsf{feat\_set}(\_ \triangleright \langle n_1 \mapsto c_1, \ ... \ , n_n \mapsto c_n \rangle) \stackrel{\text{def}}{=} \cup_{i \in [1..m]} \{n_i \mapsto c_i\}$
t_feat_set	Fonctionnalités (transitives)	$\mathsf{t\_feat\_set}(fc) \stackrel{\text{def}}{=} \mathsf{feat\_set}(fc) \cup \cup_{t \in \mathsf{supertypes}(fc)} \{\mathsf{t\_feat\_set}(t)\}$

Relation de sous-typage : on définit la relation subtype entre deux FlexoConcepts. Cette relation est réflexive et transitive et se définit comme suit :

$$fc_1 <: fc_2 \stackrel{\text{def}}{=} \forall f_1 \in \mathsf{t\_feat\_set}(fc_1), \exists f_2 \in \mathsf{t\_feat\_set}(fc_2), \ \mathsf{name}(f_1) \stackrel{\mathbb{N}}{=} \mathsf{name}(f_2) \wedge (\mathsf{content}(f_1) <: \mathsf{content}(f_2) \vee \mathsf{content}(f_1) = \mathsf{content}(f_2))$$

On montre trivialement que : $fc_2 \in \mathsf{supertypes}(fc_1) \Rightarrow fc_1 <: fc_2$.

FlexoConcept abstrait : un FlexoConcept peut être déclaré abstract. Cette qualité permet au concept de définir des features abstraites ($\mathsf{abstract}(n \mapsto c) \implies \mathsf{content}(n \mapsto c) = \emptyset$). Un concept abstrait ne peut être instancié directement.

$$\overline{\mathsf{abstract}(fc)} \implies \forall f \in \mathsf{t\_feat\_set}(fc), \ \mathsf{content}(f) \neq \emptyset$$

Identification (*lookup*) d'un FlexoConcept : un FlexoConcept est identifié par un nom local, qui permet de construire une Uniform Resource Identifier (URI), concrétisée par une chaîne de caractères (bien formée) [1]. Pour reprendre la notation de MF^2 et dans le contexte FML, l'ensemble $\mathbb{I}$ se définit comme l'ensemble de toutes les chaînes de caractères bien formées qui représentent une *URI* qui est résolvable. Nous définissons ainsi la fonction lookup qui permet de retrouver, dans un contexte donné (ici une unité de compilation FML), un FlexoConcept à partir de son *URI*.

1. Dans la pratique, on utilise les URL qui permettent d'identifier une ressource sur le web.

$$\mathsf{lookup}(i) = fc \overset{\text{def}}{=} \exists fc \in \mathcal{M}, i \in \mathbb{I}, fc ::= i \triangleright \langle ... \rangle$$

L'unicité de cette *URI* est garantie par cette assertion :

$$\forall i_1, i_2 \in \mathbb{I}, \mathsf{lookup}(i_1) \overset{\mathcal{M}}{=} \mathsf{lookup}(i_2) \implies i_1 \overset{\mathbb{I}}{=} i_2$$

FlexoConcept bien formé : le prédicat wf (*well formed term*) permet de caractériser les FlexoConcept légaux : toutes les fonctionnalités doivent avoir des noms distincts, les fonctionnalités redéfinies doivent être consistentes en terme de typage avec les déclarations des types parents, le graphe des supertypes doit être acyclique et les clause de contenance consistentes avec l'arbre d'héritage :

$$\begin{aligned}\mathsf{wf}(fc) \overset{\text{def}}{=} & \forall f_1, f_2 \in \mathsf{feat\_set}(fc),\ \mathsf{name}(f_1) \overset{\mathbb{N}}{=} \mathsf{name}(f_2) \Rightarrow f_1 = f_2 \\ & \bigwedge \forall f_1 \in \mathsf{feat\_set}(fc), \forall f_2 \in \mathsf{t\_feat\_set}(fc) \setminus \mathsf{feat\_set}(fc) \\ & \quad \mathsf{name}(f_1) \overset{\mathbb{N}}{=} \mathsf{name}(f_2) \Rightarrow \mathsf{content}(f_1) <: \mathsf{content}(f_2) \\ & \bigwedge \forall t \in \mathsf{supertypes}(fc), \mathsf{wf}(t) \\ & \bigwedge fc \notin \mathsf{t\_supertypes}(fc) \\ & \bigwedge \forall t \in \mathsf{supertypes}(fc), \mathsf{container}(fc) <: \mathsf{container}(t)\end{aligned}$$

B.3 VirtualModel

Il permet la structuration d'une fédération. Un VirtualModel contient des FlexoConcepts et est également (héritage) un FlexoConcept. En tant que tel, un VirtualModel définit des propriétés et des comportements (fonctionnalités). L'ensemble de tous les VirtualModel est noté $\mathcal{M}_{\mathsf{VM}} \subset \mathcal{M}$.

Un VirtualModel vm avec le nom local $i \in \mathbb{I}$ qui hérite des supertypes $vm_1 ... vm_m$ et qui contient les FlexoConcepts [2] $fc_1 ... fc_k$ se définit formellement ainsi :

$$\begin{gathered}vm ::= i\ [: vm_1 ... : vm_m] \triangleright\ \langle n_1 \mapsto c_1,\ ...\ , n_n \mapsto c_n \mid fc_1, ..., fc_k \rangle \\ vm_i \in \mathcal{M}_{\mathsf{VM}},\ n_i \in \mathbb{N},\ c_i \in \mathbb{C}_{\mathsf{FML}},\ fc_i \in \mathcal{M}\end{gathered}$$

Les fonctions et les prédicats définis pour un FlexoConcept restent valides pour un VirtualModel avec les spécialisations suivantes :

$$\begin{gathered}\forall vm \in \mathcal{M}_{\mathsf{VM}}, \mathsf{container}(vm) \in \mathcal{M}_{\mathsf{VM}} \\ \forall vm \in \mathcal{M}_{\mathsf{VM}}, \forall t \in \mathsf{supertypes}(vm), t \in \mathcal{M}_{\mathsf{VM}}\end{gathered}$$

Le prédicat wf (*well formed term*) pour les VirtualModels se complète avec la gestion de la covariance pour les FlexoConcepts contenus :

2. Les concepts contenus peuvent être d'autres VirtualModels (qui sont aussi des FlexoConcepts).

$$
\begin{cases}
\mathsf{wf}(vm) \stackrel{\text{def}}{=} \forall f_1, f_2 \in \mathsf{feat\_set}(vm), \\
\qquad \mathsf{name}(f_1) \stackrel{\mathbb{N}}{=} \mathsf{name}(f_2) \Rightarrow f_1 = f_2 \\
\qquad \bigwedge \forall f_1 \in \mathsf{feat\_set}(vm), \forall f_2 \in \mathsf{t\_feat\_set}(vm) \setminus \mathsf{feat\_set}(vm) \\
\qquad \mathsf{name}(f_1) \stackrel{\mathbb{N}}{=} \mathsf{name}(f_2) \Rightarrow \mathsf{content}(f_1) <: \mathsf{content}(f_2) \\
\qquad \bigwedge \forall fc_1, fc_2 \in \mathsf{contained}(vm), \\
\qquad \mathsf{id}(fc_1) \stackrel{\mathbb{I}}{=} \mathsf{id}(fc_2) \Rightarrow fc_1 = fc_2 \\
\qquad \bigwedge \forall fc \in \mathsf{contained}(vm), \mathsf{wf}(fc) \\
\qquad \bigwedge \forall t \in \mathsf{supertypes}(vm), \mathsf{wf}(t) \\
\qquad \bigwedge vm \notin \mathsf{t\_supertypes}(vm) \\
\qquad \bigwedge \forall t \in \mathsf{supertypes}(fc), \mathsf{container}(vm) <: \mathsf{container}(t) \\
\qquad \bigwedge \forall fc \in \mathsf{contained}(vm), \mathsf{wf}(fc, vm) \\
\mathsf{wf}(fc, vm) \stackrel{\text{def}}{=} \forall t \in \mathsf{supertypes}(fc), vm <: \mathsf{container}(t)
\end{cases}
$$

FMLCompilationUnit : le code source de FML est organisé en unités de compilation (FMLCompilationUnit). Une unité de compilation contient une unique définition de VirtualModel et porte le nom de ce VirtualModel. La définition d'un VirtualModel nommé `MyModel`, par exemple, apparaît dans une ressource nommée `MyModel.fml`. Une unité de compilation FML est typiquement un fichier avec une extension `.fml`. Par souci de concision, nous appellerons simplement "fichier" une unité de compilation.

Le listing B.2 `MyModel.fml` montre un exemple d'une telle unité de compilation définissant un VirtualModel `MyModel`, contenant un FlexoConcept FlexoConcept `MyConcept`. L'*URI* du VirtualModel `MyModel` est explicitement définie (ligne 5) avec la définition de métadonnées (dont la syntaxe est très similaire aux annotations Java). Notez également que la ligne 2 de la déclaration import explicite une dépendance envers un VirtualModel `AnOtherModel` requis par le lien d'héritage entre `MyModel` et `AnOtherModel` (ligne 6). La gestion des dépendances et des imports sera détaillée plus longuement dans la section B.7.

```
1  // Import FML model addressed by specified URI
2  import ["http://www.openflexo.org/test/AnOtherModel.fml"] as AnOtherModel;
3
4  // This is the main declaration with explicit URI declaration
5  @URI("http://www.openflexo.org/test/MyModel.fml")
6  model MyModel extends AnOtherModel {
7      ...
8      concept MyConcept {
9          ...
10     }
11     ...
12 }
```

FIGURE B.2 – Extrait de code FML montrant la définition d'une unité de compilation

B.4 Métamodèle FML@runtime

On se réfèrera à la partie droite de la figure 4.5 pour une illustration graphique de ce métamodèle simplifié.

FlexoConceptInstance : c'est le concept qui représente une instance (ontologique) d'un FlexoConcept (son type), et qui en tant que tel, définit une valeur pour toutes les propriétés structurelles (plus précisemment, pour toutes celles qui ne sont ni une indirection, ni le résultat d'un calcul). On définit $\mathbb{V}$ comme sous-ensemble de $\mathbb{C}$ qui représente l'ensemble des valeurs [3].

$$fci ::= i : fc \triangleright \langle n_1 \mapsto v_1 \ ... \ n_n \mapsto v_n \rangle$$
$$fc \in \mathcal{M}, n_i \in \mathbb{N}, v_i \in \mathbb{V}$$

L'ensemble de toutes les FlexoConceptInstances est appelé $\mathcal{I}$. Les fonctions name et value extraient respectivement le nom et la valeur d'une fonctionnalité, tandis que values_set référence l'ensemble des valeurs. Enfin, type fait référence au type (unique) d'une instance.

name	Nom d'une fonctionnalité	$\mathsf{name}(n \mapsto v) \stackrel{\text{def}}{=} n$
value	Valeur d'une fonctionnalité	$\mathsf{value}(n \mapsto v) \stackrel{\text{def}}{=} v$
values_set	Ensemble des valeurs	$\mathsf{values\_set}(\_ \triangleright \langle n_1 \mapsto v_1, \ ... \ , n_m \mapsto v_m \rangle) \stackrel{\text{def}}{=} \cup_{i \in [1..m]} \{n_i \mapsto v_i\}$
type	Type	$\mathsf{type}(i : fc \triangleright \langle ... \rangle) \stackrel{\text{def}}{=} fc$

Relation de typage : la relation typeof (conforme à) entre une FlexoConceptInstance et son type FlexoConcept est définie comme suit :

$$fci ::= \_ : fc \triangleright \langle \_ \rangle \Rightarrow fci : fc$$

avec :

$$fci : fc \stackrel{\text{def}}{=} \forall f \in \mathsf{t\_feat\_set}(fc), \mathsf{isRequired(f)} \implies$$
$$\exists v \in \mathsf{values\_set}(fci), (\mathsf{name}(f) \stackrel{\mathbb{N}}{=} \mathsf{name}(v)) \wedge (\mathsf{value}(v) : \mathsf{content}(f))$$

Gestion de la contenance : de la même manière qu'un FlexoConcept peut contenir d'autres FlexoConcepts, FlexoConceptInstance peut contenir d'autres FlexoConceptInstances :

$$fci ::= i : fc \triangleright \langle n_1 \mapsto v_1 \ ... \ n_n \mapsto v_n \mid fci_1, ..., fci_k \rangle$$
$$fc \in \mathcal{M}, n_i \in \mathbb{N}, v_i \in \mathbb{V}, fci \in \mathcal{I}$$

Les FlexoConceptInstances contenues doivent avoir leurs types conformes à la définition de container.

3. Formellement, ce sont les termes non réductibles.

Nous définissons la fonction container qui fait référence au conteneur (unique) d'une FlexoConceptInstance et la fonction contained qui fait référence à l'ensemble des FlexoConceptInstances contenues :

contained	*contained set*	$\mathsf{contained}(\_ \triangleright \langle \_ \mid fci_1, ..., fci_k \rangle) \stackrel{\text{def}}{=} \cup_{i \in [1..k]} \{fci_i\}$
container	*container*	$\mathsf{container}(fci) = ct \stackrel{\text{def}}{=} \exists ct \in \mathcal{I}, fci \in \mathsf{contained}(ct)$

FlexoConceptInstances bien formées : le prédicat wf permet de caractériser les FlexoConceptInstances légales (le type doit être identifié et être valide, bien formé, et la relation typeof (conforme à) doit être respectée) :

$$\begin{aligned}\mathsf{wf}(fci) \stackrel{\text{def}}{=}\ & \exists fc \in \mathcal{M}, fci : fc \wedge \mathsf{wf}(fc) \\ & \bigwedge \forall fci_{child} \in \mathsf{contained}(fci), \mathsf{wf}(fc_{child}) \\ & \bigwedge \forall fci_{child} \in \mathsf{contained}(fci), \exists t \in \mathsf{contained}(fc), fci_{child} : t\end{aligned} .$$

La même problématique de lookup que celle soulevée dans la section B.2 s'applique ici : d'un point de vue syntaxique, le FlexoConcept (type) d'une FlexoConceptInstance nécessite son identification (*lookup*) à partir de son *URI* elle-même calculée à partir de son nom local.

VirtualModelInstance : c'est naturellement le concept qui représente une instance (ontologique) d'un VirtualModel (son type). Le sous-ensemble de toutes les VirtualModelInstances est appelé $\mathcal{I}_{\mathsf{VM}} \subset \mathcal{I}$.

$$vmi ::= i : vm \triangleright \langle n_1 \mapsto v_1 \ ... \ n_n \mapsto v_n \mid fci_1, ..., fci_k \rangle$$
$$vm \in \mathcal{M}_{\mathsf{VM}},\ n_i \in \mathbb{N},\ v_i \in \mathbb{V},\ fci_i \in \mathcal{I}$$

Une VirtualModelInstance spécialise (linguistiquement) une FlexoConceptInstance (héritage), et donc la relation de typage ainsi que les fonctions et les prédicats définis pour une FlexoConceptInstance sont également valides pour une VirtualModelInstance avec les spécialisations suivantes :

$$\forall vmi \in \mathcal{I}_{\mathsf{VM}}, \mathsf{container}(vmi) \in \mathcal{I}_{\mathsf{VM}}$$

$$\begin{aligned}\mathsf{wf}(vmi) \stackrel{\text{def}}{=}\ & \exists vm \in \mathcal{M}_{\mathsf{VM}}, vmi : vm \wedge \mathsf{wf}(vm) \\ & \bigwedge \forall fci_{child} \in \mathsf{contained}(vmi), \mathsf{wf}(fc_{child}) \\ & \bigwedge \forall fci_{child} \in \mathsf{contained}(vmi), \exists t \in \mathsf{contained}(vm), fci_{child} : t\end{aligned} .$$

B.5 Gestion des objets externes et des technologies tierces

Avant de nous concentrer sur le système de types et sur les propriétés et les comportements des FlexoConcepts, nous devons présenter la manière dont nous traitons les objets externes.

Notion de Resource : la fédération de modèles implique la gestion de multiples sources d'information considérées comme des modèles. La notion de Resource est une abstraction utilisée pour représenter l'accès à un "modèle" [4], comme le montre la figure 4.13.

Une ressource (Resource) :

— est identifiée par un identifiant (*URI*) représenté par une chaîne de caractères qui doit être unique,
— a un nom local (représenté par une chaîne de caractères),
— fournit les accès à la donnée (ResourceData),
— a son propre cycle de vie (peut être loaded ou unloaded),
— déclare (explicitement ou implicitement) ses dépendances à d'autres Resources (les dépendances cycliques sont autorisées).

ResourceData est une abstraction générique qui représente le contenu d'un modèle externe et donc une donnée dans laquelle se trouvent les MirrorObjects. MirrorObject représente un objet (externe) accessible depuis l'espace conceptuel (mais qui ne se trouve pas dedans). Un MirrorObject vise à refléter une "donnée externe" (un objet, un élément de modélisation, un élément d'information quel qu'il soit) qui se trouve dans l'espace d'information (plus précisemment au sein d'un espace technologique). Un MirrorObject peut être considéré comme un *proxy* et une représentation locale d'un objet spécifique externe.

Considérant une Resource $\mathcal{R}$ nous définissons ces fonctions :

uri	*URI*	uri($\mathcal{R}$) : *Universal resource identifier* (`String`)
name	Nom local	name($\mathcal{R}$) : nom local de la ressource (`String`)
data	Donnée	data($\mathcal{R}$) : contenu de la ressource (ResourceData)

Notion de ResourceCenter : l'abstraction ResourceCenter ($\mathcal{RC}$) complète le mécanisme de gestion des ressources en fournissant des instances de Resource au moteur de fédération. D'un point de vue technique, un ResourceCenter fournit une couche d'accès à des Ressource à partir de différents supports (système de fichiers, réseau, bases de données, serveurs, etc.). Un ResourceCenter est organisé en dossiers (structure arborescente). Un ResourceCenter est également identifié par un identifiant (une *URI* unique, représentée sous la forme d'une chaîne de caractères). Pour une meilleure portabilité des ressources, l'*URI* des ressources contenues peut être calculée à partir de l'*URI* de base du ResourceCenter et de la structure des dossiers. Nous appelons $\mathbb{RC}$ l'ensemble de tous les ResourceCenters.

$$\mathcal{RC} ::= uri \triangleright \{\mathcal{R}_1 ... \mathcal{R}_n\}$$
$$\mathcal{RC} \subset \mathbb{RC}$$

Considérant un ResourceCenter $\mathcal{RC} ::= uri \triangleright \{\mathcal{R}_1 ... \mathcal{R}_n\}$ nous définissons ces fonctions :

uri	*URI*	$\text{uri}(uri \triangleright \{\mathcal{R}_1 ... \mathcal{R}_n\}) \stackrel{\text{def}}{=} uri$
lookup	Nom local	$\text{lookup}(uri) = \mathcal{R} \stackrel{\text{def}}{=} (\exists \mathcal{R} \in \mathcal{RC}, \text{uri}(\mathcal{R}) = uri)$

4. L'idée est celle d'une indirection vers la donnée proprement dite, via une abstraction dont la préoccupation liée est celle des I/O (entrées/sorties).

L'unicité de l'*URI* d'une Resource peut s'exprimer ainsi :

$$\forall \mathcal{R}_1, \mathcal{R}_2 \in \mathcal{RC}, \mathsf{uri}(\mathcal{R}_1) = \mathsf{uri}(\mathcal{R}_2) \implies \mathcal{R}_1 = \mathcal{R}_2$$

Notion d'*adaptateur technologique* : l'intégration d'une technologie spécifique dans le cœur de la fédération de modèles est rendue possible par l'utilisation d'un TechnologyAdapter $\mathcal{TA}$ dédié à cette technologie. L'objectif principal d'un TechnologyAdapter est de fournir des types et des constructions spécifiques à une technologie, pour pouvoir les utiliser au sein de l'espace conceptuel FML. Ceci permet aux éléments de modélisation qui se situent dans l'espace d'information d'être réfléchis (référencés) en tant que MirrorObject et d'être manipulés vers/depuis l'espace conceptuel [5].

Un TechnologyAdapter définit un ensemble de ModelSlotKind (des types de ModelSlot), qui représentent les différentes manières possibles fournies par le TechnologyAdapter pour exposer les données d'une ressource d'une technologie donnée. Ceci correspond aux différentes façons d'interpréter une telle donnée. Un TechnologyAdapter identifié par id et fournissant n ModelSlotKind $\mathcal{MS}_1$ à $\mathcal{MS}_n$ est défini comme suit :

$$\mathcal{TA} ::= id \triangleright \{\mathcal{MS}_1 ... \mathcal{MS}_n\}$$

La combinaison d'une ressource $\mathcal{R}$ et d'un ModelSlotKind $\mathcal{MS}$ définit $\mathbb{O}_{\mathcal{MS}}(\mathcal{R})$, l'ensemble de tous les MirrorObjects mo_i accessibles dans la ressource $\mathcal{R}$ à partir de l'interprétation proposée par $\mathcal{MS}$.

$$\mathbb{O}_{\mathcal{MS}}(\mathcal{R}) ::= uri \triangleright \{mo_1 ... mo_n\}, mo_i \in \mathbb{O}$$

La table 4.1 montre comment cette couche réflexive spécifique à une technologie est fournie au cœur FML en apportant des types et des primitives structurelles et comportementales au langage. Nous présentons ici un TechnologyAdapter Java exposant deux ModelSlotKind différents (`RepositoryModelSlot` et `FileModelSlot`), fournissant une interprétation de code source Java à deux niveaux de granularité différents. Ce TechnologyAdapter fournit également certains TechnologyTypes (des types spécifiques à cette technologie). La table 4.2 détaille ces deux ModelSlotKinds et présente les FlexoRoleKinds, EditionActionKinds et FlexoBehaviourKinds qui sont proposés pour `RepositoryModelSlot` et `FileModelSlot`.

La figure 4.12 présente comment un TechnologyAdapter donné étend le métamodèle FML de base. Un TechnologyAdapter définit en premier lieu un ensemble de TechnologyTypes (ce sont les types manipulés par le TechnologyAdapter et incorporés au système de type FML). On nomme $\mathbb{T}_{\mathcal{TA}}$ cet ensemble de types.

Un TechnologyAdapter définit également un ensemble de ModelSlotKind. Chaque ModelSlotKind définit un ensemble de primitives (structurelles et comportementales) qui sont mises à la disposition du noyau FML :

— un ensemble de FlexoRoleKind de tous les types de FlexoRole disponibles dans le contexte de ce ModelSlotKind,

— un ensemble de FlexoBehaviourKind de tous les types de FlexoBehaviour disponibles fournis par cette technologie (comportements spécifiques auxquels on peut se "brancher") dans le contexte de ce ModelSlotKind,

5. Comme on le verra plus tard, l'idée n'est pas seulement de fournir un accès à la donnée en lecture, mais aussi de pouvoir la modifier.

— un ensemble de EditionActionKind $\mathbb{EA} \subset \mathbb{C}$, qui sont tous les types de EditionAction fournis par cette technologie (primitives comportementales spécifiques utilisables) dans le contexte de ce ModelSlotKind.

Définition formelle de l'*espace d'information* : on définit donc l'espace d'information (*Information Space*) comme l'ensemble de tous les MirrorObjects accessibles, étant donnés un ensemble de TechnologyAdapters $\{\mathcal{TA}_1...\mathcal{TA}_m\}$ et un ensemble de ResourceCenters $\{\mathcal{RC}_1...\mathcal{RC}_n\}$:

$$\mathcal{IS} ::= \{\mathcal{RC}_1...\mathcal{RC}_n\} \times \{\mathcal{TA}_1...\mathcal{TA}_m\}$$

Un espace d'information $\mathcal{IS}$ représente l'ensemble de tous les éléments de modélisation accessibles dans n'importe quel Ressources de n'importe quel ResourceCenter et interprétables par n'importe quel ModelSlotKind composant cet espace d'information. Cet espace peut être considéré comme une représentation matricielle de toutes les Ressources et de tous les ModelSlotKind.

$$\mathcal{IS} = \bigcup_{(\mathcal{MS},\mathcal{R})\in\mathcal{IS}} \{\ \mathbb{O}_{\mathcal{MS}}(\mathcal{R})\}$$

L'unicité de l'*URI* d'une resource doit être garanti dans un espace d'information $\mathcal{IS}$ donné :

$$\forall \mathcal{R}_1, \mathcal{R}_2 \in \mathcal{IS}, \mathsf{uri}(\mathcal{R}_1) = \mathsf{uri}(\mathcal{R}_2) \implies \mathcal{R}_1 = \mathcal{R}_2$$

On définit enfin $\mathbb{T}_{\mathcal{TA}}$ comme l'ensemble de tous les types spécifiques exposés à l'espace conceptuel par un TechnologyAdapter $\mathcal{TA}$ donné.

B.6 Notion de fédération en FML

De tout ce qui précède, et pour faire le lien avec d'une part l'approche *fédération de modèles* présentée dans la section 3.3, et d'autre part la proposition présentée dans le *framework* MF², on définit formellement une fédération $\mathcal{F}$ comme la donnée d'une VirtualModelInstance dans le contexte d'un espace d'information $\mathcal{IS}$:

$$\mathcal{F} ::= (vmi, \mathcal{IS})\ ,\ vmi \in \mathcal{I}_{\mathsf{VM}}$$

La problématique du *lookup* de tous les types et de toutes les dépendances dont il est fait mention dans la VirtualModelInstance vmi doit être résolue par la donnée de l'espace d'information $\mathcal{IS}$. Ce dernier est conçu pour être accessible de l'environnement d'exécution de la fédération. Il peut être partagé ou répliqué, et constitue donc l'élément-clef de la portabilité de la fédération. Sa mise à disposition et le respect de son intégrité font partie des préoccupations d'une implémentation d'un interpréteur FML.

A noter ici que la fédération telle qu'elle est présentée ici juxtapose la notion d'instance de fédération et de "type" de fédération. Le "type" de la fédération est ici trivialement défini par le couple $(\mathsf{type}(vmi), \mathcal{IS})$.

B.7 Unité de compilation FML

Comme indiqué dans la section B.3 page 195, le code FML est organisé en unités de compilation. Chaque VirtualModel est défini dans une FMLCompilationUnit unique. Une FMLCompilationUnit décrit un VirtualModel unique mais définit également un contexte (avec des espaces de nom et des constantes) et une portée où toutes les dépendances (adaptateurs technologiques, types, primitives) peuvent être résolues.

Le listing B.3 présente un exemple d'unité de compilation définissant un VirtualModel `MyModel` qui hérite de deux VirtualModel (définis dans d'autres ressources FML), et qui utilise la technologie OWL pour définir un lien vers une ontologie. En outre, cette unité de compilation FML définit un type spécifique OWL (un individual dont le type est précisé).

Gestion des espaces de noms : namespace fournit un espace de noms (une chaîne de caractère) qui permet d'identifier et de regrouper un ensemble logique d'éléments au sein d'une CompilationUnit. Deux namespaces `LOCAL` et `NS` (lignes 1 et 2) sont par exemple définis dans cet exemple. Ces raccourcis sont ensuite utilisés et composés pour désigner des ressources externes en précisant les *URI* de VirtualModel `MyModel`. Ces espaces de nom concourent à la portabilité du langage.

Gestion des adaptateurs technologiques : pour pouvoir utiliser une primitive propre à un *adaptateur technologique* dans la définition d'un VirtualModel, il faut déclarer l'utilisation d'un ModelSlotKind au moyen de la déclaration use. L'extrait de code FML présenté ci-dessus montre aux lignes 4 et 5 les déclarations de deux ModelSlotKind nommés `FMLRT` et `OWL` (ontologie OWL).

Gestion des imports (dépendances) : la définition d'une CompilationUnit est complétée par la déclaration de certains imports, requis par la définition du VirtualModel. Ces importations peuvent être des types, des ressources externes ou des objets externes contenus dans des ressources externes. L'extrait de code FML présenté ci-dessus montre quelques exemples de déclarations d'import :

— Le type Java `java.lang.String` est importé ligne 8 avec le nom local `JavaString` (valide dans la portée de cette unité de compilation).

— Le type Java `java.util.List` est importé ligne 10 avec son nom par défaut (`List`).

— Le type `org.openflexo.technologyadapter.owl.model.OWLOntology` est importé li-gne 11 avec son nom par défaut (`OWLOntology`).

— Le VirtualModel `AnOtherModel` est importé ligne 14 avec le nom local `AnOtherModel`. Ce VirtualModel est désigné par son *URI*, calculée à partir du namespace `NS`.

— Le VirtualModel `AThirdModel` est importé ligne 16 avec son nom par défaut.

— L'ontologie désignée par l'URI "http ://www.openflexo.org/test/animals" est importée ligne 21 avec le nom local "ANIMALS". Son type est inféré du lookup de la resource qui est obligatoire.

— La classe OWL identifiée par son URI "http ://www.openflexo.org/test/animals#Cat" et trouvée depuis l'ontologie "ANIMALS" est importée ligne 25 et est nommée "CAT".

Redéfinition de types spécifiques : l'utilisation d'un ModelSlotKind implique l'utilisation du TechnologyAdapter requis et donc de tous les types spécifiques exposés. En revanche, certains types peuvent nécessiter des éléments de configuration (à définir dans la portée

```
namespace "http://www.openflexo.org/test/" as LOCAL;
namespace "http://www.openflexo.org/an.other.namespace/" as NS;

use org.openflexo.foundation.fml.rt.FMLRTModelSlot as FMLRT;
use org.openflexo.technologyadapter.owl.OWLModelSlot as OWL;

// import java String definition as 'JavaString'
import java.lang.String as JavaString;
// otherwise 'as' is implicit and values last path component, here 'List'
import java.util.List;
import org.openflexo.technologyadapter.owl.model.OWLOntology;

// Import FML model addressed by specified URI, locally identified by AnOtherModel
import [NS+"AnOtherModel.fml"] as AnOtherModel;
// Import FML model addressed by specified URI, 'as' is implicit
import [NS+"AThirdModel.fml"];

// Import OWL ontology with URI http ://www.openflexo.org/test/animals
// This ontology must be present in InformationSpace
// ANIMALS designates this OWL ontology
import [LOCAL+"/animals"] as ANIMALS;

// Import a particular object in ontology
// CAT designates here an OWL class with URI http ://www.openflexo.org/test/animals#Cat
import [ANIMALS:"Cat"] as CAT;

// We build here a technology-specific type configured with CAT OWL class
// This type represents an OWL individual whose types contains CAT class
// We call it 'Cat'
typedef OWLIndividualType(owlClass=CAT) as Cat;

// This is the main declaration
// URI is here explicit, but might be infered from RC URI
@URI(LOCAL+"MyModel.fml")
model MyModel extends AnOtherModel,AThirdModel {

    // We define an OWL ontology with OWLModelSlot
    OWLOntology ontology with OWLModelSlot();
    ...

    concept MyConcept {
        // Cat is usable as a type
        Cat aCat;
        ...
    }

    ...
}
```

FIGURE B.3 – Code FML montrant quelques constructions d'une FMLCompilationUnit

de l'unité de compilation, et non du VirtualModel). La ligne 30 présente un exemple de cette construction de type (un individual typé dans l'ontologie liée avec la classe OWL qui correspond à "CAT").

Calcul de la fermeture (*Closure computation*) : une unité de compilation FML modélise un environnement dans lequel toutes les dépendances requises (constructions spécifiques à la technologie, types, ressources externes et objets référencés dans cette FMLCompilationUnit) peuvent être résolues. Ces dépendances incluent transitivement les relations d'héritage et de contenance. Formellement, nous définissons la fonction closure qui "ferme" l'environnement local, en capturant tous les éléments nécessaires à la résolution de l'environnement local.

$$\begin{cases} \mathsf{closure}(vmi, \mathcal{IS}) ::= \mathsf{closure}(\mathsf{type}(vmi)), vmi \in \mathcal{I}_{\mathsf{VM}} \\ \mathsf{closure}(vm, \mathcal{IS}) ::= \mathsf{types\_closure}(vm) \cup \mathbb{R}_{vm} \cup \mathbb{TA}_{vm}, vm \in \mathcal{M}_{\mathsf{VM}} \\ \mathbb{TA}_{vm} = \{\mathcal{TA}_1 ... \mathcal{TA}_n\}_{\mathcal{TA}_i \in \mathbb{TA}}, \text{(adaptateurs technologiques requis)} \\ \mathbb{R}_{vm} = \{\mathcal{R}_1 ... \mathcal{R}_m\}_{\mathcal{R}_i \in \mathcal{IS}}, \text{(ressources externes requises)} \end{cases}$$

Notez que la fonction types_closure identifie les VirtualModels et les types FML requis, mais inclut également les types spécifiques (aux technologies référencées) requis.

Le calcul de cette fermeture est transitif avec une sémantique de calcul profonde (tous les liens sont explorés en profondeur).

$$\forall t \in \mathsf{closure}(vm, \mathcal{IS}) \cap \mathcal{M}, \mathsf{closure}(t, \mathcal{IS}) \subset \mathsf{closure}(vm, \mathcal{IS})$$

B.8 Le système de types FML

En tant que langage à typage statique, toutes les constructions de FML sont explicitement et fortement typées (propriétés, comportements, variables locales, etc...). Ce système de typage permet de formaliser et de renforcer les règles de typage très en amont lors de la compilation (assignation de variables, le calcul de l'assignabilité de types, les valeurs de retour, les appels de comportements).

Nous nous proposons d'explorer ici le système de types FML, qui se définit comme l'union des ensembles de types suivants :

1. **Les types représentés par les FlexoConcept** : tous les FlexoConcept sont considérés comme des types. On utilise la notation $\mathcal{M}$ pour désigner cet ensemble de types.
2. **Les types natifs FML** : le langage définit nativement un certain nombre de types, parmi lesquels on retrouve notamment :
   - les types primitifs (`int`, `long`, `short`, `byte`, `float`, `double`, `char`, `boolean`) : ce sont les mêmes types primitifs que ceux du langage Java ;
   - les types génériques : `List<Foo>` désigne par exemple une liste d'instances du FlexoConcept `Foo` ;
   - les types "*wildcard*" : également très semblables à Java, ils permettent de spécifier une bonne inférieure et/ou supérieure et permettent par exemple de construire le type `List<? extends Foo>` qui désigne une liste d'instances d'un FlexoConcept qui spécialise `Foo` ;

— des types spécifiques au langage (par exemple le type `Resource<RD>` [6], ou `MatchingSet<T>` [7]);

— les types réflexifs : `Concept<? extends Foo>` désigne par exemple le type FML d'une instance d'un FlexoConcept qui hérite du (ou qui est le) FlexoConcept <Foo>.

On note $\mathbb{T}_{\text{FML}}$ l'ensemble de tous ces types FML.

3. **Les types Java** : le langage FML étant construit au dessus du langage Java, tous les types Java sont utilisables. On définit $\mathbb{T}_{\text{JAVA}}$ l'ensemble de tous les types Java. A noter que tous les types Java (incluant les types génériques et les "*wilcards*" sont utilisables).
4. **Les types spécifiques à une technologie** : le système de types FML comprend en outre tous les types exposés par tous les TechnologyAdapter utilisés dans un contexte donné. Nous avons déjà évoqué et nommé $\mathbb{T}_{\mathcal{TA}}$ comme l'ensemble des types spécifiques à une technologie pour un TechnologyAdapter $\mathcal{TA}$. A noter que cet ensemble de types comprend également les types personnalisables et configurables dans la portée d'une unité de compilation (comme par exemple le type `Cat` décrit dans la section B.7). Enfin, ces types exposent au langage FML leur navigabilité, spécifique à la technologie sous-jascente [8].

La partie rouge de la figure 4.6 illustre une partie du système de types FML qui se définit donc ainsi :

$$\mathbb{T} = \mathcal{M} \cup \mathbb{T}_{\text{FML}} \cup \mathbb{T}_{\text{JAVA}} \cup_{\mathcal{TA} \in \mathbb{TA}} \mathbb{T}_{\mathcal{TA}}$$

Tous les types faisant partie du système de type FML implantent une fonction permettant de calculer l'assignabilité d'un type par rapport à un autre. Cette fonction permet de définir les relations transitives de sous-typage et de conformance d'une instance à son type, qui s'applique en premier lieu à tout l'espace conceptuel, mais aussi à tout l'espace d'information exposé par le biais de l'API MirrorObject [9].

Espace de typage *(typing space)*

Etant donné un VirtualModel vm, on définit l'espace de typage $\mathbb{T}_{vm}$ comme la fermeture transitive du graphe de dépendances des types requis.

$$\mathbb{T}_{vm} = \text{closure}(vm) \cap \mathbb{T}$$

Nous étendrons cette définition à FMLCompilationUnit parce qu'une FMLCompilationUnit concerne un VirtualModel unique, ainsi qu'à VirtualModelInstance pour la même raison. L'espace de typage d'une unité de compilation représente donc l'ensemble transitif de tous les types requis dans ce contexte.

Cette notion d'espace de typage est également généralisable à tout FlexoConcept fc :

$$vm = \text{container}(fc), vm \in \mathcal{I}_{\text{VM}} \implies \mathbb{T}_{fc} = \mathbb{T}_{vm}$$

6. où `RD` désigne un type de contenu de Resource
7. où `T` désigne un FlexoConcept et qui représente un type utilisé pour les constructions de mise en correspondance (*matching*), voir plus loin dans le document
8. Nous verrons plus loin (section B.9) que FML inclut un langage d'expression qui permet de naviguer dans les modèles fédérés (et donc dans les objets externes). Dans ce cadre, les types spécifiques à une technologie permettent de définir des API spécifiques de navigation.
9. Tous les objets externes sont ainsi typés, et leur type est exposé et manipulables dans le langage FML.

Identification de types (*types lookup*)

Les mécanismes d'identification de types (*types lookup*) s'effectuent au niveau de l'unité de compilation (FMLCompilationUnit), au moyen de l'identifiant local fourni dans la syntaxe concrète, en recherchant ce type dans toutes les ressources extraites du calcul de closure qui contiennent des types.

Par exemple étant donné l'identifiant `Foo`, le processus de *lookup* sera le suivant :

1. identification dans l'unité de compilation locale (ou dans les dépendances transitives) d'une définition de type explicitement nommé ainsi (cf le mot-clef type,ligne 30 du listing B.3) ;
2. identification dans l'unité de compilation locale (ou dans les dépendances transitives) d'un FlexoConcept portant ce nom local ;
3. identification d'un type spécifique à une technologie dans les déclarations use dans l'unité de compilation locale (ou dans les dépendances transitives) ;
4. identification du type comme un type Java au moyen des clauses import dans l'unité de compilation locale (ou dans les dépendances transitives).

Les ambiguïtés d'identification de type sont interdites et explicitement levées par le compilateur.

Espaces de valeurs(*value spaces*)

De la même manière que nous avons défini les types, et avec des notations très similaires, nous définissons les espaces de valeurs suivants, qui représentent l'ensemble des valeurs possibles conformes à un type donné :

$$\mathbb{V}_{\mathsf{FML}} ::= \{o \mid \exists t \in \mathbb{T}_{\mathsf{FML}},\ o : t\}$$
$$\mathbb{V}_{\mathsf{JAVA}} ::= \{o \mid \exists t \in \mathbb{T}_{\mathsf{JAVA}},\ o : t\}$$
$$\mathbb{V}_{\mathcal{TA}} ::= \{o \mid \exists t \in \mathbb{T}_{\mathcal{TA}},\ o : t\}$$

On notera $\mathbb{V}$ l'espace des valeurs FML, qui représente l'ensemble de tous les termes terminaux (non réductibles). $null$ est un terme désignant une valeur vide.

$$\mathbb{V} ::= \mathcal{M} \cup \mathcal{I} \cup \mathbb{V}_{\mathsf{FML}} \cup_{ta \in \mathcal{TA}} \mathbb{V}_{\mathcal{TA}} \cup null$$

Conformément aux définitions de la section B.5, toutes les valeurs de l'espace d'information (de tous les espaces techniques) sont typées dans $\mathbb{T}_{\mathcal{TA}}$ et sont adressables depuis l'espace conceptuel en utilisant un ModelSlotKind $\mathcal{MS}$ et une ressource $\mathcal{R}$.

$$\mathcal{IS} = \bigcup_{(\mathcal{MS},\mathcal{R}) \in \mathcal{IS}} \{\ \mathbb{O}_{\mathcal{MS}}(\mathcal{R})\} \subset \mathbb{V}_{\mathcal{TA}}$$
$$\forall mo \in \mathbb{O}_{\mathcal{MS}}(\mathcal{R})\ \exists t \in \mathbb{T}_{\mathcal{TA}},\quad t : mo$$

B.9 Le langage d'expression FML

Un langage d'expression fait partie de la définition de FML. Ce langage d'expression est défini dans le contexte d'un espace de typage $\mathbb{T}_{\mathsf{TS}} \subset \mathbb{T}$. Ce langage se définit par une combinaison d'opérateurs avec des terminaux appelés BindingPath.

La figure B.4 présente ce langage d'expression, dans le contexte d'un espace de typage $\mathbb{T}_{\mathsf{TS}}$, sous la forme d'une grammaire abstraite.

B.9.1 BindingPath (chemin)

Un BindingPath se présente sous la forme d'une liste chaînée d'éléments typés, telle que :

$$binding\_path ::= var\,.\,n_1\,.\,\ldots\,.\,n_n,\ n_i \in \mathbb{N}$$

Ce $binding\_path$ représente ici un "chemin" calculable où var est une variable résolvable dans un contexte donné, et où l'élément n_i représente une fonctionnalité résolvable dans le contexte de la résolution du type de l'élément n_{i-1} (ou var si $i < 2$).

Nous appellerons f_i la fonctionnalité dénotée par l'élément $n_i \in \mathbb{N}$ (simple propriété) ou $n_i(e_1...e_m)$ (un appel de fonction paramétré, par exemple une procédure ou un comportement). Bien que certains appels de fonction puissent ne pas renvoyer de valeur et ne soient donc pas typés, nous considérons que toutes les fonctionnalités sont typées selon la convention $\perp$ représentant le type void. Le prédicat type peut être utilisé pour récupérer le type d'une expression ou d'une fonctionnalité ($\mathsf{type}(n_i) \in \mathbb{T}_{TS}$ and $\mathsf{type}(n_i(e_1...e_m)) \in \mathbb{T}_{TS} \cup \perp$). La fonctionnalité f_i est résolue en fonction de n_i (ou $n_i(e_1...e_m)$) et du type résolu du "sous-BindingPath" formé avec les prédécesseurs $var\,.\,n_1\,.\,\ldots\,.\,n_{i-1}$. A noter que la description de ce mécanisme de liaison concerne la phase de compilation et de vérification du typage. Il s'agit de s'assurer, statiquement, que le chemin est valide. A l'exécution, ce mécanisme pourra être spécialisé par un mécanisme de liaison dynamique (où c'est le type "réel" de l'objet qui va déterminer quelle fonctionnalité précise sera exécutée).

B.9.2 Expression FML

Une expression FML peut être :

— un chemin (BindingPath) ;
— une constante (une valeur $v \in \mathbb{V}$) ;
— l'assignation d'une expression à une propriété pointée par un BindingPath ;
— un opérateur unaire appliqué à une expression ;
— un opérateur binaire appliqué à deux expressions ;
— une conditionnelle avec une expression qui exprime une condition, et deux expressions alternatives ;
— une expression dont le type est converti dans un type donné (*cast*) ;
— une expression de vérification de typage avec le prédicat typeof (*instanceof*).

Les opérateurs FML supportés sont très semblables à ceux utilisés dans le langage Java et comprennent les opérateurs unaires, binaires, logiques et arithmétiques (!,+, −, ∗, /,

$$
\begin{array}{rll}
\textit{expression} & e ::= & \\
\textit{simple binding path} & & p \\
\textit{constant} & | & v \\
\textit{assign} & | & p\texttt{.}n \leftarrow e \\
\textit{unary operator} & | & \textit{un\_op}\ e \\
\textit{binary operator} & | & e\ \textit{bin\_op}\ e \\
\textit{conditional} & | & e\ ?\ e\ :\ e \\
\textit{cast} & | & (t)e \\
\textit{instance of} & | & e\ \textit{instanceof}\ t \\
\textit{binding path} & p ::= & \\
\textit{null} & & \textit{null} \\
\textit{variable} & | & \textit{var} \\
\textit{property} & | & n \\
\textit{current} & | & \textit{this} \\
\textit{parent} & | & [t_{\mathcal{M}}\texttt{.}]\textit{super} \\
\textit{container} & | & \textit{container} \\
\textit{property call} & | & p\texttt{.}n \\
\textit{container call} & | & p\texttt{.}\textit{container} \\
\textit{function call} & | & p\texttt{.}n(\textit{args}) \\
\textit{new instance} & | & \nu\ t(\textit{args})\ [\textit{in}\ p] \\
\textit{arguments} & \textit{args} ::= & \_ \mid e...e
\end{array}
$$

$n \in \mathbb{N}$ (noms des fonctionnalités), $t_{\mathcal{M}} \in \mathcal{M}$ (types), $t \in \mathbb{T}_{TS}$

$var \in \mathcal{V}$ (noms des variables), $v \in \mathbb{V}$ (valeurs)

FIGURE B.4 – Grammaire abstraite du langage d'expression FML

++, −−, &, |, &&, ||, <, <=, >, >=, ==, !=, >>, <<) ainsi que les opérateurs d'assignation (=,+=,− =), avec des priorités bien définies [10].

Les expressions FML sont en outre équipées d'un système de parenthésage. On se référera à [97] pour plus de détails sur ces aspects syntaxiques ainsi que sur les opérateurs et les priorités.

B.9.3 Environnement de typage (*typing environment*)

(aussi appelé *typing context*)

Une expression e est définie relativement à un *environnement de typage* $\mathcal{C}$ (plus précisemment $\mathcal{C}_{fc}$ quand l'environnement de typage est défini pour le FlexoConcept fc) :

$$\mathcal{C} := (fc, var_1 : t_1, ..., var_n : t_n)$$
$$fc \in \mathcal{M}, var_i \in \mathcal{V}, t_i \in \mathbb{T}_{\mathsf{TS}}$$

Les prédicats wf (*well formed*) et type se définissent ainsi pour un chemin (BindingPath) :

$$\begin{cases} let\ p = var\,.\,f_1\,.\,....\,.\,f_n \\ let\ \mathcal{C} = (fc, var_1 : t_1, ..., var_n : t_n) \\ \mathsf{wf}(p, \mathcal{C}) \stackrel{\text{def}}{=} (var = this) \vee (var \in \{var_1 ... var_n\}) \\ \qquad\qquad \wedge \mathsf{type}(var) <: \mathsf{type}(n1) \\ \qquad\qquad \wedge_{i \in \{2..n\}}\ (\mathsf{type}(var\,.\,f_1 ... f_{i-1}) \neq \bot \\ \qquad\qquad\quad \wedge\ \mathsf{type}(var\,.\,f_1 ... f_{i-1}) <: \mathsf{type}(f_i)) \\ \mathsf{type}(p, \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{type}(f_n) \end{cases}$$

Certaines fonctionnalités peuvent être modifiables (*settable*). Ce sont les propriétés simples (sans paramètres) et qui autorisent explicitement leur modification. Un BindingPath est *settable* si et seulement si l'élément final du chemin est déclaré *settable*. Une expression est *settable* si elle est formée d'un unique chemin BindingPath qui est *settable* :

$$let\ e = var\,.\,f_1\,.\,....\,.\,f_n$$
$$\mathsf{settable}(\mathcal{C}, e) \stackrel{\text{def}}{=} \mathsf{wf}(\mathcal{C}, e) \wedge \mathsf{settable}(f_n)$$

Tout opérateur unaire un_op définit, étant donné un type d'entrée, deux prédicats wf et type définissant respectivement la valididé de son application et le type retourné lorsque l'opérateur est appliqué. De même, tout opérateur binaire bin_op définit ces deux prédicats wf et type étant donné les deux types d'entrée sur lesquels cet opérateur s'applique.

10. et conformes aux spécifications Java

Ceci nous permet de construire le prédicat wf (*well formed*) qui définit les expressions légales :

$$
\begin{cases}
\mathsf{wf}(var\,.\,f_1 ... f_n, \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{wf}(p, \mathcal{C})\ (simple\ path) \\
\mathsf{wf}(v, \mathcal{C}) \stackrel{\text{def}}{=} true\ (constant) \\
\mathsf{wf}(p\,.\,n \leftarrow e, \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{settable}(p\,.\,n, \mathcal{C}) \wedge \mathsf{wf}(e, \mathcal{C}) \\
\mathsf{wf}(un\_op\ e, \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{wf}(un\_op, \mathsf{type}(e, \mathcal{C})) \\
\qquad\qquad \wedge \mathsf{wf}(e, \mathcal{C}) \\
\mathsf{wf}(e1\ bin\_op\ e2, \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{wf}(bin\_op, \\
\qquad\qquad \mathsf{type}(e1, \mathcal{C}), \mathsf{type}(e2, \mathcal{C})) \\
\qquad\qquad \wedge \mathsf{wf}(e1, \mathcal{C}) \wedge\ \mathsf{wf}(e2, \mathcal{C}) \\
\mathsf{wf}(c\ ?\ e1 : e2, \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{wf}(c, \mathcal{C}) \wedge \mathsf{type}(c) <: bool \\
\qquad\qquad \wedge \mathsf{wf}(e1, \mathcal{C}) \wedge\ \mathsf{wf}(e2, \mathcal{C}) \\
\qquad\qquad \wedge (\mathsf{type}(e1, \mathcal{C}) <: \mathsf{type}(e2, \mathcal{C}) \\
\qquad\qquad\quad \vee \mathsf{type}(e2, \mathcal{C}) <: \mathsf{type}(e1, \mathcal{C})) \\
\mathsf{wf}((t)e, \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{wf}(e, \mathcal{C}) \\
\mathsf{wf}(e\ instanceof\ t, \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{wf}(e, \mathcal{C})
\end{cases}
$$

Une expression FML est typée. Ce type est calculable en utilisant le prédicat type dont la définition suit :

$$
\begin{cases}
\mathsf{type}(var\,.\,f_1 ... f_n, \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{type}(p, \mathcal{C})(simple\ binding\ path) \\
\mathsf{type}(v, \mathcal{C}) \stackrel{\text{def}}{=} t \in \mathbb{T} \mid v : t \\
\mathsf{type}(p\,.\,n \leftarrow e, \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{type}(p\,.\,n, \mathcal{C}) \\
\mathsf{type}(un\_op\ e, \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{type}(un\_op, \mathsf{type}(e, \mathcal{C})) \\
\mathsf{type}(e1\ bin\_op\ e2, \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{type}(bin\_op, \\
\qquad\qquad \mathsf{type}(e1, \mathcal{C}), \mathsf{type}(e2, \mathcal{C})) \\
\mathsf{type}(c\ ?\ e1 : e2, \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{type}(e1, \mathcal{C}) <: \mathsf{type}(e2, \mathcal{C})\ ? \\
\qquad\qquad \mathsf{type}(e2, \mathcal{C}) : \mathsf{type}(e1, \mathcal{C}) \\
\mathsf{type}((t)e, \mathcal{C}) \stackrel{\text{def}}{=} t \\
\mathsf{type}(e\ instanceof\ t, \mathcal{C}) \stackrel{\text{def}}{=} bool
\end{cases}
$$

Nous appellerons $e_t,\ t \in \mathbb{T}$ une expression dont l'exécution produit une valeur conforme au type t :

$$\mathsf{type}(e_t, \mathcal{C}) \stackrel{\text{def}}{=} t$$

B.9.4 Points d'extension du langage d'expressions FML

Le langage d'expression FML est défini sur l'espace de typage complet de $\mathbb{T}_{\mathsf{TS}}$, qui comprend tous les types et les FlexoConcepts FML, mais aussi tous les types définis sur des

espaces technologiques liés. Dans ce contexte, ce langage vise en outre à fournir une navigation de données au sein de ces objets typés, et donc à rendre des services opérationnels (désigner et fournir des services par le biais d'un chemin). Pour ce faire, tout TechnologyAdapter fournit au langage d'expression FML des points d'extension via une BindingFactory qui définit des types spécifiques et des associations entre ces types et une liste de fonctionnalités (à la fois des propriétés simples et des appels de fonction paramétrés), ainsi que d'éventuels opérateurs unaires et binaires supplémentaires.

Ceci offre une très grande expressivité pour le langage FML. Si les notions de ModelSlot et FlexoRole permettent dans un premier temps l'accès à la donnée dans l'espace technologique, c'est le langage d'expression FML qui permet ensuite la navigation dans les espaces technologiques, à partir de ces points d'entrée, dans un paradigme adapté à la technologie visée. En pratique, cela rend le langage très opérationnel et adapté à la manipulation de données métier diverses.

Par exemple, la navigation dans une ontologie OWL permet de suivre des chemins à partir d'un *individual* en suivant les *object properties* et les *data properties* qui permettent de typer les données accédées (par exemple : `myDog.owner.address.city` si on imagine manipuler une ontologie avec les classes `Dog`, `Person`, `Address` et `City` et les *object properties* associées, ou encore exprimer `myCell.upper.value.asInt` sur une feuille de calcul pour désigner à partir d'une cellule la cellule immédiatement au dessus, et d'obtenir sa valeur en tant que nombre entier). Enfin, une même expression FML peut combiner différents paradigmes, pour par exemple exprimer `myDog.owner.address.city.name + myCell.upper.value.asString`.

B.9.5 Environnement d'exécution (*execution environment*)

Une expression e bien formée est évaluable étant donné un *environnement d'exécution* $\mathcal{E}$ (or $\mathcal{E}_{fci}$) conforme à un *environnement de typage* $\mathcal{C}$:

$$\mathcal{E} := (fci, var_1 \mapsto v_1, ..., var_n \mapsto v_n)$$
$$fci \in \mathcal{I}, var_i \in \mathcal{V}, v_i \in \mathbb{V}$$

La relation *conforms to* entre un environnement d'exécution et un environnement de typage ($\mathcal{E} : \mathcal{C}$) implique que toutes les variables définies dans $\mathcal{C}$ soient évaluables et que leur valeur soit du type attendu :

$$let\ \mathcal{C}_{fc} = (fc, var_1 : t_1, ..., var_n : t_n)$$
$$let\ \mathcal{E}_{fci} = (fci, var_1 \mapsto v_1, ..., var_n \mapsto v_n)$$
$$\mathcal{E}_{fci} : \mathcal{C}_{fc} \stackrel{\text{def}}{=} fci : fc$$
$$\wedge\ \forall\ (var_i, t_i) \in \mathcal{C},\ \exists\ (var_i, v_i) \in \mathcal{E}_{fci},\ v_i : t$$

On définit la fonction $\mathsf{env}_{\mathcal{E}} : \mathcal{V} \cup \mathbb{N} \mapsto \mathbb{V}$ qui associe à un identifiant valide (nom de variable ou fonctionnalité) $ident \in \mathcal{V} \cup \mathbb{N}$ une valeur $v = \mathsf{env}_{\mathcal{E}}(ident) \in \mathbb{V}$.

$$\begin{cases} let\ \mathcal{E}_{fci} = (fci, var_1 \mapsto v_1, ...var_n \mapsto v_n) \\ \mathsf{env}_{\mathcal{E}}(var_i) = v_i, \text{ if } var_i \in \{var_1 ... var_n\} \\ \mathsf{env}_{\mathcal{E}}(n) = (\lambda x\ x.n) fci, \text{ if } n \in \mathsf{feat\_set}(type(fci)) \end{cases}$$

B.9.6 Evaluation des expressions FML

Nous pouvons désormais décrire la fonction d'évaluation d'une expression FML dans le contexte d'un environnement d'exécution $\mathcal{E}_{fci}$:

$$[\![e_t]\!]\mathcal{E}_{fci} \longrightarrow v \ , \ v \in \mathbb{V}, \ t \in \mathbb{T}$$
$$([\![e_t]\!]\mathcal{E}_{fci} \longrightarrow v) \Rightarrow (v : t)$$

La fonction d'évaluation eval pour un BindingPath se définit ainsi :

$$\begin{aligned}
[\![null]\!]\mathcal{E}_{fci} &\longrightarrow null \\
[\![var]\!]\mathcal{E}_{fci} &\longrightarrow \mathsf{env}_{\mathcal{E}}(var) \\
[\![n]\!]\mathcal{E}_{fci} &\longrightarrow \mathsf{env}_{\mathcal{E}}(n) \\
[\![this]\!]\mathcal{E}_{fci} &\longrightarrow fci \\
[\![[t.]super]\!]\mathcal{E}_{fci} &\longrightarrow fci \\
[\![container]\!]\mathcal{E}_{fci} &\longrightarrow \mathsf{container}(fci) \\
[\![p.n]\!]\mathcal{E}_{fci} &\longrightarrow (\lambda x \ x.n)[\![p]\!]\mathcal{E}_{fci} \\
[\![p.container]\!]\mathcal{E}_{fci} &\longrightarrow \mathsf{container}([\![p]\!]\mathcal{E}_{fci}) \\
[\![p.n(args)]\!]\mathcal{E}_{fci} &\longrightarrow (\lambda x \ par \ x.n(par))[\![p]\!]\mathcal{E}_{fci}
\end{aligned}$$

Les règles suivantes complètent la fonction d'évaluation eval pour les expressions :

$$\begin{aligned}
[\![un_op \ e]\!]\mathcal{E}_{fci} &\longrightarrow un_op \ [\![e]\!]\mathcal{E}_{fci} \\
[\![e_1 \ bin_op \ e_2]\!]\mathcal{E}_{fci} &\longrightarrow [\![e_1 \ ]\!]\mathcal{E}_{fci} bin_op \ [\![e_2]\!]\mathcal{E}_{fci} \\
[\![e_{bool} ? e_1 : e_2]\!]\mathcal{E}_{fci} &\longrightarrow [\![e_{bool} \ ]\!]\mathcal{E}_{fci} ? \\
&\qquad [\![e_1]\!]\mathcal{E}_{fci} \ : \ [\![e_2]\!]\mathcal{E}_{fci} \\
[\![p \, . \, n \leftarrow e]\!]\mathcal{E}_{fci} &\longrightarrow [\![e]\!]\mathcal{E}_{fci} \\
[\![(t)e]\!]\mathcal{E}_{fci} &\longrightarrow [\![e]\!]\mathcal{E}_{fci} \\
[\![e \ instanceof \ t]\!]\mathcal{E}_{fci} &\longrightarrow [\![e]\!]\mathcal{E}_{fci} <: t \ ? \ true \ : \ false
\end{aligned}$$

Nous terminons par définir la fonction set_eval$(e_t, v, \mathcal{E})$ qui est valide si et seulement si l'expression e_t est *settable* et si v est du type attendu $t = \mathsf{type}(e_t)$:

$$\forall e_t \mid \mathsf{settable}(e_t) \ \forall v \in \mathbb{V} \mid v : t$$
$$\mathsf{set\_eval}(e_t, v, \mathcal{E}) \Rightarrow \mathsf{eval}(p, \mathcal{E}) = v$$

B.10 La logique impérative FML

Graphes de contrôle FML

La notion d'expression typée définie plus haut nous permet de décrire FML en tant que langage impératif fortement typé. Les constructions de base sont appelés graphes de contrôle (FMLControlGraph, cf figure 4.6). Une grammaire simplifiée de la syntaxe abstraite de ces graphes de contrôle est présentée figure B.5.

Un graphe de contrôle FML peut ainsi être :

$$
\begin{array}{rll}
\textit{control graph} & cg ::= & \\
\textit{nop} & & \_ \\
\textit{declaration} & | & t\ var\ [\leftarrow e] \\
\textit{expression} & | & e \\
\textit{assignment} & | & e \leftarrow e \\
\textit{edition action} & | & ea(args) \\
\textit{sequence} & | & cg\ cg \\
\textit{conditional} & | & if\, e_{bool}\ then\ cg\ [else\ cg] \\
\textit{loop} & | & while\ e_{bool}\ cg \\
\textit{iteration} & | & for\ var\ in\ e\ cg \\
\textit{return} & | & return\ e \\
\textit{arguments} & args ::= & \_ \mid e...e
\end{array}
$$

$ea \in \mathbb{EA}$ (edition action kinds), $t \in \mathbb{T}$ (types), $var \in \mathcal{V}$ (variables names)

FIGURE B.5 – Grammaire (simplifiée) de la syntaxe abstraite des graphes de contrôle FML

— nop (*no operation*) ;

— une déclaration (avec une éventuelle valeur initiale définie comme une expression) ;

— une expression FML ;

— une assignation d'une expression à une expression qui est *settable* ;

— une primitive comportementale (EditionAction) exposée par un TechnologyAdapter avec d'éventuels arguments définis comme expressions ;

— une séquence de deux graphes de contrôle ;

— une instruction conditionnelle exprimée avec une expression et deux graphes de contrôle définissant les deux alternatives ;

— une boucle avec une expression qui traduit la condition de bouclage et le graphe de contrôle sur lequel boucler ;

— une itération avec une expression qui définit l'itération et le graphe de contrôle sur lequel itérer ;

— une instruction de retour (*return*).

A noter que d'autres constructions sont aussi disponibles dans le langage (opérateurs de sélection, de *matching*, *logging*, notifications, etc...). Ces constructions sont évoquées dans la section B.13.

Le prédicat wf définit ainsi les graphes de contrôle légaux :

$$
\begin{cases}
\mathsf{wf}(\_, \mathcal{C}) \stackrel{\text{def}}{=} true \\
\mathsf{wf}(t\ var[\leftarrow e], \mathcal{C}) \stackrel{\text{def}}{=} (var \notin \mathcal{C} \vee\ \exists (var, t_s) \in \mathcal{C} \mid t <: t_s)\ [\wedge\ \mathsf{wf}(e, \mathcal{C})] \\
\mathsf{wf}(e, \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{wf}(e, \mathcal{C})\ (plain\ expression) \\
\mathsf{wf}(e1 \leftarrow e2, \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{wf}(e1, \mathcal{C}) \wedge \mathsf{wf}(e2, \mathcal{C}) \wedge\ \mathsf{type}(e2) <: \mathsf{type}(e1) \\
\mathsf{wf}(ea(args), \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{lookup}(ea, args) \bigwedge_{arg \in args} \mathsf{wf}(arg, \mathcal{C}) \\
\mathsf{wf}(cg1\ cg2, \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{wf}(cg1, \mathcal{C}) \wedge \mathsf{wf}(cg2, \mathcal{C}) \\
\mathsf{wf}(if\ e_{bool}\ then\ cg1\ [else\ cg2], \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{wf}(e_{bool}, \mathcal{C}) \wedge \mathsf{wf}(cg1, \mathcal{C})\ [\wedge \mathsf{wf}(cg2, \mathcal{C})] \\
\mathsf{wf}(while\ e_{bool}\ cg, \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{wf}(e_{bool}, \mathcal{C}) \wedge \mathsf{wf}(cg, \mathcal{C}) \\
\mathsf{wf}(for\ var\ in\ e\ cg, \mathcal{C}) \stackrel{\text{def}}{=} (var \notin \mathcal{C}\ \vee\ \exists (var, t_s) \in \mathcal{C} \mid t <: t_s) \\
\qquad \wedge \mathsf{wf}(e, \mathcal{C}) \wedge \mathsf{type}(e) <: List \\
\qquad \wedge \mathsf{wf}(cg, \mathcal{C}) \\
\mathsf{wf}(return\ e, \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{wf}(e, \mathcal{C})
\end{cases}
$$

Les graphes de contrôle peuvent être typés ou de typage indéterminé, ce que décrit le prédicat type :

$$
\begin{cases}
\mathsf{type}(\_, \mathcal{C}) \stackrel{\text{def}}{=} \emptyset \\
\mathsf{type}(t\ var[\leftarrow e], \mathcal{C}) \stackrel{\text{def}}{=} \emptyset \\
\mathsf{type}(e, \mathcal{C}) \stackrel{\text{def}}{=} \emptyset \\
\mathsf{type}(e1 \leftarrow e2, \mathcal{C}) \stackrel{\text{def}}{=} \emptyset \\
\mathsf{type}(ea(args), \mathcal{C}) \stackrel{\text{def}}{=} \emptyset \\
\mathsf{type}(cg1\ cg2, \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{type}(cg2, \mathcal{C}) \\
\mathsf{type}(if\ e_{bool}\ then\ cg1\ [else\ cg2], \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{type}(cg1, \mathcal{C}) \\
\mathsf{type}(while\ e_{bool}\ cg, \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{type}(cg, \mathcal{C}) \\
\mathsf{type}(for\ var\ in\ e\ cg, \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{type}(cg, \mathcal{C}) \\
\mathsf{type}(return\ e, \mathcal{C}) \stackrel{\text{def}}{=} \mathsf{type}(e, \mathcal{C})
\end{cases}
$$

Enfin, l'exécution d'un graphe de contrôle est décrite par la fonction exec dont la sémantique opérationnelle d'exécution est évoquée dans la section C.7.

B.11 Fonctionalités structurelles (propriétés)

La partie gauche de la figure 4.6 montre la hiérarchie de concepts pour l'abstraction FlexoProperty qui représente une propriété. Toutes les FlexoProperty ont un nom, une cardinalité, une visibilité définissant la portée, et un type (qui est soit déduit, soit défini explicitement, selon la nature de la propriété).

Voici la grammaire de la syntaxe abstraite d'une FlexoProperty :

$$
\begin{array}{rrl}
\textsf{FlexoProperty} & prop ::= & \\
variable & & card\ t_{\textsf{FML}}\ n \\
role & | & card\ t_R\ n\ [with\ properties] \\
model\ slot & | & card\ t_{MS}\ n\ [with\ properties] \\
expression\ property & | & n\ e \\
get\ property & | & n\ cg \\
get/set\ property & | & n\ cg\ cg \\
abstract\ property & | & t\ n\ \bot \\
cardinality & card ::= & 0..1 \mid 1 \mid 0..* \mid 1..* \\
properties & properties ::= & \_ \mid property...property \\
property & property ::= & t\ n
\end{array}
$$

$n \in \mathbb{N}$ (espace de noms), $t \in \mathbb{T}$ (types), $t_{\textsf{FML}} \in \mathbb{T}_{\textsf{FML}} \cup \mathbb{T}_{\textsf{JAVA}}$

$t_R \in \mathbb{T}_{\mathcal{TA}}, t_R <:$ FlexoRole (FlexoRoleKind)

$t_{MS} \in \mathbb{T}_{\mathcal{TA}}, t_{MS} <:$ ModelSlot (ModelSlotKind)

Le listing B.6 présente un exemple d'une unité de compilation qui définit diverses propriétés structurelles.

Les propriétés FlexoRole et ModelSlot peuvent déclarer des clauses contractuelles (en utilisant le mot-clé de la syntaxe concrète with) relativement à la donnée liée. L'intuition du langage est de pouvoir également définir "comment" la propriété est réalisée, à côté de son nom ("qui ?"), de sa cardinalité ("combien ?") et de son type résultant dans l'espace conceptuel ("quoi ?"). Le listing B.6 présente l'exemple d'une telle définition ligne 25. L'instanciation de ces propriétés prend la forme d'un dictionnaire clés-valeurs où les clés sont des chaînes et les valeurs des données typées représentées par une expression. La nature de ces propriétés dépend du type de FlexoRole ou de ModelSlot (il peut s'agir d'une spécification de contenance, d'un sous-type ou d'une relation d'instanciation, etc...).

— **FMLVariable** : c'est une variable typée, avec une cardinalité et une visibilité. Les lignes 15 à 21 du listing B.6 montrent des exemples de telles variables.

— **FlexoRole** : cette propriété se définit comme une référence à un objet. Cet objet peut être une instance de FlexoConcept ou bien un objet externe, spécifique à une technologie. La donnée référencée est accessible au sein d'une Resource en utilisant un ModelSlot.

— **ModelSlot** : un ModelSlot spécialise FlexoRole en référençant un modèle externe en tant que Resource. La ligne 10 du listing B.6 montre un exemple de ModelSlot permettant d'accéder à une ontologie OWL. Ceci est rendu possible par la déclaration use de OWLModelSlot déclarée à la ligne 1.

— **FlexoRole spécifique à une technologie** : la ligne 25 du listing B.6 montre un tel exemple de FlexoRole permettant d'accéder à une classe OWL. Ce rôle est défini comme un contrat assorti d'une clause contraignant la classe OWL concernée à se situer dans l'ontologie OWL désignée par l'expression 'owl' (qui est l'ontologie OWL référencée par le ModelSlot owl).

— **FlexoConceptInstanceRole** : cette propriété spécialise FlexoRole en désignant une instance (ou une liste d'instances) d'un FlexoConcept (exemples aux lignes 28 à 30

```
use org.openflexo.technologyadapter.owl.OWLModelSlot as OWL;

import java.lang.String;
import java.util.List;
import java.util.Map;

model MyModel {

    // OWL-specific model slot : owl designates here a OWL ontology
    OWLOntology owl with OWL::OWLModelSlot();

    concept ConceptA {

      // FML variables
      String aString;
      String[1,1] projectName; // A non-null String
      String[1,*] atLeastAValue;
      String[0,*] someValues;
      String... equivalentToPrevious;
      String[2,8] atLeast2AtMost8;
      int anInteger = 2+3;

      // OWL-specific FlexoRole
      // owlClass is contractually located in OWL ontology 'owl'
       OWLClass owlClass with OWL::OWLClassRole(container=owl);

      // FlexoConceptInstanceRole
      ConceptB aConceptB with ConceptInstance();
      ConceptB anOtherConceptB; // abbreviated notation
      ConceptB[0,*] someConceptB;

      // ExpressionProperty
      int aComputedValue values a.path.to.a.computed.value;
      int anOtherComputedValue values a.path().to.a.computed.value(1,2,"foo");
    }

    concept ConceptB {

      private int internalValue;

      // A GetSetProperty
      int value {
        int get() {
          return internalValue;
        }
        set(int aValue) {
          internalValue = aValue;
        }
      }
    };

    abstract concept ConceptC {
       abstract int anAbstractProperty;
    }
}
```

FIGURE B.6 – Extrait de code FML illustrant diverses propriétés structurelles

du listing B.6). Le mot-clef with permet de spécifier des clauses contractuelles. Notez que la syntaxe abrégée sans la clause with est également possible.

— **ExpressionProperty** : c'est une propriété dérivée qui s'appuie sur une expression (le type est alors calculé). Les lignes 33 et 34 du listing B.6 montrent de tels exemples.

— **GetProperty** : c'est une propriété calculée qui définit un graphe de contrôle typé (qui retourne une valeur typée). Le type est ici calculé (lignes 42-44 du listing B.6).

— **GetSetProperty** : elle spécialise GetProperty en fournissant également un graphe de contrôle dédié à la modification de la propriété (lignes 46-48 du listing B.6). Une telle propriété est *settable*.

— **AbstractProperty** : elle est définie comme une simple déclaration d'une propriété explicitement typée qui doit être implantée dans les sous-FlexoConcepts non abstraits (seul un FlexoConcept déclaré comme abstract peut définir une telle propriété). La ligne 52 du listing B.6 donne un exemple de propriété abstraite.

B.12 Fonctionnalités comportementales (comportements)

Les fonctionnalités comportementales (les comportements) d'un FlexoConcept sont représentées par la hiérarchie de FlexoBehaviour, comme illustré sur la partie droite de la figure 4.6. Un FlexoBehaviour est très similaire au concept de méthode d'une classe dans la programmation orientée objet. CreationScheme s'apparente à un constructeur et permet d'instancier un FlexoConcept, ActionScheme représente une méthode paramétrée tandis que DeletionScheme s'apparente à un destructeur. Un FlexoBehaviour a un nom, une visibilité qui détermine sa portée, et peut déclarer certains paramètres dont les types définissent une signature. Un FlexoBehaviour définit un graphe de contrôle, en tant que "morceau" de code FML suivant la logique de programmation impérative (cf section B.10). FML met en œuvre le polymorphisme dans le contexte d'une implantation de la liaison dynamique (cf section C).

Voici une grammaire (simplifiée) de la syntaxe abstraite d'un FlexoBehaviour :

FlexoBehaviour $behaviour ::=$

$$
\begin{aligned}
\textit{constructor} &\quad new\ n\ params\ cg\ [with\ properties] \\
\textit{destructor} &\quad |\ delete\ n\ cg\ [with\ properties] \\
\textit{method} &\quad |\ n\ params\ cg\ [with\ properties] \\
\textit{technology-specific} &\quad |\ n\ params\ cg\ [with\ properties] \\
\textit{parameters} \quad params &::= \_\ |\ param...param \\
\textit{parameter} \quad param &::= t\ n \\
\textit{properties} \quad properties &::= \_\ |\ property...property \\
\textit{property} \quad property &::= t\ n
\end{aligned}
$$

$n \in \mathbb{N}$ (espace de noms), $t \in \mathbb{T}$ (types)

De même que pour les propriétés, un FlexoBehaviour peut également déclarer des clauses contractuelles (en utilisant le mot-clé with) dans laquelle certaines propriétés peuvent être définies.

Le listing B.7 présente un exemple d'une unité de compilation FML avec divers comportements.

```
model MyModel extends AModel {

   // An anonymous CreationScheme with a parameter
   create(String aName) {
      super(42);
      ...
   }

   // A named CreationScheme with two parameters
   create::createNewModel(int id, String aName) {
      super(parameters.id);
      ...
   }

   // An action without arguments and returning an integer
   int update() {
      return 0;
   }

   concept MyConceptA {
      create() {
         ...
      }

      create::anOtherConstructor(int anInt) {
         ...
      }

      // An action with two arguments and returning an integer
      int action1(String name, int id) {
         return parameters.name.length()+parameters.id;
      }

      // Another action without parameters
      void action2() {
         ...
      }

      // No return type is required
      action3() {
         ...
      }

      // An action with a visibility modifier
      protected int action4() {
         return 42;
      }

      delete() {
         ...
      }

      delete::recursiveDelete() {
         ...
      }

   }
}
```

FIGURE B.7 – Extrait de code FML code illustrant divers comportements

— **Constructeurs (CreationSchemes)** : les constructeurs peuvent être anonymes (listing B.7, lignes 4-7 ou 21-23) ou nommés (listing B.7, lignes 10-13 ou 25-27). Ils définissent des arguments typés.

- Etant donné l'environnement de typage $\mathcal{C} := (\texttt{MyModel}, \_)$ (dans le contexte d'une instance de `MyModel`), un FlexoConcept s'instancie ainsi :

```
1  myConceptA1 = new MyConceptA();
2  myConceptA2 = new MyConceptA::anOtherConstructor(42);
```

- Etant donné l'environnement de typage $\mathcal{C} := (\_, \texttt{myModel} : \texttt{MyModel})$, l'appel du constructeur doit être contextualisé :

```
1  myConceptA1 = myModel.new MyConceptA();
2  myConceptA2 = myModel.new MyConceptA::anOtherConstructor(42);
```

- L'instanciation d'un VirtualModel doit préciser le nom de la nouvelle instance VirtualModelInstance :

```
1  MyModel myModel1 = new MyModel("foo") with (name="Foo");
2  MyModel myModel2
3     = new MyModel::createNewModel(42,"foo") with (name="Foo");
```

— **Destructeurs (DeletionSchemes)** : les constructeurs suivent la même logique. Ils peuvent être anonymes (listing B.7, lines 49-51) ou nommés (listing B.7, lines 53-55). Etant donné l'environnement de typage $\mathcal{C} := (\_, \texttt{myConcept} : \texttt{MyConceptA})$, l'appel au destructeur s'effectue ainsi :

```
1  myConcept.delete();
2  myConcept.delete::recursiveDelete();
```

— **Comportements simples (ActionSchemes)** : ils sont très similaires à une méthode. Le listing présente des exemples de ces autres comportements.
Etant donné l'environnement de typage $\mathcal{C} := (\_, \texttt{myModel} : \texttt{MyModel}, \texttt{myConcept} : \texttt{MyConceptA})$, les appels suivent une syntaxe classique :

```
1  myModel.update();
2  int i = myConcept.action1("foo",42);
3  myConcept.action2();
4  myConcept.action3();
5  int j = myConcept.action4();
```

— **Comportements spécifiques à une technologie** : comme on l'a vu dans la section B.5, une technologie peut également exposer des comportements spécifiques à l'espace conceptuel (cf ModelSlotKind). Il s'agit de comportements propres à une pile technologique donnée (et qui donc s'exécutent dans l'espace technologique concerné), auxquels le langage FML permet de se "brancher" pour programmer le comportement de la fédération. Pour reprendre les notations de 3.4, c'est l'action "globale" $a_{\mathcal{F}}$ qui doit répondre à l'action "locale" a. La déclaration d'un tel comportement permet de définir un mécanisme pour la fédération qui sera déclenché par une ressource externe fédérée. Cela permet par exemple de lier le code FML aux interacteurs des outils externes, et enfin d'implémenter des liens externes avec un comportement réactif.
Le listing B.8 montre par exemple l'utilisation d'un comportement spécifique appelé ListenClassAdded et qui est défini dans le cadre d'un modèle UML (cf la déclaration use de UMLModelSlot ligne 1). Cet exemple est extrait du cas d'utilisation présenté dans la section 3.4.1. Dans cet exemple, nous "écoutons" le diagramme de classes UML `classDiagram` déclaré à la ligne 5, et nous définissons à la ligne 8 le graphe de contrôle à exécuter lorsque ce comportement est déclenché.

```
1 use org.openflexo.ta.emf.UMLModelSlot as UML;
2
3 model MyModel {
4
5    UMLClassDiagram classDiagram with UML::UMLModelSlot();
6
7    newClassCreated(UMLClass umlClass)
8       with UML::ListenClassAdded(observed=classDiagram) {
9       // Some FML code here
10      ...
11   }
12
13 }
```

FIGURE B.8 – Extrait de code FML illustrant un comportement spécifique

B.13 Autres constructions du langage

Le langage FML définit un certain nombre de fonctionnalités accessibles à travers des constructions spécifiques. Sans prétendre être exhaustif par manque de place, nous nous proposons de passer en revue certaines de ces constructions.

B.13.1 Actions de requêtage

Le TechnologyAdapter peut fournir une API de requêtage de données (d'éléments de modèle), à travers la fourniture de FetchRequest, qui sont des spécialisations de la notion d'EditionAction accessibles via une construction syntaxique spécifique.

Le listing B.9 présente des exemples d'utilisation de cette construction, sur la base d'une fédération impliquant une ontologie OWL. Le TechnologyAdapter OWL fournit deux primitives SelectOWLClass et SelectOWLIndividual qui permettent de requêter respectivement des classes et des *individuals* dans une ontologie. Une première requête ligne 17 permet de retourner toutes les classes OWL de l'ontologie, et d'itérer. Un autre exemple ligne 25 explicite l'emploi de clauses de sélection (une ou plusieurs expressions conditionnelles derrière le mot-clef `where` qui contraignent la requête). Notons également l'utilisation du mot-clef `unique` qui contraint le résultat de la requête à un singleton [11]. La ligne 38 montre un exemple de requête sur des *individuals*, et la ligne 47 montre l'utilisation d'une requête avec un type spécifique OWL (défini ligne 8).

B.13.2 Actions de suppression/destruction

De la même façon, le TechnologyAdapter peut fournir une API de suppression de données, à travers la fourniture d'actions spécifiques (spécialisations de la notion d'EditionAction). Ces fonctionnalités sont utilisables via une construction syntaxique spécifique (mot-clef `delete`). La figure B.10 montre par exemple la suppression de formes dans un diagramme (lignes 11 et 16), dans la construction de deux destructeurs (DeletionScheme) encodant une sémantique différente.

11. La sémantique précise est que la première instance correspondante est retournée, c'est-à-dire que le respect de la contrainte d'unicité est de la responsabilité de l'utilisateur de la requête.

```
use org.openflexo.technologyadapter.owl.OWLModelSlot as OWL;

import ["http://openflexo.org/test/animals"] as ANIMALS;
import [ANIMALS:"Cat"] as CAT;

import org.openflexo.technologyadapter.owl.model.OWLOntology;

typedef OWLIndividualType(owlClass=CAT) as Cat;

@URI("http://www.openflexo.org/test/owl/Animals.fml")
public model Animals {
 OWLOntology ontology with OWLModelSlot();
      ...

 public List<OWLClass> listAllOWLClasses() {
  List<OWLClass> returned = new ArrayList();
  for (OWLClass aClass : select OWLClass from ontology) {
   log "OWLClass: " + aClass;
   returned.add(aClass);
  }
  return returned;
 }

 public OWLClass selectUniqueOWLClass() {
  return select unique OWLClass from ontology where (selected.name == "Cat");
 }

 public List<OWLIndividual> listAllOWLIndividuals2() {
  return select OWLIndividual from ontology;
 }

 public OWLIndividual selectUniqueOWLIndividual() {
  return select unique OWLIndividual from ontology where (selected.name == "Jerry")
      ;
 }

 public List<OWLIndividual> listAllCats() {
  List<OWLIndividual> returned = new ArrayList();
  for (OWLIndividualType(owlClass=CAT) aCat : select OWLIndividualType(owlClass=CAT
      ) from ontology) {
   log "Cat: " + aCat;
   returned.add(aCat);
  }
  return returned;
 }

 public List<Cat> listAllCats2() {
  List<Cat> returned = new ArrayList();
  for (Cat aCat : select Cat from ontology) {
   log "Cat: " + aCat;
   returned.add(aCat);
  }
  return returned;
 }

}
```

FIGURE B.9 – Extrait de code FML illustrant des actions de requêtage sur une ontologie

```
concept MyConceptGR {

  MyConcept myConcept;
  DiagramShape shape with DIAGRAM::ShapeRole;

  delete::deleteConcept() {
    delete myConcept;
    delete shape from diagram;
  }

  delete::removeFromDiagram() {
    myConcept = null;
    delete shape from diagram;
  }
}
```

FIGURE B.10 – Extrait de code FML illustrant des actions de suppression

B.13.3 Opérations de *matching*

D'un point de vue méthodologique et comme on l'a vu sur quelques scénarios, la mise en œuvre de la fédération de modèles implique souvent le *pattern* de mise en correspondance (alignement de deux modèles). Dans ce cadre, le langage FML a été doté de constructions spécifiques permettant de décrire de manière compacte ces opérations de synchronisation, qui s'apparentent à un appariement de deux modèles avec des instances d'un concept dédié (un FlexoConcept qui réifie un lien).

La figure B.11 illustre ce *pattern* par un exemple inspiré du cas d'utilisation décrit sur le listing 4.11. Il s'agit ici d'aligner un diagramme de classes (`classDiagram`, ligne 12) avec des classes Java (`javaSources`, ligne 13). C'est le concept `EntityClass` qui réfie ce lien, avec les deux rôles `umlClass`, ligne 17 et `javaClass`, ligne 18. Ce FlexoConcept définit un constructeur (lignes 20-23), un destructeur (lignes 29-32) et un comportement (lignes 25-27). Le comportement de synchronisation figure lignes 101-111, et repose sur l'instanciation d'un MatchingSet (une collection tagguée d'instances de `EntityClass`), comme indiqué ligne 102. Une itération est opérée sur toutes les instances UMLClass du diagramme de classe (ligne 103), et l'opérateur de matching est appliqué (lignes 104-107) pour identifier dans le MatchingSet l'instance qui vérifie l'assertion `selected.umlClass=umlClass` (ligne 105). Si cette instance est identifiée, elle est tagguée dans le MatchingSet, sinon une nouvelle instance est créée (lignes 106-107). Dans tous les cas, `myEntity` n'est pas `null` ligne 108 et le comportement `myEntity.doSomething()` peut être exécuté. Après l'itération, toutes les instances non tagguées du MatchingSet peuvent être appelées avec un comportement (ici le comportement `delete()` ligne 110 pour supprimer toutes les instances qui pointent vers une UMLClass `null` ou qui n'existe plus).

B.13.4 Gestion des évènements

La langage FML supporte nativement un modèle évènementiel (ces évènements sont à distinguer des comportements spécifiques fournis par des adaptateurs technologiques).

La figure B.12 montre un exemple de définition et d'utilisation d'un modèle évènementiel. Un évènement (FlexoEvent) est une spécialisation de FlexoConcept (définition lignes 15-

```
7 model ModelMapping {
                ⋮
11
12  UMLClassDiagram classDiagram with UML::UMLModelSlot();
13  SourceRepository javaSources with JAVA_REPO::RepositoryModelSlot(jdk="Java11+");
14
15  concept EntityClass {
16
17    UMLClass umlClass with UML::UMLClassRole(container=classDiagram);
18    JavaClass javaClass with JAVA::JavaClassRole(container=javaSources);
19
20    create(UMLClass umlClass,JavaClass javaClass) {
21      this.umlClass = umlClass;
22      this.javaClass = javaClass;
23    }
24
25    doSomething() {
26       ...
27    }
28
29    delete() {
30      this.umlClass = null;
31      this.javaClass = null;
32    }
33    ...
34  }
                ⋮
101 synchronize() {
102   MatchingSet<EntityClass> matchingSet = begin match EntityClass from this;
103   for (UMLClass umlClass : select UMLClass from classDiagram) {
104     EntityClass myEntity = match EntityClass in matchingSet from this
105       where (selected.umlClass=umlClass)
106       create(umlClass,
107            javaSources.getJavaFile(umlClass.name+".java").getMainClass()));
108     myEntity.doSomething();
109   }
110   end match EntityClass in matchingSet delete();
111 }
                ⋮
151 }
```

FIGURE B.11 – Extrait de code FML illustrant des actions de *matching*

```
1 public model TestEvent {
2
3   int result;
4
5   create() {
6     log "TestEvent";
7     this.sendMyEvent();
8   }
9
10  sendMyEvent() {
11    log "Send event";
12    fire new MyEvent(42);
13  }
14
15  event MyEvent {
16    int value;
17    create (int aValue) {
18      value = parameters.aValue;
19    }
20  }
21
22  listen MyEvent from this {
23    log "Event has been received with value="+evt.value;
24    result = evt.value;
25  }
26
27 }
```

FIGURE B.12 – Extrait de code FML illustrant la gestion des évènements

20). Sur tout FlexoConcept, il est possible de définir un comportement d'"écoute" d'un certain type d'évènement (sémantique d'héritage), en provenance d'une instance de FlexoConcept (ou VirtualModel) désigné par une expression. Par exemple sur la figure B.12, le VirtualModel `TestEvent` définit un tel comportement lignes 22-25 (le comportement est ici parfaitement réflexif, puisque l'instance de VirtualModel s'"écoute" elle-même). Les évènements sont déclenchés avec le mot-clef fire (ligne 12).

L'intérêt de ce support pour la programmation évènementielle dépend bien évidemment du modèle de concurrence dans lequel s'exécutent les VirtualModelInstance. L'implantation Openflexo fournit différentes stratégies pour le modèle d'exécution [12]. Nous abordons très rapidement ce point dans l'annexe C, avant de le traiter dans de futurs travaux.

B.13.5 Contraintes et invariants

En complément des contraintes imposées par le système de types du langage, FML fournit également une notion d'*invariant*, à prendre dans le sens de la programmation par contrat (en anglais, *Design by Contract* ou DbC), tel que formulé par Bertrand Meyer dans le langage Eiffel [157]. L'idée est de définir des règles, appelées *assertions*, qui forment un contrat précisant des responsabilités entre composants logiciels. Ce mécanisme complète le typage en fournissant un support plus souple dans un contexte de modélisation.

12. A ce jour 2 stratégies sont proposées : (1) exécution sans concurrence ou (2) une mémoire partagée et un fil d'exécution (*thread*) par VirtualModelInstance.

La figure B.13 présente un exemple d'usage de ces invariants, avec le mot-clef `assert` (lignes 10, 11-13, 14-18, 19-24). Une caractéristique particulière du langage FML est la possibilité de couplage de ces invariants avec des comportements liés à la manipulation des modèles (mot-clef `onfailure`), et qui peuvent être exécutés pour "réparer" des erreurs (pour essayer de remplir à nouveau le contrat).

Le support pour les invariants a été récemment introduit dans le langage et lève la problématique de la sémantique d'exécution de la vérification, et de l'exécution éventuelle de comportements de compensation (risque de boucles infinies, hystérésis, etc.). Des travaux futurs doivent permettre de définir une sémantique précise pour l'exécution de ces invariants.

```
1 model TestInvariants {
2
3  concept AConcept {
4
5   int a;
6   boolean h;
7   float i;
8   ASubConcept[0,*] subConcepts with ConceptInstance(container=this);
9
10  assert (i != -1);
11  assert h onfailure: {
12   h = true;
13  }
14  assert (a > 1) onfailure: {
15   while (a <= 1) {
16    this.incrementA();
17   }
18  }
19  assert (ASubConcept c : subConcepts) {
20   assert c.aBoolean onfailure: c.aBoolean = true;
21   assert c.anOtherBoolean onfailure: {
22    log "This is a failure case";
23   }
24  }
25
26  void incrementA() {
27   a=a+1;
28  }
29
30  concept ASubConcept {
31   boolean aBoolean;
32   boolean anOtherBoolean;
33  }
34 }
35
36 }
```

FIGURE B.13 – Extrait de code FML illustrant la gestion des invariants

Annexe C

Sémantique du langage FML

Cette annexe traite de la sémantique du langage FML. Nous ne donnons ici que quelques éléments de sémantique opérationnelle, sans prétendre couvrir toute la sémantique du langage. On se référera au rapport scientifique [97] pour plus de détails.

C.1 Notion d'état mémoire (*memory state*)

Etant donnée une fédération $\mathcal{F}_{\text{FML}} ::= (vmi, IS)$, un état mémoire se définit comme la fermeture transitive d'une fédération $\text{closure}(\mathcal{F}_{\text{FML}})$. Concrètement, nous définissons un état mémoire $\mathcal{M}em$ comme un ensemble de FlexoConceptInstances (y compris toutes les VirtualModelInstances qui sont également des FlexoConceptInstances), et tous les MirrorObject utilisés dans la fédération concernée :

$$\mathcal{M}em ::= \langle fci_1, ..., fci_n \mid mo_1, ..., mo_m \rangle, fci_i \in \mathcal{I}, mo_i \in \mathbb{O} \setminus \mathbb{O}_{\text{FML}}$$

On définit l'opérateur $\oplus$ qui permet de déclarer une nouvelle instance dans un état mémoire $\mathcal{M}em$:

$$\langle fci_1, ..., fci_n \mid ... \rangle \oplus fci \longrightarrow \langle fci, fci_1, ..., fci_n \mid ... \rangle,\ fci \in \mathcal{I}$$

$$\langle ... \mid mo_1, ..., mo_m \rangle \oplus mo \longrightarrow \langle ... \mid mo, mo_1, ..., mo_m \rangle,\ mo \in \mathbb{O} \setminus \mathbb{O}_{\text{FML}}$$

Inversement on définit l'opérateur $\ominus$ qui représente la suppression d'une instance d'un état mémoire $\mathcal{M}em$:

$$\langle fci_1, ..., fci_n \mid ... \rangle \ominus fci_i \longrightarrow \langle fci_1, ..., fci_{i-1}, fci_{i+1}, ..., fci_n \mid ... \rangle$$

$$\langle ... \mid mo_1, ..., mo_m \rangle \ominus mo_i \longrightarrow \langle ... \mid mo_1, ..., mo_{i-1}, mo_{i+1}, ..., mo_m \rangle$$

Nous définissons l'opérateur d'assignation suivant, qui s'applique à une instance FlexoConceptInstance. La valeur v est ici assignée à la *feature* n de l'instance fci :

$$fci \xleftarrow{n} v$$

Dans le formalisme de l'annexe B :

$$(\_ \triangleright \langle ..., n \mapsto v_{ini}, ... \rangle \xleftarrow{n} v) \longrightarrow \_ \triangleright \langle ..., n \mapsto v, ... \rangle$$

Nous complétons ces définitions par l'opérateur $\otimes$ qui désigne un état de mémoire $\mathcal{M}em$ dans lequel la *feature* n d'une instance FlexoConceptInstance a été assignée à une nouvelle valeur :

$$\mathcal{M}em \otimes (fci_i \xleftarrow{n} v) \longrightarrow \mathcal{M}em'$$

Notez qu'on présente ici une version simplifiée de la sémantique, où les FlexoConcept fc sont immutables, mais qu'ils pourraient l'être. On pourrait par exemple définir cette sémantique sur un état mémoire :

$$\mathcal{M}em ::= \langle fci_1, ..., fci_n \mid fc_1, ..., fc_n \mid mo_1, ..., mo_m \rangle, fci_i \in \mathcal{I}, fc_i \in \mathcal{M}, mo_i \in \mathbb{O} \setminus \mathbb{O}_{\text{FML}}$$

C.2 Notion d'état de calcul (*computation state*)

Nous définissons un état de calcul $\mathcal{U}$ comme un triplet définissant l'état de la mémoire, une liste ordonnée d'événements à déclencher et une pile (liste ordonnée) d'opérations à effectuer (op_1 est ici au sommet de la pile et sera la première opération exécutée).

$$\mathcal{U} ::= \langle \mathcal{M}em \mid evt_1, ..., evt_n \mid op_1, ..., op_m \rangle$$

Nous passons ici sous silence la gestion des évènements et des opérations pour lesquels les opérateurs d'ajout et de suppression devraient être explicitement définis.

Le traitement des évènements dans la pile d'événements a la priorité la plus élevée :

(1.1) EVENT
$$\frac{\text{handle\_event}(evt_1) = op \qquad (\langle \mathcal{M}em \mid evt_1, ..., evt_n \mid op_1, ..., op_m \rangle, op) \underset{\mathcal{U}}{\rightarrow} \langle \mathcal{M}em' \mid evt_1, ..., evt_n \mid op_1, ..., op_m \rangle}{\langle \mathcal{M}em \mid evt_1, ..., evt_n \mid op_1, ..., op_m \rangle \underset{\mathcal{U}}{\rightarrow} \langle \mathcal{M}em' \mid evt_2, ..., evt_n \mid op_1, ..., op_m \rangle}$$

(1.2) EVENT/ERR
$$\frac{\text{handle\_event}(evt_1) = op \qquad (\langle \mathcal{M}em \mid evt_1, ..., evt_n \mid op_1, ..., op_m \rangle, op) \underset{\mathcal{U}}{\rightarrow} Err \in \mathbb{E}}{\langle \mathcal{M}em \mid evt_1, ..., evt_n \mid op_1, ..., op_m \rangle \underset{\mathcal{U}}{\rightarrow} Err \text{ [END OF PROGRAM]}}$$

Les opérations sont traitées dans l'ordre dans lequel elles ont été rangées dans la pile. Quand la pile est vide, le programme s'arrête et l'état mémoire reflète le résultat attendu du calcul.

$$\text{(1.3) OPERATION} \quad \frac{(\langle \mathcal{M}em \mid \emptyset \mid op_1, ..., op_m \rangle, op_1) \underset{\mathcal{U}}{\rightarrow} \langle \mathcal{M}em' \mid \emptyset \mid op_1, ..., op_m \rangle}{\langle \mathcal{M}em \mid \emptyset \mid op_1, ..., op_m \rangle \underset{\mathcal{U}}{\rightarrow} \langle \mathcal{M}em' \mid \emptyset \mid op_2, ..., op_m \rangle}$$

$$\text{(1.4) END OF PROGRAM} \quad \frac{(\langle \mathcal{M}em \mid \emptyset \mid op \rangle, op) \underset{\mathcal{U}}{\rightarrow} \langle \mathcal{M}em' \mid \emptyset \mid op \rangle}{\langle \mathcal{M}em \mid \emptyset \mid op \rangle \underset{\mathcal{U}}{\rightarrow} \langle \mathcal{M}em' \mid \emptyset \mid \emptyset \rangle \text{ [END OF PROGRAM]}}$$

$$\text{(1.5) OPERATION/ERR} \quad \frac{(\langle \mathcal{M}em \mid \emptyset \mid op_1, ..., op_m \rangle, op_1) \underset{\mathcal{U}}{\rightarrow} Err \in \mathbb{E}}{\langle \mathcal{M}em \mid \emptyset \mid op_1, ..., op_m \rangle \underset{\mathcal{U}}{\rightarrow} Err \text{ [END OF PROGRAM]}}$$

Nous donnons ici une version simplifiée d'un modèle d'exécution sans concurrence. De futurs travaux pourraient étudier et comparer divers modèles de concurrence, par exemple une mémoire partagée et un fil d'exécution (*thread*) par VirtualModelInstance.

C.3 Sémantique des opérations

Les différentes opérations exécutables sont présentées sur la figure C.1.

Operation	$op ::=$
Evaluation d'une expression	$\mathsf{EVAL}(e)$
Assignation d'une valeur à une *feature*	$\mid \mathsf{SET}(fci, n \mapsto c_p, v)$
Exécution d'un graphe de contrôle	$\mid \mathsf{EXEC}(cg)$
Création d'une FlexoConceptInstance	$\mid \mathsf{NEW}(t_{\mathcal{M}}, n, \{arg_i\}, fci)$
Exécution d'un comportement	$\mid \mathsf{EXEC}(fci, n, \{arg_i\})$
Suppression d'une FlexoConceptInstance	$\mid \mathsf{DELETE}(fci, n, \{arg_i\})$

$fci \in \mathcal{I},\ e \in \mathbb{C}_{\mathsf{EXP}}$ (expressions), $cg \in \mathbb{C}_{\mathsf{CG}}$ (graphes de contrôle)

$n \in \mathbb{N},\ c_p \in \mathbb{P}_{\mathsf{FML}},\ v \in \mathbb{V},\ arg_i \in \mathbb{V}$

FIGURE C.1 – Opérations FML

Voici la syntaxe générale d'une règle d'inférence, étant donnés n prémices. op est le terme à réduire tandis que r est le terme réduit. La règle d'inférence s'applique à un terme à réduire, étant donné un état de calcul $\mathcal{U}$, un environnement de typage $\mathcal{C}_{fc}$ et un environnement d'exécution $\mathcal{E}_{fci}$. Elle produit un nouveau terme r, et peut modifier l'état de calcul, l'environnement de typage et/ou l'environnement d'exécution.

$$\text{RÈGLE} \quad \frac{Premice_1 \quad ... \quad Premice_n}{(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![op]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}',\ \mathcal{C}'_{fc},\ \mathcal{E}'_{fci},\ r)}$$

Une déclaration est ajoutée à un *environnement de typage* $\mathcal{C}_{fc}$ avec l'opérateur $\oplus$:

$$\mathcal{C}_{fc} \xrightarrow{t\ var} (var : t) \oplus \mathcal{C}_{fc}$$

Un opérateur similaire permet de décrire l'ajout d'une déclaration à un *environnement d'exécution* $\mathcal{E}_{fci}$ (une nouvelle variable est initialisée avec la valeur $null$) :

$$\mathcal{E}_{fci} \xrightarrow{t\ var} (var, null) \oplus \mathcal{E}_{fci}$$

Si la déclaration s'accompagne d'une valeur d'initialisation :

$$\mathcal{E}_{fci} \xrightarrow{t\ var \leftarrow e} (var, [\![e]\!]_{\mathcal{E}_{fci}}) \oplus \mathcal{E}_{fci}$$

C.4 Sémantique de l'évaluation des expressions FML

On considère en premier lieu l'opération qui décrit l'évaluation d'une expression FML pour une FlexoConceptInstance donnée :

$$op ::= \mathsf{EVAL}(e), e \in \mathbb{C}_{\mathsf{EXP}}$$

Formellement, cette expression est évaluée dans un contexte défini par : $(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci})$.

Pour des raisons de lisibilité, on notera cette opération $op ::= [\![e]\!]_{\mathcal{E}_{fci}}$ et on considérera :

$$\textsc{Règle}\quad \frac{Premice_1 \quad ... \quad Premice_n}{[\![e]\!]_{\mathcal{E}_{fci}} \longrightarrow r}$$

comme une notation simplifiée mais équivalente à :

$$\textsc{Règle}\quad \frac{Premice_1 \quad ... \quad Premice_n}{(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ op) \longrightarrow (\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ r)}$$

Voici les règles d'inférence qui décrivent la sémantique formelle pour la syntaxe abstraite de la figure B.4.

(2.1) NULL EXPRESSION $$\frac{}{[\![null]\!]_{\mathcal{E}_{fci}} \longrightarrow null}$$

(2.2) VALUE EXPRESSION $$\frac{v \in \mathbb{V}}{[\![v]\!]_{\mathcal{E}_{fci}} \longrightarrow v}$$

(2.3) VARIABLE EXPRESSION $$\frac{\mathcal{E}_{fci} ::= (this \mapsto fci, var_1 \mapsto v_1, ...var_n \mapsto v_n)}{[\![var_i]\!]_{\mathcal{E}_{fci}} \longrightarrow v_i}$$

(2.4) THIS EXPRESSION $$\frac{}{[\![this]\!]_{\mathcal{E}_{fci}} \longrightarrow fci}$$

(2.5) IMPLICIT SUPER EXPRESSION $$\frac{\mathsf{supertypes}(fci) = \{t\},\ t \in \mathcal{M}}{(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![super]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U},\ \mathcal{C}_t,\ \mathcal{E}_{fci},\ fci)}$$

(2.6) EXPLICIT SUPER EXPRESSION $$\frac{t_{\mathcal{M}} \in \mathsf{supertypes}(fci),\ t_{\mathcal{M}} \in \mathcal{M}}{(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![t_{\mathcal{M}}.super]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U},\ \mathcal{C}_{t_{\mathcal{M}}},\ \mathcal{E}_{fci},\ fci)}$$

(2.7) CONTAINER EXPRESSION $$\frac{\mathsf{container}(fci) \longrightarrow vmi,\ vmi \in \mathcal{I}_{VM} \qquad \mathsf{container}(fci) \neq \emptyset}{[\![container]\!]_{\mathcal{E}_{fci}} \longrightarrow vmi}$$

(2.8) INSTANCE PROPERTY CALL $$\frac{[\![p]\!]_{\mathcal{E}_{fci}} \longrightarrow fci_2,\ fci_2 \in \mathcal{I} \qquad [\![n]\!]_{\mathcal{E}_{fci_2}} \longrightarrow v,\ v \in \mathbb{V}}{[\![p.n]\!]_{\mathcal{E}_{fci}} \longrightarrow v}$$

(2.9) CONTAINER PROPERTY CALL $$\frac{\begin{array}{c}[\![p]\!]_{\mathcal{E}_{fci}} \longrightarrow fci_2,\ fci_2 \in \mathcal{I} \\ \mathsf{container}(fci_2) \longrightarrow vmi,\ vmi \in \mathcal{I}_{\mathsf{VM}} \\ \mathsf{container}(fci_2) \neq \emptyset\end{array}}{[\![p.container]\!]_{\mathcal{E}_{fci}} \longrightarrow vmi}$$

(2.10) INSTANCE BEHAVIOUR CALL $$\frac{\begin{array}{c}[\![p]\!]_{\mathcal{E}_{fci}} \longrightarrow fci_2,\ fci_2 \in \mathcal{I} \qquad fci_2 : fc_2,\ fc_2 \in \mathcal{M} \\ \forall i \in \{1..m\}\ [\![e_i]\!]_{\mathcal{E}_{fci}} \longrightarrow v_i,\ e_i \in \mathbb{C}_{exp},\ v_i \in \mathbb{V} \\ \forall i \in \{1..m\}\ (v_i : t_i) \vee (v_i = null),\ t_i \in \mathbb{T}_{TS} \\ \mathsf{lookupFeature}(fc_2, n, \mathsf{type}(v_1)...\mathsf{type}(v_m)) = b \\ b ::= n\ var_1 : t_1 ... var_m : t_m\ cg \\ \mathcal{E'}_{fci_2} = ((var_1, [\![e_1]\!]_{\mathcal{E}_{fci}}), ..., (var_m, [\![e_m]\!]_{\mathcal{E}_{fci}})) \oplus \mathcal{E}_{fci_2} \\ [\![cg]\!]_{\mathcal{E'}_{fci_2}} \longrightarrow v\end{array}}{[\![p.n(e_1...e_m)]\!]_{\mathcal{E}_{fci}} \longrightarrow v}$$

(2.11) TA PROPERTY CALL $$\frac{[\![p]\!]_{\mathcal{E}_{fci}} \longrightarrow v,\ v \in \mathbb{V} \qquad v : t,\ t \in \mathbb{T}_{TS} \setminus \mathcal{M} \qquad n \in \mathsf{featuresSet}(t)}{[\![p.n]\!]_{\mathcal{E}_{fci}} \longrightarrow v.\mathsf{executeFeature}(n)}$$

(2.12) TA PARAMETERED CALL
$$\frac{\begin{array}{c}[\![p]\!]\varepsilon_{fci} \longrightarrow v,\ v \in \mathbb{V} \qquad v : t,\ t \in \mathbb{T}_{TS} \setminus \mathcal{M} \\ \forall i \in \{1..m\}\ [\![e_i]\!]\varepsilon_{fci} \longrightarrow v_i,\ e_i \in \mathbb{C}_{exp},\ v_i \in \mathbb{V} \\ \forall i \in \{1..m\}\ (v_i : t_i) \vee (v_i = null),\ t_i \in \mathbb{T}_{TS} \\ \mathsf{lookupFeature}(t, n, \mathsf{type}(v_1)...\mathsf{type}(v_m)) = b\end{array}}{[\![p.n(e_1...e_m)]\!]\varepsilon_{fci} \longrightarrow v.\mathsf{executeFeature}(b, [\![e_1]\!]\varepsilon_{fci}, ..., [\![e_m]\!]\varepsilon_{fci})}$$

(2.13) UNARY OPERATOR
$$\frac{[\![e]\!]\varepsilon_{fci} \longrightarrow v \qquad v \in dom(un\_op)}{[\![un\_op\ e]\!]\varepsilon_{fci} \longrightarrow un\_op\ v}$$

(2.14) BINARY OPERATOR
$$\frac{[\![e_1]\!]\varepsilon_{fci} \longrightarrow v_1 \qquad [\![e_2]\!]\varepsilon_{fci} \longrightarrow v_2 \qquad (v_1, v_2) \in dom(bin\_op)}{[\![e_1\ bin\_op\ e_2]\!]\varepsilon_{fci} \longrightarrow v_1\ bin\_op\ v_2}$$

(2.15) TRUE CONDITIONAL
$$\frac{[\![e_{bool}]\!]\varepsilon_{fci} \longrightarrow true \qquad [\![e_1]\!]\varepsilon_{fci} \longrightarrow v}{[\![e_{bool}\ ?\ e_1 :\ e_2]\!]\varepsilon_{fci} \longrightarrow v}$$

(2.16) FALSE CONDITIONAL
$$\frac{[\![e_{bool}]\!]\varepsilon_{fci} \longrightarrow false \qquad [\![e_2]\!]\varepsilon_{fci} \longrightarrow v}{[\![e_{bool}\ ?\ e_1 :\ e_2]\!]\varepsilon_{fci} \longrightarrow v}$$

C.5 Sémantique de l'évaluation d'une propriété

On définit la fonction $\mathsf{lookupFeature}(fc, n), fc \in \mathcal{M}, n \in \mathbb{N}$ qui s'applique sur le système de types FML. Cette fonction retourne la FlexoProperty nommée n la plus spécialisée qui s'applique au FlexoConcept fc. A noter que la sémantique de la liaison sémantique dans FML pour les propriétés se traduit donc par : $\mathsf{lookupFeature}(\mathsf{type}(fci), n)$ qui donne la propriété à considerer pour le propriété n de l'instance fci.

La sémantique de l'évaluation d'une propriété d'une FlexoConceptInstance dépend de la nature de la propriété. Les règles suivantes décrivent la sémantique opérationnelle de cette évaluation. La règle (3.1) traduit l'évaluation d'une variable simple tandis que la règle (3.2) traduit l'évaluation d'une propriété de type FlexoRole ou ModelSlot. La règle (3.3) décrit l'évaluation de l'expression déclarée comme contenu de la propriété (la valeur calculée est le résultat d'un calcul encodé comme une expression). Enfin la règle (3.4) décrit cette évaluation comme le résultat de l'évaluation d'un graphe de contrôle déclaré comme contenu d'une propriété (la valeur calculée est le résultat d'un calcul encodé comme un graphe de contrôle).

(3.1) FMLVARIABLE
$$\frac{fci ::= \_ \triangleright \langle ..., n \mapsto v, ... \rangle,\ n \in \mathbb{N},\ v \in \mathbb{V}_{\mathsf{FML}} \qquad v : t_{\mathsf{FML}},\ t_{\mathsf{FML}} \in \mathbb{T}_{\mathsf{FML}}}{[\![n]\!]\varepsilon_{fci} \longrightarrow v}$$

(3.2) FLEXOROLE/MODELSLOT
$$\frac{fci ::= \_ \triangleright \langle ..., n \mapsto v, ... \rangle,\ n \in \mathbb{N},\ v \in \mathbb{V}_{\mathcal{TA}} \quad v : t_R,\ t_R \in \mathbb{T}_{\mathcal{TA}}}{[\![n]\!]_{\mathcal{E}_{fci}} \longrightarrow v}$$

(3.3) EXPRESSIONPROPERTY
$$\frac{\begin{array}{c} fc = \mathsf{type}(fci),\ fc \in \mathcal{M} \\ fc = \_ \triangleright \langle ..., n \mapsto e, ... \rangle,\ n \in \mathbb{N},\ e \in \mathbb{C}_{\mathsf{EXP}} \\ \mathsf{lookupFeature}(fc, n) = n \mapsto e \\ (\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![e]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}', \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ v),\ v \in \mathbb{V} \end{array}}{(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![n]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}',\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ v)}$$

(3.4) GETPROPERTY
$$\frac{\begin{array}{c} fc = \mathsf{type}(fci),\ fc \in \mathcal{M} \\ fc = \_ \triangleright \langle ..., n \mapsto get\_cg, ... \rangle,\ n \in \mathbb{N},\ get\_cg \in \mathbb{C}_{\mathsf{CG}} \\ \mathsf{lookupFeature}(fc, n) = n \mapsto get\_cg \\ (\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![get\_cg]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}',\ \mathcal{C}'_{fc},\ \mathcal{E}'_{fci},\ v),\ v \in \mathbb{V} \end{array}}{(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![n]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}',\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ v)}$$

Il convient de noter que l'évaluation des propriétés FMLVariable, FlexoRole et ModelSlot ne modifie pas l'état de calcul (les règles d'inférence sont décrites à l'aide de la notation simplifiée). L'évaluation de ces propriétés est exécutée sans effet de bord. En revanche, l'évaluation des propriétés de type ExpressionProperty et de GetProperty peut modifier l'état de calcul (par exemple en modifiant la valeur d'une propriété d'une FlexoConceptInstance) et donc produire des effets de bord.

La règle (3.5) décrit l'évaluation d'une propriété abstraite qui conduit à une erreur d'exécution.

(3.5) ABSTRACTPROPERTY
$$\frac{fc = \mathsf{type}(fci),\ fc \in \mathcal{M} \qquad \mathsf{lookupFeature}(fc, n) = n \mapsto \emptyset}{[\![n]\!]_{\mathcal{E}_{fci}} \longrightarrow [ERR]}$$

La règle (3.6) traduit la sémantique de la contenance pour l'évaluation des propriétés : si la propriété nommée n n'est pas décrite dans l'instance courante et qu'elle l'est dans son container, alors l'évaluation se fera dans l'environnement d'exécution du conteneur.

(3.6) CONTAINER SCOPE
$$\frac{\begin{array}{c} \mathsf{container}(fci) \longrightarrow vmi,\ vmi \in \mathcal{I}_{VM} \\ n \notin \mathsf{feat\_set}(\mathsf{type}(fci)) \qquad [\![n]\!]_{\mathcal{E}_{vmi}} \longrightarrow v,\ v \in \mathbb{V} \end{array}}{[\![n]\!]_{\mathcal{E}_{fci}} \longrightarrow v}$$

C.6 Sémantique de l'assignation de valeur à une propriété

On considère maintenant l'opération qui traduit l'assignation d'une expression réductible à une valeur à une FlexoProperty :

$$op ::= \mathsf{SET}(fci, n \mapsto c_p, v),\ fci \in \mathcal{I},\ n \in \mathbb{N},\ c_p \in \mathbb{P}_{\mathsf{FML}},\ v \in \mathbb{V},\ arg_i \in \mathbb{V}$$

Seules les propriétés déclarées *settable* sont sujettes à des assignations (règle 4.1).

(4.1) NONSETTABLEPROPERTY
$$\frac{e \in \mathbb{C}_{\mathsf{EXP}} \qquad fc = \mathsf{type}(fci),\ fc \in \mathcal{M} \qquad \mathsf{lookupFeature}(fc, n) = n \mapsto c, c \in \mathbb{C} \qquad \overline{\mathsf{settable}(c)}}{(\mathcal{U}, \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![n \leftarrow e]\!]_{\mathcal{E}_{fci}}) \longrightarrow [ERR]}$$

Le type déclaré pour la propriété doit être compatible (règle 4.2).

(4.2) INVALIDTYPE
$$\frac{e \in \mathbb{C}_{\mathsf{EXP}} \qquad fc = \mathsf{type}(fci),\ \overline{fc \in \mathcal{M}} \qquad \mathsf{lookupFeature}(fc, n) = n \mapsto c, c \in \mathbb{C} \qquad \overline{\mathsf{type}(e) <: \mathsf{type}(c)}}{(\mathcal{U}, \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![n \leftarrow e]\!]_{\mathcal{E}_{fci}}) \longrightarrow [ERR]}$$

Les règles suivantes (règles 4.3 à 4.5) décrivent la sémantique opérationnelle de l'assignation pour les propriétés déclarées *settable*.

(4.3) VARIABLE ASSIGN
$$\frac{\begin{array}{c} fci ::= \_ \triangleright \langle ..., n \mapsto v_1, ... \rangle,\ n \in \mathbb{N},\ v \in \mathbb{V}_{\mathsf{FML}} \\ v_1 : t_{\mathsf{FML}},\ t_{\mathsf{FML}} \in \mathbb{T}_{\mathsf{FML}} \\ (\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![e]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}',\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ v_2),\ v_2 \in \mathbb{V}_{\mathsf{FML}} \end{array}}{(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![n \leftarrow e]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}' \otimes (fci \xleftarrow{n} v_2),\ \mathcal{C}_{fc},\ \mathcal{E}_{fci}, v_2)}$$

(4.4) ROLE/M.SLOT ASSIGN
$$\frac{\begin{array}{c} fci ::= \_ \triangleright \langle ..., n \mapsto v_1, ... \rangle,\ n \in \mathbb{N},\ v \in \mathbb{V}_{\mathcal{TA}} \qquad v_1 : t_R,\ t_R \in \mathbb{T}_{\mathcal{TA}} \\ (\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![e]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}',\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ v_2),\ v_2 \in \mathbb{V}_{\mathcal{TA}} \end{array}}{(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![n \leftarrow e]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}' \otimes (fci \xleftarrow{n} v_2),\ \mathcal{C}_{fc},\ \mathcal{E}_{fci}, v_2)}$$

(4.5) GET/SET P. ASSIGN
$$\frac{\begin{array}{c} fc = \mathsf{type}(fci),\ fc \in \mathcal{M} \\ fc = \_ \triangleright \langle ..., n \mapsto get\_cg\ set\_cg, ... \rangle,\ n \in \mathbb{N},\ get\_cg, set\_cg \in \mathbb{C}_{\mathsf{CG}} \\ \mathsf{lookupFeature}(fc, n) = n \mapsto get\_cg\ set\_cg \\ (\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![e]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}',\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ v),\ v \in \mathbb{V} \\ (\mathcal{U}',\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![set\_cg]\!]_{(value,v) \oplus \mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}'',\ \mathcal{C}'_{fc},\ \mathcal{E}'_{fci},\ \_) \end{array}}{(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![n \leftarrow e]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}'',\ \mathcal{C}_{fc},\ \mathcal{E}_{fci}, v)}$$

C.7 Sémantique des graphes de contrôle FML

Nous terminons cette présentation de quelques éléments de sémantique opérationnelle du langage en produisant les règles qui concernent l'exécution des graphes de contrôle FML.

$$op ::= \mathsf{EXEC}(cg),\ cg \in \mathbb{C}_{\mathsf{CG}}$$

La figure C.2 présente un aperçu non exhaustif de ces règles, sur la base de la grammaire abstraite des graphes de contrôle FML présentée figure B.5.

La règle (5.1) traduit l'opération *nop* (ne fait rien).

Les règles (5.2.1) et (5.2.2) concernent une nouvelle déclaration, qui modifie l'environnement de typage en lui adjoignant une variable typée. Une simple déclaration (règle 5.2.1) modifie l'environnement d'exécution en associant la valeur $null$ à la nouvelle variable, tandis qu'une déclaration/assignation associe l'évaluation de l'expression $[\![e]\!]\varepsilon_{fci}$ à cette variable (règle 5.2.2).

La règle (5.4) décrit l'exécution de graphes de contrôle en séquences. L'exécution du graphe de contrôle cg_2 bénéficie des environnements de typage et d'exécution résultant de l'exécution de cg_1.

Les règles (5.5.1-5.5.4) décrivent la sémantique de l'opérateur conditionnel, dans sa version simple et dans sa version ternaire avec une alternative.

Enfin les règles (5.6.1) et (5.6.2) décrivent la sémantique d'une boucle en FML.

(5.1) SKIP
$$\frac{}{(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![\_]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci})}$$

(5.2.1) DECLARATION
$$\frac{t \in \mathbb{T}_{TS},\ var \in \mathcal{V}}{(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![t\ var]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U},\ (var : t) \oplus \mathcal{C}_{fc},\ (var, null) \oplus \mathcal{E}_{fci})}$$

(5.2.2) DECL./ASSIGN.
$$\frac{\begin{array}{c}(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![e]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}',\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ v) \\ v \in \mathbb{V},\ t \in \mathbb{T}_{TS},\ var \in \mathcal{V}\end{array}}{(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![t\ var \leftarrow e]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U},\ (var : t) \oplus \mathcal{C}_{fc},\ (var, v) \oplus \mathcal{E}_{fci})}$$

(5.4) SEQUENCE
$$\frac{\begin{array}{c}(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![cg1]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}',\ \mathcal{C}'_{fc},\ \mathcal{E}'_{fci}) \\ (\mathcal{U}',\ \mathcal{C}'_{fc},\ \mathcal{E}'_{fci},\ [\![cg2]\!]_{\mathcal{E}'_{fci}}) \longrightarrow (\mathcal{U}'',\ \mathcal{C}''_{fc},\ \mathcal{E}''_{fci})\end{array}}{(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![cg1\ cg2]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}'',\ \mathcal{C}''_{fc},\ \mathcal{E}''_{fci})}$$

(5.5.1) TRUE COND. 1
$$\frac{\begin{array}{c}(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![e_{bool}]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}',\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ true) \\ (\mathcal{U}',\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![cg_1]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}'',\ \mathcal{C}'_{fc},\ \mathcal{E}'_{fci})\end{array}}{(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![if\ e_{bool}\ then\ cg_1\ else\ cg_2]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}'',\ \mathcal{C}_{fc},\ \mathcal{E}'_{fci})}$$

(5.5.2) TRUE COND. 2
$$\frac{\begin{array}{c}(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![e_{bool}]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}',\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ true) \\ (\mathcal{U}',\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![cg_1]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}'',\ \mathcal{C}'_{fc},\ \mathcal{E}'_{fci})\end{array}}{(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![if\ e_{bool}\ then\ cg_1]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}'',\ \mathcal{C}_{fc},\ \mathcal{E}'_{fci})}$$

(5.5.3) FALSE COND. 1
$$\frac{\begin{array}{c}(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![e_{bool}]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}',\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ false) \\ (\mathcal{U}',\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![cg_2]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}'',\ \mathcal{C}'_{fc},\ \mathcal{E}'_{fci})\end{array}}{(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![if\ e_{bool}\ then\ cg_1\ else\ cg_2]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}'',\ \mathcal{C}_{fc},\ \mathcal{E}'_{fci})}$$

(5.5.4) FALSE COND. 2
$$\frac{(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![e_{bool}]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}',\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ false)}{(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![if\ e_{bool}\ then\ cg_1]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}',\ \mathcal{C}_{fc},\ \mathcal{E}_{fci})}$$

(5.6.1) WHILE 1
$$\frac{(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![e_{bool}]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}',\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ false)}{(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![while\ e_{bool}\ then\ cg]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}',\ \mathcal{C}_{fc},\ \mathcal{E}_{fci})}$$

(5.6.2) WHILE 2
$$\frac{\begin{array}{c}(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![e_{bool}]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}',\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ true) \\ (\mathcal{U}',\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![cg]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}'',\ \mathcal{C}'_{fc},\ \mathcal{E}'_{fci}) \\ (\mathcal{U}'',\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![while\ e_{bool}\ then\ cg]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}''',\ \mathcal{C}_{fc},\ \mathcal{E}_{fci})\end{array}}{(\mathcal{U},\ \mathcal{C}_{fc},\ \mathcal{E}_{fci},\ [\![while\ e_{bool}\ then\ cg]\!]_{\mathcal{E}_{fci}}) \longrightarrow (\mathcal{U}''',\ \mathcal{C}_{fc},\ \mathcal{E}_{fci})}$$

FIGURE C.2 – Sémantique des graphes de contrôle FML

Annexe D

L'infrastructure Openflexo

D.1 Site web et resources

Le site web du projet Openflexo est https://openflexo.org.

L'infrastructure Openflexo est open-source sous double licence (GNU GPL v3 et EUPL)). Elle est développée en Java (Java 8 et Java11+), et versionnée sur git dans de nombreux dépôts. L'outil de *build* utilisé est GRADLE [1]. Le code source est disponible sur GitHub (https://github.com/openflexo-team/).

Des *releases* pour les principaux systèmes d'exploitation sont accessibles depuis le site web : https://downloads.openflexo.org/openflexo/.

L'intégration continue est accessible sur https://jenkins.openflexo.org.

La gestion de faits techniques est assurée sur GitHub, dans les dépôts concernés.

Site web	https://openflexo.org
Code source	https://github.com/openflexo-team/
Téléchargements	https://downloads.openflexo.org/openflexo/
Intégration continue	https://jenkins.openflexo.org

D.2 Composants logiciels

D.2.1 Environnement de développement et de build

- **openflexo-dev** Environnement de développement (*build* composite GRADLE)
  Code source : https://github.com/openflexo-team/openflexo-dev
- **openflexo-buildplugin** Plug-in GRADLE (environnement de build)
  Code source : https://github.com/openflexo-team/openflexo-buildplugin
- **openflexo-buildconfig** Plug-in GRADLE (environnement de build : configuration)
  Code source : https://github.com/openflexo-team/openflexo-buildconfig

1. https://gradle.org/

- **openflexo-production** Outils de production
  Code source : https://github.com/openflexo-team/openflexo-production

D.2.2 Composants de bas-niveau

- **connie** Bibliothèque de manipulation d'expressions typées
  https://openflexo.org/connie
  Code source : https://github.com/openflexo-team/connie
- **pamela** *Framework* de programmation orientée modèles
  https://openflexo.org/pamela
  Code source : https://github.com/openflexo-team/pamela
- **gina** Bibliothèque de définition et exécution d'IHM formulaires
  https://openflexo.org/gina
  Code source : https://github.com/openflexo-team/gina
- **diana** Bibliothèque de définition et exécution d'IHM diagrammatiques
  https://openflexo.org/diana
  Code source : https://github.com/openflexo-team/diana

D.2.3 Composants du cœur de fédération

- **openflexo-utils** Utilitaires pour openflexo-core
  https://openflexo.org/openflexo-utils
  Code source : https://github.com/openflexo-team/openflexo-utils
- **openflexo-core** Implantation du langage FML, environnement d'exécution, environnement de développement intégré, environnement de scripting et de test
  https://openflexo.org/openflexo-core
  Code source : https://github.com/openflexo-team/openflexo-core
- **integration-tests** Tests d'intégration
  Code source : https://github.com/openflexo-team/openflexo-integration-tests
- **http-server** Serveur Openflexo (HTTP)
  Code source : https://github.com/openflexo-team/http-server
  (aujourd'hui sur https://github.com/openflexo-team/openflexo-http/tree/2.99/http-server)

D.2.4 Adaptateurs technologiques

- **openflexo-diagram** Diagrammes (technologie DIANA)
  https://openflexo.org/openflexo-diagram
  Code source : https://github.com/openflexo-team/openflexo-diagram
- **openflexo-gina** Interfaces graphiques (technologie GINA)
  https://openflexo.org/openflexo-gina
  Code source : https://github.com/openflexo-team/openflexo-gina
- **openflexo-owl** Ontologies OWL
  https://openflexo.org/openflexo-owl
  Code source : https://github.com/openflexo-team/openflexo-owl
- **openflexo-emf** Modèles/métamodèles EMF (Eclipse Modelling Framework)
  https://openflexo.org/openflexo-emf
  Code source : https://github.com/openflexo-team/openflexo-emf

- **openflexo-http** Web-services (REST et XMLRPC)
  https://openflexo.org/openflexo-http
  Code source : https://github.com/openflexo-team/openflexo-http
- **openflexo-jdbc** Bases de données relationnelles (via JDBC)
  https://openflexo.org/openflexo-jdbc
  Code source : https://github.com/openflexo-team/openflexo-jdbc
- **openflexo-docx** Resources Microsoft word
  https://openflexo.org/openflexo-docx
  Code source : https://github.com/openflexo-team/openflexo-docx
- **openflexo-pptx** Resources Microsoft Powerpoint
  https://openflexo.org/openflexo-pptx
  Code source : https://github.com/openflexo-team/openflexo-pptx
- **openflexo-xlsx** Resources Microsoft Excel
  https://openflexo.org/openflexo-xlsx
  Code source : https://github.com/openflexo-team/openflexo-xlsx
- **openflexo-xml** Resources XML et XSD
  https://openflexo.org/openflexo-xml
  Code source : https://github.com/openflexo-team/openflexo-xml
- **openflexo-pdf** Resources PDF
  https://openflexo.org/openflexo-pdf
  Code source : https://github.com/openflexo-team/openflexo-pdf
- **openflexo-oslc** Resources OSLC (support limité)
  https://openflexo.org/openflexo-oslc
  Code source : https://github.com/openflexo-team/openflexo-oslc
- **openflexo-kafka** Resources KAFKA [2] (support limité)
  https://openflexo.org/openflexo-kafka
  Code source : https://github.com/openflexo-team/openflexo-kafka
- **openflexo-rhapsody** Resources IBM Rhapsody [3] (support limité)
  https://openflexo.org/openflexo-rhapsody
  Code source : https://github.com/openflexo-team/openflexo-rhapsody
- **openflexo-technology-adapters** Archétypes d'adaptateurs technologiques
  https://openflexo.org/openflexo-technology-adapters
  Code source : https://github.com/openflexo-team/openflexo-technology-adapters

D.2.5 Composants applicatifs

- **openflexo-modeller** Module applicatif générique offrant un environnement de développement et d'exécution de code FML
  https://openflexo.org/openflexo-modeller
  Code source : https://github.com/openflexo-team/openflexo-modeller
- **enterprise-architecture-editor** Module métier (environnement et outils de modélisation, BPMN, UML, OWL, etc.)
  https://openflexo.org/enterprise-architecture-editor
  Code source : https://github.com/openflexo-team/enterprise-architecture-editor

2. https://kafka.apache.org/
3. https://www.ibm.com/products/systems-design-rhapsody

- **free-modelling-editor** Editeur de modélisation libre
  https://openflexo.org/free-modelling-editor
  Code source : https://github.com/openflexo-team/free-modelling-editor
- **modelers** Centre de resources partagées (métamodèles, modeleurs, etc...)
  https://openflexo.org/modelers
  Code source : https://github.com/openflexo-team/modelers

D.3 *Releases* et gestion des dépendances

L'infrastructure elle-même est versionnée (*releases*) :

— **Infrastructure 1.9.1** (2019)
— **Infrastructure 2.0.0** (juin 2020)
— **Infrastructure 2.0.1** (dernière version stable, mars 2023)
— **Infrastructure 2.0.2** (en cours de développement, évolution 2.0.1 avec correctifs)
— **Infrastructure 2.99** (en cours de développement, inclut le support de la version textuelle de FML après une très importante ré-ingénierie, version transitoire qui encode la migration des VirtualModels XML vers FML)
— **Infrastructure 3.0.x** (en cours de développement, inclut le support de la version textuelle de FML + Java 11)
— **Infrastructure 3.1.x** (future *release* avec le support des modules Java)

L'infrastructure étant encodée dans différents dépôts GIT, la gestion des dépendances est relativement complexe. Ces différentes dépendances sont décrites dans les tableaux D.1 et D.2.

Infrastructure	**1.9.1**	**2.0.0**	**2.0.1** [a]	**2.0.2** [b]	**2.99** [c]	**3.0.0** [d]	**3.1.0** [e]
Version Java	Java 8	Java 8	Java 8	Java 8	Java 8	Java 11	Java 11
openflexo-buildconfig	0.2	0.2	0.2	0.2	0.3	0.3	0.4 [f]
openflexo-buildplugin	1.9.1	2.0.0	2.0.1	2.0.2	2.99	3.0.0	3.1.0?
openflexo-production	0.5	0.5	0.5	0.5	0.5	0.5	0.5
connie	1.4.2	1.5	1.5.0.1	1.5.0.2	2.0.0	2.1.0	3.1.0?
pamela	1.4.2	1.5	1.5.0.1	1.5.0.2	1.6	1.6.1	1.7?
gina	2.1.2	2.2	2.2.0.1	2.2.0.2	2.3	2.4	2.5?
diana	1.4.2	1.5	1.5.0.1	1.5.0.2	1.6	1.7	1.8?
openflexo-utils	1.4.2	1.5	1.5.0.1	1.5.0.2	1.6	1.7	1.8?
openflexo-core	1.9.1	2.0.0	2.0.1	2.0.2	2.99	3.0.0	3.1.0?
integration-tests	1.9.1	2.0.0	2.0.1	2.0.2	2.99	3.0.0	3.1.0?
Adaptateurs technologiques							
technology-adapters	1.9.1	2.0.0	2.0.1	2.0.2	2.99	3.0.0	3.1.0?
openflexo-diagram	✗	2.0.0	2.0.1	2.0.2	2.99	3.0.0	3.1.0?
openflexo-gina	✗	2.0.0	2.0.1	2.0.2	2.99	3.0.0	3.1.0?
openflexo-owl	✗	2.0.0	2.0.1	2.0.2	2.99	3.0.0	3.1.0?
openflexo-emf	✗	2.0.0	2.0.1	2.0.2	2.99	3.0.0	3.1.0?
openflexo-http	1.9.1	2.0.0	2.0.1	2.0.2	2.99	3.0.0	3.1.0?
openflexo-jdbc	1.9.1	2.0.0	2.0.1	2.0.2	2.99	3.0.0	3.1.0?
openflexo-docx	✗	✗	✗	✗	2.99	3.0.0	3.1.0?
openflexo-pptx	✗	✗	✗	✗	2.99	3.0.0	3.1.0?
openflexo-xlsx	✗	✗	✗	✗	2.99	3.0.0	3.1.0?
openflexo-xml	✗	✗	✗	✗	2.99	3.0.0	3.1.0?
openflexo-pdf	✗	✗	✗	✗	2.99	3.0.0	3.1.0?
openflexo-oslc [g]	1.9.1	2.0.0	2.0.1	2.0.2	2.99	3.0.0	3.1.0?
openflexo-kafka [g]	1.9.1	2.0.0	2.0.1	2.0.2	2.99	3.0.0	3.1.0?
openflexo-rhapsody [g]	✗	2.0.0	2.0.1	2.0.2	2.99	3.0.0	3.1.0?

a. Dernière *release* stable
b. En cours de développement (correction de bugs)
c. Version transitoire vers la version 3.0 (support pour la migration vers FML textuel), en développement
d. Prévision mi-2024
e. Non plannifié
f. Utilisation prévue des modules Java
g. Support limité

TABLE D.1 – Versions et dépendances des composants logiciels (cœur et adaptateurs technologiques)

Infrastructure	**1.9.1**	**2.0.0**	**2.0.1** [a]	**2.0.2** [b]	**2.99** [c]	**3.0.0** [d]	**3.1.0** [e]
Version Java	Java 8	Java 8	Java 8	Java 8	Java 8	Java 11	Java 11
openflexo-buildconfig	0.2	0.2	0.2	0.2	0.3	0.3	0.4 [f]
openflexo-buildplugin	1.9.1	2.0.0	2.0.1	2.0.2	2.99	3.0.0	3.1.0?
openflexo-production	0.5	0.5	0.5	0.5	0.5	0.5	0.5
connie	1.4.2	1.5	1.5.0.1	1.5.0.2	2.0.0	2.1.0	3.1.0?
pamela	1.4.2	1.5	1.5.0.1	1.5.0.2	1.6	1.6.1	1.7?
gina	2.1.2	2.2	2.2.0.1	2.2.0.2	2.3	2.4	2.5?
diana	1.4.2	1.5	1.5.0.1	1.5.0.2	1.6	1.7	1.8?
openflexo-utils	1.4.2	1.5	1.5.0.1	1.5.0.2	1.6	1.7	1.8?
openflexo-core	1.9.1	2.0.0	2.0.1	2.0.2	2.99	3.0.0	3.1.0?
integration-tests	1.9.1	2.0.0	2.0.1	2.0.2	2.99	3.0.0	3.1.0?
Modules applicatifs							
openflexo-modules	1.9.1	2.0.0	2.0.1	2.0.2	✗	✗	✗
openflexo-modeller	✗	✗	✗	✗	2.99	3.0.0	3.1.0?
architecture-editor	✗	✗	✗	✗	2.99	3.0.0	3.1.0?
free-modelling-editor	✗	✗	✗	✗	2.99	3.0.0	3.1.0?
http-server	✗	✗	✗	✗	✗	3.0.0	3.1.0?
modelers	1.9.1	2.0.0	2.0.1	2.0.2	2.99	3.0.0	3.1.0?
openflexo-packaging	1.9.1	2.0.0	2.0.1	2.0.2	2.99	3.0.0	3.1.0?

a. Dernière *release* stable
b. En cours de développement (correction de bugs)
c. Version transitoire vers la version 3.0 (support pour la migration vers FML textuel), en développement
d. Prévision mi-2024
e. Non plannifié
f. Utilisation prévue des modules Java

TABLE D.2 – Versions et dépendances des composants logiciels (cœur et modules applicatifs)

Glossaire

FML Federation Modelling Language. 82

Adaptateur technologique *(technology adapter)* Composant logiciel (bibliothèque) lié à une implantation du langage FML et exposant des fonctionnalités permettant d'utiliser une technologie donnée dans l'espace conceptuel du langage FML. L'objectif principal d'un adaptateur technologique est d'offrir une API claire et explicite de l'espace technique à l'espace conceptuel. 86, 87, 95, 120, 181, 200, 202

BPEL Business Process Execution Language. 61

BPMN Business Process Model and Notation. 61, 124

CIM Computation Independant Model. 22

CVE Common Vulnerabilities and Exposures. 154

CWE Common Weakness Enumeration. 154

DSL Domain Specific Language. 60, 62, 163

DSML Domain Specific Modeling Language. 40, 46

EMF Eclipse Modelling framework. 57, 60, 61, 76, 104, 124, 139, 238

Environnement d'exécution (*Execution environment*) Un environnement de d'exécution permet de valuer un environnement de typage, c'est-à-dire d'associer à chaque variable d'un environnement de typage donné sa valeur (conforme au type attendu). 123, 211, 230

Environnement de typage (*Typing environment* ou *Typing context*) En théorie des types, un environnement de typage représente une relation entre des noms de variables et leur type associé. 122, 209, 211, 229

Espace conceptuel *(conceptual space)* Espace de modélisation dédié à la conceptualisation et essentiellement composé de FlexoConcepts. Cette conceptualisation s'effectue de manière découplée des espaces techniques, dans lesquels résident les sources de données fédérées. 84, 86

Espace d'information *(information space)* Espace qui désigne l'ensemble des espaces techniques, et donc dans lequel "vivent" tous les modèles fédérés dans leur environnement technique d'origine. Cet espace s'oppose à l'espace conceptuel dans lequel s'effectue la conceptualisation. 84, 86, 95, 121, 201

Espace technique *(technical space)* Espace qui désigne un (ou des) modèle(s) à fédérer dans son environnement technique dans lequel il a été conçu et dans lequel s'opére son cycle de vie (objets externes à la fédération). 84, 86

EUPL EUropean Public Licence, version 1.2. 118, 237

FlexoBehaviour Comportement d'un FlexoConcept : un FlexoBehaviour se définit notamment par une signature typée et un graphe de contrôle dans un langage impératif, et spécifie un comportement. 84, 192, 217

FlexoConcept Concept central du langage FML : il permet la conceptualisation d'un concept métier. Un FlexoConcept définit un ensemble de propriété structurelles (des FlexoProperties) et comportementales (des FlexoBehaviours). 83, 191

FlexoConceptInstance C'est le concept qui représente une instance (ontologique) d'un FlexoConcept (son type). 197

FlexoProperty Propriété (structurelle) d'un FlexoConcept : une FlexoProperty définit l'accès à une donnée typée. 84, 192, 214, 232

FlexoRole Désigne une FlexoProperty permettant l'accès à une donnée externe (typée). Un FlexoRole est exposé à l'espace conceptuel par un *adaptateur technologique* et permet également de définir des clauses contractuelles (un contrat) concernant l'accès à la donnée sous-jascente. Cette notion s'apparente à une notion de pointeur (capable de franchir des frontières techniques). 84

GNU GPL v3 GNU General Public License, version 3.0. 118, 237

Graphe de contrôle (FMLControlGraph) Construction de base du langage impératif FML. 212

IDE Integrated Development Environment. 119

IDM Ingénierie Dirigée par les Modèles. 18, 19, 21, 27, 51, 53, 56, 57, 60–62, 122

MBSE Model Based System Engineering. 41, 172

MDA Model Driven Architecture. 22

MDD Model Driven Development. 23

MDE Model Driven Engineering. 21

MirrorObject Représentation locale dans le langage FML d'une instance d'un objet externe (au sein d'un modèle fédéré dans une technologie donnée): un MirrorObject maintient le lien vers un objet externe. 84

MOF Meta Object Facility. 21, 22, 47, 48, 68, 139

OMG Object Management Group. 18, 22

OOP Object Oriented Programming. 192

OWL Web Ontology Language. 76, 101, 124, 202, 211, 238

PDM Platform Description Model. 23

PIM Platform Independent Model. 22

PSM Platform Specific Model. 22

Resource La notion de Resource est une abstraction utilisée pour représenter l'accès à un "modèle". 199

ResourceCenter Un ResourceCenter fournit une couche d'accès à des Resources à partir de différents supports (système de fichiers, réseau, bases de données, serveurs, etc.). 99, 120, 199

ResourceData La notion de ResourceData représente le contenu d'une Resource, et donc une donnée dans laquelle se trouvent les MirrorObjects. 97, 199

REST HTTP/REST (*Representational State Transfer*) est un style d'architecture logicielle pour créer des services web. Les services web conformes au style d'architecture REST, aussi appelés services web RESTful, établissent une interopérabilité entre les ordinateurs sur Internet, à partir de requêtes sans état. 106, 129, 154

SSOT Single source of Truth. 59, 60

SysML Systems Modeling Language. 25, 41, 45, 61, 124

UML Unified Modeling Language. 21, 22, 25, 56, 61, 66, 68–70, 124

URI Uniform Resource Identifier. 191, 194

VirtualModel Unité de structuration des concepts (conteneur de FlexoConcepts). Un VirtualModel est également un FlexoConcept. 84, 195

VirtualModelInstance C'est le concept qui représente une instance (ontologique) d'un VirtualModel (son type). 198

XMI XML Metadata Interchange. 22

Bibliographie

[1] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[2] João Paulo A Almeida, Thomas Kühne, Adrian Rutle, and Manuel Wimmer. The MULTI process challenge–EMISAJ special issue version. 2021.

[3] C. Atkinson and T. Kuhne. Model-driven development : a metamodeling foundation. *IEEE Software*, 20(5) :36—-41, September 2003.

[4] Colin Atkinson and Ralph Gerbig. Flexible deep modeling with melanee. In *Modellierung 2016 – Workshopband*, pages 117–121, 2016.

[5] Colin Atkinson and Thomas Kühne. The essence of multilevel metamodeling. In *International Conference on the Unified Modeling Language*, pages 19–33. Springer, 2001.

[6] Colin Atkinson and Thomas Kühne. The essence of multilevel metamodeling. In *International Conference on the Unified Modeling Language*, pages 19–33. Springer, 2001.

[7] Colin Atkinson and Thomas Kühne. Profiles in a strict metamodeling framework. *Science of Computer Programming*, 44(1) :5–22, 2002.

[8] Colin Atkinson and Thomas Kuhne. Model-driven development : a metamodeling foundation. *IEEE software*, 20(5) :36–41, 2003.

[9] Colin Atkinson and Thomas Kühne. Concepts for comparing modeling tool architectures. In *International Conference on Model Driven Engineering Languages and Systems*, pages 398–413. Springer, 2005.

[10] Colin Atkinson, Dietmar Stoll, and Philipp Bostan. Supporting view-based development through orthographic software modeling. In *ENASE*, pages 71–86, 2009.

[11] Bruno Bachimont. Herméneutique matérielle et artéfacture : des machines qui pensent aux machines qui donnent à penser. *mémoire de Thèse en épistémologie, édition Ecole Polytechnique*, 1996.

[12] Bruno Bachimont. Arts et sciences du numérique : ingénierie des connaissances et critique de la raison computationnelle. *Mémoire de HDR*, 2004.

[13] Manas Bajaj, Jonathan Backhaus, Tim Walden, Manoj Waikar, Dirk Zwemer, Chris Schreiber, Ghassan Issa, Intercax, and Lockheed Martin. Graph-based digital blueprint for model based engineering of complex systems. In *INCOSE International Symposium*, volume 27, pages 151–169. Wiley Online Library, 2017.

[14] Manas Bajaj, Dirk Zwemer, Russell Peak, Alex Phung, Andrew G Scott, and Miyako Wilson. Slim : collaborative model-based systems engineering workspace for next-generation complex systems. In *2011 Aerospace Conference*, pages 1–15. IEEE, 2011.

[15] Manas Bajaj, Dirk Zwemer, Rose Yntema, Alex Phung, Amit Kumar, Anshu Dwivedi, and Manoj Waikar. MBSE++ — Foundations for Extended Model-Based Systems Engineering Across System Lifecycle. In *INCOSE International Symposium*, volume 26, pages 2429–2445. Wiley Online Library, 2016.

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

[17] Mikael Barbero, MDD Fabro, and Jean Bézivin. Traceability and provenance issues in global model management. In *Proc. of 3rd ECMDA Traceability Workshop (ECMDA-TW)*, pages 47–55. Citeseer, 2007.

[18] Matthias Barkowsky and Holger Giese. Modular and incremental global model management with extended generalized discrimination networks. *arXiv preprint arXiv :2301.00624*, 2023.

[19] Drouot Bastien, Valery Monthe, Sylvain Guérin, and Joël Champeau. Security Analysis : From model to system analysis. In *CRiSiS 2022 : International Conference on Risks and Security of Internet and Systems*, Sousse, Tunisia, December 2022.

[20] Amel Bennaceur, Thein Tun, Yijun Yu, and Bashar Nuseibeh. Requirements engineering. handbook of software engineering. https ://hal.archives-ouvertes.fr/hal-01758502/, 2018.

[21] Gábor Bergmann, István Dávid, Ábel Hegedüs, Ákos Horváth, István Ráth, Zoltán Ujhelyi, and Dániel Varró. Viatra 3 : A reactive model transformation platform. In *International Conference on Theory and Practice of Model Transformations*, pages 101–110. Springer, 2015.

[22] Jacques Bertin. Semiology of graphics. 1983.

[23] A. Beugnard, F. Dagnat, S. Guérin, and Christophe Guychard. Des situations de modélisation pour évaluer les outils de modélisation. In *INFORSID*, 2014.

[24] Antoine Beugnard. OO languages late-binding signature. In *The Ninth International Workshop on Foundations of Object-Oriented Languages*, 2002.

[25] Antoine Beugnard. A software engineering perspective on digital twin : many candidates, none elected. In *DigitalTwin 2023*, 2023.

[26] Antoine Beugnard, Fabien Dagnat, Sylvain Guérin, and Christophe Guychard. Des situations de modélisation pour décrire un processus de modélisation. *Ingénierie des systèmes d'information*, 20(2) :41 – 66, 2015.

[27] Jean Bézivin. La transformation de modèles. *INRIA-ATLAS & Université de Nantes*, 13, 2003.

[28] Jean Bézivin. On the unification power of models. *Software & Systems Modeling*, 4(2) :171–188, 2005.

[29] Jean Bézivin, Salim Bouzitouna, Marcos Didonet Del Fabro, Marie-Pierre Gervais, Frédéric Jouault, Dimitrios Kolovos, Ivan Kurtev, and Richard F Paige. A canonical scheme for model composition. In *Model Driven Architecture–Foundations and Applications : Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006. Proceedings 2*, pages 346–360. Springer, 2006.

[30] Jean Bézivin and Jean-Pierre Briot. Sur les principes de base de l'ingénierie des modèles. *Obj. Logiciel Base données Réseaux*, 10(4) :145–157, 2004.

[31] Jean Bézivin, Hugo Bruneliere, Frédéric Jouault, and Ivan Kurtev. Model engineering support for tool interoperability. In *Workshop in Software Model Engineering (WiSME'2005)-a MODELS 2005 Satellite Event*, 2005.

[32] Jean Bézivin, Sébastien Gérard, Pierre-Alain Muller, and Laurent Rioux. MDA components : Challenges and opportunities. In *Workshop on Metamodelling for MDA*, pages 23–41, 2003.

[33] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the omg/mda framework. In *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 273–280. IEEE, 2001.

[34] Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. On the need for megamodels. In *Proceedings of the OOPSLA/GPCE : Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–9. Citeseer, 2004.

[35] Dominique Blouin, Yvan Eustache, and Jean-Philippe Diguet. Extensible global model management with meta-model subsets and model synchronization. In *GEMOC@ MoDELS*, pages 43–52, 2014.

[36] Lossan Bondé. *Transformations de Modèles et Interopérabilité dans la Conception de Systèmes Hétérogènes sur Puce à Base d'IP*. PhD thesis, Thèse de doctorat, Université des Sciences et Technologies de Lille, 2006.

[37] Emma Borg, Antonio Scarafone, and Marat Shardimgaliev. Meaning and communication. Internet Encyclopedia of Philosophy https://iep.utm.edu/meaning-and-communication/, 2021.

[38] Artur Boronat, Roberto Bruni, Alberto Lluch Lafuente, Ugo Montanari, and Generoso Paolillo. Exploiting the hierarchical structure of rule-based specifications for decision planning. In John Hatcliff and Elena Zucca, editors, *Formal Techniques for Distributed Systems*, pages 2–16, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[39] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-driven software engineering in practice*. Morgan & Claypool Publishers, 2017.

[40] Hugo Bruneliere, Erik Burger, Jordi Cabot, and Manuel Wimmer. A feature-based survey of model view approaches. *Software & Systems Modeling*, 18(3) :1931–1952, 2019.

[41] Hugo Bruneliere, Jokin Garcia Perez, Manuel Wimmer, and Jordi Cabot. EMF views : A view mechanism for integrating heterogeneous models. In *International Conference on Conceptual Modeling*, pages 317–325. Springer, 2015.

[42] Greg Brunet, Marsha Chechik, Steve Easterbrook, Shiva Nejati, Nan Niu, and Mehrdad Sabetzadeh. A manifesto for model merging. In *Proceedings of the 2006 international workshop on Global integrated model management*, pages 5–12, 2006.

[43] Roberto Bruni, Alberto Lluch-Lafuente, and Ugo Montanari. On structured model-driven transformations. *International journal of software and informatics*, 5(1-2) :185–206, 2011.

[44] Erik Burger, Jörg Henss, Martin Küster, Steffen Kruse, and Lucia Happe. View-based model-driven software development with ModelJoin. *Software & Systems Modeling*, 15(2) :473–496, 2016.

[45] Guy Caplat. *Modèles et métamodèles*. PPUR presses polytechniques, 2008.

[46] David Chen, Nicolas Daclin, et al. Framework for enterprise interoperability. In *Proc. of IFAC Workshop EI2N*, pages 77–88. Bordeaux, 2006.

[47] Lianping Chen, Muhammad Ali Babar, and Nour Ali. Variability management in software product lines : a systematic review. 2009.

[48] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating co-evolution in model-driven engineering. In *Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*, EDOC '08, pages 222–231, Washington, DC, USA, 2008. IEEE Computer Society.

[49] ClearSy. Atelier B : B System. https://www.atelierb.eu/.

[50] Anthony Cleve, Ekkart Kindler, Perdita Stevens, and Vadim Zaytsev. Multidirectional Transformations and Synchronisations (Dagstuhl Seminar 18491). *Dagstuhl Reports*, 8(12) :1–48, 2019.

[51] Benoît Combemale. Ingénierie dirigée par les modèles (IDM)-état de l'art, 2008. *URL https :/ /hal. archives-ouvertes. fr/hal-00371565/document*, 2019.

[52] Benoit Combemale, Julien Deantoni, Benoit Baudry, Robert B France, Jean-Marc Jézéquel, and Jeff Gray. Globalizing modeling languages. *Computer*, 47(6) :68–71, 2014.

[53] Krzysztof Czarnecki, J Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F Terwilliger. Bidirectional transformations : A cross-discipline perspective. In *International Conference on Theory and Practice of Model Transformations*, pages 260–283. Springer, 2009.

[54] Merijn De Jonge. Pretty-printing for software reengineering. In *International Conference on Software Maintenance, 2002. Proceedings.*, pages 550–559. IEEE, 2002.

[55] Juan De Lara and Esther Guerra. Deep meta-modelling with metadepth. In *International conference on modelling techniques and tools for computer performance evaluation*, pages 1–20. Springer, 2010.

[56] Ferdinand de Saussure. *Cours de linguistique générale / Ferdinand de Saussure ; publié par Charles Bailly [i.e. Bally] et Albert Séchehaye avec la collaboration de Albert Riedlinger édition critique préparée par Tullio De Mauro postface de Louis-Jean Calvet*. Grande bibliothèque Payot. [reproduction en fac-similé] edition, 1916.

[57] Thomas Degueule, Benoit Combemale, Arnaud Blouin, Olivier Barais, and 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*, SLE 2015, pages 25–36, New York, NY, USA, 2015. ACM.

[58] Andreas Demuth, Markus Riedl-Ehrenleitner, Alexander Nöhrer, Peter Hehenberger, Klaus Zeman, and Alexander Egyed. Designspace : an infrastructure for multi-user/multi-tool engineering. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1486–1491, 2015.

[59] Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. Specifying overlaps of heterogeneous models for global consistency checking. In *Proceedings of the First International Workshop on Model-Driven Interoperability*, pages 42–51, 2010.

[60] Bastien Drouot. *Fédération de modèles pour l'analyse de cybersécurité du point de vue d'un attaquant.* PhD thesis, École Nationale Supérieure de Techniques Avancées Bretagne, 2021.

[61] Bastien Drouot and Joël Champeau. Model federation based on role modeling. In *MODELSWARD*, pages 72–83, 2019.

[62] Mickaël Duruisseau. *Améliorer la compréhension d'un programme à l'aide de diagrammes dynamiques et interactifs*. PhD thesis, 2019. Thèse de doctorat dirigée par Le Pallec, XavierGérard, Sébastien et Tarby, Jean-Claude Informatique Université de Lille (2018-2021) 2019.

[63] Umberto Eco. Sémiotique et philosophie du langage. 1988.

[64] Seventh Edition. The authoritative dictionary of ieee standards terms. In *IEEE Std. 100-2000*, pages 1–1362. 2000.

[65] Marc Ehrig. *Ontology alignment : bridging the semantic gap*, volume 4. Springer Science & Business Media, 2006.

[66] Yosser El Ahmar. *Améliorer l'efficacité cognitive des diagrammes UML : Application de la Sémiologie Graphique*. PhD thesis, Lille 1, 2018.

[67] Matthew Emerson and Janos Sztipanovits. Techniques for metamodel composition. In *OOPSLA–6th Workshop on Domain Specific Modeling*, pages 123–139, 2006.

[68] Vincent Englebert and Patrick Heymans. Towards more extensible metacase tools. In J. Krogstie, A.L. Opdahl, and G. Sindre, editors, *CAiSE 2007*, LNCS 4495, page 454–468, Berlin Heidelberg, 2007. Springer-Verlag.

[69] Patrice Enjalbert. De l'interprétation (sens, structures et processus). *Intellectica*, 23(2) :79–120, 1996.

[70] Nicole Everaert-Desmedt. "la sémiotique de peirce", dans louis hébert (dir.), signo [en ligne], rimouski. http://www.signosemio.com/peirce/semiotique.asp, 2011 (accessed November 10, 2020).

[71] Jean-Marie Favre. Foundations of model (driven)(reverse) engineering : models–Episode I : stories of the fidus papyrus and of the solarus. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2005.

[72] Jean-Marie Favre, Jacky Estublier, and Mireille Blay-Fornarino. *L'ingénierie dirigée par les modèles : au-delà du MDA*. Hermes-Lavoisier, 2006.

[73] Jean-Marie Favre and Tam Nguyen. Towards a megamodel to model software evolution through transformations. *Electronic Notes in Theoretical Computer Science*, 127(3) :59–74, 2005.

[74] Matthias Felleisen and Daniel P. Friedman. A reduction semantics for imperative higher-order languages. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE Parallel Architectures and Languages Europe*, pages 206–223, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[75] Formose ANR project. http://formose.lacl.fr.

[76] Steve Jeffrey Tueno Fotso, Marc Frappier, Régine Laleau, and Amel Mammar. Back propagating B system updates on SysML/KAOS domain models. In *Proceedings of the 23rd International Conference on Engineering of Complex Computer Systems, ICECCS 2018, Melbourne, Australia, December 12-14*, pages 160–169. IEEE Computer Society, 2018.

[77] Steve Jeffrey Tueno Fotso, Amel Mammar, Régine Laleau, and Marc Frappier. Event-B expression and verification of translation rules between SysML/KAOS domain models and B System specifications. In *ABZ*, volume 10817 of *LNCS*, pages 55–70. Springer, 2018.

[78] Robert France and Bernhard Rumpe. Model-driven development of complex software : A research roadmap. In *2007 Future of Software Engineering*, pages 37–54. IEEE Computer Society, 2007.

[79] Felipe Furtado and Andrea Zisman. Trace++ : A traceability approach to support transitioning to agile software engineering. In *2016 IEEE 24th International Requirements Engineering Conference (RE)*, pages 66–75, 2016.

[80] Holger Giese and Robert Wagner. From model transformation to incremental bidirectional model synchronization. *Software & Systems Modeling*, 8(1) :21–43, 2009.

[81] Gabriella Gigante and Domenico Pascarella. Formal methods in avionic software certification : The DO-178C perspective. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies*, pages 205–215, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[82] Thomas Goldschmidt, Steffen Becker, and Erik Burger. Towards a tool-oriented taxonomy of view-based modelling. *Modellierung 2012*, 2012.

[83] F. R. Golra, A. Beugnard, F. Dagnat, S. Guerin, and C. Guychard. Continuous requirements engineering using model federation. In *2016 IEEE 24th International Requirements Engineering Conference (RE)*, pages 347–352, 2016.

[84] Fahad R. Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guerin, and Christophe Guychard. Addressing modularity for heterogeneous multi-model systems using model federation. In *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, page 206–211, New York, NY, USA, 2016. Association for Computing Machinery.

[85] Fahad R Golra and Fabien Dagnat. The lazy initialization multilayered modeling framework (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, pages 924–927. ACM, 2011.

[86] Fahad Rafique Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guérin, and Christophe Guychard. Using free modeling as an agile method for developing domain specific modeling languages. In *MODELS 2016 : ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 24–34, Saint Malo, France, October 2016.

[87] Fahad Rafique Golra and Fabien Dagnat. The lazy initialization multilayered modeling framework. In Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 924–927. ACM, 2011.

[88] Fahad Rafique Golra, Fabien Dagnat, Jeanine Souquières, Imen Sayar, and Sylvain Guerin. Bridging the gap between informal requirements and formal specifications using model federation. In Einar Broch Johnsen and Ina Schaefer, editors, *Software Engineering and Formal Methods*, pages 54–69, Cham, 2018. Springer International Publishing.

[89] Cesar Gonzalez-Perez and Brian Henderson-Sellers. A powertype-based metamodelling framework. *Software & Systems Modeling*, 5(1) :72–90, 2006.

[90] Jack Goody, Jean Bazin, and Alban Bensa. *La raison graphique : la domestication de la pensée sauvage*. 1979.

[91] Sandra Greiner, Thomas Buchmann, and Bernhard Westfechtel. Bidirectional transformations with QVT-R : a case study in round-trip engineering UML class models and java source code. In *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 15–27. IEEE, 2016.

[92] Richard C Gronback. *Eclipse modeling project : a domain-specific language (DSL) toolkit*. Pearson Education, 2009.

[93] Object Management Group. Model driven architecture (MDA). https://www.omg.org/mda/, accessed November 10, 2020.

[94] Object Management Group. Unified modeling language (UML) 2.5.1. https://www.omg.org/spec/UML/2.5.1, December 2017 (accessed November 10, 2020).

[95] Object Management Group. Meta object facility (MOF) 2.0 core specification. https://www.omg.org/spec/MOF/, January 2006 (accessed November 10, 2020).

[96] Sylvain Guérin, Joel Champeau, Jean-Christophe Bach, Antoine Beugnard, Fabien Dagnat, and Salvador Martínez. Multi-level modeling with Openflexo/FML : A contribution to the multi-level process challenge. *Enterprise Modelling and Information Systems Architectures (EMISAJ)*, 17 :9–1, 2022.

[97] Sylvain Guérin, Joel Champeau, Jean-Christophe Bach, Antoine Beugnard, Fabien Dagnat, and Salvador Martínez. FML language : formal definition, syntax and semantics. Technical report, ENSTA Bretagne, IMT Atlantique, 2023.

[98] Esther Guerra and Juan de Lara. On the quest for flexible modelling. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 23–33, 2018.

[99] Esther Guerra, Juan De Lara, Dimitrios S Kolovos, and Richard F Paige. Inter-modelling : from theory to practice. In *International Conference on Model Driven Engineering Languages and Systems*, pages 376–391. Springer, 2010.

[100] Esther Guerra, Juan de Lara, and Fernando Orejas. Inter-modelling with patterns. *Software & Systems Modeling*, 12(1) :145–174, 2013.

[101] Christophe Guychard, Sylvain Guerin, Ali Koudri, Antoine Beugnard, and Fabien Dagnat. Conceptual interoperability through models federation. In *Semantic Information Federation Community Workshop / 16th International Conference, MODELS 2013*, 2013.

[102] Christophe Guychard, Sylvain Guerin, Ali Koudri, Antoine Beugnard, and Fabien Dagnat. Free the modeling ! In *Poster de présentation de la plateforme Openflexo de fédération de modèles*, 2013.

[103] Sylvain Guérin, Joel Champeau, Antoine Beugnard, and Salvador Martínez. Association constraints in model-oriented programming. In *MLE'2023 workshop, MODELS'23*, 2023.

[104] Sylvain Guérin, Guillaume Polet, Caine Silva, Joel Champeau, Jean-Christophe Bach, Salvador Martínez, Fabien Dagnat, and Antoine Beugnard. PAMELA : An annotation-based java modeling framework. *Science of Computer Programming*, 210 :102668, 2021.

[105] Matthew Hause et al. The SysML modelling language. In *Fifteenth European Systems Engineering Conference*, volume 9. Citeseer, 2006.

[106] Regina Hebig, Djamel Eddine Khelladi, and Reda Bendraou. Approaches to co-evolution of metamodels and models : A survey. *IEEE Transactions on Software Engineering*, 43(5) :396–414, 2016.

[107] Regina Hebig, Andreas Seibel, and Holger Giese. On the unification of megamodels. *Electronic Communications of the EASST*, 42, 2012.

[108] Ábel Hegedüs, Ákos Horváth, István Ráth, and Dániel Varró. Query-driven soft interconnection of EMF models. In *International Conference on Model Driven Engineering Languages and Systems*, pages 134–150. Springer, 2012.

[109] Rogardt Heldal, Patrizio Pelliccione, Ulf Eliasson, Jonn Lantz, Jesper Derehag, and Jon Whittle. Descriptive vs prescriptive models in industry. In *Proceedings of the acm/ieee 19th international conference on model driven engineering languages and systems*, pages 216–226, 2016.

[110] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. COPE-automating coupled evolution of metamodels and models. In *ECOOP*, volume 9, pages 52–76. Springer, 2009.

[111] Soichiro Hidaka, Massimo Tisi, Jordi Cabot, and Zhenjiang Hu. Feature-based classification of bidirectional transformation approaches. *Software & Systems Modeling*, 15(3) :907–928, 2016.

[112] Nicolas Hili. A metamodeling framework for promoting flexibility and creativity over strict model conformance. In *2nd Workshop on Flexible Model Driven Engineering Co-Located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016)*, page 2–11, 2016.

[113] Guillaume Hillairet, Frédéric Bertrand, Jean Yves Lafaye, et al. Bridging EMF applications and RDF data sources. In *Proceedings of the 4th International Workshop on Semantic Web Enabled Software Engineering, SWESE*, 2008.

[114] Wai-Ming Ho, Jean-Marc Jézéquel, François Pennaneac'h, and Noël Plouzeau. A toolkit for weaving aspect oriented UML designs. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 99–105, 2002.

[115] IEC 61508 Edition 2.0. International Electrotechnical Commission (IEC), 2010.

[116] ISO/TC 184/SC 5 Interoperability, integration, and architectures for enterprise systems and automation applications committee. Industrial automation systems and integration – concepts and rules for enterprise models. Standard ISO 14258 :1998, International Organization for Standardization, Geneva, CH, 1998.

[117] Shafagh Jafer, Bharvi Chhaya, and Umut Durak. OWL ontology to Ecore metamodel transformation for designing a domain specific language to develop aviation scenarios. In *Proceedings of the symposium on model-driven approaches for simulation engineering*, pages 1–11, 2017.

[118] Johannes Jakob, Alexander Königs, and Andy Schürr. Non-materialized model view specification with triple graph grammars. In *International Conference on Graph Transformation*, pages 321–335. Springer, 2006.

[119] Matthias Jarke, Rainer Gallersdörfer, Manfred A Jeusfeld, Martin Staudt, and Stefan Eherer. Conceptbase—a deductive object base for meta data management. *Journal of Intelligent Information Systems*, 4(2) :167–192, 1995.

[120] Cédric Jeanneret, Martin Glinz, and Thomas Baar. Modeling the purposes of models. In *Modellierung 2012*, number 201 in Lecture Notes in Informatics, pages 11–26, Bonn, Germany, March 2012. Gesellschaft für Informatik.

[121] Manfred Jeusfeld, Matthias Jarke, and John Mylopoulos. *Metamodeling for method engineering*. MIT press Cambridge, 2009.

[122] Manfred A Jeusfeld. Metamodeling and method engineering with conceptbase. In *Metamodeling for Method Engineering*, pages 89–168. MIT Press, 2009.

[123] Manfred A Jeusfeld and Bernd Neumayr. Deeptelos : Multi-level modeling with most general instances. In *International Conference on Conceptual Modeling*, pages 198–211. Springer, 2016.

[124] Jean-Marc Jézéquel. Modeling : From case tools to sle and machine learning, 2023.

[125] Jean-Marc Jézéquel, Benoit Combemale, and Didier Vojtisek. *Ingénierie Dirigée par les Modèles : des concepts à la pratique...* Références sciences. Ellipses, February 2012.

[126] Jean-Marc Jézéquel, Noël Plouzeau, Torben Weis, and Kurt Geihs. From contracts to aspects in UML designs. 2002.

[127] Ralph Johnson and Bobby Woolf. *Type Object*, chapter 4, page 47–65. Addison-Wesley Longman Publishing Co., Inc., USA, 1997.

[128] Nathalie Julien and Éric Martin. *Le jumeau numérique : De l'intelligence artificielle à l'industrie agile*. Dunod, 2020.

[129] Stuart Kent. Model driven engineering. In *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, pages 286–298, London, UK, UK, 2002. Springer-Verlag.

[130] A. Kleppe. *Software Language Engineering : Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, Boston, MA, USA, 2008.

[131] Wolfgang Kling, Frédéric Jouault, Dennis Wagelaar, Marco Brambilla, and Jordi Cabot. MoScript : A DSL for Querying and Manipulating Model Repositories. In Anthony Sloane and Uwe Aßmann, editors, *Software Language Engineering*, pages 180–200, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[132] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. Merging models with the Epsilon Merging Language (EML). In *International Conference on Model Driven Engineering Languages and Systems*, pages 215–229. Springer, 2006.

[133] Kurt Kosanke. ISO standards for interoperability : a comparison. In *Interoperability of enterprise software and applications*, pages 55–64. Springer, 2006.

[134] Ali Koudri, Christophe Guychard, Sylvain Guérin, Antoine Beugnard, Joël Champeau, and Fabien Dagnat. De la nécessité de fédérer des modèles dans une chaîne d'outils. In *Génie logiciel, numéro 105*, volume 105, pages 18–23, 2013.

[135] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. A metamodel family for role-based modeling and programming languages. In *International Conference on Software Language Engineering*, pages 141–160. Springer, 2014.

[136] Ivan Kurtev, Jean Bézivin, and Mehmet Aksit. Technological spaces : An initial appraisal. *CoopIS, DOA*, 2002, 2002.

[137] Sous la direction de M. Nivat. Premier rapport de l'observatoire de la recherche en informatique. Technical report, Ministère de l'Enseignement Supérieur et de la recherche, 1994.

[138] Alfons Laarman and Ivan Kurtev. Ontological metamodeling with explicit instantiation. In *Software Language Engineering : Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers 2*, pages 174–183. Springer, 2010.

[139] Cathy P Lachapelle and Christine M Cunningham. Engineering is elementary : Children's changing understandings of science and engineering. In *ASEE Annual Conference & Exposition*, volume 33, 2007.

[140] Philippe Lahire, Brice Morin, Gilles Vanwormhoudt, Alban Gaignard, Olivier Barais, and Jean-Marc Jézéquel. Introducing variability into aspect-oriented modeling approaches. In *International Conference on Model Driven Engineering Languages and Systems*, pages 498–513. Springer, 2007.

[141] Régine Laleau, Farida Semmak, Abderrahman Matoussi, Dorian Petit, Ahmed Hammad, and Bruno Tatibouet. A first attempt to combine SysML requirements diagrams and B. *Innovations in Systems and Software Engineering*, 6(1-2) :47–54, 2010.

[142] Philip Langer, Konrad Wieland, Manuel Wimmer, Jordi Cabot, et al. EMF profiles : A lightweight extension approach for EMF models. *J. Object Technol.*, 11(1) :1–29, 2012.

[143] Juan De Lara and Esther Guerra. A posteriori typing for model-driven engineering : Concepts, analysis, and applications. *ACM Trans. Softw. Eng. Methodol.*, 25(4), May 2017.

[144] Juan de Lara and Hans Vangheluwe. Atom 3 : A tool for multi-formalism and metamodelling. In *International Conference on Fundamental Approaches to Software Engineering*, pages 174–188. Springer, 2002.

[145] Matias Ezequiel Vara Larsen, Julien Deantoni, Benoit Combemale, and Frédéric Mallet. A behavioral coordination operator language (BCOoL). In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 186–195. IEEE, 2015.

[146] Tom Leclerc, Soumya Paul, Jussi Roberts, Fabien Dagnat, Florian Ledoux, Jean-Christophe Bach, Marcus Wallum, Nicky Mezzina, Daniel Fischer, Sylvain Guerin, Ihab Benamer, and Pierre JeanJean. A flexible and robust framework for the secure systems engineering of space missions. In *17th International Conference on Space Operations, Dubai, United Arab Emirates, 6-10 march 2023*, 2023.

[147] Thierry Lecomte, David Déharbe, Étienne Prun, and Erwan Mottin. Applying a formal method in industry : A 25-year trajectory. In Simone André da Costa Cavalheiro and José Luiz Fiadeiro, editors, *Formal Methods : Foundations and Applications - 20th Brazilian Symposium, SBMF 2017, Recife, Brazil, November 29 - December 1, 2017, Proceedings*, volume 10623 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2017.

[148] Ákos Lédeczi, Arpad Bakay, Miklos Maroti, Peter Volgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai. Composing domain-specific design environments. *Computer*, 34(11) :44–51, 2001.

[149] Michael Leuschel and Michael J. Butler. Prob : A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003 : Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874. Springer, 2003.

[150] Levi Lúcio, Sadaf Mustafiz, Joachim Denil, Hans Vangheluwe, and Maris Jukss. FTG+ PM : An integrated framework for investigating model transformation chains. In *SDL 2013 : Model-Driven Dependability Engineering : 16th International SDL Forum, Montreal, Canada, June 26-28, 2013. Proceedings 16*, pages 182–202. Springer, 2013.

[151] Nuno Macedo. *A relational approach to bidirectional transformation*. PhD thesis, 09 2014.

[152] Nuno Macedo, Tiago Jorge, and Alcino Cunha. A feature-based classification of model repair approaches. *IEEE Transactions on Software Engineering*, 43(7) :615–640, 2016.

[153] Fernando Macias Gomez de Villar, Adrian Rutle, and Volker Stolz. MultEcore : Combining the best of fixed-level and multilevel metamodelling. In *Proceedings of the 3rd International Workshop on Multi-Level Modelling*, volume 1722 of *CEUR Workshop Proceedings*, pages 66–75, 2016.

[154] Azad M Madni, Dan Erwin, and Carla C Madni. Digital twin-enabled MBSE testbed for prototyping and evaluating aerospace systems : Lessons learned. In *2021 IEEE Aerospace Conference (50100)*, pages 1–8. IEEE, 2021.

[155] Salvador Martínez, Massimo Tisi, and Rémi Douence. Reactive model transformation with atl. *Science of Computer Programming*, 136 :1–16, 2017.

[156] Johannes Meier and Andreas Winter. Model consistency ensured by metamodel integration. In *MoDELS (Workshops)*, pages 408–415, 2018.

[157] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10) :40–51, 1992.

[158] Marvin Minsky. Matter, mind, and models. Semantic Information Processsing, pages 425–432, 1968.

[159] William T Morris. On the art of modeling. *Management science*, 13(12) :B–707, 1967.

[160] Patrick Mpondo-Dicka. Sémiotique, numérique et communication. *Revue française des sciences de l'information et de la communication*, (3), 2013.

[161] Pierre-Alain Muller, Franck Fleurey, and 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.

[162] Pierre-Alain Muller, Frédéric Fondement, Benoit Baudry, and Benoît Combemale. Modeling modeling modeling. *Software & Systems Modeling*, 11(3) :347–359, 2012.

[163] Denisse Munante. *Une approche basée sur l'Ingénierie Dirigée par les modèles pour identifier, concevoir et évaluer des aspects sécurité*. PhD thesis, 2014. Thèse de doctorat dirigée par Philippe Aniorté et Laurent Gallon.

[164] Sadaf Mustafiz, Joachim Denil, Levi Lúcio, and Hans Vangheluwe. The FTG+ PM framework for multi-paradigm modelling : An automotive case study. In *Proceedings of the 6th International Workshop on Multi-paradigm Modeling*, pages 13–18, 2012.

[165] Jörg Niemöller, Leonid Mokrushin, Konstantinos Vandikas, Stefan Avesand, and Lars Angelin. Model federation and probabilistic analysis for advanced OSS and BSS. In *2013 Seventh International Conference on Next Generation Mobile Apps, Services and Technologies*, pages 122–129. IEEE, 2013.

[166] OMG. OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1, 2013. February 23^{rd}, 2022.

[167] Richard F Paige, Dimitrios S Kolovos, Louis M Rose, Nicholas Drivalos, and Fiona AC Polack. The design of a conceptual framework and technical infrastructure for model management language engineering. In *2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, pages 162–171. IEEE, 2009.

[168] Candy Pang and Duane Szafron. Single source of truth (SSOT) for service oriented architecture (SOA). In *International Conference on Service-Oriented Computing*, pages 575–589. Springer, 2014.

[169] Fernando Silva Parreiras, Steffen Staab, and Andreas Winter. On marrying ontological and metamodeling technical spaces. In *The 6th joint meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering : companion papers*, pages 439–448, 2007.

[170] C.S. Peirce and G. Deledalle. *Écrits sur le signe*. Collection L'ordre philosophique. Seuil, 1978.

[171] Benjamin C Pierce. *Types and programming languages*. MIT press, 2002.

[172] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. *Software product line engineering : foundations, principles and techniques*. Springer Science & Business Media, 2005.

[173] Cosmina Cristina Raţiu, Wesley KG Assunção, Rainer Haas, and Alexander Egyed. Reactive links across multi-domain engineering models. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, pages 76–86, 2022.

[174] The Rodin Platform. https://wiki.event-b.org/index.php/Rodin_Platform.

[175] D. Ross and K. Schoman. Structured analysis for requirements definition. *IEEE Transactions on Software Engineering*, 3(1), January 1977.

[176] Rick Salay, Sahar Kokaly, Alessio Di Sandro, Nick LS Fung, and Marsha Chechik. Heterogeneous megamodel management using collection operators. *Software and Systems Modeling*, 19(1) :231–260, 2020.

[177] Rick Salay, John Mylopoulos, and Steve Easterbrook. Using macromodels to manage collections of related models. In *International Conference on Advanced Information Systems Engineering*, pages 141–155. Springer, 2009.

[178] Jean-Philippe Schneider, Joël Champeau, Loic Lagadec, and Eric Senn. Role framework to support collaborative virtual prototyping of system of systems. In *2015 IEEE 24th International Conference on Enabling Technologies : Infrastructure for Collaborative Enterprises*, pages 144–149. IEEE, 2015.

[179] Jean-Philippe Schneider, Joël Champeau, Ciprian Teodorov, Eric Senn, and Loïc Lagadec. A role language to interpret multi-formalism system of systems models. In *2015 Annual IEEE Systems Conference (SysCon) Proceedings*, pages 200–205. IEEE, 2015.

[180] Andy Schürr. Specification of graph translators with triple graph grammars. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163. Springer, 1994.

[181] Andreas Seibel. *Traceability and model management with executable and dynamic hierarchical megamodels*. PhD thesis, Universität Potsdam, 2012.

[182] Andreas Seibel, Stefan Neumann, and Holger Giese. Dynamic hierarchical mega models : comprehensive traceability and its efficient maintenance. *Software & Systems Modeling*, 9(4) :493–528, 2010.

[183] Edwin Seidewitz. What models mean. *IEEE software*, 20(5) :26–32, 2003.

[184] Sagar Sen, Jean-Marie Mottu, Massimo Tisi, and Jordi Cabot. Using models of partial knowledge to test model transformations. In *International Conference on Theory and Practice of Model Transformations*, pages 24–39. Springer, 2012.

[185] Caine Silva, Sylvain Guérin, Raúl Mazo, and Joel Champeau. Contract-based design patterns : A design by contract approach to specify security patterns. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, ARES '20, New York, NY, USA, 2020. Association for Computing Machinery.

[186] Freddy Kamdem Simo. *Model-based federation of systems of modelling*. PhD thesis, Université de Technologie Compiègne (UTC), 2017.

[187] Dan Sperber and Deirdre Wilson. Mutual knowledge and relevance in theories of comprehension. Routledge, 1998.

[188] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. 3 metamodelling. In *Dagstuhl Workshop on Model-Based Engineering of Embedded Real-Time Systems*, pages 57–76. Springer, 2007.

[189] Friedrich Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data & Knowledge Engineering*, 35(1) :83–106, 2000.

[190] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF : Eclipse Modeling Framework*. Pearson Education, 2008.

[191] Perdita Stevens. Bidirectional transformations in the large. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 1–11. IEEE, 2017.

[192] Perdita Stevens. Connecting software build with maintaining consistency between models : Towards sound, optimal, and flexible building from megamodels. *Software and Systems Modeling*, March 2020.

[193] Patrick Stünkel, Harald König, Yngve Lamo, and Adrian Rutle. Comprehensive Systems : A formal foundation for Multi-Model Consistency Management. *Formal Aspects of Computing*, 33(6) :1067–1114, 2021.

[194] Patrick Stünkel, Ole von Bargen, Adrian Rutle, and Yngve Lamo. GraphQL federation : A model-based approach. 2020.

[195] Patrick Stünkel, Harald König, Adrian Rutle, and Yngve Lamo. Multi-model evolution through model repair. *Journal of Object Technology*, 20(1) :1 :1–25, January 2021. Workshop on Models and Evolution (ME 2020).

[196] Tithnara Nicolas Sun, Bastien Drouot, Fahad Golra, Joël Champeau, Sylvain GUERIN, Luka Le Roux, Raúl Mazo, Ciprian Teodorov, Lionel Van Aertryck, and Bernard L' Hostis. A Domain-specific Modeling Framework for Attack Surface Modeling. In *ICISSP 2020 : 6th International Conference on Information Systems Security and Privacy*, volume 1 of *Proceedings of the 6th International Conference on Information Systems Security and Privacy*, pages 341 – 348, Valetta, Malta, February 2020.

[197] Eugene Syriani, Lechanceux Luhunu, and Houari Sahraoui. Systematic mapping study of template-based code generation. *Computer Languages, Systems & Structures*, 52 :43–62, 2018.

[198] Juha-Pekka Tolvanen and Matti Rossi. MetaEdit+ defining and using domain-specific modeling languages and code generators. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 92–93, 2003.

[199] Weslley Torres, Mark van den Brand, and Alexander Serebrenik. Model management tools for models of different domains : a systematic literature review. In *2019 IEEE International Systems Conference (SysCon)*, pages 1–8. IEEE, 2019.

[200] Weslley Torres, Mark GJ Van den Brand, and Alexander Serebrenik. A systematic literature review of cross-domain model consistency checking by model management tools. *Software and Systems Modeling*, pages 1–20, 2020.

[201] Steve Tueno, Marc Frappier, Régine Laleau, Amel Mammar, and Hector Ruiz Barradas. The Generic SysML/KAOS Domain Metamodel. *ArXiv e-prints, cs.SE, 1811.04732*, November 2018.

[202] Steve Tueno, Régine Laleau, Amel Mammar, and Marc Frappier. Towards Using Ontologies for Domain Modeling within the SysML/KAOS Approach. *IEEE proceedings of MoDRE workshop, 25th IEEE International Requirements Engineering Conference*, pages 1–5, 2017.

[203] Dániel Urbán, Gergely Mezei, and Zoltán Theisz. Formalism for static aspects of dynamic metamodeling. *Periodica Polytechnica Electrical Engineering and Computer Science*, 61(1) :34–47, 2017.

[204] Axel van Lamsweerde. *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[205] Andrés Vignaga, Frédéric Jouault, María Cecilia Bastarrica, and Hugo Brunelière. Typing artifacts in megamodeling. *Software & Systems Modeling*, 12(1) :105–119, 2013.

[206] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić. Sirius : A rapid development of DSM graphical editor. In *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*, pages 233–238. IEEE, 2014.

[207] Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. Towards user-friendly projectional editors. In *Software Language Engineering : 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings 7*, pages 41–61. Springer, 2014.

[208] Guido Wachsmuth. Metamodel adaptation and model co-adaptation. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, ECOOP'07, pages 600–624, Berlin, Heidelberg, 2007. Springer-Verlag.

[209] Bernhard Westfechtel. Merging of EMF models. *Software & Systems Modeling*, 13(2) :757–788, 2014.

[210] Dustin Wüest, Norbert Seyff, and Martin Glinz. FlexiSketch : a lightweight sketching and metamodeling approach for end-users. *Software and Systems Modeling*, 18(2) :1513–1541, 2019.

COLLEGES SCIENCES
BRETAGNE POUR L'INGENIEUR
LOIRE ET LE NUMERIQUE

ENSTA BRETAGNE

Titre : FML, un langage de fédération de modèles pour l'interopérabilité sémantique de sources d'information hétérogènes

Mots clés : Ingénierie Dirigée par les Modèles, Fédération de modèles, Métamodélisation

Résumé : La modélisation est une pratique universelle à la base de la pensée humaine, pour concevoir, comprendre, calculer, imaginer, analyser, communiquer. L'Ingénierie Dirigée par les Modèles (IDM) se propose de systématiser l'utilisation de modèles dans toutes les tâches liées au cycle de vie des systèmes et des logiciels (développement, déploiement, intégration, maintenance et évolution). De ce contexte émerge le besoin de gérer en cohérence de multiples modèles (ou des sources d'information interprétées comme des modèles), qui ont chacun leur autonomie et leur propre cycle de vie. Le cœur de la contribution de cette thèse repose sur la formalisation d'une approche que nous nommons "fédération de modèles" et qui se propose de réifier explicitement les liens entre les modèles fédérés et de leur associer un comportement. Plus précisément, nous proposons un langage de modélisation appelé FML, qui est à la fois un langage permettant la conceptualisation et la réification de l'interprétation, mais qui est également doté d'un mécanisme de désignation qui permet l'établissement de liens de fédération vers des sources de données hétérogènes. FML permet, à la fois, de réagir aux comportements et évolutions des modèles fédérés et de programmer des comportements agissant sur ces modèles. Nous proposons un environnement outillé du langage au sein d'une infrastructure logicielle appelée Openflexo. Enfin, nous validons l'approche sur quatre cas d'utilisation, prototypes de recherche et expérimentations industrielles.

Title : FML : a model federation language for semantic interoperability of heterogeneous information sources

Keywords : Model Driven Engineering, Model Federation, Metamodelling

Abstract : Modelling is a universal practice at the root of human thought, for designing, understanding, calculating, imagining, analyzing and communicating. Model Driven Engineering (MDE) aims to systematize the use of models in all activities related to the whole life cycle of systems and software (development, deployment, integration, maintenance and evolution). From this context emerges the need to coherently manage multiple models (or information sources interpreted as models), where each model comes with its own autonomy and lifecycle. The core contribution of this thesis is founded on the formalization of an approach called "model federation", which aims to explicitly reify the links between federated models, while associating behaviour with those links. More specifically, we propose a modelling language called FML, which is both a language for conceptualization and reification of interpretation, but also features a referencing mechanism that enables federation links to point on heterogeneous data sources. FML allows both to react to the behaviour and evolution of federated models and to program behaviours acting on these models. We propose a tooling environment for the language within a software infrastructure called Openflexo. Finally, we validate the approach on four use cases, research prototypes and industrial experiments.