Habilitation à diriger des recherches

Faciliter la vérification et la validation de méta-modèles dans le cadre de l'ingénierie dirigée par les modèles : une approche agile, outillée et orientée données

présentée et soutenue publiquement le 28 Novembre 2012

par

Alain Plantec

Laboratoire en sciences et techniques de l'information, de la communication et de la connaissance

Lab-STICC/CNRS UMR 6285

Composition du jury :

MM.	Yvon	Kermarrec	Président
	Yamine	Ait Ameur	Rapporteurs
	Stéphane	Ducasse	
	Fabrice	Kordon	
	Jean-Philippe	Babau	Examinateurs
	Gaëlle	Calvary	
	Frank	Singhoff	

À mon amour, Bénédicte
À mes enfants adorés, Louise, Martin et Adèle

Je tiens à remercier tout particulièrement

Yamine Ait Ameur, Stéphane Ducasse et Fabrice Kordon pour l'honneur qu'ils me font d'avoir accepté de rapporter ce mémoire.

Jean-Philippe Babau, Gaëlle Calvary, Yvon Kermarrec et Frank Singhoff pour l'honneur qu'ils me font d'avoir accepté d'être membre de mon jury.

À tous les membres du département informatique de l'Université de Bretagne Occidentale, sachez que mes travaux ne seraient pas présentés ici sans votre participation, votre soutient ou tout simplement votre présence au quotidien ; un grand merci.

Merci plus particulièrement à Daniel Priour, Loic Lagadec, Bernard Pottier, Mickaël Kerboeuf, Jean-Philippe Babau, Vincent Ribaud.

Merci aux membres du projet Cheddar, Pierre Dissaux, Jérôme Legrand, Laurent Lemarchand, Stéphane Rubini, Vincent Gaudel ; c'est un plaisir de travailler avec vous !

Résumé

Nous nous situons dans le cadre de la spécification, de la vérification et de la validation de langages et des outils connexes spécifiquement développés pour un domaine et une application cible. L'ingénierie dirigée par les modèles (IDM) et les environnements dédiés apportent une réponse partielle à ces besoins. Les problèmes qui persistent concernent le coût de l'application de l'IDM et l'adaptabilité des environnements.

Nous considérons qu'une partie du problème du coût est très fortement corrélé à l'approche générative prônée par l'IDM. Cependant, le problème doit être envisagé dans sa globalité car le système cible, obtenu après génération de code demeure un élément important notamment pour la validation. Nous étudions notamment les possibilités offertes par les méthodes agiles pour faciliter la vérification et la validation des langages. Nous montrons que la vérification et la validation d'un langage peut s'intégrer dans un processus agile, itératif et incrémental. Chaque itération correspond à un incrément dont il s'agit de réduire le coût. Notre approche consiste à renforcer l'importance du méta-environnement au bénéfice de la validation. Les contributions de nos travaux sont :

- la définition d'une méthode de méta-modélisation basée sur une approche semi-formelle et itérative permettant d'une part, la méta-programmation et la validation précoce, à haut niveau d'abstraction, et d'autre part, la prise en compte des validations opérées au sein du système cible ;
- un étayage concret de la méthode par la mise en œuvre d'un environnement permettant la vérification et la validation précoce de langages mais aussi leur adaptation homogène au domaine et à l'application ; nous exploitons un typage optionnel des méta-modèle ; notre mise en œuvre, nommée Platypus, est basée sur l'implantation d'un environnement de modélisation STEP/EXPRESS dans un système Smalltalk ;
- trois expérimentations dans le cadre de projets significatifs et pour des domaines différents qui montrent la pertinence de l'approche et des outils que nous avons développés.

Mots clés : Ingénierie Dirigée par les Modèles, Langage spécifique au domaine, Vérification, Validation, Prototypage, Langages Dynamiques, Méthodes agiles, Smalltalk, Interopérabilité par les données, Norme STEP.

Table des matières

Table des figures

Première partie

Introduction générale et état de l'art

Chapitre 1

Introduction

Ce chapitre présente mes activités de recherche. La présentation est chronologique depuis mes travaux de thèse. L'objectif est de donner une vue d'ensemble de mon parcours scientifique. Mes travaux de thèse sont tout d'abord brièvement décrits. Ensuite, mes travaux à l'UBO concernant la vérification et la validation de langages sont présentés. Mes publications sont données en fin de chapitre.

1.1 De la formation initiale à la soutenance de thèse

Après un DUT informatique obtenu à Nantes en 1987, j'ai travaillé comme analyste programmeur. J'ai été successivement développeur puis responsable d'application. En 1990, après trois ans d'expérience professionnelle, j'ai repris mes études à Brest. J'ai ainsi pu obtenir une licence puis une maitrise d'informatique en 1992. C'est pendant ma maitrise que j'ai très concrètement pu m'initier à la recherche. J'ai en effet participé à un projet de parallélisation d'*algorithmes génétiques*. Ce projet s'est concrétisé par la présentation d'un poster à OOPSLA'92 [LPPZ92]. Encouragé par cette expérience, j'ai poursuivi par un DEA d'informatique de Rennes I en 1993. Je me suis inscrit en thèse en 1994, bénéficiant d'une convention CIFRE entre Thalès et l'Université de Bretagne Occidentale (UBO). Pendant ma thèse CIFRE, je me suis plus particulièrement intéressé à la modélisation, la méta-modélisation et à la génération de code.

1.1.1 Le contexte et les objectifs de la thèse

Au début des années 90, malgré un important effort en recherche autour des méthodes formelles [KB89], on constate une sous-utilisation de ces dernières dans l'industrie du logi-

ciel. L'utilisation des méthodes de conception se concentre principalement sur la définition et l'exploitation des modèles conceptuels des données. Les ingénieurs savent exploiter ces descriptions au travers d'outils de conception. Ces spécifications sont alors exploitées pour la documentation et la production automatique de squelettes de composants logiciels. Les techniques de génération de code à partir de la description des données manipulées par les composants logiciels à générer sont alors employées en alternative aux approches formelles. On parle alors de méthodes semi-formelles. Les techniques associées à ces méthodes exploitent les informations contenues dans les méta-modèles du langage de description pour la spécification et la génération du code cible [Des94]. L'exemple le plus typique, mais aussi le plus courant dans son utilisation, consiste à générer des classes C++ ou Java à partir d'un diagramme statique des classes OMT ou UML.

Cependant, les capacités des générateurs de codes sont limitées et ces outils demeurent sous-exploités. Une des causes principales est le manque d'adaptation du code généré pour les besoins des projets avec la nécessité d'adapter manuellement les composants générés. En effet, les outils sont alors conçus comme des boites noires intégrées aux environnements, les outils sont difficiles voire impossibles à adapter sans modifier les environnements.

L'enjeu scientifique est de permettre un plus haut niveau d'abstraction pour les développements. La problématique vise alors à répondre à la question suivante : *Comment élever le niveau d'abstraction pour le développement logiciel tout en favorisant l'adaptation des composants logiciels au contexte, leur réutilisation et leur évolution ?* Concrètement, il s'agissait de pouvoir manipuler des spécifications amont de haut niveau d'abstraction, lisibles et compréhensibles par un ingénieur mais aussi pouvant être compilées pour l'automatisation de la production de code.

A cet enjeu scientifique s'ajoute un défi à la foi humain et technique consistant à montrer comment produire une suite d'outils de génie logiciel pour la génération de code pouvant être manipulée par les ingénieurs. La contrainte est donc aussi d'adapter la technologie au savoir faire concret pour favoriser le transfert technologique. En 1997, dans [OFT97], l'OFTA [1] établit des recommandations quant à l'utilisation des méthodes formelles. L'objectif de mes travaux de thèse s'inscrit pour une grande part dans ces recommandations.

1.1.2 Résultat des travaux

1.1.2.1 La méthode *Eugene*

Nous avons développé et valorisé la méthode *Eugene* pour la conception et la mise en œuvre de générateurs de code. La particularité de cette méthode est, pour la spécification, de s'abstraire des notations et des syntaxes concrètes :

- en s'appuyant sur la spécification de modèles de données pour décrire les concepts manipulés par un générateur,
- et en spécifiant, indépendamment de la plateforme d'exécution, les actions sémantiques associées.

La mise en œuvre d'un générateur est alors automatisée par génération de code vers la plateforme d'exécution cible. Un générateur de code automatiquement produit est conçu comme un composant interopérable par les données qui consomme une spécification source constituée d'un lot de données encodé de façon neutre. La réalisation produite constitue soit un lot de données soit une représentation textuelle.

1. Observatoire Français des Techniques Avancées

1.1.2.2 Bilan

Mes travaux ont été valorisés notamment, par une revue internationale [PR96], cinq conférences internationales [PR98a, PR98b, PR99a, PR99b, PR00], trois applications industrielles et enfin cinq projets d'étudiants de Master (un étudiant a été recruté chez Thalès à la suite de son projet).

Les projets industriels ont montrés des gains de différentes natures. Ces gains sont liés d'une part à la génération de code (quantité, complexité et qualité du code généré), d'autre part à l'interopérabilité des outils inhérente au paradigme de l'orienté données et enfin à la simplicité de la méthode qui permet l'intégration de l'activité de conception et de mise en œuvre de générateurs de code dans le cycle de développement d'un projet. Cependant des problèmes subsistent, notamment en ce qui concerne la validation et la maintenance des générateurs :

1. *validation des générateurs* : même si la méthode *Eugene* a pu être intégrée dans le cycle de développement de projets, il est par ailleurs observé qu'il est difficile de développer des générateurs de code spécifiques à une application en faisant l'économie de développements manuels de composants exemples ; en effet, il est nécessaire d'acquérir une bonne expertise concernant le système cible avant de développer un générateur de code [Cle88] ;
2. *maintenance et évolution des systèmes* : il se pose le problème de la maintenance et de l'évolution des systèmes [MWD+05], en effet, les bénéfices ont été observés pour la production de la première version des projets ; par contre, les problèmes se posent au niveau de la maintenance car la génération de code implique plus de difficultés pour la gestion des versions des méta-modèles et pour la maitrise des configurations, de la coévolution et de la validation des systèmes [Ams10].

J'ai soutenu ma thèse en 1999 et j'ai été recruté en septembre 2000 comme Maître de Conférence au département d'informatique de l'UBO. J'ai alors orienté mes travaux autour de l'application des méthodes agiles dans le contexte de l'ingénierie dirigée par les modèles.

1.2 Mes travaux de recherche à l'UBO depuis 2000

Le défi est de répondre aux problèmes de la validation et de la maintenance des générateurs et plus généralement des langages sous-jacents, en agissant sur les deux leviers de la méthode et des environnements logiciels utilisés pour spécifier les langages :

- du point de vue de la méthode, la spécification et la validation d'un langage procède de son élaboration par itérations et incréments ; l'adaptation et la justesse par rapport au domaine et à l'application est obtenue progressivement par des mises à l'épreuve les plus réalistes et complètes possibles ;
- du point de vue des environnements, pour la spécification et la validation d'un langage spécifique à un domaine, un haut niveau d'abstraction est primordial, par contre, l'environnement logiciel doit aussi être conçus et évalué en tenant compte des besoins fonctionnels et des contraintes non-fonctionnelles (consommation de ressource, temps d'exécution, ...) du système cible ; le problème est de pouvoir converger vers une solution satisfaisante, en agissant à haut niveau d'abstraction tout en tenant compte du système cible réellement exécuté.

Les expérimentations sont menées dans le cadre de projets ou de contrats. Elles sont basées sur la mise en œuvre semi-automatique d'ateliers spécialisés dont les projets en bénéficient directement. Comme dans [Kli93], nous considérons qu'un atelier est un ensemble cohérent d'outils pour un domaine d'application particulier. Les outils produits peuvent être alors des éditeurs de modèles, des environnements de test ou encore des interpréteurs ou des générateurs de code [WWM+07]. Un tel atelier doit non seulement permettre la génération de code pour une application cible mais aussi faciliter l'évolution et la validation des systèmes produits [Ams10]. Un méta-atelier est un outil pour la spécification et l'instrumentation d'un langage permettant de produire un atelier spécialisé pour le langage.

1.2.1 Problématique

Nous nous situons dans le cadre de la spécification, de la vérification et de la validation d'un langage et des outils connexes pour un domaine et une application cible. Tout concepteur qui doit résoudre un tel problème désire plus particulièrement :

- *agir au maximum à haut niveau d'abstraction* : le concepteur désire spécifier et valider son langage par la manipulation de concepts et de règles indépendamment du système cible et de sa plateforme d'exécution ;
- *vérifier et valider :* en agissant au niveau des spécifications (analyse des types, méthodes formelles,...) et par tests dynamiques pour mettre à l'épreuve le langage par des expérimentations ;
- *minimiser les coûts et les délais* : la contrainte est forte pour le passage à l'échelle, une méthode et l'outillage associé ne sont pas utilisés ou ne sont que sous-utilisés si leurs mises en application sont perçues comme trop coûteuses ;
- *disposer d'un outillage adapté et adaptable* : pour un domaine ou une application particulière, l'outillage doit pouvoir être adapté ou même construit spécifiquement pour répondre au plus près aux besoins de vérification et de validation.

L'ingénierie dirigée par les modèles (IDM) et les environnements dédiés apportent une réponse partielle à ces besoins. L'IDM permet de bénéficier d'un haut niveau d'abstraction pour les spécifications. L'IDM permet de vérifier les spécifications statiquement ou en exploitant des méthodes formelles. L'IDM permet d'expérimenter et de mettre à l'épreuve les langages notamment par une approche générative. Les problèmes qui persistent se résument en deux points : le premier point concerne le coût de l'application de l'IDM et le second point concerne l'adaptabilité des environnements.

1.2.1.1 Problème du coût

Le typage autorise de manière efficace certaines vérifications précoces des spécifications sources. Les méthodes formelles sont plus difficiles à appliquer et aussi plus coûteuses. Pour accroitre les capacités en termes de validation, il est bénéfique de disposer de spécifications sources exécutables. Dans le cadre de l'IDM, un méta-modèle exécutable résulte de la traduction d'une spécification qui elle n'est pas exécutable. Les spécifications sources sont transformées en méta-modèles exécutables par génération de code pour la plateforme d'exécution cible. C'est l'expérimentation qui, par l'utilisation du système cible, permet au concepteur d'apprécier la justesse et la conformité par rapport aux besoins. De fait, une grande partie

de la vérification et de la validation est effectuée au sein du système cible, après les transformations des spécifications et la génération de code. Le coût d'une itération peut être alors conséquent. [2]

1.2.1.2 Adaptabilité des environnements

La grande force des environnements pour l'IDM concerne essentiellement les possibilités offertes pour la spécification et la mise en œuvre des transformations de modèles. Par contre, pour la vérification et la validation, l'ensemble des outils disponibles est prédéfini. Il est difficile de prévoir le besoin précis en termes d'outillage nécessaire pour la spécification et la validation d'un langage. Un environnement doit pouvoir être adapté pour correspondre au besoin en termes de vérification et de validation.

1.2.2 Approche et contributions

Nous considérons que le problème du coût est très fortement corrélé à l'approche générative. Cependant, le problème doit être envisagé dans sa globalité car le système cible, obtenu après génération de code demeure un élément important notamment pour la validation.

Nous considérons que la vérification et la validation d'un langage s'intègre dans un processus itératif. Chaque itération correspond à un incrément dont il s'agit de réduire le coût. Notre approche consiste à renforcer l'importance du méta-atelier au bénéfice de la validation :

- les spécifications sources sont exécutables à haut niveau d'abstraction au sein du méta-atelier ce qui permet non seulement de vérifier mais aussi de valider le langage (au moins partiellement) par le biais de prototypes, d'animations des modèles ou de simulations ;
- il est possible d'adapter le méta-atelier au besoin en construisant des outils de vérification et de validation spécifiquement adaptés au domaine et à l'application avant la génération de code ;
- après génération de code, la validation est renforcée par exécution du système produit au sein de la plateforme d'exécution cible et les mises à l'épreuve réalistes effectuées dans la plateforme d'exécution cible sont bénéfiques pour la validation effectuée dans le méta-atelier.

Les contributions de nos travaux sont :

- la définition d'une méthode de méta-modélisation autorisant des validations précoces, au sein d'un méta-atelier, pour la production semi-automatique d'un système cible ; la méthode est basée sur une approche semi-formelle et itérative permettant d'une part, la méta-programmation à haut niveau d'abstraction et d'autre part, le prise en compte des validations opérées au sein du système cible ;
- un étayage concret de la méthode par la mise en œuvre d'un environnement permettant la vérification et la validation précoce d'un langage mais aussi une adaptation

2. Par exemple, en 2002, j'ai initié le projet *Eugene2* avec comme base d'expérimentation, un méta-atelier basé sur l'orienté donnée pour la méta-modélisation. Il s'agit de permettre la gestion de versions spécialisées d'environnements pour l'édition, la validation et l'échange de modèles. Une version particulière est basée sur un méta-modèle domaine. Les versions successives sont produites par générations de code. J'ai développé un noyau pour *eugene2* permettant la gestion d'un repository de méta-modèles. Ce noyau développé en C++ comporte 300 classes et environ 150000 lignes de code. 90% est généré automatiquement à partir d'un méta-modèle de 1686 lignes. L'effet de levier permis par la génération de code a donc été très bénéfique pour la production de la première version du noyau. Par contre, pour chaque modification de la spécification source, afin de pouvoir tester et évaluer une version de ce noyau, 15 minutes de compilation sont nécessaires.

homogène au domaine et à l'application permettant la réutilisation cohérente des outils (analyseurs, interpréteurs, gestionnaire de version, ...) ;

– trois expérimentations dans le cadre de projets significatifs et pour des domaines différents qui montrent la pertinence de l'approche et des outils que nous avons développés.

1.2.2.1 La méthode de méta-modélisation Platypus

Nous proposons la méthode Platypus qui vise à faciliter la spécification, la vérification et la validation d'un langage par la construction d'un atelier spécialisé. La finalité est de maintenir un système cible construit à partir d'un atelier. Aucune hypothèse n'est faite concernant la plateforme d'exécution du système cible. La méthode est défini suivant le point de vue de la construction de l'atelier spécialisé et suivant les relations établies entre l'atelier et le système cible.

Construction de l'atelier. Notre objectif est de permettre la mise au point progressive et itérative de méta-modèles en autorisant un processus naturel d'élaboration d'une conception. Afin de bénéficier pleinement des apports de l'IDM tout en facilitant la vérification et la validation des évolutions, Platypus est schématiquement basé sur un principe de méta-modélisation suivant deux niveaux complémentaires :

– un niveau *prototypage*, pour la définition des outils spécifiques nécessaires à la méta-modélisation et pour la manipulation de méta-modèles exécutables à haut niveau d'abstraction,

– un niveau *typage*, pour affiner la vérification et la validation au regard du système cible et pour la transformation des modèles.

Ces deux niveaux sont complémentaires. Ils permettent d'élargir l'éventail des utilisations possibles de la méta-modélisation et favorisent l'instrumentation précoce des méta-modèles.

Comme le montre la figure 1.1, le niveau d'utilisation dit de *prototypage* permet la construction de prototypes, de simulateurs ou d'animations d'un méta-modèle. Le niveau d'abstraction est élevé et la déclaration des types du méta-modèle est optionnelle.

Une fois qu'un méta-modèle est considéré comme conforme au besoin, une définition précise de ces éléments peut être implantée pour enrichir sa spécification. Un méta-modèle fortement typé peut alors être spécifié. Cette spécification consiste à enrichir le méta-modèle déjà manipulé à haut niveau d'abstraction. L'instrumentation peut alors exploiter la définition des types pour compléter la vérification et la validation.

Intégration entre un atelier et un système cible. Pour aider à vérifier que les besoins fonctionnels et que les contraintes non fonctionnelles sont réellement satisfaites, le système cible est un élément fondamental. Au niveau de l'atelier, il est possible d'exploiter les informations issues du fonctionnement du système cible. L'intégration entre l'atelier et le système cible est pour cela un point essentiel. Cette intégration doit notamment permettre l'échange ou le partage de modèles conformes. Les modèles conformes en provenance du système cible permettent la mise à l'épreuve réaliste de l'atelier et du langage sous-jacent. Pour l'intégration entre un atelier et le système cible, une approche par l'orienté donnée est exploitée.

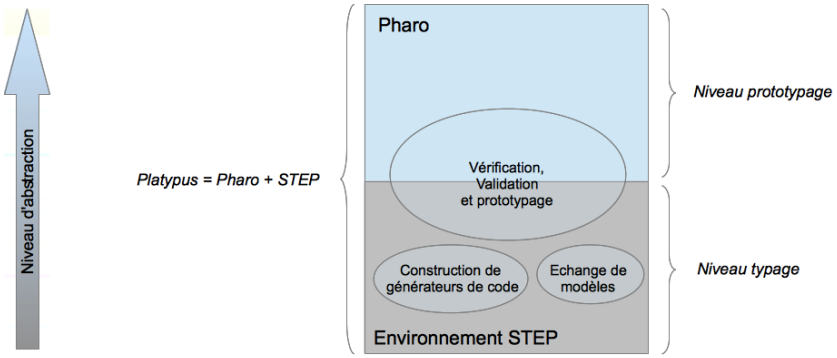

FIGURE 1.1 – Les niveaux prototypage et typage dans Platypus

1.2.2.2 Un méta-atelier adapté à l'IDM

Nous avons développé le méta-atelier Platypus pour supporter la méthode de spécification et de validation de langages. Le processus d'utilisation de ce méta-atelier permet la construction d'ateliers spécialisés pour un domaine et une application. Nous avons conçu Platypus suivant les fondamentaux techniques suivants :

- *Environnement hôte dynamique :* le méta-atelier et les ateliers spécialisés sont intégrés à un environnement hôte permettant l'exécution des méta-modèles ;
- *Interactivité et maintient dynamique du lien de causalité au sein du méta-atelier :* l'interactivité du méta-atelier est un point fondamental pour l'agilité du processus ; Notamment, une modification opérée sur un méta-modèle doit entrainer automatiquement et dynamiquement une adaptation des modèles conformes ;
- *Réutilisation :* un atelier spécialisé pour un domaine particulier est produit par spécialisation du méta-atelier dans l'environnement hôte ; le rôle de l'atelier est de permettre l'élaboration d'un méta-modèle, de permettre de maximiser la validation pour un domaine particulier et de produire et maintenir une application cible ;
- *homogénéité :* tous les outils disponibles au niveau du méta-atelier, y compris l'analyseur de méta-modèles et celui permettant d'instancier les modèles conformes, l'interpréteur, le gestionnaire de version, ..., sont réutilisables de manière homogène ;
- *Intégration :* aucune hypothèse n'est effectuée concernant la plateforme d'exécution de l'application cible cependant, pour une application particulière, le méta-atelier, l'atelier et le système cible doivent être finement intégrés ; notamment, au niveau de l'atelier, le retour d'informations en provenance du système doit pouvoir être instrumenté.

La figure 1.2 montre les éléments importants de Platypus et l'articulation entre un système cible et Platypus. Pour un domaine particulier, un atelier spécialisé est construit par méta-modélisation. Classiquement, les éléments constitutifs de l'atelier s'articulent autour du ou des méta-modèles. Ces éléments comprennent principalement :

- un composant d'édition de modèles et
- des composants pour la validation et l'échange de modèles.

La plateforme d'exécution de Platypus et des ateliers conçus dans Platypus est Pharo [Pha], un système Smalltalk libre. La technologie STEP [ISO94a] est exploitée comme méthode de description et de mise en œuvre des méta-modèles.

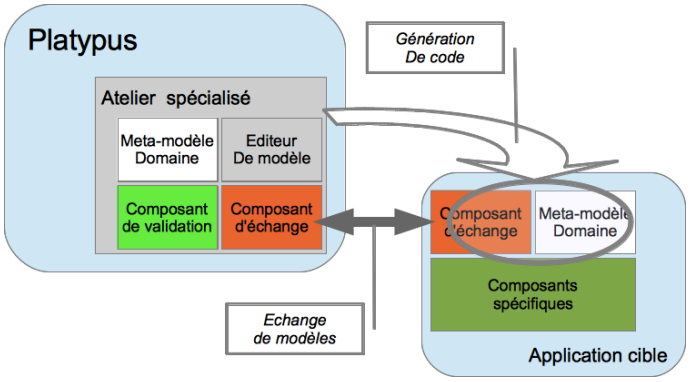

FIGURE 1.2 – Platypus : construction d'ateliers spécialisés

À partir d'un atelier, des générateurs de code sont mis en œuvre pour produire une partie d'une application cible. Pour permettre l'intégration entre le système cible et l'atelier, les composants générés comprennent au minimum :

- un gestionnaire de modèles conformes aux méta-modèles du domaine,
- un composant d'échange de modèles conformes.

1.3 Conclusion

L'IDM apporte des méthodes et des outils pour la spécification, la vérification et la validation de langages et des outils connexes pour des domaines et des applications cibles. L'IDM permet d'agir à haut niveau d'abstraction et de spécifier des transformations de modèles permettant de produire tout ou partie d'un système. L'approche est générative et les concepteurs bénéficient d'environnements puissants pour la transformation des modèles. Un système est spécifié à haut niveau d'abstraction et le système exécutable est obtenu par transformation des spécifications sources. Mes travaux de thèse constituent une contribution au domaine de la construction des générateurs de code. Par contre, nous avons compris que l'approche générative n'apporte pas de réponse satisfaisante aux problèmes du coût et des délais imposés par le contexte industriel. De plus, les environnements pour l'IDM sont rigidifiés et faiblement adaptables aux besoins spécifiques liés au domaine et à l'application cible.

Nous proposons une méthode et un environnement appelés Platypus. La finalité est de permettre la mise au point itérative des méta-modèles, leur vérification et leur validation progressive en autorisant un processus naturel d'élaboration d'une conception. Platypus permet de bénéficier des apports de l'IDM tout en facilitant la vérification et la validation des évolutions à haut niveau d'abstraction. Platypus permet d'élargir l'éventail des possibilités pour la vérification et la validation d'un langage en permettant l'instrumentation précoce des méta-modèles et l'exploitation des retours d'informations obtenus par exécution du système cible.

Depuis 2005, *Platypus* a été directement valorisé par quatre revues internationales [SPDL09, GSP+11, SP07, PS06], une revue nationale [PS07], neuf conférences internationales [PSGR11, DLP+11, DLP+10, PSDL10, SP07, SPD08, PR06b, PR06a, BLPL07].

Platypus a été appliqué dans le cadre de trois projets importants dont deux Européens :

– Dans *Premecs II*, un projet Européen de l'Ifremer, *Platypus* a été utilisé pour la spécification et le prototypage d'un langage permettant la définition de la structure et des propriétés de chaluts ;

– Dans le cadre du projet Européen *Morpheus*, *Madeo*, est un environnement pour la spécification, la validation et l'implantation de circuits reconfigurables, *Platypus* a été utilisé pour la ré-ingénierie du langage de spécification des circuits et la synthèse automatique des composants d'échange des circuits ;

– Dans le cadre du projet *Cheddar*, l'environnement *Cheddar* permet la validation de l'ordonnancement des architectures temps-réel ; *Platypus* est utilisé pour la production semi-automatique de *Cheddar* et est actuellement au cœur d'un contrat de transfert technologique avec la société Ellidiss Technologies ; ce contrat a permis notamment le cofinancement d'une thèse régionale.

1.4 Publications

Revue internationales à comité de sélection

[GSP+11] V. Gaudel, F. Singhoff, A. Plantec, S. Rubini, P. Dissaux, and J. Legrand. An ada design pattern recognition tool for aadl performance analysis. *Ada Lett.*, 31(3) :61–68, Nov. 2011.

[SPDL09] F. Singhoff, A. Plantec, P. Dissaux, and J. Legrand. Investigating the usability of real-time scheduling theory with the cheddar project. *Journal of Real Time Systems. Volume 3. Number 43. Springer Verlag. ISSN :0922-6443*, pp. 259–295, Nov. 2009.

[DPP+09] J. Delange, L. Pautet, A. Plantec, M. Kerboeuf, F. Singhoff, and F. Kordon. Validate, simulate and implement arinc653 systems using the aadl. *ACM SIGAda Ada Letters. Volume 29. Number 3, Pages 31-44. ISSN :1094-3641.*, Nov. 2009.

[SP07] F. Singhoff and A. Plantec. Aadl modeling and analysis of hierarchical schedulers. *ACM SIGAda Letters, volume 27, number 3, pages 41-50, ISSN :1094-3641.*, Nov. 2007.

[PS06] A. Plantec and F. Singhoff. Refactoring of an ada 95 library with a meta case tool. *ACM SIGAda Ada Letters, volume 26, number 3, pages 61-70, ISSN :1094-3641.*, Nov. 2006.

[PR96] A. Plantec and V. Ribaud. Data management : From express schemata to user interface. *Journal of Computing and Information, Volume 2, Number 1, pages 1203-1224, ISSN :1201-8511*, Nov. 1996.

Revues nationales à comité de sélection

[PS07] A. Plantec and F. Singhoff. Un processus d'ingénierie de cheddar pour la simulation de systèmes temps réel à grande échelle. *Revue Génie Logiciel, numéro 83, pages 26-35. ISSN :0295-6322*, Dec. 2007.

Conférences internationales avec actes et comité de sélection

[CSR+12] J. P. Craveiro, J. L. R. Souza, J. Rufino, V. Gaudel, L. Lemarchand, A. Plantec, S. Rubini, and F. Singhoff. Scheduling analysis principles and tool for time and space partitioned systems. In *INFORUM 2012 symposium - Simpósio de Informática*, pp. 582–585, Lisbone, Portugal, September 2012.

[PSGR11] A. Plantec, F. Singhoff, V. Gaudel, and V. Ribaud. Forward engineering and early model validation with Smalltalk. In *Smalltalk 2011 conference, Research track*, Nov. 2011.

[LPPB11] M. Louvel, J. Pulou, A. Plantec, and J.-P. Babau. Ensuring qos of multimedia applications in heterogeneous home networks : the cpu use case. In *IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC)*, pp. 19–26. IEEE, October 2011.

[DLP+11] P. Dissaux, J. Legrand, A. Plantec, V. Gaudel, S. Rubini, and F. Singhoff. AADL real-time design-pattern automatic recognition. In *AeroTech 2011 Proceedings* , pp. Oct. 2011.

[LBPB11] M. Louvel, P. Bonhomme, A. Plantec, and J.-P. Babau. A network resource management framework for multimedia applications distributed in heterogeneous home networks. In *Proceedings of the Advanced Information Networking and Applications (AINA), ISBN : 978-1-61284-313-1*, pp. 724–731, march 2011.

[PSDL10] A. Plantec, F. Singhoff, P. Dissaux, and J. Legrand. Enforcing applicability of Real-time Scheduling Theory Feasibility Tests with the use of Design-Patterns. In L. N. on Computer Science/LNCS, editor, *4th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2010), Amirandes, Heraklion, Greece*, volume 6415, pp. 4–17, October 2010.

[KPS+10] M. Kerboeuf, A. Plantec, F. Singhoff, A. Schach, and P. Dissaux. Comparison of six ways to extend the scope of cheddar to aadl v2 with osate. In *5th international workshop on AADL and UML. In the proceedings of the 15th IEEE International Conference on Engineering of Complex Computer Systems*, pp. 367–372, 2010.

[LPPB10] M. Louvel, J. Pulou, A. Plantec, and J.-P. Babau. Quantity of resource aggregation for heterogeneous resource reservation for multimedia applications. In *Work In Progess session of the IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), ISBN : 978-1-4244-6848-5*, pp. 1–4, 2010.

[DLP+10] P. Dissaux, J. Legrand, A. Plantec, M. Kerboeuf, and F. Singhoff. AADL design-pattern and tools for modelling and performance analysis of real-time systems. In *5th European congress ERTSS, Embedded Real-Time Software and System*, May 2010.

[PRV09] A. Plantec, V. Ribaud, and V. Varma. Building a semantic virtual museum : from wiki to semantic wiki using named entity recognition. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pp. 769–770, New York, NY, USA, 2009. ACM.

[SPD08] F. Singhoff, A. Plantec, and P. Dissaux. Can we increase the usability of real time scheduling theory ? the cheddar project. In *Proceedings of the 13th Ada-*

Europe international conference on Reliable Software Technologies, Ada-Europe '08, pp. 240–253, Berlin, Heidelberg, 2008. Springer-Verlag.

[BLPL07] J. Boukhobza, L. Lagadec, A. Plantec, and J.-C. Lelann. Cdfg platform in morpheus. In *The First AETHER - MORPHEUS Workshop- Autumn School From Reconfigurable to Self - Adaptive Computing (AMWAS'07)*, October 2007.

[SP07] F. Singhoff and A. Plantec. Towards user-level extensibility of an ada library : an experiment with cheddar. In *Proceedings of the 12th international conference on Reliable software technologies*, Ada-Europe'07, pp. 180–191, Berlin, Heidelberg, 2007. Springer-Verlag.

[PR06a] A. Plantec and V. Ribaud. PLATYPUS : A STEP-based Integration Framework. In *14th Interdisciplinary Information Management Talks (IDIMT-2006)*, pp. 261–274, République Tchèque, Sept. 2006.

[PR06b] A. Plantec and V. Ribaud. Reengineering of a data processing line with platypus. In *DSPD06 : Proceedings of the 1st Workshop on Domain-Specific Program Development*, July 2006.

[RSP03] V. Ribaud, P. Saliou, and A. Plantec. Database access control within a java application. In *International Workshop on Encapsulation and Access Rights in Object-Oriented Design and Programming, WEAR 2003 OOIS 2003*, pp. 63–69, Sept. 2003.

[SPR02] P. Saliou, A. Plantec, and V. Ribaud. Metaprogramming with express and sql. In *International Workshop Declarative Meta Programming, DMP'02. University of Edinburgh*, Dec. 2002.

[PR00] A. Plantec and V. Ribaud. Step-based case tools cooperation. In *ICSE'00 International Workshop on Constructing Software Engineering Tools (COSET'00)*, June 2000.

[PR99a] A. Plantec and V. Ribaud. Experiences using an application generator builder. In *12th IEEE International Conference on Software Engineering and Knowlege Engineering (SEKE'99)*, June 1999.

[PR99b] A. Plantec and V. Ribaud. Using and re-using application generators. In *ICSE'99 International Workshop on Constructing Software Engineering Tools (COSET'99)*, May 1999.

[PR98a] A. Plantec and V. Ribaud. Eugene : a step-based framework to build application generators. In *First Australian Workshop on Constructing Software Engineering Tools (AWCSET'98)*, Nov. 1998.

[PR98b] A. Plantec and V. Ribaud. The step standard as an approach for design and prototyping. In *9th IEEE International Workshop on Rapid System Prototyping (RSP'98)*, June 1998.

[LPPZ92] L. Lemarchand, A. Plantec, B. Pottier, and S. Zanati. An object-oriented environment for specification and concurrent execution of genetic algorithms (poster). *SIGPLAN OOPS Mess.*, 4 :163–165, December 1992.

Conférences nationales avec actes et comité de sélection

[GSP+11] V. Gaudel, F. Singhoff, A. Plantec, P. Dissaux, and J. Legrand. Sélection automatique de tests de faisabilité à l'aide de patrons de conception. In *Actes de l'École d'été temps réel 2011, Université de Bretagne Occidentale*, Aug. 2011.

[KPB11] M. Kerboeuf, A. Plantec, and J.-p. Babau. An experiment of a mde approach for the design of reusable dsl tools. In *journées sur l'Ingénierie Dirigée par les Modèles (IDM)*, June 2011.

[KPR07] M. Kerboeuf, A. Plantec, and V. Ribaud. Motif pour la métamodélisation : Flot de contrôle. In *Atelier de IDM07 sur les motifs de métamodélisation*, Jan. 2007.

[PKR07] A. Plantec, M. Kerboeuf, and V. Ribaud. Motif pour la métamodélisation : Commentaire attribué. In *Atelier de IDM07 sur les motifs de métamodélisation*, Jan. 2007.

[PR06] A. Plantec and V. Ribaud. Motif pour la métamodélisation : Interfa cage entre contextes. In *Premier atelier de travail sur les motifs de métamodélisation, dans les actes des 2èmes journées sur l'Ingénierie Dirigée par les Modèles*, June 2006.

[PR05] A. Plantec and V. Ribaud. Un procédé de validation des métamodèles par les métadonnées. In *1ères journées sur l'Ingénierie Dirigée par les Modèles*, June 2005.

Séminaires et présentations diverses

[GSP+11] V. Gaudel, F. Singhoff, A. Plantec, S. Rubini, P. Dissaux, and J. Legrand. Automatic selection of feasibility tests with the use of aadl design patterns. In *AADL SAE working group meeting*, 2011.

[Pla11] A. Plantec. Élaboration de méta-modèles avec platypus. In *Séminaire IDL EA3883*, 2011.

[PGR+11] A. Plantec, V. Gaudel, S. Rubini, F. Singhoff, P. Dissaux, and J. Legrand. Automatically adapt cheddar to users need. In *AADL SAE working group meeting*, 2011.

[DLPS08] P. Dissaux, J. Legrand, A. Plantec, and F. Singhoff. Aadl performance analysis with cheddar : a summary. In *AADL SAE working group meeting*, Apr. 2008.

[Pla07a] A. Plantec. Construction d'environnement sous squeak avec platypus. In *Séminaire LISYC EA3883*, 2007.

[SP07b] F. Singhoff and A. Plantec. A propos de l'applicabilité de la théorie de l'ordonnancement temps réel : le projet cheddar. In *Séminaire de l'IRIT.*, Nov. 2007.

[PSK07] A. Plantec, F. Singhoff, and M. Kerboeuf. Une expérience de ré ingénierie d'une application avec platypus. In *Séminaire chez Altran*, 2007.

[SP07a] F. Singhoff and A. Plantec. Implementing tools for the cheddar programming language with platypus. In *LISYC - Journées de l'Aber-Wrac'h EA3883*, June 2007.

[Pla07b] A. Plantec. Platypus, un méta-environnement orienté donnée. Action IDM : Journée Outils pour l'IDM, January 2007.

[Pla06] A. Plantec. Méta-programmation dans platypus. In *LISYC - Journées de l'Aber-Wrac'h EA3883*, 2006.

Délivrables de projet ou de contrat de recherche

[PTV+07] E. M. Panainte, F. Thoma, N. Voros, B. Pottier, L. Lagadec, J. Boukhobza, A. Plantec, J. C. L. Lann, R. Taylor, A. Grasset, and P. Millet. Morpheus project, toolset modules report (d2.3.1). 2007.

[Pla05] A. Plantec. Utilisation du méta-environnement platypus pour la réingéniérie de phobos. Technical report, Rapport de recherche, Délivrable contractuel, Contrat IFREMER/ – EA3883 (LISYC), Nov. 2005.

Mémoires et rapports techniques

[Pla04] A. Plantec. Tamaris : an object explorer for squeak. Technical report, Equipe IDM, LISyC, EA3883, March 2004.

[DP02] C. Dezan and A. Plantec. Simulation distribuée de systèmes vhdl : étude de cas. Technical report, Laboratoire Architectures & Systèmes, UFR Sciences, Université de Bretagne Occidentale, October 2002.

[Pla01] A. Plantec. expgensh : génération de code c++ et smalltalk à partir d'express. Technical report, Laboratoire Architectures & Systèmes, UFR Sciences, Université de Bretagne Occidentale, November 2001.

[RRP+01] H. Roussain, J. Regnault, B. Pottier, C. Dezan, A. Plantec, and L. Lemarchand. Simulation distribuée de systèmes vhdl. Technical report, Laboratoire Architectures & Systèmes, UFR Sciences, Université de Bretagne Occidentale, September 2001.

[Pla00] A. Plantec. La norme step (standard iso 10303). Technical report, Laboratoire Architectures & Systèmes, UFR Sciences, Université de Bretagne Occidentale, February 2000.

[Pla99] A. Plantec. *Exploitation de la norme STEP pour la spécification et la mise en œuvre de générateurs de code.* PhD thesis, Université de Rennes I, 35065 Rennes cedex, France, 1999.

[SYS97] SYSECA. L'application rafos. Technical report, SYSECA, 32 quaie de la douane, 29200, Brest, France, 1997.

[Pla97] A. Plantec. *daigen* : un outil de production de code à partir d'EXPRESS. Technical report, LIBr, Université de Bretagne Occidentale, Brest and SYSECA, 34 quai de la douane, Brest, France, 1997.

[Pla96a] A. Plantec. Génération de code à partir d'EXPRESS. Technical report, LIBr, Université de Bretagne Occidentale, Brest and SYSECA, 34 quai de la douane, Brest, France, 1996.

[Pla96b] A. Plantec. Utilisation de la norme step pour la mise en œuvre de la structure d'accueil VITAC. *Le bulletin technique,* SYSECA*, 66, rue Brossolette, 92240, Malakoff, France, 4(3)*, October 1996.

Chapitre 2

Une terminologie documentée de l'ingénierie dirigée par les modèles

Depuis la fin des années 1990 et les travaux de l'OMG autour du MDA [OMG03], la communauté scientifique considère l'*Ingénierie Dirigée par les Modèles* [Sch06] (IDM) comme un des fondements favorables à l'optimisation de la qualité et à la rationalisation des coûts. L'IDM est une approche pour le développement logiciel considérant le modèle comme l'élément pivot du processus. De nombreux travaux ont abouti à la clarification des principes fondamentaux de l'IDM [Bé05, Ken02, FEBF06, Com08, Fav04] :

- *haut niveau d'abstraction* : l'IDM permet de se concentrer en premier lieu sur le domaine et les fonctions d'un système tout en s'abstrayant des solutions techniques tout au long du cycle de vie ; le savoir faire en termes de représentation de l'information et de processus est capitalisé sous la forme de modèles ;
- *séparation des préoccupations* :

  - différenciation temporelle : tout au long du cycle de vie, une application est représentée par différents modèles (exigences, cas d'utilisation, conception abstraite et détaillée, programme, tests, anomalies, ...)
  - différenciation fonctionnelle : pour un même niveau dans le cycle de vie, suivant les caractéristiques fonctionnelles de l'application à développer, différents modèles spécifiques sont élaborés (gestion des données, de la sécurité, des interactions Homme-Machine, ...) ;
- *réutilisation et raffinement des modèles* : les différents points de vue modélisés peuvent être combinés et transformés en d'autre modèles représentant des points de vue différents ;
- *automatisation de la mise en œuvre* : les applications sont tout d'abord spécifiées indépendamment des plates-formes et des systèmes cibles ; les vues de haut niveau d'abstraction peuvent être combinées et raffinées par étapes successives pour obtenir des représentations de plus faible niveau d'abstraction ; l'objectif est d'aboutir à une réalisation exécutable via un processus de transformation automatisé.

Ce chapitre a pour objectif de présenter les contours de mon activité de recherche liés à l'IDM en introduisant le vocabulaire et les différents concepts qui sous-tendent mes travaux et sur lesquels je m'appuie dans toute la suite de ce rapport. Ce chapitre est donc à lire comme une présentation bibliographique des points clés de mon domaine de recherche.

Dans la première partie, les notions de modèle et de relations entre modèles sont brièvement introduites en s'appuyant classiquement sur le concept de *représentation*. Ensuite, les notions de langage et sémantique de langage sont introduites. En fin de chapitre les activités de modélisation et de méta-modélisation sont abordées avec la description d'outils utilisés pour ces activités.

2.1 Notions de modèle

Ce domaine de recherche s'articule donc autour de la notion de modèle. Les auteurs s'accordent à dire qu'il n'y a pas de définition claire de la notion de modèle. Suivant sa nature et son utilisation, un modèle est la représentation, la description ou la spécification d'un système. On parle aussi de *modèle conceptuel*, définit comme un ensemble comprenant les informations, les hypothèses de modélisation et les fonctions mathématiques qui décrivent un système ou un processus. Un *système* se définit de façon très générale comme un ensemble d'éléments en interaction [FEBF06].

2.1.1 Caractéristiques désirables d'un modèle

Dans [MFBC10], neuf définitions du concept de *modèle* sont données. De façon très générale, ces définitions considèrent le modèle comme une représentation. En toute généralité, nous dirons qu'un *modèle* est un *système* constituant une représentation simplificatrice d'un autre *système*. Les systèmes sont donc associés par une relation de *représentation* et le modèle constitue l'élément source de cette relation [Kü05, Bé05].

Un modèle est ainsi construit pour comprendre le système qu'il représente [BG01] et doit pouvoir être utilisé pour répondre à des questions sur le système modélisé [Kü05]. La caractérisation de la notion de modèle ne fait pas l'objet d'un consensus [MFBC10]. Nous retenons celle de Kühne [Kü06] qui indique qu'un modèle doit respecter les trois caractéristiques suivantes :

– un modèle est basé sur un sujet qui est le système que la représentation simplifie ;
– un modèle est une réduction ou une simplification de son sujet puisque certains aspects du système sont omis ;
– un modèle doit être utilisable comme représentant du système.

Par exemple, le cycle de vie du logiciel se constitue de plusieurs étapes. Chaque étape a pour objectif de produire un ou plusieurs modèles. Un modèle produit à une étape N permet de décrire ce qui doit être produit à l'étape $N+1$. En début de cycle de vie, l'analyse des exigences vise à produire la spécification des besoins. Ce modèle décrit ce qui doit être produit à l'étape de spécification fonctionnelle de l'application. Dans ce cas, le sujet sur lequel est basée la spécification des besoins est la spécification fonctionnelle. Il s'agit bien d'une réduction qui sert de base de raisonnement à l'ingénieur pour concevoir les fonctionnalités à développer.

2.1.2 La relation de représentation

La relation de *représentation* n'établit aucune hypothèse quant à la préexistence de l'une ou l'autre extrémité de la relation. Deux cas sont donc possibles :

– le modèle peut être conçu pour capturer certaines caractéristiques d'un système préexistant ; la contrainte est alors pour le modèle d'être conforme aux caractéristiques du système ;
– le système peut être construit à partir du modèle et c'est le système qui doit être conforme au modèle ;

Si le système préexiste alors il est *décrit* par le ou les modèles alors que si le modèle préexiste alors le système est *spécifié* par le ou les modèles [1]. Par contre, la théorie, le raisonnement demeure cependant toujours au niveau du modèle [FEBF06] et le modèle est dit de niveau d'abstraction supérieur à celui du système modélisé.

2.1.3 Description versus prescription

Un modèle peut être une description un-à-un des concepts du système modélisé ou bien une description universelle, basée sur la notion de type [Kü06, Kü05] :

– *représentation directe* : un modèle peut être constitué d'un ensemble de symboles en relation ; chaque symbole correspond à un élément du système représenté dont il constitue le représentant dans le modèle ; il s'agit d'une définition par extension ; on parle aussi de représentation symbolique ou de modèle d'instance ou de modèle descriptif ;
– *représentation intentionnelle* : un modèle est défini par un ensemble de types ou de formules qui décrit l'ensemble des systèmes qu'il est possible de représenter ; suivant le domaine, on parle aussi de modèle de type, de prescription, de schéma ou de classification.

Dans le cas d'une représentation directe, le modèle est considéré comme le représentant du sujet. Dans le cadre de l'IDM, ces modèles sont utilisés comme représentation interne du sujet.

Dans le cas de la représentation intentionnelle, le modèle formule le sujet. Le modèle intentionnel est exploité pour la synthèse (automatique ou non) de composants logiciels ou d'autres modèles permettant de manipuler ou de raisonner sur des représentations directes.

1. Suivant les auteurs, les termes employés diffèrent

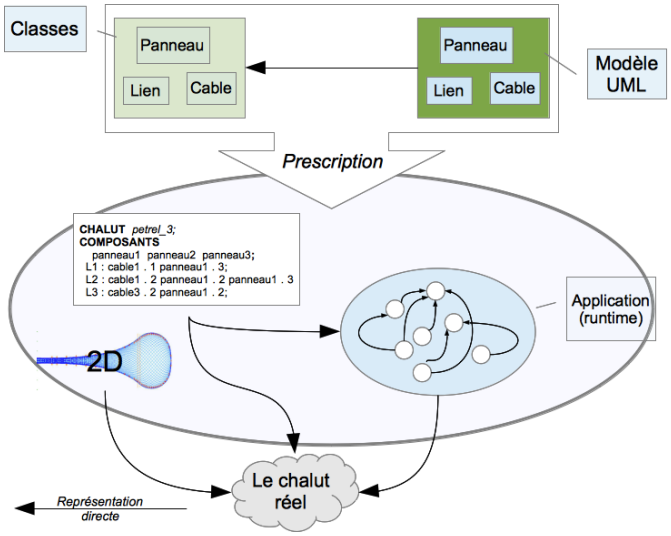

FIGURE 2.1 – Représentation directe et prescriptive : exemple du chalut

Un exemple est illustré par la figure 2.1. Un filet de pêche industriel, plus précisément *un talon de chalut*, est une structure construite à l'aide de structures constituées de câbles et cordages assemblés pour former des panneaux. Les panneaux sont eux-même assemblés pour former le talon de chalut. Une application d'analyse des caractéristiques physiques et dynamiques peut-être conçue sur la base de modèles de telles structures. Le domaine cible est donc la caractérisation physique en immersion de structures encordées. Dans un modèle textuel, un chalut peut être modélisé par un ensemble de constructions syntaxiques. Une telle déclaration peut correspondre à un câble, un panneau composé de câbles associés par des liens ou à une caractéristique ou un attribut d'un câble tel que une longueur ou une élasticité. Le modèle décrit alors un chalut particulier et peut servir de base à une représentation en 2D ou 3D du chalut et plus généralement à sa mise en œuvre par une application.

Un modèle constitué des types de câble et de lien en UML est une représentation intentionnelle de talon de chalut qui décrit l'ensemble des talons de chalut que l'on souhaite manipuler. Cette prescription peut être utilisée par un générateur de code pour produire les classes permettant de manipuler des modèles descriptifs de talons de chalut. Le modèle UML et les classes produites sont des représentations intentionnelles par rapport au modèle textuel décrit précédemment.

2.1.4 Rôle de la représentation

Décider si un modèle est une représentation directe ou une représentation intentionnelle est impossible si on ne tient compte que du modèle lui-même. En d'autres termes la nature du modèle n'est pas une caractéristique intrinsèque du modèle mais dépend de comment le modèle est utilisé.

Concernant l'exemple du chalut donné précédemment, le modèle décrivant les types de câbles est prescriptif par rapport à un modèle de talon de chalut particulier. Ce dernier est lui-même une représentation intentionnelle d'une partie d'une occurrence de filet de pêche réel embarqué sur un bateau particulier.

2.1.4.1 Niveau de représentation

Un modèle est descriptif par rapport à d'autres modèles descriptifs ou par rapport au sujet (du monde) réel. Cette relation de description est transitive [Kü06] : Si un modèle $m1$ est une représentation directe d'un système S et que le modèle $m2$ est une représentation directe du modèle $m1$, alors $m2$ est aussi transitivement une représentation directe du système S. Autrement dit, $m1$ et $m2$ se situe au même niveau de représentation.

La relation de représentation intentionnelle n'est par contre pas transitive. Si un modèle $m1$ est une représentation intentionnelle d'un système S et que le modèle $m2$ est une représentation intentionnelle du modèle $m1$, alors $m2$ n'est pas une représentation intentionnelle du système S. Le domaine de $m2$ est alors différent de celui de $m2$ et le niveau de modélisation de $m2$ est immédiatement supérieur à celui de $m1$.

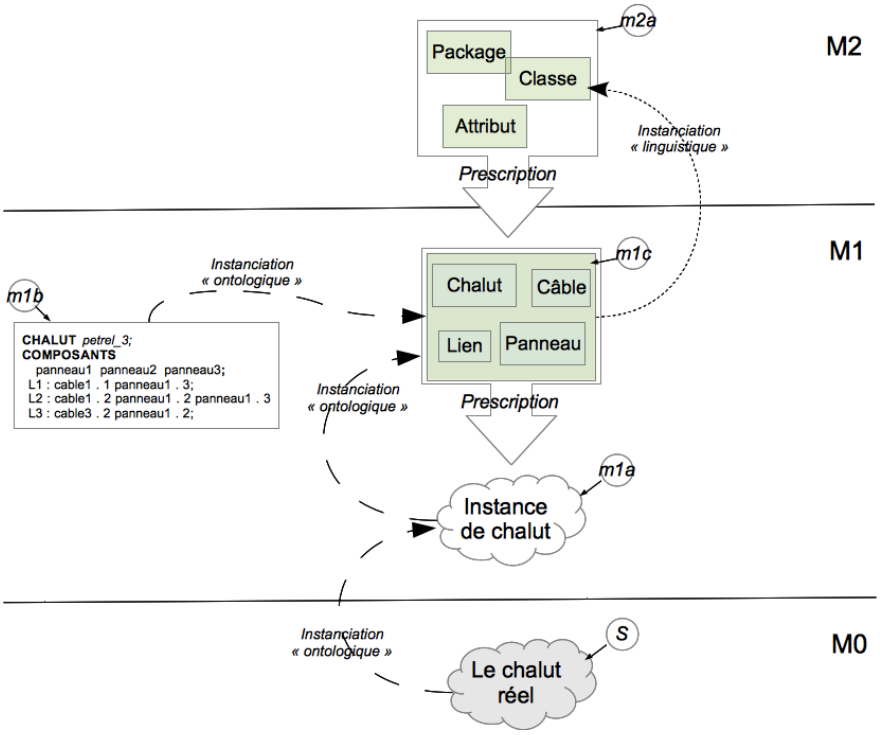

FIGURE 2.2 – Niveaux de modélisation

La figure 2.2 montre une représentation de ces différents niveaux :

- *niveau M0* : conventionnellement il comprend le sujet du monde réel ;
- *niveau M1* : il comprend des représentations directes du sujet ; le modèle étiqueté $m1a$ est une instance de chalut manipulée au travers de l'application ; c'est une instance ontologique du modèle étiqueté $m1c$; la représentation textuelle $m1b$ est aussi une représentation directe du modèle $m1a$ et une instance ontologique de $m1c$;
- *niveau M2* : il est constitué du modèle $m2a$ prescriptif pour tous les modèles du niveau $M1$. Il spécifie les concepts de classe et d'instance utilisés au niveau inférieur pour représenter le système.

Un modèle qui est une instance d'un modèle prescriptif est dit *conforme* au modèle prescriptif. Le schéma de la figure 2.2 montre qu'il y a deux natures d'instanciation. On parle d'instanciation *ontologique* si le domaine de l'instance est identique à celui du modèle. On parle d'instance *linguistique* dans le cas contraire.

2.1.4.2 Modèle et mise en œuvre d'une abstraction

Les relations de *représentation* et d'*abstraction* sont très fortement corrélées puisque la notion de modèle est aussi décrite comme étant l'abstraction d'un système modélisé sous la forme d'un ensemble de faits construits dans une intention particulière [FEBF06, Kü06]. Une abstraction permet d'extraire et de manipuler les propriétés d'un problème considérées comme essentielles tout en omettant les détails uniquement nécessaires à la mise en œuvre du problème [Boo87]. L'abstraction est ainsi une notion indissociable de la notion de modèle et de l'utilisation des modèles dans l'IDM.

L'IDM est définie comme une approche générative. Dans ce cadre, le modèle préexiste par rapport au système généré. L'abstraction constitue alors le support qui permet aux utilisateurs de se consacrer au problème à mettre en œuvre (le modèle qui représente le *quoi*) sans se préoccuper du résultat à produire (le système qui représente le *comment*).

Une abstraction peut être considérée à deux niveaux : le niveau de la *spécification* et celui de la *réalisation* [Kru92]. Une *réalisation* constitue l'élément cible de la relation de représentation. La réalisation d'une abstraction peut être la spécification d'une autre abstraction. Par exemple, la représentation conceptuelle des données exprimée par la modélisation statique des classes d'UML correspond à un haut niveau d'abstraction par rapport aux composants écrits dans un langage de programmation. Ces composants sont eux même de plus haut niveau d'abstraction que celui de la codification en langage d'assemblage.

Une abstraction comporte une partie variable, une partie fixe et une partie cachée [Kru92]. La partie cachée comprend toutes les informations relatives à la réalisation et ignorées du point de vue de la spécification. Les parties variables et fixes sont visibles depuis la spécification. La partie fixe correspond à la sémantique de l'abstraction. Elle peut être formulée de manière formelle par exemple avec une axiomatique de *Hoare* ou de manière informelle. La partie variable représente toutes les informations qui peuvent être différentes d'une réalisation à une autre, ce sont les paramètres de l'abstraction.

Cette définition de l'abstraction est directement exploitée par des constructions de certains langages, tel que les *patrons* de C++, les *paquetages* d'ADA ou les classes génériques d'Eiffel. Par exemple, dans un *patron* C++, la partie variable correspond aux types paramètres du patron et la partie cachée est décrite par son code. La sémantique n'est, par contre, pas spécifiée en C++. La spécification est ainsi totalement décrite dans un fichier entête. Une réalisation particulière du patron est produite automatiquement par le compilateur lors de la déclaration d'une variable dans un module.

2.1.4.3 Qu'est ce qu'un méta-modèle ?

Comme Tomas Kühne le précise dans [Kü06], le préfixe "méta" signifie qu'une opération est appliquée deux fois. Ainsi, le modèle d'un modèle est qualifié de méta-modèle. Selon cette définition, le modèle étiqueté $m1c$ de niveau $M1$ dans la figure 2.2 pourrait être qualifié de "méta-modèle" pour l'objet du monde réel. Cependant, le modèle $m1c$ et le système réel ne sont séparés que d'un seul niveau (les représentations directes du niveau $M1$ sont transitives).

Donc strictement, dans cet exemple, le modèle $m1c$ n'est pas un méta-modèle si on s'en tient à la définition du terme "méta".

Intuitivement, un méta-modèle est un modèle prescriptif dont les instances sont-elles même des modèles prescriptifs. Plus formellement, un modèle prescriptif MM est un méta-modèle pour un ensemble de modèles conformes Ms si, tous les modèles de l'ensemble Ms sont eux-même prescriptifs [2].

Ainsi, dans notre exemple de la figure 2.2, le modèle $m2a$ est un méta-modèle pour le modèle $m1c$.

2.1.5 Espace technique

La relation de méta-description est théoriquement infinie : un modèle est décrit par un modèle qui est lui-même décrit par un modèle...

Suivant le point de vue de la description, les modèles peuvent être considérés comme faisant partie d'une tour de modèles. Cette tour se caractérise par des niveaux empilés. Les modèles d'une tour n'appartiennent qu'à un et un seul niveau.

Dans une tour de modélisation, les niveaux sont ordonnés suivant la relation de lien de causalité qui implique qu'un changement dans un modèle de niveau N est suivi d'une adaptation des modèles appartenant au niveau $N-1$ de la tour de modélisation.

Pratiquement, une tour de modélisation est mise en œuvre au sein d'un espace technique.

La suite de ce chapitre décrit les notions de tour de modélisation et de lien de causalité. La notion de transformation de modèle est ensuite explicitée par rapport à celle d'espace technique.

2.1.5.1 Tour de modélisation

Un méta-modèle MM est lui-même conforme à un modèle prescriptif appelé alors un méta-méta-modèle pour les modèles prescriptifs décrits par MM. Le méta-méta-modèles prescrit quels sont les éléments utilisés pour constituer les méta-modèles. Il s'agit d'un modèle ou d'un langage minimal qui décrit formellement les concepts de classe (ou concept, entité ...), d'attribut (ou propriété) de classe et enfin de relation (ou association) entre classes [Fla02].

Le langage Entité-Association [Che76] peut-être utilisé pour exprimer des méta-modèles. Suivant le rôle du méta-langage, d'autres méta-méta-langages sont utilisés. Par exemple EBNF est utilisé pour la spécification de la syntaxe concrète des langages textuels ou encore le langage EXPRESS [ISO94b] dans le domaine de la mécanique et de la conception assistée par ordinateur.

D'après Robin Milner [Mil09], une tour de modélisation est une collection de modèles reliés par une relation de description. Dans une tour, un modèle peut être construit par combinaison d'autres modèles. Un exemple est montré par la figure 2.3. Dans cet exemple, le modèle C est auto-descriptif. B et C sont combinés pour produire le modèle A qui décrit le modèle C.

L'architecture à quatre niveaux mise en œuvre par l'OMG [OMG03, BBF03] est un exemple de tour de modélisation. Les principes sont repris du concept de tour réflexive illustré par la mise en œuvre d'un dialecte réflexif de Lisp appelé 3-Lisp [Tan09]. La tour de modélisation de l'OMG est orienté donnée et ne considère que quatre niveaux. En effet, la prise en compte de quatre niveaux est suffisant pour la représentation complète d'un système

2. Par souci de simplification, les publications et exemples donnés dans les documentations se dédouanent souvent de cette restriction en décrivant $m1c$ comme un méta-modèle.

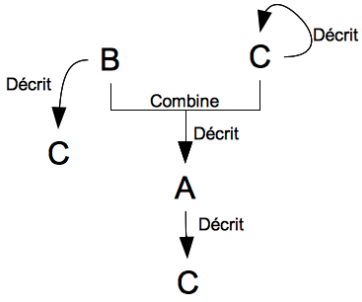

FIGURE 2.3 – Exemple de tour de modélisation (extraite de [Mil09])

d'information [RH]. Le niveau le plus haut (M3) se constituant du méta-méta-modèle de la tour de modélisation.

Un environnement pour l'IDM est toujours conçu comme une mise en œuvre et un outillage particulier d'une ou plusieurs tours de modélisation. Un environnement peut imposer un langage de modélisation (par exemple UML) ou permettre plusieurs langages de modélisation. Il permet l'utilisation d'analyseurs, de vérificateurs ou de traducteurs de modèles.

Comme le montre la figure 2.4, un *espace technique* [KBJV06] est défini par une tour de modélisation associée à un ensemble d'outils pour manipuler les modèles et les méta-modèles de la tour.

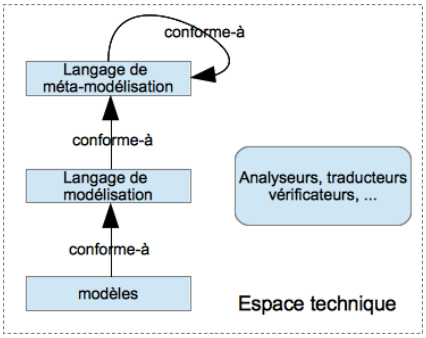

FIGURE 2.4 – Espace technique

2.1.5.2 Notions de lien de causalité et de réflexivité

Le concept de lien de causalité [Mae87] est fondamental si on considère un espace technique comme continuellement en évolution. Pour qu'un espace technique demeure valide, il est nécessaire qu'un modèle de niveaux $N-1$ soit conforme à un modèle de niveaux N et que tout changement dans un niveau N soit suivi d'une adaptation du niveau $N-1$. On dit que les niveaux N et $N-1$ sont en relation par un lien de causalité [Foo92].

Dans [RFBlO01] la définition suivante est donnée : "un niveau de modélisation est en lien de causalité avec son niveau immédiatement supérieur si le niveau inférieur est conforme au niveau supérieur et si les changements du niveau supérieur sont suivi, au niveau inférieur, de changements cohérents par rapport aux changements du niveau supérieur".

Dans [Tan09], le lien de cause à effet est considéré à double sens : un système est construit pour agir et raisonner sur un domaine ; un système est décrit par un programme et le programme doit être à jour par rapport au domaine ; le lien de causalité fait qu'un changement dans le domaine est reporté sur le programme et vice et versa. Cette relation est montrée par la figure 2.5(a). Le système est ici considéré comme une application qui s'exécute. Selon cette définition, un méta-système est un système dont le domaine est un autre système (figure 2.5(b)).

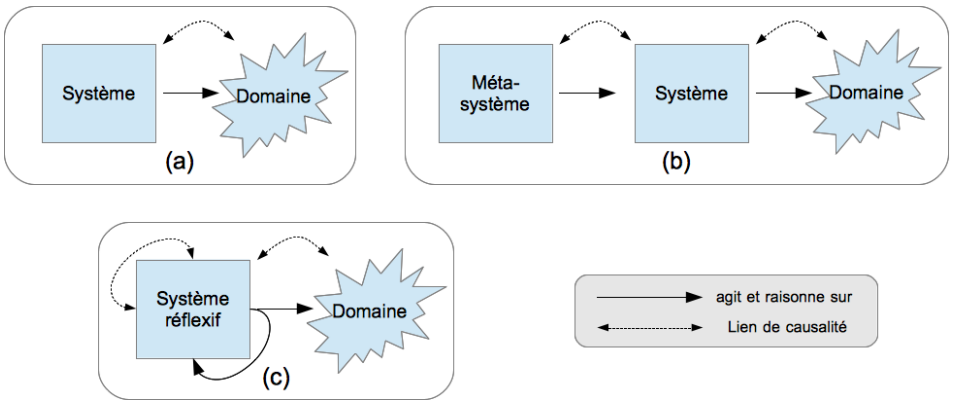

FIGURE 2.5 – Système (a), méta-système (b) et système réflexif (c) d'après [Tan09]

La notion de lien de causalité permet de définir celle de système réflexif [MJD96, Tan09]. Un système réflexif est un méta-système qui permet d'interagir avec un domaine mais dont le système sur lequel il agit et raisonne est lui-même (figure 2.5(c)). Une architecture logicielle réflexive est auto-décrite et entretient un lien de cause à effet avec son auto-représentation. Un tel système permet l'introspection (la capacité à s'auto-observer) et l'intercession (la capacité à modifier son comportement ou sa propre structure).

Le paradigme des objets est particulièrement bien adapté à la mise en œuvre de systèmes réflexifs [Tan09]. La réflexivité est mise en œuvre à l'aide d'un protocole à deux niveaux. Les classes et les méthodes sont accessibles par réification, c'est-à-dire qu'elles sont représentées dans le système par des objets avec lesquels il est possible d'interagir. Ces objets sont les méta-objets. Le comportement et les règles de gestion de ces méta-objets sont spécifiés par un protocole appelé le protocole à méta-objets ou MOP [3] [Tan09, KR91]. Le MOP définit un mécanisme réflexif : le méta-niveau gère le niveau représenté d'un point de vue structurel et comportemental. Les objets applicatifs sont modélisés par les méta-objets. Le méta-niveau a pour rôle d'observer, d'interpréter et de gérer les messages que les objets reçoivent, les propriétés structurelles et l'état des objets. La réflexivité est autorisée du fait que le méta-niveau est auto-décrit et que tout objet peut accéder aux objets du méta-niveau. Une modification du méta-niveau (modification structurelle ou comportementale) est automatiquement suivie d'une adaptation au niveau du code. Les langages CLOS et Smalltalk sont typiquement basés sur la mise en œuvre d'un MOP. Java dispose de mécanismes plus limités qui reposent essentiellement sur l'introspection et une version limitée de l'intercession comportementale [Tan09].

3. MOP pour *Meta-Object Protocol*

2.1.5.3 Espace technique et transformation de modèle

L'espace technique est la brique de base de construction des environnements pour l'IDM. Dans ce contexte, une transformation de modèle est défini comme la construction automatisée d'un modèle cible depuis un modèle source.

La transformation de modèle est utilisée depuis les débuts de l'informatique et est exploitée systématiquement par les compilateurs [ASU86]. L'utilisation de la transformation de modèle c'est étendu à tous les domaines de l'ingénierie du logiciel depuis les années 80-90 avec notamment la popularité des méthodes de modélisation orienté-objet telle que la méthode OMT [RBP+91]. En plaçant le modèle au centre du développement logiciel et ce, à tous les niveaux du cycle de développement, la transformation de modèle c'est généralisé et on parle de l'IDM comme d'une approche générative.

L'approche générative est très intéressante pour le passage à l'échelle, notamment pour l'effet de levier dont on bénéficie en termes de quantité, de qualité du code produit (respect des normes de codage et des règles architecturales) mais aussi en termes d'intégration de code complexe. L'approche générative permet aussi la portabilité des spécifications et constitue une alternative à l'interprétation ou à la compilation [FEBF06]. Enfin, les transformations de modèle permettent de capitaliser le savoir faire technique en termes d'implantation de spécifications de haut niveau d'abstraction dans une plate-forme d'exécution.

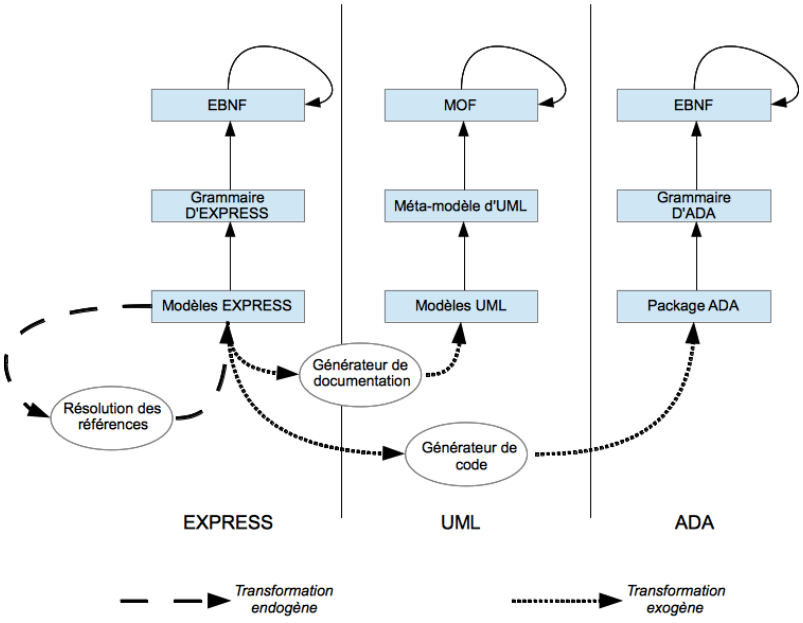

FIGURE 2.6 – Espace technique et transformation de modèle

Deux types de transformations sont classiquement considérés :

- une transformation de modèle à modèle transforme un modèle conforme à un méta-modèle source vers un modèle cible conforme à un méta-modèle cible ;
- une transformation de modèle vers du texte transforme un modèle source en une représentation textuelle.

Les modèles source et destination peuvent être spécifiés dans le même espace technique. Il

s'agit alors d'une *transformation endogène*. Si le modèle source et le modèle cible sont spécifiés dans deux espaces techniques différents, on parle alors de *transformation exogène*.

Prenons l'exemple d'un environnement de modélisation basé sur le langage EXPRESS. Il s'agit d'un langage textuel basé sur une grammaire EBNF. Les modèles EXPRESS peuvent se référencer entre eux. La résolution des références produit un nouveau modèle EXPRESS. Cette transformation est endogène. Un modèle EXPRESS peut aussi être traduit en UML ou en ADA par des traductions exogènes. La figure 2.6 schématise ces transformations de modèle par rapport aux trois espaces techniques mis en œuvre par cet environnement.

Plusieurs approches de transformation de modèles ont été développées (voir par exemple [CH03, CH06] pour une description et une classification de ces approches). Trois techniques sont principalement utilisées pour la mise en œuvre des transformations de modèle. Pour un méta-modèle particulier :

- la technique par programmation consiste à réifier les éléments du méta-modèle vers des classes, des structures ou des fonctions d'un langage pour lequel on dispose d'un compilateur ou d'un interpréteur ; les transformations sont alors programmées dans ce langage sur la base des éléments réifiés ;
- la technique par patron consiste déclarer un canevas pour la réalisation souhaitées ; un tel canevas se présente sous la forme d'une spécification dans le formalisme désiré pour la réalisation comprenant des expressions ou *escapes* [Cle88] qu'un interpréteur spécifique sait évaluer et remplacer par le résultat de l'évaluation ; la réalisation est produite par interprétation de toutes les expressions intégrées ;
- la méta-programmation consiste à exprimer les transformations de modèle à l'aide d'un langage dédié à la transformation de modèle et permettant de manipuler directement les méta-modèles ; l'approche peut-être impérative ou déclarative.

2.1.5.4 Maintien du lien de causalité par transformation de modèle

Dans le cadre de l'IDM, la transformation de modèle constitue le moyen le plus couramment utilisé pour maintenir le lien de causalité entre niveaux de modélisation. Typiquement, à l'aide d'éditeurs, les modèles de niveaux $M2$ sont instanciés pour obtenir des modèles de niveaux $M1$ (par exemple pour obtenir des modèles UML ou EXPRESS suivant le langage de niveaux $M2$). Ces modèles de niveaux $M1$ ne sont pas utilisés directement. Ils sont transformés en une représentation de niveaux $M1$, dans un langage pour lequel on dispose d'un compilateur (par exemple le modèle en Ada obtenu par la transformation du modèle EXPRESS dans la figure 2.6). Le code obtenu est compilé en une représentation exécutable et des instances du modèle peuvent être manipulées. Cette approche exploite deux espaces techniques séparés (transformation exogène). En pratique ces deux espaces techniques sont mis en œuvre de façon séparée, d'une part dans l'outil de modélisation et d'autre part dans l'environnement qui s'exécute [RFBlO01].

2.2 IDM et langage

Dans le contexte de l'IDM, les modèles sont manipulés via l'utilisation de *langages*. Du point de vue de ce qui est directement visible, un tel langage est classiquement soit graphique, soit textuel, soit composé de texte et de symboles graphiques. Mais les symboles graphiques ou les mots d'un langage textuel ne peuvent pas être exploités si on ne leur donne pas d'interprétation par rapport aux éléments du système modélisé. Ainsi, pour comprendre

et manipuler un élément de modèle, il faut qu'il puisse être interprété. Un système et un modèle qui le décrit peuvent être considérés comme deux ensembles associés par une relation d'interprétation. L'interprétation fournit "le sens" ou la *sémantique* du modèle par rapport au système qu'il décrit. L'interprétation établie un lien entre un ou plusieurs éléments du modèle et un ou plusieurs éléments du système modélisé [HR00, FEBF06]. Pour attribuer un sens aux éléments d'un modèle, la définition explicite d'une sémantique, qu'elle soit formelle ou non, est indispensable pour relier les éléments de modèles aux informations [HR04]. L'ensemble des informations décrites constitue le domaine sémantique. La ou les données d'un modèle qui décrivent une information du domaine sémantique sont appelées des méta-informations.

Dans les environnements, une méta-information est produite automatiquement à partir de l'analyse de spécifications formelles décrites à l'aide d'un langage. Un langage consiste en une notation ou *syntaxe* associée à une *sémantique*.

2.2.1 Aspects syntaxiques

La syntaxe d'un langage est spécifiée par un ensemble d'éléments valides pouvant être des mots, des phrases, des instructions, mais aussi des boites, des flèches ou des diagrammes [HR00]. De façon générique, ses éléments sont appelés des *expressions*. On distingue les expressions simples, non décomposables des expressions construites par combinaison d'autres expressions. Deux approches sont principalement employées pour exprimer la syntaxe : l'approche grammaticale et l'approche par méta-modélisation.

Généralement, pour un langage, deux types de syntaxe sont utilisés, la *syntaxe concrète* et la *syntaxe abstraite*. La *syntaxe concrète* ou *syntaxe statique* [Kai89] est destinée à être manipulée par l'utilisateur alors que la syntaxe abstraite est elle destinée à être manipulée par l'ordinateur [ASU86]. La syntaxe concrète est spécifiée par des règles syntaxiques formelles permettant la construction d'analyseurs. La syntaxe abstraite est dérivée de la syntaxe concrète par épuration des décorations utiles à l'utilisateur et aux analyseurs mais inutiles pour l'interprétation du modèle représenté. Elle exprime, de manière structurelle, l'ensemble des concepts manipulés au travers du langage [Com08].

2.2.2 Modèle, méta-modèle et langage

Dans l'IDM, un *méta-modèle* est considéré comme la syntaxe abstraite du langage qui sert à exprimer les modèles. Dans ce cadre, la syntaxe abstraite décrit comment un modèle doit être constitué et qu'elle sont les hypothèses de validités des constituants du modèle. Il s'agit tout d'abord d'une spécification purement structurelle (le contenant) qui indique comment les éléments de modèles sont organisés et quelle sont les informations contenues. Cette spécification structurelle peut être enrichie de contraintes. Un modèle est alors considéré comme bien formé si les règles structurelles sont respectées et si l'évaluation de toutes les contraintes renvoie *vrais*.

Typiquement, un analyseur syntaxique pour un langage produit des modèles structurellement bien formés par construction. Après l'analyse syntaxique, l'évaluation des contraintes permet de garantir qu'un modèle est valide par rapport aux règles exprimées dans la syntaxe abstraite.

Dans la figure 2.7, une tour de modélisation est présentée. Le modèle de chalut $m1b$ est une représentation directe du modèle $m1a$ ("manipulé" par l'application). Le modèle $m1b$ est exprimé à l'aide du langage $l1$. Le modèle prescriptif $m1c$ spécifie la syntaxe abstraite de $l1$. $m1b$ est une instance de $m1c$. Parallèlement, le modèle $m1c$ est exprimé à l'aide du langage

$l2$. $l2$ est donc un méta-langage pour $m1c$. Le méta-modèle $m2a$ spécifie la syntaxe abstraite du méta-langage $l2$ et $m1c$ est une instance du méta-modèle $m2a$. Le langage $l1$ est lui dit conforme au méta-langage $l2$ et par extension on dit aussi qu'un modèle est conforme à son méta-modèle.

Cette organisation est régulière jusqu'au niveau du méta-méta-langage utilisé pour exprimer le méta-modèle $m2a$ et dont le méta-méta-modèle $m3$ spécifie la syntaxe abstraite. Par contre, le méta-méta-langage sert aussi à exprimer sa propre syntaxe abstraite $m3$.

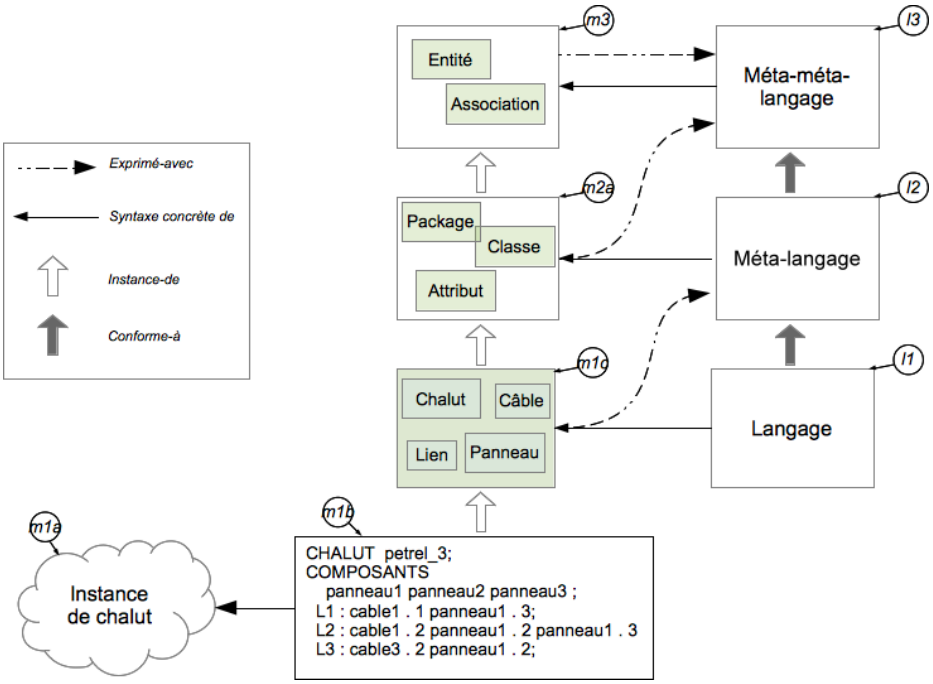

FIGURE 2.7 – Tour de (méta-)modélisation : modèle, méta-modèle et langage

2.2.3 Sémantique

Le raisonnement et la prise de décision implique l'existence d'une sémantique quelle soit formelle ou non. La sémantique d'un langage permet d'associer un sens aux éléments du langage utilisé pour le raisonnement. Le sens donné à un élément de modèle comporte toujours une part sociologique dépendante de la communauté qui manipule le modèle [Usc03, HR04]. Dans [Usc03], quatre catégories de sémantique sont décrites dans le cadre d'un "continuum sémantique" illustré par la figure 2.8 :

- une sémantique peut être implicite si le sens porté fait l'objet d'un consensus intellectuel dans un groupe de personnes sans retranscription sur aucun support ;
- une sémantique peut-être informelle mais explicitement établie ;
- la sémantique peut être explicitée de manière formelle dans un but documentaire ;
- et elle peut-être explicitée formellement pour être exploitée automatiquement.

Bien que donnée dans le contexte du web sémantique et des ontologies, la notion de continuum sémantique est transposable dans le domaine de la conceptualisation de l'IDM. Notamment,

la sémantique d'un modèle ou d'un méta-modèle s'appuie toujours sur un consensus établi, des coutumes de représentation et des codes informels implicites ou explicites [4].

Les modèles conceptuels et les méta-modèles s'appuient sur une formalisation, la nature des informations représentées est prescrite et imposée par le modèle pour une application particulière [JPAA06]. La sémantique peut-être exprimée suivant différentes approches, in-

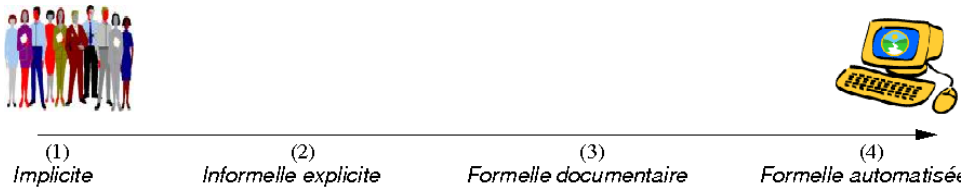

FIGURE 2.8 – Le continuum sémantique [Usc03]

formelle, en langage courant ou formellement par une sémantique dénotationnelle, opérationnelle, axiomatique ou algébrique. Si le modèle est un automate ou un programme, la sémantique comprend son activité ou son comportement. Pour un modèle comprenant des instructions logiques, la sémantique inclut la valuation des instructions logiques (par interprétation de fonctions ou des symboles des prédicats). Mais le terme "sens" inclut aussi une description informelle de comment les éléments du modèles "fonctionnent" [Mil09]. L'objectif est d'exprimer le lien entre les éléments de modèles construits et un domaine sémantique.

2.2.4 Domaine sémantique

Une sémantique associe un sens aux éléments de modèle dans un contexte ou le domaine est clairement défini. Ce contexte ou domaine est le *domaine sémantique* [HR04].

Par exemple, pour un modèle représentant une expression arithmétique, l'interprétation est possible en considérant le domaine mathématique des nombres naturels, pour un modèle composé de boites graphiques nommées et de flèches reliant les boites, le domaine sémantique peut être celui des flots de données, celui des communications dans un réseau téléphonique ou encore le paradigme de l'orienté-objet.

Rien ne prédestine un modèle à un domaine sémantique particulier de la même façon que plusieurs domaines sémantiques peuvent être utilisés pour un même modèle et induire des interprétations différentes suivant le contexte.

2.2.5 Règles sémantiques

Les règles sémantiques permettent d'expliciter le sens des éléments de modèle. Un langage clairement défini se constitue de sa syntaxe et de la spécification des règles sémantiques. Cette spécification peut être informelle, exprimée en langage courant. Dans ce cas, les éléments de la syntaxe abstraite sont systématiquement associés à la description de leur sens dans le domaine sémantique. Les descriptions informelles sont généralement utilisées pour les langages de programmation ou de modélisation généralistes. L'importance de la communauté des utilisateurs et les applications aident à préciser et corriger la description de la sémantique.

4. Par exemple, comme le montre David Mc Candles, suivant les cultures, le sens des couleurs varie très sensiblement : http://www.informationisbeautiful.net/vizualizations/colours-in-cultures/ (exemple cité dans [Wei11])

De cette façon, la sémantique peut-être suffisamment clairement et précisément décrite de façon à permettre le développement d'outils corrects par rapport à la sémantique du langage. Mais cette démarche n'est pas automatisable : la description de la sémantique ne peut pas être utilisée automatiquement pour construire les compilateurs ou des outils spécifiques par exemple pour valider ou prouver les modèles. Pour ce faire, les règles sémantiques doivent aussi être exprimées de façon formelle. On parle alors d'approche formelle.

Dans [CRC+06, Com08] Benoit Combemale décrit, par l'exemple, une taxonomie des différentes méthodes pour l'expression des règles sémantiques. On distingue la sémantique statique de la sémantique dynamique. La sémantique statique correspond principalement au type des éléments de modèle. Les éléments d'un modèle sont considérés comme décrivant les états possibles d'un système. Cette partie de la sémantique est directement analysée, vérifiée et exploitée pour vérifier que le modèle est bien formé et cohérent.

La sémantique dynamique se rapporte au comportement des éléments de modèle. Plusieurs techniques de spécification sont utilisées, elles sont classées en trois grandes catégories : opérationnelle, dénotationnelle ou axiomatique. Une sémantique opérationnelle décrit le fonctionnement d'un modèle en s'appuyant sur un programme. Une sémantique dénotationnelle décrit l'effet d'un programme sur son contexte d'exécution. Une sémantique axiomatique décrit l'évolution des caractéristiques des éléments de modèle pendant l'exécution du programme.

Si les règles sémantiques sont associées à une spécification formelle des syntaxes concrètes et abstraites, il est alors possible de produire automatiquement des environnements de programmation permettant à la fois l'édition, la vérification syntaxique, le déverminage et l'exécution de programmes pour un langage spécialisés pour un domaine [Kli93, Kai89, BCD+88].

Par exemple, le formalisme TYPOL est un représentant de cette approche formelle. TYPOL est exploité dans l'environnement Centaur [BCD+88] pour la spécification de la sémantique dite *dynamique*. La compilation d'un programme TYPOL produit du Prolog. Toutes les règles d'inférence sont traduites en Prolog sous la forme de clauses de *Horn*. Le programme Prolog peut alors être exécuté depuis CENTAUR.

TYPOL est un langage basé sur la *Sémantique Naturelle*. Un programme TYPOL comprend une liste de règles d'inférence de la forme :

$$\frac{H_1 \vdash T_1 : S_1 \ \& \ \cdots \ \& \ H_n \vdash T_n : S_n}{H \vdash T : S}$$

Pour ces règles, chaque séquence $H_i \vdash T_i : S_i$ est appelée un *prémisse* et la séquence $H \vdash T : S$ est appelée la conclusion. Les termes $T, T_1, \cdots, T_n$ sont les *phyla*, le tuple H_i représente les hypothèses et le tuple S_i représente les conséquences de la règle. Dans ce système à déduction, étant donnée une hypothèse H_0, calculer la valeur S_0 d'un terme syntaxique abstrait T_0 signifie prouver la séquence $H_0 \vdash T_0 : S_0$.

L'exemple 2.9 donné dans [AAFZ92], montre la spécification de la sémantique de l'instruction conditionnelle avec TYPOL. Les deux règles s'appliquent à l'opérateur *if*. D'après la règle de gauche, si la valeur calculée de *expr* est *true* et que la valeur calculée de *instr_s1* est *s1*, alors, le résultat de la conditionnelle est *s1*. La règle de gauche spécifie de la même manière le résultat de la conditionnelle lorsque la valeur calculée de *expr* est *false*.

Des fonctions TYPOL peuvent être associées aux règles d'inférence pour spécifier des actions sémantiques. Ces fonctions sont évaluées lorsque la règle est-elle même réduite. Elles sont déclarées dans un *package* et importées dans le programme TYPOL. L'exemple 2.10 montre une fonction TYPOL et son utilisation dans une règle d'inférence. La partie gauche montre la déclaration de la fonction *add* dans le *package expfunc*. La partie gauche montre

```
s |- expr : true &                    s |- expr : false &
s |- instr_s1 : s1                    s |- instr_s2 : s2
----------------                      ----------------
s |- if (expr, instr_s1, instr_s2) : s1   s |- if (expr, instr_s1, instr_s2) : s2
```

FIGURE 2.9 – La sémantique d'une instruction conditionnelle avec TYPOL

```
expfunc$add(A,B,C) :=                 import add(-,-,-) from expfunc ;
        C is A + B,

                                      |- expr1 : integer x1&
                                      |- expr2 : integer x2&
                                      add(x1,x2,x)
                                      ----------------
                                      |- plus(expr1,expr2) : integer x ;
```

FIGURE 2.10 – La déclaration et l'exécution d'une action sémantique avec TYPOL

son utilisation dans une règle d'inférence. L'interprétation informelle de cette règle est : la valeur de l'opérateur plus avec deux fils expr1 et expr2 est x. x1 est le résultat de l'évaluation de expr1, x2 est celui de expr2 et x est le résultat de l'addition de x1 et x2.

2.3 Activité de modélisation

Le génie logiciel peut être vu comme l'art de manipuler des abstractions. La forme dans laquelle une abstraction est manipulée provient du domaine pour lequel l'abstraction est utilisée [Cle88]. La modélisation et la méta-modélisation sont des activités où l'on cherche à cerner les caractéristiques d'un domaine d'application ou d'un domaine technique pour une utilisation particulière. Cette définition a deux implications importantes : une activité de méta-modélisation est toujours liée à un domaine et, étant donné qu'il n'y a pas de représentation unique et définitive d'un domaine, ces activités sont complexes et sujet à élaboration et remise en cause.

L'activité de modélisation est un processus humain de *catégorisation* du monde réel. Ce processus est double. Il est décrit comme deux traductions successives : le monde réel ou un système préexistant est traduit en un modèle conceptuel ; le modèle conceptuel est ensuite traduit en un modèle de simulation [TSL+05]. Dans le cadre d'un système formel, l'interprétation du modèle conceptuel vers le modèle de simulation est rendue opérationnelle et donc automatique. Par contre, la construction du modèle conceptuel représentant un système préexistant peut-être sujet à variation [TSL+05]. L'activité de modélisation s'effectue dans un contexte mouvant et dans ce cadre les modèles obtenus et en définitive le système construit

n'est jamais satisfaisant ou définitivement correct. Il s'agit d'un processus itératif qui théoriquement est infini. Dans la pratique, les résultats de la modélisation constitue un compromis notamment entre des délais, des coûts et un niveau de qualité requis [FEBF06].

2.3.1 Notions de catégorisation

Confronté à un ensemble important d'informations, l'être humain réagit en organisant les informations : par regroupement, en fonction du temps, dans l'espace, alphabétiquement ou suivant un continuum. Le regroupement des informations en sous-ensembles est la *catégorisation* [Wei11]. La catégorisation se base sur les similarités, des connaissances a priori et un contexte d'utilisation pour regrouper les informations. Selon la théorie de la perception, les éléments similaires d'un système sont perçus comme plus naturellement en relation. Toute catégorisation est liée à un contexte et n'a de sens que par rapport à son contexte.

Il existe principalement trois théories de la catégorisation [FGS11] :

- *la théorie classique* considère qu'une catégorie est une entité caractérisée par un ensemble de propriétés que partagent tous ses membres ;
- *la théorie des prototypes* est une évolution de la théorie classique ; la théorie des prototypes considère aussi qu'une catégorie est constituée p ar un ensemble de propriétés mais les membres d'une catégorie peuvent ne pas avoir certaines des propriétés, les exceptions sont donc autorisées ;
- *la théorie des exemplaires* considère qu'une catégorie est constituée de l'ensemble de ses membres sans notion de factorisation des propriétés.

Dans le cadre des systèmes informatiques et des modèles, la catégorisation peut procéder par *classification* ou par *généralisation*. Ces deux opérations sont bien distinctes [Kü06] :

- la classification est une relation entre plusieurs éléments d'un modèles Ma et un élément d'un autre modèle Mb. Les éléments de Mb sont prescriptifs pour ceux de l'ensemble Ma ; la catégorisation par classification est exploitée dans les tours de modélisation et typiquement, le concept de classe des langages Classe-Instance permet de mettre en œuvre la classification ;
- la généralisation est une relation entre plusieurs éléments de modèle et un élément particulier appartenant au même modèle M ; par exemple, un filet de pêche peut être constitué de plusieurs types de panneaux, des losanges ou des rectangles ; ces types peuvent être généralisés par le type panneau qui dans ce cas appartient bien au même modèle que losange et rectangle ; dans les langages à objets, la généralisation se traduit par l'héritage ; la conceptualisation dans les langages à prototypes [SA04] ou dans les ontologies [JPAA06, MPAA03] sont d'autres exemples de catégorisation par généralisation.

De façon inverse, la compréhension d'un ensemble de catégories nécessite de connaitre le contexte dans lequel la catégorisation est construite. La compréhension peut procéder (et procède souvent) de la construction d'exemples [Wei11]. Pour la construction d'un système, les cas d'utilisation, les tests et le développement de prototypes d'application pour simuler le système ou une fonctionnalité du système à produire constituent des "exemples" qui aident à la compréhension du problème à résoudre.

La catégorisation et la construction d'exemples permettent conjointement la constitution d'un *modèle mental* du problème sujet à modélisation. La constitution d'un modèle mental est la première étape du processus de modélisation.

2.3.2 Modèle mental, modèle conceptuel et système cible

Un *modèle mental* représente la pensée d'une personne qui tente de comprendre comment un système fonctionne ou plus généralement le monde qui l'entoure. Un modèle mental est basé sur des faits incomplets, des expériences passées et même des perceptions intuitives. Il définit notamment comment une personne intègre mentalement et résout des problèmes [wika, LHB10, Wei11]. Plusieurs théories sont développées dans le cadre des sciences cognitives. Dans le domaine de la conception, le concept du raisonnement par abduction [wikb, BL05] est aussi présenté. Dans la suite, nous utiliserons le terme de modèle mental pour désigner de façon générique la représentation mentale d'un problème.

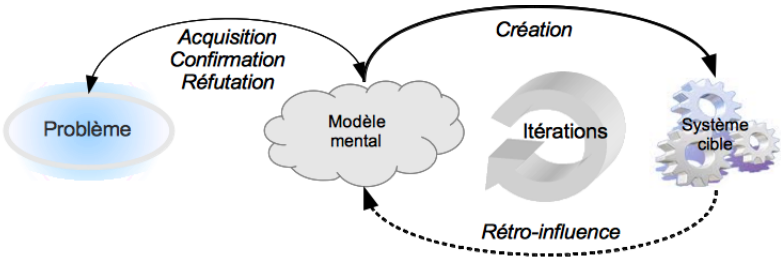

FIGURE 2.11 – Une vue simple du cycle de création d'un système

Le modèle mental peut être créé avant même que la personne qui le crée ait une expérience concrète avec l'objet pensé. Une personne fait appel à son modèle mental pour anticiper, prédire le comportement d'un système ou ce qu'il est possible de faire avec un système. Enfin, un modèle mental évolue notamment en fonction des expériences concrètes de la personne [Wei11]. Comme le montre la figure 2.11, la création d'un système consiste en trois relations entre le problème à résoudre (incluant ici tout le contexte), le système cible et le modèle mental. Le modèle mental est créé et évolue en deux grandes étapes :

- le modèle mental est créé à partir du problème à résoudre, le contexte du problème peut inclure ou non un système préexistant et les informations utiles peuvent être au départ mal définies et imprécises ; dans la figure, le terme d'*acquisition* est utilisé,
- le système cible est créé à partir du modèle mental et l'analyse du système cible induit un effet de rétro-influence sur le modèle mental ; la connaissance acquise par observation du système cible est confronté au problème ; la solution peut être alors affinée, complétée, confirmée, ou réfutée.

La figure 2.11 montre que la construction d'un système est le résultat d'itérations. Une itération est définie comme la production cyclique de code fonctionnel et de tests [Bec99]. D'autres définitions sont données par Nicholas Berente et al dans [BL05] et la notion d'itération est rapprochée de celle de prototypage. Un processus itératif est aussi considéré comme constitué d'approximations successives permettant de converger vers une solution satisfaisante. Ce point de vue implique une notion de progression vers l'obtention d'un résultat

satisfaisant des objectifs. Dans le cadre de la méta-modélisation, l'objectif premier est de produire un ou des méta-modèles utilisables et valides par rapport au domaine d'application et au besoin. Idéalement, le nombre d'itérations est infini. Concrètement, un processus itératif se termine lorsqu'un compromis est établi entre les différents objectifs, le coût et la qualité.

Un *modèle conceptuel* représente comment un objet du monde réel est conçu, ces caractéristiques fonctionnelles, son apparence ou son interface. Il est donc imposé par ce qu'est ou ce que doit être réellement l'objet représenté. Donc, même si l'objet représenté est anticipé (en phase de création), le modèle conceptuel doit correspondre à la réalité (future).

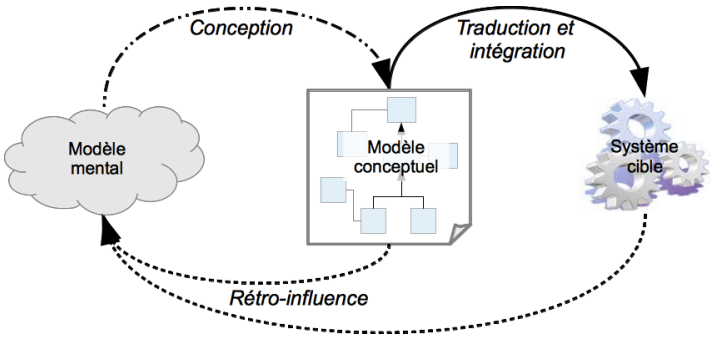

FIGURE 2.12 – Cycle de conception : relations entre le modèle mentale, le modèle conceptuel et le système cible

La figure 2.12 montre de façon très simplifiée la relation entre un modèle mental et un modèle conceptuel produit par rapport à un système à mettre en œuvre. La construction du modèle conceptuel intervient avant celle du système cible :

- le modèle mental est construit et exploité en amont, il est traduit en un modèle conceptuel qui prend en compte le besoin, le domaine et la façon dont la personne envisage l'utilisation du système ; le modèle conceptuel est analysé et vérifié et cette observation du modèle influe en retour sur le modèle mental ;
- le modèle conceptuel est exploité pour produire le système, la production peut-être manuelle, semi-automatique ou automatique ;
- par confrontation avec le système construit, la personne évalue l'adéquation du système par rapport à son modèle mental ; le modèle mental peut ainsi évoluer ; la conséquence de cette évolution peut être une modification du modèle conceptuel et celle du système.

Pour construire les modèles conceptuels, les concepteurs appliquent une technique de modélisation. Une technique comprend une partie outillée à l'aide d'un ou plusieurs langages de modélisation et comprend une méthode. La méthode décrit les étapes et les procédures qui doivent être appliquées pour produire les modèles conceptuels. La méthode peut être générique ou spécifiques au domaine [KK02]. La technique de modélisation est adaptée par rapport au problème à résoudre, par rapport aux facteurs qui influencent la modélisation et notamment par rapport à la manière dont les modèles conceptuels évoluent.

2.3.3 Les facteurs qui influencent sur la modélisation

La figure 2.13 montre les différents facteurs qui influencent la production du modèle conceptuel pour un système cible nécessitant plusieurs concepteurs ou plusieurs équipes de

conception et des utilisateurs. Dans ce cadre, l'activité de modélisation est classiquement contrainte par le domaine de l'application et les besoins fonctionnels. Le domaine peut imposer la forme du modèle. Suivant le domaine, nous manipulons constamment des abstractions par des représentations textuelles, des formules mathématiques, des patrons ou des représentations graphiques.

Le domaine comporte une part purement ontologique et une part technique. La part ontologique correspond à ce pour quoi les modèles sont développés. Par exemple, la finance ou les télécommunications constituent des domaines applicatifs. On parle de composante verticale du domaine. La part technique correspond à des composantes horizontales du domaine tel que le paradigme des objets, les bases de données, les architectures temps-réel, les interactions Homme-Machine ou encore le domaine du processus industriel.

Le besoins fonctionnels sont aussi décomposés en besoins verticaux liés au métier et en besoins horizontaux liées aux techniques informatiques. Du point de vue vertical, il peut s'agir par exemple de paramétrage du domaine ou de vérification par rapport aux règles métier. Du point de vue horizontal, il peut s'agir de besoins particuliers d'interopérabilité pour l'import et l'export de modèles, de présentation ou de mécanismes d'interaction particuliers ou encore des besoins d'exécution, de compilation ou d'analyse et de vérification des modèles.

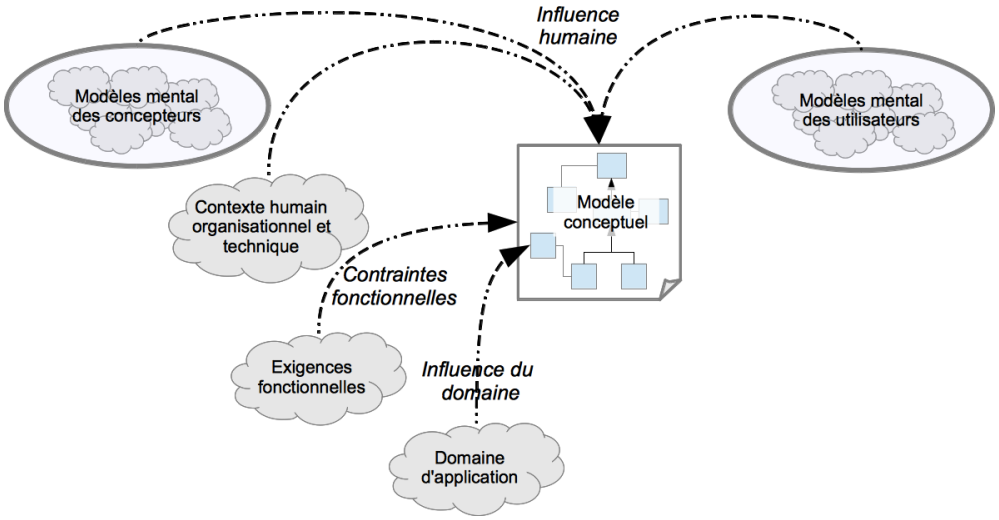

FIGURE 2.13 – Les différentes formes de contraintes qui influent sur la modélisation

Cette activité est aussi influencée par un contexte humain et technique. En particulier, l'organisation du ou des modèles conceptuels (par exemple leur catégorisation) et en définitive celle du système construit reflète de l'organisation des équipes qui produisent l'application [GGD07]. Les modèles mentaux des concepteurs et des utilisateurs influent sur la modélisation, notamment, les concepteurs doivent avoir conscience du modèle mental des (futurs) utilisateurs du système par exemple pour une conception adéquate des modèles d'interaction.

Chaque concepteur à son propre modèle mentale du système à produire. Pour un système complexe, les modèles mentaux peuvent être incomplets ou parcellaires. Dans le cas d'un tel système, le schéma de la figure 2.13 représenterait le processus de modélisation au niveau d'un sous-système. Les modèles conceptuels des sous-systèmes sont alors intégrés pour constituer le système complet.

Potentiellement, la rétro-influence peut modifier tous les facteurs en amont par analyse des modèles de conception et du système cible produit. En premier lieu, les modèles mentaux des concepteurs et des utilisateurs avec pour conséquence une adaptation soit des modèles mentaux ("le système a raison") soit des modèles conceptuels produits ("le système doit être corrigé").

Le contexte organisationnel et technique peut aussi être adapté. L'organisation des équipes, la redistribution des responsabilités ou le changement de choix techniques sont des conséquences possibles de la rétro-influence.

La confrontation entre le système produit et les utilisateurs peut entrainer une adaptation des besoins fonctionnels. Enfin le domaine lui-même peut être adapté ou plus précisément la façon dont les concepteurs et les utilisateurs considèrent le domaine peut évoluer.

L'outillage permet d'interagir avec le modèle, ou de questionner ou d'actionner le modèle en vue de comprendre, d'exploiter le système, de produire le système ou de le transformer.

2.3.4 Méta-modélisation

La méta-modélisation est une activité de modélisation qui consiste à mettre en œuvre un langage. Pour cette activité, il s'agit non seulement de produire des méta-modèles mais aussi de définir la sémantique du langage, de mettre en œuvre des analyseurs, des compilateurs [ASU86] des générateurs de code [CE00] et plus généralement, à construire un ensemble d'outils exploitant les méta-modèles [Kle07].

L'activité de méta-modélisation et celle de modélisation sont influencées par les mêmes facteurs. Cependant, la méta-modélisation se différencie de la modélisation du fait de la nature double des facteurs d'influences et du double niveau d'expertise nécessaire à cette activité.

Le domaine est tout d'abord celui de la définition d'un langage. Il est aussi lié à la fois aux spécificités du domaine d'application et à un ou plusieurs paradigmes techniques. Les influences humaines sont définies suivant le rôle des intervenants [KK02]. Par exemple, dans [ABE10], Van Amstel considère quatre rôles type :

- l'expert en méthode est responsable de la définition précise de la méthode et du processus de méta-modélisation ; il doit aussi maitriser les concepts liés au domaine applicatif ;
- l'expert en langage spécifie le langage de modélisation ; cette activité inclut la définition de la sémantique et des notations ou mécanismes employés pour construire les modèles, il participe à la définition du processus de modélisation, de configuration et d'intégration des outils produits ;
- la connaissance de la plateforme d'exécution cible peut aussi nécessiter l'intervention d'un expert ;
- l'utilisateur de l'outillage expérimente les outils de modélisation ; c'est un expert du domaine et il influence en retour les autres experts concernant l'adéquation technique et métier des outils de modélisation construits.

Chaque rôle peut nécessiter l'intervention de plusieurs personnes. Par exemple, dans [ABE10], chaque rôle est tenu par 2 à 4 intervenants.

2.3.4.1 Méta-modélisation et espace technique

Par rapport au concept d'espace technique (voir chapitre 2.1.5), méta-modéliser consiste à utiliser un ou des langages de niveau $M3$ pour spécifier et mettre en œuvre au niveau

$M2$ un ensemble comprenant des méta-modèles et des outils spécifiques. Sur la base d'un tel langage, une architecture de modélisation est définie.

Le niveau $M3$ rigidifie l'espace technique autour d'un langage commun pour la spécification des langages de niveau $M2$. Dans l'IDM, l'approche est orienté-donnée, le langage de niveau $M3$ permet de spécifier des structures nommées et attribuées : le langage de niveau $M3$ permet la définition de la syntaxe abstraite de langages de niveau $M2$. La mise en œuvre d'une telle définition permet de représenter les modèles conformes et de constituer des représentations internes. Une syntaxe concrète permet à chaque niveau d'échanger les modèles conformes. Actuellement, les langages de niveau $M3$ les plus utilisés sont le MOF [Gro04] et ECore [BBM03] et la syntaxe concrète majoritairement utilisée est XML.

Les exemples suivants sont très souvent cités :

- le *CASE Data Interchange Format* (CDIF) a été normalisé pour permettre l'échange de modèles entre différents outils de modélisation ; CDIF n'est plus développé mais à servi de base notamment pour le développement du *MOF* ;
- le *Meta Object Facility* (MOF) [Gro04] et ses dérivés (E-MOF, C-CORE...) sont normalisés par l'OMG [OMG92] comme spécification standard de niveau $M3$ pour permettre la spécification et l'échange des modèles à objets (principalement spécifiés avec UML et ses différents profiles) ; le MOF est un élément central du MDA [OMG03, Bro04], l'architecture de modélisation standard de l'OMG ;
- le *Resource Description Framework* (RDF) spécifie une hiérarchie de modélisation pour les réseaux sémantiques ; RDF est basé sur trois concepts : *Ressource*, *Propriété* et *Instruction* pour la spécification de concepts nommés et de leurs propriétés au niveau $M2$.
- MetaEdit+ [Met] est un environnement de la société MetaCase basé sur un langage de niveau $M3$ permettant la spécification et la mise en œuvre automatisée de langages graphiques spécifiques à une application et à un domaine.

2.3.4.2 Intégration par l'espace technique

L'intégration caractérise les moyens par lesquels il est possible de composer plusieurs éléments de modèle et d'adapter la composition à un contexte d'utilisation. L'espace technique peut être vu comme un espace d'intégration de méta-modèles. L'intégration peut être considérée suivant deux niveaux distincts :

- le niveau du méta-modèle pour lequel plusieurs méta-modèles décrivant des points de vue différents mais au même niveau d'abstraction sont intégrés pour constituer un méta-modèle intégré, on parle d'*intégration horizontale* ;
- le niveau du système pour lequel plusieurs méta-modèles sont exploités pour constituer le système à plus bas niveau d'abstraction, on parle d'*intégration verticale*.

Pour l'intégration horizontale, l'objectif est d'assurer une homogénéisation lexicale, syntaxique et sémantique des différents méta-modèles. L'intégration est tout d'abord lexicale et syntaxique et le rôle de l'espace technique est d'apporter la syntaxe et les structures unificatrices permettant l'intégration. L'intégration est aussi sémantique puisque les méta-modèles sont conformes à un méta-méta-modèle commun. L'intégration sémantique est fondée sur un accord concernant les types de données.

Pour l'intégration verticale, les méta-modèles sont combinés et exploités dynamiquement par un outil pour produire une réalisation à partir d'un modèle source. L'intégration verticale est de nature fonctionnelle et est orchestrée par l'outillage de l'espace technique.

2.3.4.3 Interopérabilité et espace technique

L'intégration, qu'elle soit horizontale ou verticale pose le problème de l'interopérabilité entre sous-systèmes. L'interopérabilité est la capacité de deux ou plusieurs composants à coopérer malgré les différences relatives aux langages, aux interfaces et aux plateformes d'exécution [Weg96].

Cette définition convient au problème de l'intégration lorsque, par exemple dans une architecture client-serveur, le client et le serveur sont générés automatiquement. Le client et le serveur doivent être compatibles d'un point de vue statique et d'un point de vue dynamique.

La compatibilité statique est réglée par un accord sur les types alors que la compatibilité dynamique est réglée par un accord sur un protocole d'échange ou de partage de l'information [Weg96]. Le protocole peut être basé sur la définition :

- d'un format de données commun reposant sur la spécification d'un langage pour la représentation des données, on parle alors d'interopérabilité par les données ou par échange de données,
- d'une interface logicielle commune, on parle alors d'interopérabilité fonctionnelle.

Le rôle de l'espace technique est de définir un consensus permettant l'intégration horizontale et verticale visant une adaptation *a priori* standardisée. On constate que l'interopérabilité par les données est systématiquement utilisée pour l'intégration des méta-modèles. Le standard représenté par le langage de niveau $M3$ normalise les types, les structures et le protocole d'échange des méta-méta-données. Le niveau d'interopérabilité fonctionnelle est plus complexe à atteindre car elle implique un consensus non seulement au niveau de la représentation des données mais aussi un consensus concernant les interfaces logicielles plus difficiles à obtenir et moins pérennes du fait de l'évolution et de la diversité des langages et des plateformes d'exécution.

La norme STEP (*Standard for the Exchange of Product model data*) [ISO94a] est le standard ISO 10303 développé pour l'échange de modèles principalement dans le domaine de la conception assistée par ordinateur. Dans le cadre de cette norme les deux niveaux d'interopérabilité sont standardisés. La technologie STEP est basée sur le langage EXPRESS pour la spécification des méta-modèles, sur un protocole d'échange de modèles basé sur la définition d'un format d'échange neutre et d'une méthode de partage de modèles basée sur la spécification d'une interface logicielle standard.

2.3.4.4 Méta-modélisation et domaine

L'objectif de la méta-modélisation peut-être non seulement de spécifier des méta-modèles et de permettre l'échange de modèles conformes mais aussi d'outiller spécifiquement et le plus complètement possible un domaine, une application ou une famille d'applications. On parle de langage spécifique à un domaine [Kle08, DKV00], de modélisation spécifique à un domaine [KT08] ou de lignes de produits logiciels.

Les usines logicielles [GSCK04] représentent le point de vue de Microsoft pour l'IDM. La vision de Microsoft intègre une part dominante du domaine dans l'outillage qui doit être développé spécifiquement. Les éditeurs de modèles sont spécifiques et les développeurs sont spécialisés pour le domaine. Plus généralement, les langages employés, les outils d'analyse, de vérification et de transformation sont spécifiques. La réutilisation de composants logiciels est aussi un point fort de cette vision.

Un langage ou environnement de modélisation spécifique à un domaine permet d'élever le niveau d'abstraction des outils de l'environnement par la prise en compte des spécificités du

domaine et des besoins fonctionnels. Cette spécialisation pour le domaine et éventuellement l'application, favorise la vérification et la validation des spécifications. Le concepteur ou le développeur d'application qui utilise un environnement spécifique à un domaine manipule alors directement les concepts du domaine et utilise des compilateurs et des composants logiciels spécifiques.

```
1 CHALUT petrel_3;
2 COMPOSANTS
3       panneau1 panneau2 panneau3 ;
4       cable1 cable2 cable3 ;
5 L1  : cable1.1 panneau1.2;
6 L2  : cable2.2 panneau1.1 panneau3.2;
7 L3  : cable3.3 panneau3.1;
```

FIGURE 2.14 – Exemple de modèle spécifique à un domaine : la description de la structure d'un chalut

Par exemple, pour notre application d'analyse des propriétés physiques et dynamiques d'un chalut, la modélisation d'un filet de pêche en langage C devrait s'appuyer sur un ensemble de structures et de bibliothèques de visualisation 2D ou 3D et de bibliothèques mathématiques. La manipulation d'une représentation de filet de pêche peut ainsi s'avérer complexe et coûteuse en termes de temps de développement.

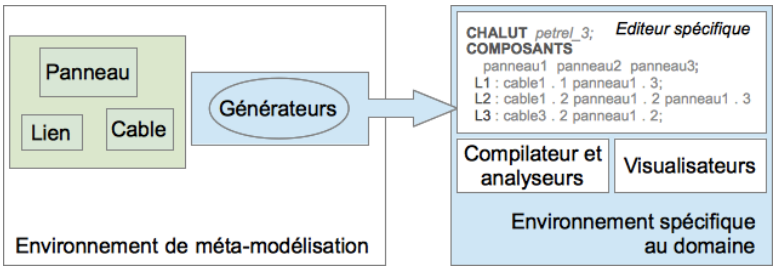

FIGURE 2.15 – Production d'un environnement spécifique à un domaine à partir d'un environnement de méta-modélisation

La figure 2.15 montre les constituants d'un environnement spécifique (à droite) qui pourraient être produit automatiquement ou semi-automatiquement à partir de la spécification d'un langage pour le domaine. Ce langage est spécifié au sein d'un environnement de méta-modélisation (à gauche). Dans l'environnement de méta-modélisation, les concepts du domaine, la syntaxe concrète du langage, éventuellement sa sémantique et des générateurs de code sont spécifiés et mis en œuvre.

L'environnement spécifique au domaine produit pourrait permettre d'exprimer des instances de chalut, par exemple, la spécification montrée par la figure 2.14. L'utilisateur manipule alors directement les concepts du domaine. Cette spécification doit être compilée et vérifiée. Le compilateur et les analyseurs peuvent aussi être automatiquement construits à partir de la spécification du langage spécifique au domaine.

Le résultat de la compilation, une représentation interne d'un filet, peut servir d'entrée

pour la visualisation des propriétés physiques du chalut. Les outils utilisés pour les visualisations peuvent aussi être construits par l'environnement de méta-modélisation.

2.3.5 Ateliers et méta-ateliers pour l'IDM

Un atelier pour l'IDM permet à l'utilisateur de spécifier des modèles en utilisant un ou plusieurs langages de modélisation. La figure 2.16 montre qu'un atelier spécifique à un langage est construit en deux grandes étapes. Un atelier est basé sur la spécification de méta-modèles constituant la ou les syntaxes abstraites des langages de modélisation utilisés et éventuellement les ou les syntaxes concrètes. C'est la première étape de construction d'un atelier. Ces spécifications sont intégrées à une structure d'accueil générique. Cette structure générique permet d'exploiter la spécification du langage pour construire des analyseurs de modèles. Les outils spécifiques au langage, au domaine et à l'application sont construits en utilisant les composants génériques.

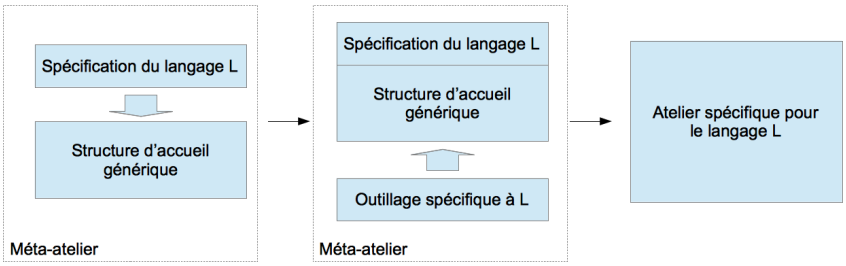

FIGURE 2.16 – Processus de construction d'un atelier spécifique à un langage

Outre le stockage et la gestion des modèles et des éléments de modèle, un atelier spécialisé permet tout ou partie des fonctionnalités suivantes :

- présenter les informations sous la forme de graphes, de tables, à l'aide de butineurs ou d'inspecteurs,
- exécuter des actions pour analyser, vérifier ou transformer les modèles,
- interagir avec les utilisateurs à l'aide d'interfaces graphiques ou de moniteurs,
- communiquer et échanger les modèles (fichiers d'échange, partage via un bus logiciel ou une interface client-serveur)
- gérer les configurations et les versions des modèles.

Un méta-atelier est un outil qui sert à produire des ateliers. Il permet donc de spécifier et d'exploiter des méta-modèles pour le développement d'ateliers spécifiques. Les fonctions principales d'un méta-atelier sont :

- l'édition et la documentation de méta-modèles, la spécification de contraintes de validité associées ;
- de spécifier les processus qui mettent en œuvre les modèles pour l'automatisation de certaines tâches, la simulation, le prototypage ou l'animation [MPGK00, vHTVMP04] des modèles ;
- de permettre la spécification et la mise en œuvre des transformations de modèles et des générateurs de code ;
- la définition des interfaces utilisateurs et la production des ateliers spécifiques.

2.3.5.1 Une architecture générique

La figure 2.17 montre une architecture générique pour un (méta-)atelier inspirée de celle présentée dans [KK02].

La structure d'accueil générique est le noyau du méta-atelier. Il est constitué d'un méta-méta-modèle fixe et d'un ensemble de mécanismes mis en œuvre spécifiquement pour le méta-méta-modèle. Le méta-méta-modèle définit les concepts directement utilisables pour la spécification des méta-modèles. Il est fixe mais observable (introspection) depuis tous les composants de l'atelier. Les mécanismes peuvent évoluer ou être enrichis.

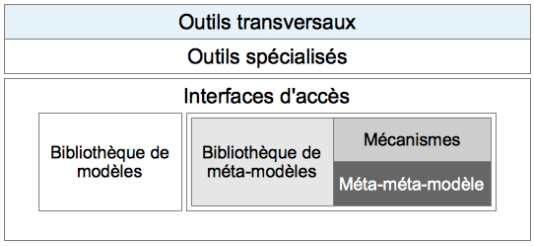

FIGURE 2.17 – Architecture générique de méta-atelier

2.3.5.2 Les mécanismes

Les mécanismes servent à accéder à la description du méta-méta-modèle. Suivant la nature du méta-atelier, les mécanismes peuvent être aussi utilisés pour accéder aux méta-modèles et aux modèles ou permettre des opérations de sérialisation/matérialisation de bas niveau. Trois catégories de mécanismes sont répertoriées suivant qu'ils sont basés (1) sur le modèle Entité-Association, (2) sur l'orienté objet ou (3) sur la théorie des graphes.

Les mécanismes basés sur le modèle Entité-Association sont les plus simples. Le méta-méta-modèle est défini structurellement à l'aide d'entités et d'associations. Les mécanismes servent essentiellement à naviguer au sein de la structure du méta-méta-modèle. Le modèle entité-association peut-être enrichi pour, par exemple, intégrer des primitives de représentation des entités, des associations ou de l'héritage. Les représentations peuvent être graphiques ou textuelles.

L'orienté-objet apporte l'instanciation, la possibilité d'associer un comportement et d'étendre les mécanismes de façon élégante pour la représentation et la manipulation des méta-méta-modèles. Typiquement, les mécanismes sont mis en œuvre par un MOP. Si les mécanismes sont utilisés uniformément pour tous les niveaux de l'architecture, le lien de causalité peut être maintenu de façon dynamique.

Les mécanismes peuvent être basés sur la théorie des graphes. La formalisation est plus importante, la mise en œuvre est basée sur la théorie des ensembles ou la logique. Certains outils utilisent les bases de données déductives permettant la spécification de requêtes et de règles logiques.

2.3.5.3 Interfaces d'accès et outils transversaux

Des interfaces d'accès mettent en œuvre l'interopérabilité pour l'échange et le partage du méta-méta-modèle, des méta-modèles et des modèles. L'intégration normalisée du méta-

atelier dans son environnement est assurée à l'aide de ces interfaces. Notamment, la persistance des bibliothèques de modèles et de méta-modèles peut être assurée grâce à ces interfaces. L'intégration par partage est assurée par une interface logicielle publique permettant d'intégrer des greffons. L'intégration par échange est assurée par échange de données encodées dans un format normalisé (comme XML, XMI, STEP...) et par des encodeurs/décodeurs de données suivant une norme d'encodage. Ces mécanismes de sérialisation/matérialisation peuvent être utilisés en interne pour la communication entre les différents composants de l'atelier, pour assurer la persistance des bibliothèques de modèles et de méta-modèles ou pour assurer l'interopérabilité entre le méta-atelier et d'autre outils.

Les outils transversaux sont utilisés pour construire des outils spécialisés pour un domaine et une application. Ces outils peuvent être, par exemple, des compilateurs de grammaire pour l'instrumentation des syntaxes concrètes, des éléments graphiques pour construire des interfaces Homme-Machine ou des bibliothèques pour construire des butineurs. Ces outils peuvent être apportés par l'environnement hôte dans lequel le méta-atelier est emboité.

2.3.5.4 Instanciation d'un méta-atelier pour une application

Un méta-atelier constitue une plateforme d'accueil générique pour la mise en œuvre de méta-modèles. La généricité est apportée par les mécanismes et les outils transversaux. Les mécanismes peuvent être automatiquement adaptés ou enrichis pour le méta-modèle. Le noyau d'un atelier spécifique à un domaine et à une application se constitue alors du méta-méta-modèle, des méta-modèles spécifiques et des mécanismes. Comme le montre la figure 2.18, un méta-atelier peut être aussi vu comme un générateur d'ateliers spécialisés pour des domaines et des applications. Dans la figure, une instance d'atelier est montrée par le cadre hachuré. Un tel atelier est, pour une application, un outil permettant de manipuler des modèles conformes aux méta-modèles spécifiques au domaine et à l'application.

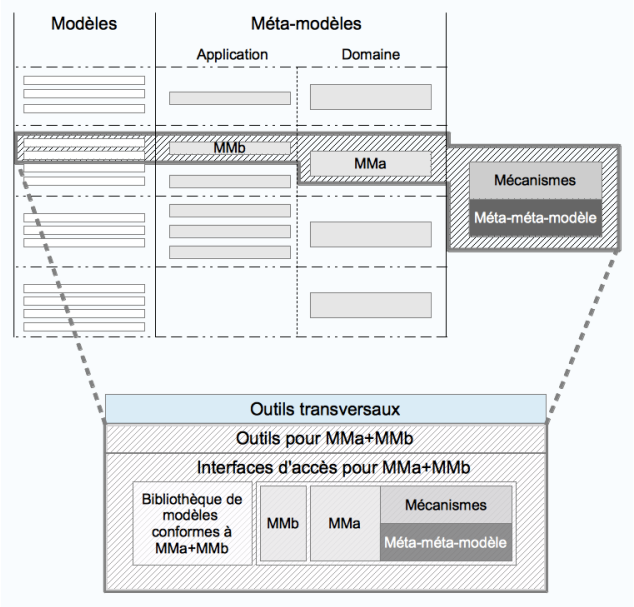

FIGURE 2.18 – Instanciation d'un méta-atelier pour une application

Le noyau d'un atelier spécialisé comprend un ensemble de méta-modèles pour le domaine (MMa dans la figure 2.18) et l'application (MMb dans la figure 2.18). Le même méta-modèle domaine peut être réutilisé pour des applications différentes. Les interfaces d'accès sont instanciées pour l'application et l'atelier comprend des outils spécialisés pour les méta-modèles.

2.3.5.5 Mise en œuvre des outils spécifiques à un domaine

La mise en œuvre des outils peut procéder par interprétation ou par génération de code. Comme le montre la figure 2.19, pour ces deux approches, une représentation interne du méta-modèle est utilisée et des mécanismes spécifiques au domaine permettent de manipuler le méta-modèle. Les cadres pointillés représentent le contexte du méta-atelier. La représentation interne est issue d'une analyse du méta-modèle.

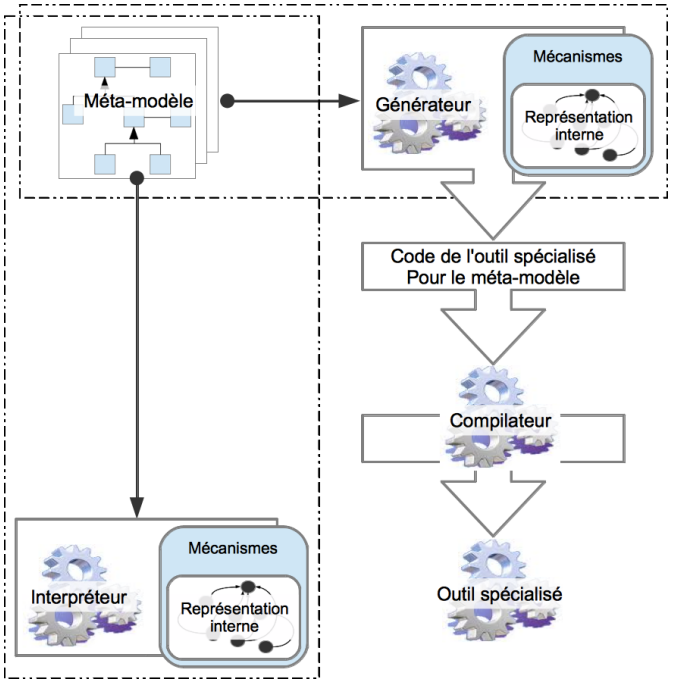

FIGURE 2.19 – Outil générique versus outil spécialisé

La partie gauche de la figure montre l'approche par interprétation. L'interpréteur est un outil générique développé une fois pour toute. Il peut être utilisé quelque soit le méta-modèle. Une interprétation exploite directement la représentation interne et les mécanismes du méta-atelier. Pour cette approche, le méta-atelier constitue une plateforme d'exécution.

La partie droite montre l'approche générative. Le générateur est un interpréteur de méta-modèle particulier. Pour une génération de code, il exploite les mécanismes et utilise une représentation interne du méta-modèle. Pour un méta-modèle, le code spécifique est généré en exploitant la représentation interne. Le code est ensuite intégré avec des composants qui mettent en œuvre les mécanismes. Ces composants additionnels sont développés une fois pour toute pour la plateforme d'exécution cible. On parle de *framework* ou de plateforme. Le code intégré est finalement compilé pour produire l'outil exécutable. Pour cette approche, la

plateforme d'exécution est distincte du méta-atelier.

2.3.6 Exemples de méta-ateliers

Dans sa thèse [Eng00], Vincent Englebert décrit l'environnement *ToolBuilder*, comme le premier méta-atelier. ToolBuilder a été développé il y a 45 ans. Depuis, de nombreux travaux ont abouti à la mise en œuvre de méta-ateliers. Les environnements suivants sont souvent mentionnés : ConceptBase [JGJ+95, Jeu], Kogge [JE97], GME/MetaGME [Dav03], MetaEdit+ [Met], Eclipse-EMF/ECore [BBM03], Dome [EK00], Moose [NDG05, Moo], AMMA/KM3 [JB06], Kermeta [MFJ05].

2.3.6.1 ConceptBase

ConceptBase est un méta-atelier dont les mécanismes sont fondés sur les bases de données actives et l'orienté objet. ConceptBase s'appuie sur une approche formelle utilisant des expressions logiques et un moteur *Prolog* pour les contraintes, le modèle des bases de données actives (Évènement-Condition-Action) et la programmation fonctionnelle pour les aspects comportementaux. ConceptBase met en œuvre une architecture à quatre niveaux complets et les données de chaque niveau sont uniformément stockées dans la base de données. ConceptBase est construit sur la base d'une architecture client-serveur, le serveur constitue le cœur du système pour l'accès aux données et aux méta-données. ConceptBase permet la création de langages spécifiques textuels ou graphiques. Les possibilités en termes de simulation et de prototypage s'appuient sur le modèle des bases de données actives fondée sur la réponse à des évènements et les exceptions. Pour le prototypage plus spécifique il est nécessaire de mettre en œuvre des générateurs de code.

2.3.6.2 Kogge

Kogge est basé sur la théorie des graphes. Un nœud est associé à une entité et un arc à une association. Les graphes sont typés, attribués et ordonnés. Les graphes sont spécifiés à l'aide du langage Z et des règles d'intégrité peuvent être spécifiées à l'aide du langage GRAAL. La spécification complète d'un atelier particulier procède de la définition d'un méta-modèle, des menus présentés à l'utilisateur et de diagrammes d'états pour les évènements et les actions activées.

2.3.6.3 GME/MetaGME

GME/MetaGME est représentatif de l'approche par intégration des méta-modèles (approche MIC pour *Model Integrated Computing*). Un atelier est constitué de composants. Les méta-modèles sont multi-vue, chaque vue décrivant un aspect particulier et est lié à un composant particulier. Dans GME, UML et OCL sont utilisés pour spécifier les méta-modèles et les règles d'intégrité. Une extension d'UML est utilisée pour exprimer la représentation des éléments de modèles dans l'atelier. L'atelier est intégralement automatiquement produit par génération de code et les composants additionnels sont intégrés par programmation en C++, en Visual Basic, C# ou encore Python. Un atelier est constitué de son interface graphique spécialisée pour modéliser, analyser les modèles et invoquer les fonctionnalités additionnelles.

2.3.6.4 MetaEdit+ et Dome

MetaEdit+ [Met] et Dome [EK00] sont aussi des représentants de l'approche MIC. ils utilisent un langage de méta-modélisation spécifique. Le langage de niveau $M3$ est le langage propriétaire GOPRR [5] pour MetaEdit+ et une variante de *Scheme* pour Dome. Du point de vue de la production des ateliers spécialisés, les approches de MetaEdit+ et de Dome diffèrent très sensiblement car un atelier spécialisé et le méta-atelier sont intimement intégrés et les mécanismes sont réutilisés.

MetaEdit+ [Met] permet de mettre en œuvre des éditeurs de modèles graphiques à partir de la spécification de langages spécifiques à un domaine. L'éditeur est conçu comme un interpréteur de méta-modèle et son exécution ne nécessite pas de génération de code. MetaEdit+ permet aussi l'interprétation de règles OCL et l'interprétation de transformation de modèles. Par contre, le point de vue traitement et comportemental repose sur l'introspection et sur la transformation de modèle. À l'aide du langage de script *Merle*, il est possible de parcourir les modèles et générer des programmes et de la documentation. Ainsi, la construction de prototypes repose sur la génération de code.

Dome est un méta-atelier intégré à un environnement Smalltalk [GR83]. Dome permet la définition de méta-modèles et de leur comportement. La spécification peut être interprétée ou directement compilée en Smalltalk. La structure des méta-modèles est spécifiée graphiquement et le comportement est programmé à l'aide d'une variante du langage *Scheme*. L'exécution peut être assurée soit directement en Scheme soit en Smalltalk. Dans sa forme, Dome est similaire à MetaEdit+. Étant donné sont intégration dans un environnement Smalltalk, les prototypes sont construits et exécuté directement dans le méta-atelier, sans génération de code et le lien des modifications apportées au méta-modèle sont directement prise en compte par les éditeurs et les prototypes.

2.3.6.5 Les méta-ateliers intégrés à Eclipse

Eclipse-EMF/ECore [BBM03] est utilisé pour spécifier la structure des méta-modèles. L'approche est minimaliste et s'appuie sur le modèle Entité-Association. Le langage de méta-modélisation est ECore. Une spécification ECore permet l'expression de concepts et des relations entre concepts. Il constitue un noyau minimal pour la spécification des méta-modèles et il permet d'assurer l'interopérabilité entre outils par échange de méta-données. EMF se constitue de trois composants : *core*, *edit* and *codegen*. *core* comprend le méta-méta-modèle de ECore et les mécanismes permettent de gérer la persistance, la sérialisation/matérialisation, un service de traces pour le suivi des modifications et un service de validation des méta-modèles.*edit* comprend un éditeur de méta-modèles et *codegen* permet de produire des classes Java. Encodé en XML, un méta-modèle se présente comme une description inerte principalement structurelle et la génération de code est indispensable pour évaluer opérationnellement les méta-modèles. Eclipse-EMF/ECore constitue en fait une base de développement d'autres méta-ateliers comme AMMA/KM3 [JB06] ou encore Kermeta [MFJ05],

AMMA/KM3 [JB06] permet la spécification de méta-modèles avec le langage KM3. Le rôle principal d'un atelier spécialisé est la transformation de modèles. Les transformations de modèles sont exprimées à l'aide du langage ATL (*ATL Transformation language*). Une transformation spécifie la construction d'un modèle cible à partir d'un ensemble de modèles sources. ATL est déclaratif, il est basé sur des règles permettant d'accéder aux éléments de modèles par navigation au sein de sa structure. Le langage OCL est utilisé pour exprimer

5. GOPRR pour *Graph, Object, Property, Relationship, Role*

les règles de navigation et d'accès aux éléments de modèles. Les modèles manipulés sont conformes à ECore. Ces spécifications en ATL sont interprétées par l'atelier.

Kermeta met en œuvre une extension de EMOF 2.0 qui permet de spécifier du comportement impératif aux entités du méta-modèle. Le prototypage et la transformation de modèles est ainsi possible directement dans le méta-atelier. Cependant, les capacités en termes d'interprétation sont limitées et sont utilisées essentiellement pour la validation précoce des modèles (interprétation de pré et post conditions et évaluation des contraintes) et l'exécution des générateurs de code. La construction de prototypes repose sur la génération de code.

2.3.6.6 Moose

Moose [NDG05, Moo] est une plateforme dédiée à l'analyse de composants logiciels et de données. De notre point de vue, Moose est un méta-atelier. Le cœur est développé sous Smalltalk. Il permet l'analyse de composant logiciels écrit dans d'autres langages que Smalltalk (Java, C, C++...). A partir de l'analyse de ces composants, un méta-modèle interne à Moose est constitué. Ce méta-modèle peut être instrumenté pour être exécuté, analysé ou visualisé sous différentes formes. L'intégration est assurée par une tour de modélisation spécifique outillée par un ensemble de services appelé Fame. Fame est construit sur la base du méta-méta-modèle FM3, du méta-modèle générique Famix et du format d'échange MSE de données. Moose est la pierre angulaire de la méthode d'évaluation et d'analyse de logiciels *Humane Assessement* [Gir10].

2.4 Conclusion

Ce chapitre nous a permis d'introduire le vocabulaire que nous utilisons dans le contexte de l'IDM. Le concept de modèle, la relation de description et le concept d'espace technique ont tout d'abord été introduits. Nous avons ensuite décrit les notions de langage et de méta-modèle toujours suivant le point de vue de l'IDM. Les activités de modélisation et de méta-modélisation ainsi que l'outillage permettant de mener ces activités ont ensuite été présentés.

Le défi motivant mon activité de recherche concerne l'amélioration de la qualité des outils et au final des applications pour lesquelles les outils sont construits. Les travaux menés dans le cadre de l"IDM ont permis des avancées pour l'amélioration de la qualité. Nous verrons cependant que ce bilan peut-être mitigé. Notre objectif de recherche est notamment d'accroitre l'impact de l'IDM au profit de la qualité et de l'adéquation des solutions aux besoins. L'amélioration de la qualité procède par vérification et validation des modèles et des méta-modèles. Dans le chapitre suivant, nous nous intéressons plus particulièrement à ces deux activités.

Chapitre 3

Vérification et validation

La qualité des composants logiciels peut être améliorée par l'ingénierie dirigée par les modèles. L'idée consiste à intégrer la spécification, la vérification et la validation dans un cycle de développement à itérations courtes. La vérification et la validation se pratiquent classiquement au sein d'un modèle exécutable déduit de la spécification. Ces activités influent en retour sur la spécification.

Selon IEEE [610], la notion de *qualité* est décrite pour un système, un composant ou un processus, en deux niveaux de satisfaction, (1) par rapport aux exigences décrites et (2) par rapport aux attentes et aux besoins des utilisateurs. Un niveau de qualité n'a de sens que par rapport au contexte. Ainsi, au besoin de qualité s'ajoute des contraintes de temps de développement et plus généralement de coût de développement. Nous faisons face à la fois aux enjeux d'optimisation des coûts de développement, d'amélioration de la qualité et, en ce qui concerne les systèmes critiques, de la nécessite absolue du bon fonctionnement des applications. La problématique générales est introduite par J.M. Jézéquel et al dans [FEBF06] : "comment réaliser une implantation de qualité avec des spécifications continuellement mouvantes". L'approche est donc souvent pragmatique, le génie logiciel devant se contenter de compromis entre les délais, les coûts et la qualité [FEBF06]. Le défi est double car les améliorations doivent porter conjointement sur le processus et sur les environnements de développement des applications.

Dans [CM00], John Claxton et Peter A. McDougall écrivent que "*évaluer la qualité de n'importe quoi, y compris des modèles consiste en deux activités ; la première activité revient à*

mesurer les bonnes caractéristiques, de la bonne manière et suivant de bonnes règles ; mais le plus important concerne la seconde activité : évaluer quelque-chose sur la base de ses fonctions et de ses usages.

Dans le domaine du calcul scientifique, dans [OR10], la qualité est abordée du point de vue de la crédibilité du modèle : les éléments qui permettent d'assurer la crédibilité des résultats obtenus d'un système sont, la qualité de l'analyste, la qualité du modèle physique, des activités de vérification et de validation, de la quantification des incertitudes et de la finesse de l'analyse. Les auteurs précisent que tous ces éléments sont nécessaires mais qu'aucun n'est suffisant à lui seul.

Dans le domaine de la méta-modélisation et de l'outillage pour l'IDM, la qualité des modèles dépends notamment de celle des langages de modélisation, des outils utilisés pour spécifier les modèles et les transformations, des connaissances et de l'expérience des développeurs en méta-modélisation, de la qualité du processus de modélisation et du processus d'assurance qualité suivi [MD07]. A ces critères de qualité s'ajoute la qualité des transformations pratiquées sur les modèles [MA07].

Les premières définitions des notions de vérification et de validation données fin des années 70 s'appliquaient à la simulation dans le contexte des sciences expérimentales. Depuis, les définitions ont été progressivement précisées. L'émergence de l'IDM a impliqué une adaptation des notions de vérification et de validation, pour tenir compte, notamment de l'impossibilité dans bien des cas, de mener des expériences telles qu'elles sont pratiquées pour les sciences fondamentales.

La suite de ce chapitre rappelle tout d'abord brièvement pourquoi améliorer la vérification et la validation est un défi important. Les aspects importants de la vérification et de la validation concernant la détection et la correction des erreurs sont ensuite introduits. Cette présentation s'appuie sur les travaux menés autour de la qualité pour l'ingénierie du logiciel. Ensuite, les travaux menés dans le domaine de la vérification et de la validation pour les sciences expérimentales sont présentés. Ces travaux ont aboutis à la définition normalisée d'un processus orienté qualité basé sur la simulation et l'expérimentation. Enfin, les spécificités de la qualité dans le cadre de l'IDM sont introduites avec les objectifs principaux de la qualité et les techniques utilisées pour la vérification et la validation pour l'IDM.

3.1 Un enjeux fondamental et des difficultés croissantes

Une grande partie du coût du développement logiciel provient de la vérification et de la validation. Dans [Ost96], Osterweil estime qu'assurer un certain niveau de qualité consomme au moins 50 à 60% des efforts nécessaires à la production d'un logiciel. Ces pourcentages augmentent naturellement pour les systèmes critiques et pour certaines applications, la validation de la conformité du logiciel peut être plus lourde en termes de temps de développement et de coût financier que le développement du logiciel. Une partie de ce surcout est inhérent à l'inadéquation des infrastructures de test et de validation [Res02, MD08]. Dans [MS10], les auteurs on établi que 60% des efforts de développement concernent la mise au point des modèles et que près de la moitié de ce coût est dédié aux modèles de simulation. Améliorer les outils dédiés à la simulation et la validation est donc toujours un enjeu primordial.

D'autre part, les plateformes d'exécution sont maintenant mobiles et hétérogènes. La tendance est à l'augmentation de la complexité avec notamment de plus grandes difficultés pour la mise au point et la validation. Cette augmentation de la complexité du logiciel est permise par un accroissement régulier de la puissance de calcul et de la diminution de la taille

du matériel. La complexité progresse aussi de part les capacités accrues des techniques de développements, des langages et des infrastructures. Par contre, comme le montre le rapport du CAST [SCS12], les qualités non fonctionnelles (performance, sécurité, facilité de maintenance,...) ne sont pas particulièrement favorisées par les langages et techniques actuelles[1]. Dans ce contexte, l'IDM apparait comme bénéfique notamment pour l'apport en terme de validation précoce des artéfacts.

3.2 Détection et correction des erreurs dans l'ingénierie du logiciel

Selon Osterweil, le défi de la recherche en qualité du logiciel est de fournir des outils et des technologies qui permettent à l'industrie du logiciel de déployer des logiciels et des services qui sont fiables et utilisables dans un cadre économique permettant aux entreprises de se concurrencer [Ost96].

La notion d'erreur est définie comme une déviation de l'exécution par rapport à l'intention du programme et à son comportement envisagé. Il s'agit donc d'une notion relative qui dépend fondamentalement de la spécification de l'intention du programme [Ost96]. Certains auteurs élargissent la notion d'erreur. Par exemple, dans [IAE00], une erreur est définie comme la violation mesurable d'une règle générale, d'une méthode, d'une spécification ou d'un résultat. Les erreurs peuvent être introduites pendant toute phase du cycle de développement : pendant l'expression des besoins, la conception, le codage, dans la documentation, ou pendant la correction d'une erreur [Jon08].

3.2.1 Processus de détection et de correction des erreurs

Le processus qui aboutit à la correction d'une erreur (ou anomalie), comprend trois phases : (1) la détection, (2) l'étude de la solution et (3) la correction :

- la détection consiste en une prise de conscience par l'analyse statique ou dynamique d'un comportement non attendu du système ou de l'absence d'un comportement attendu ;
- l'étude de la solution vise à déterminer la ou les causes, les conséquences et la sévérité de l'erreur ;
- la correction consiste soit en l'élimination de l'erreur, soit en son inhibition ou son contournement.

Par rapport au cycle de vie, il peut être relativement simple de détecter une erreur mais très difficile d'en mesurer la sévérité dans les phases amont. Par contre, il peut être plus aisé d'identifier la sévérité d'une erreur dans les phases avancées du cycle de vie mais plus complexe d'en déterminer la cause. La phase d'étude peut-être très coûteuse. D'après Marc Eisenstadt dans [Eis97], la correction d'une erreur est souvent difficile à statuer du fait de la difficulté de relier la cause réelle en amont aux effets constatés et du fait de l'impossibilité de pouvoir utiliser des outils.

1. Les données collectées concernent 745 applications et 160 organisations ; la moitié des applications étudiées sont développées en Java-EE ; les autres langages, à savoir, .Net, ABAP, Cobol et Oracle Forms constituent entre 7 et 11% des applications étudiées ; ce rapport met notamment en avant que les applications écrites en Cobol sont les plus sures

Les auteurs mettent en avant l'intérêt de la détection précoce des erreurs. La détection précoce d'une erreur consiste à prévenir une anomalie avant qu'elle impacte négativement sur le comportement d'un système réellement utilisé. La détection précoce des erreurs permet donc leur prévention. Par exemple, d'après Boehm et Basili dans [BB01], il est 100 fois plus coûteux de corriger une erreur après le déploiement d'une application que pendant l'analyse ou la conception et près de la moitié du développement concerne des modifications qui auraient pu être évitées. De façon similaire, les travaux de Evanco présentés dans [Eva01] montrent que les erreurs trouvées par les tests unitaires sont moins coûteuses que celles découvertes en aval, pendant le test du système. Comme il est indiqué dans [RSS00], l'expérience des développeurs est un facteur important du coût de la maintenance et de la détection des erreurs ; les plus expérimentés trouvent plus rapidement les problèmes. Le niveau d'expertise est aussi noté comme un point important dans [Sin98]. Magne Jørgensen précise cependant que le niveau d'expertise n'est pas à lui seul suffisant, la personne en charge de la maintenance doit aussi bénéficier d'informations et de retours sur les exécutions pour améliorer la qualité de la maintenance et des prédictions possibles [JS02].

Une erreur peut se manifester et être découverte par le test ou l'inspection. Cependant, des erreurs persistent dans le logiciel utilisé. Ces erreurs latentes peuvent être difficiles à détecter [Jon08].

La détection des erreurs nécessite la composition de trois spécifications :

- la spécification du comportement réel,
- la spécification du comportement attendu (intention),
- une procédure pour déterminer la cohérence de ces deux spécifications.

Trois catégories de techniques sont généralement présentées : les *tests dynamiques*, l'*analyse statique* et les *méthodes formelles*. Ces techniques sont décrites dans les trois chapitres suivants.

3.2.2 Les tests dynamiques

La spécification du comportement réel est donnée par le programme. Le comportement attendu peut-être simplement décrit ou formellement spécifié via un langage d'assertions. Un autre moyen est la spécification rigoureuse des résultats intermédiaires ou finaux attendus.

La procédure qui détermine la cohérence des résultats peut reposer sur l'observation et l'expertise humaine. Des outils spécifiques sont aussi développés. On parle d'*oracle* pour désigner l'outil ou la fonction qui automatise l'évaluation de la cohérence des résultats. L'oracle le plus simple procède par comparaison directe entre résultat attendu et résultat réel. Des outils statistiques ou mathématiques peuvent aussi être utilisés. Idéalement, l'oracle est automatiquement déduit des spécifications de résultats attendus et consiste en un composant directement exécutable.

Les tests dynamiques sont très couramment utilisés. Leur succès provient notamment du fait qu'ils s'intègrent naturellement dans l'activité de développement. Par contre, l'adéquation par rapport aux attendus des utilisateurs ou la facilité d'utilisation sont des caractéristiques difficilement contrôlés par l'analyse statique notamment du fait de la difficulté de spécifier formellement ces contraintes et de disposer d'un oracle capable de les exploiter. Une autre difficulté importante provient du choix des éléments du contexte à tester. De plus, les résultats collectés sont relatifs à une exécution dans un contexte donné, en d'autres termes, les résultats ne permettent pas de qualifier le modèle ou le code de façon absolue.

Les tests dynamiques peuvent être de nature diverse. Certains tests ne comportent d'une seule ligne de code alors que d'autres sont constitués d'un grand nombre d'assertions et combinent plusieurs éléments de programme [GLN05].

3.2.2.1 Tests et cycle de vie

Suivant l'étape du cycle de vie pendant laquelle ils sont exploités, on distingue les tests précoces, qui sont intégrés très tôt dans le cycle de vie des tests tardifs qui eux s'intègrent dans le système exécutable final :

- les tests précoces ont pour utilité de découvrir et de prévenir les erreurs ; ils sont considérés comme des outils d'aide au développement dédiés à une qualité "a priori" ;
- les tests tardifs sont implantés pour garantir qu'il n'y a pas d'erreur dans l'exécutable final

Suivant le point de vue de l'étape dans le cycle de développement, on distingue trois grandes catégories de tests : les tests unitaires, les tests fonctionnels et les tests d'intégration :

- les tests unitaires sont des tests précoces utilisés pour contrôler très finement une exécution ; idéalement, un seul élément de programme est évalué dans un contexte d'exécution particulier ;
- les tests d'intégration sont des tests tardifs dont l'objectif est de contrôler le bon fonctionnement d'un système du point de vue de sa composition en sous-systèmes ;
- les tests de système ou tests fonctionnels visent à contrôler une fonctionnalité considérée comme une boite noire ;
- les tests d'acceptation ou dits de recette sont effectués à la livraison, ils sont contractuels et pratiqués en présence de l'utilisateur ou de l'acheteur.

3.2.2.2 Caractéristique testée

Les caractéristiques testées peuvent être de différentes natures. Elles peuvent être fonctionnelles ou non fonctionnelles.

Les caractéristiques fonctionnelles concernent les fonctionnalités requises du point de vue du domaine et de l'application. On s'intéresse à ce que fait le logiciel et ces caractéristiques sont testées sans tenir compte de comment elles sont mise en œuvre.

Une caractéristique non-fonctionnelle concerne un aspect transversal comme la sécurité, la performance en termes de consommation d'une ressource comme la mémoire, le CPU, le réseau. On parle aussi de tests de charge ou de performance. Ces tests dynamiques sont aussi employés pour évaluer un exécutable systématiquement dans des conditions extrêmes dites de "stress". Un stress peut reposer sur les flots de données ou sur les transitions.

Les tests dits de craquement et les tests de récupération après craquement sont aussi utilisés pour évaluer la solidité d'un système et sa capacité à reprendre un niveau d'exécution normal dans le cas ou l'exécution du système est subitement stoppée (volontairement ou non).

3.2.2.3 Implantation des tests dynamiques

Différentes techniques d'implantation des tests sont employées. On distingue deux techniques principales suivant le niveau d'implantation :

- la technique la plus répandue est d'intégrer les tests au niveau du code ; les tests sont programmés comme des fonctions particulières ; les outils comme *sunit* ou *junit* permettent de gérer les tests, de les invoquer et de gérer les résultats obtenus [Bec99] ;
- les tests peuvent être implantés au niveau méta par rapport au code ; ils peuvent reposer sur une représentation en graphe du programme : les nœuds représentent les instructions et les arêtes les transitions possibles ; d'autres techniques s'appuient sur une représentation des flots de données et permettent de tester dynamiquement les accès aux données pendant l'exécution.

Les caractéristiques fonctionnelles sont testées au niveau du code mis en œuvre pour le domaine et l'application. Par contre, les caractéristiques non fonctionnelles sont plus difficilement testées au niveau du code car elles correspondent à des aspects transversaux du logiciel. L'implantation des tests est effectuée au niveau méta. La programmation par aspects ou l'adaptation au niveau des compilateurs, des interpréteurs ou des machines virtuelles sont employés. Par exemple, Juan-Carlos Ruiz-Garcia [RG02], présente un protocole à méta-objets adapté dans une optique de tests.

Suivant la caractéristique des tests, on peut calculer un taux de couverture des tests. Le taux de couverture donne une indication sur le nombre de cas testés par rapport à l'ensemble des tests possibles.

3.2.3 L'analyse statique

L'analyse statique est une méthode de vérification qui repose sur l'étude des propriétés invariantes des artefacts, sans recours à leur exécution, à leur simulation ou à leur animation. Il s'agit de l'examen systématique de la structure déclarée du programme pour inférer que certaines propriétés sont vérifiées sans pour cela procéder à l'exécution de la spécification. L'analyse statique repose sur une modélisation simplificatrice ou un point de vue de l'intention [Ost96]. Les techniques d'analyse statique reposent sur l'inspection. Les outils sont des analyseurs de spécification basés sur un ensemble fixe d'algorithmes ou d'heuristiques de détection des erreurs. Ils analysent le flot de contrôle, les flots de données et les interfaces. Les erreurs trouvées ne sont pas forcément des fautes réelles, l'analyse des résultats produits par les outils de vérification est donc souvent nécessaire [ZWN+06]. La spécification de l'intention et la procédure de comparaison avec le comportement réellement spécifié consiste donc en l'expertise et l'observation humaine.

Historiquement, l'analyse statique c'est fortement développée avec la mise en œuvre des compilateurs. L'analyse se base sur la représentation interne issue de l'analyse de la spécification source. D'autres outils comme par exemple *Lint*, permettent des analyses statiques très poussées séparées de la compilation.

Dans [ZWN+06], l'analyse statique est systématiquement évaluées et quantifiées. Les conclusions de cette étude montrent que l'analyse statique, l'analyse dynamique et l'inspection manuelle sont complémentaires :

- le coût de l'analyse statique par faute détectée est du même ordre de grandeur que celui de l'inspection manuelle du code ; d'après les auteurs, cela montre le faible coût de l'analyse statique ;
- le rendement de l'élimination d'erreur permis par l'analyse statique est du même ordre que celui permis par l'inspection manuelle ;
- le rendement de l'élimination des dysfonctionnements permis par les tests dynamiques est deux à trois fois supérieur à celui permis par l'analyse statique ;

- le nombre d'erreurs trouvées par l'analyse statique d'un module est un bon indicateur de la qualité du module avant que les tests dynamiques puissent être exécutés ;
- l'analyse statique permet très principalement de détecter deux catégories d'erreurs que sont les expressions de comparaison invalides et les erreurs d'affectation alors que 90% des erreurs détectées ensuite par l'inspection manuelle sont des fautes dans les algorithmes et la documentation ;
- en grande majorité, les erreurs trouvées après mise en exploitation (trouvées par le client) sont dans les fonctions et les algorithmes ;
- 80% des erreurs trouvées correspondent à 20% des catégories d'erreurs possibles ;
- un très grand pourcentage d'erreurs détectées par l'analyse statique permet d'éviter des erreurs graves liées à la sécurité.

L'analyse statique soufre de la relative imprécision du modèle d'exécution qu'il utilise. Les contrôles sont effectués localement de point à point dans la représentation interne. Il est notamment difficile de vérifier des contraintes globales comme la validité d'un chemin dans son ensemble (problème des boucles dont le nombre d'itérations est déterminé dynamiquement par exemple). Un autre problème est la relative limitation des possibilités de contrôles consistant principalement en une analyse du flot de contrôle ou celle du flot de données.

3.2.4 Les méthodes formelles

Certaines techniques d'analyse statique sont dite formelles. L'analyse formelle est une méthode fondée sur les mathématiques permettant le raisonnement sur la sémantique du logiciel basée sur une spécification précise du comportement attendu [DHR+07]. Le terme "formel" provient du fait que la cohérence est aussi vérifiée systématiquement par raisonnement basé sur une sémantique formelle automatiquement exploitable. Une méthode formelle permet de garantir la correspondance entre la spécification du comportement et l'intention. Le bien fondé mathématique permet de garantir le résultat mais aussi la méthode utilisé pour obtenir le résultat [DHR+07]. L'oracle met en œuvre un mécanisme de preuve. Ces méthodes sont notamment employées pour les systèmes critiques dont le dysfonctionnement peut impliquer une mise en danger de personnes ou l'intégrité de matériels onéreux. Les techniques fondamentales sont le *model checking*, l'*interprétation abstraite* et les *méthodes déductives* [DHR+07].

Les méthodes formelles sont utilisées via des langages ou des environnements dédiés dont la principale qualité est de permettre une analyse automatique précise des modèles et de la sémantique des modèles. Par exemple, le langage Z [Spi92] permet de décrire un système suivant deux points de vue : celui des données et des opérations opérées sur les données. Les tests des spécifications peuvent être produits automatiquement par traduction vers une machine à états et vers un réseau de pétri qui permet le test suivant le point de vue du flot de contrôle.

Le méta-langage ASF+SDF [Kli93] permet la spécification formelle de la sémantique d'une syntaxe concrète. Dans [Ams10], un ensemble de métriques a été défini pour évaluer des transformations opérées sur du texte. Les transformations sont exprimées à l'aide d'ASF+SDF. A partir de l'analyse d'une spécification source en ASF+SDF, un outil de contrôle de la qualité des transformations de modèles est construit.

Un autre exemple de méta-environnement permettant la définition formelle de la sémantique est explicité dans [Kai89]. Kaiser décrit un méta-environnement permettant le test de

programmes écrit à l'aide d'un langage textuel dont la syntaxe et la sémantique sont spécifié par le méta-environnement.

Les méthodes formelles souffrent par contre de limitations importantes. D'une part, elles sont complexes et coûteuse à utiliser. Ce problème est bien introduit dans [KM08] et les travaux présentés dans cette publication visent à faciliter la définition de la sémantique formelle de langages. D'autre part, les points de vue pouvant être exprimés sont restreints [FR07]. Il est en effet difficile de garantir que la spécification de l'intention fonctionnelle correspond précisément au besoin et de couvrir correctement les exigences. Il est aussi souligné que leur utilisation est plus liée à la validation de l'intention fonctionnelle. Ces méthodes sont plus difficiles à adapter pour la validation d'intentions non fonctionnelles comme la facilité d'utilisation ou l'efficacité en termes de consommation de ressources.

3.3 Vérification et validation pour les sciences fondamentales

La mise en œuvre d'outils pour la validation ou la simulation dans le cadre scientifique des sciences fondamentales met en œuvre des techniques de validation éprouvées et parfois même étayées par la théorie. Par exemple, d'après Charles M. Macal dans [Mac05], en physique, le *modèle standard* est le nom donné à la théorie actuelle des particules fondamentales et de leurs interactions. Ce *modèle standard* est une théorie validée, les prédictions qu'elle permet d'établir correspondent avec précision aux expérimentations et toutes les particules prédites par la théorie ont été trouvées. La vérification et la validation s'articulent autour du modèle exécutable, représenté par le code du "simulateur".

Le cadre des sciences fondamentales est particulier suivant trois points. (1) L'intention première du développement logiciel est orientée vers la validation, l'invalidation, l'anticipation ou l'extrapolation par rapport à une théorie scientifique. (2) Les modèles sont spécialisés dans leur forme et leur nature pour cette intention. (3) L'expérimentation concrète est possible et l'objectif est que le modèle soit compatible à la fois à la simulation logicielle et à l'expérimentation.

Historiquement, les concepts de vérification et de validation sont d'abord précisés et instrumentés dans le cadre des sciences expérimentales. Les définitions des notions de vérification et de validation sont précisées depuis environ cinq décennies [OR10] :

- *Vérification* : vise à garantir qu'un modèle exécutable représente correctement un modèle conceptuel dans les limites de précision prévues.
- *Validation* : vise à garantir qu'un modèle exécutable est suffisamment précis et cohérent dans les limites de son domaine d'applicabilité et pour l'application prévue.
- *Qualification* : vise à obtenir un modèle conceptuel adapté au domaine et à l'application.

La figure 3.1 montre les relations entre la réalité, le modèle conceptuel et le modèle exécutable dans le cadre de ces définitions. Ces premières définitions considèrent la notion de *garantie* ou de *preuve*. La vérification se focalise sur le modèle exécutable et sur sa correction par rapport à ce qui est établi par le modèle conceptuel. La validation intervient après la vérification et se focalise sur la correction du modèle exécutable par rapport au domaine et à l'application. Le modèle conceptuel est dit qualifié par la vérification et la validation par rapport au système simulé.

Les définitions données ensuite par l'IEEE et surtout par le DoD précisent que la vérification et la validation sont des processus continus et qu'il n'est pas possible de déterminer la fin de ces processus. Hors mis pour des cas triviaux, il est en effet impossible de démontrer qu'un

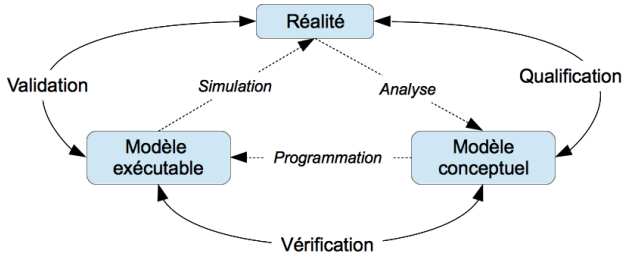

FIGURE 3.1 – Rôles de la validation et de la vérification dans le domaine de la simulation

modèle est juste par rapport à un domaine et une application [OR10]. L'objectif est d'acquérir un maximum de confiance envers les modèles conceptuels et les modèles exécutables.

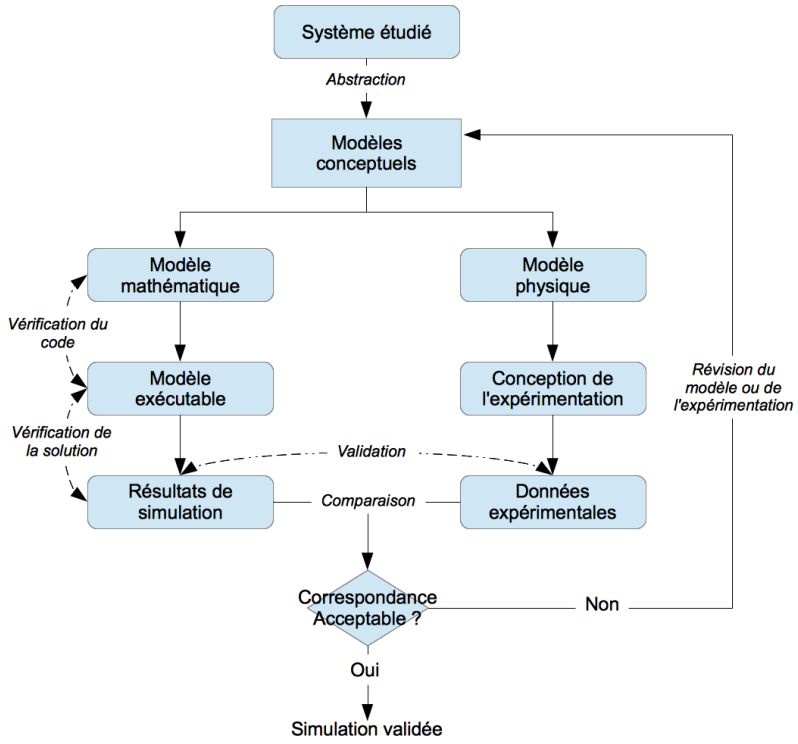

FIGURE 3.2 – Vérification et validation selon l'ASME

En 2006, la société Américaine des ingénieurs en mécanique (ASME) [2] a produit un guide de la vérification et de la validation dans le domaine de la mécanique [ASM06]. Le schéma présenté par la figure 3.2 montre une vue simplifiée des activités de vérification et de validation telle qu'elle est donnée dans le guide de l'ASME :

– la vérification se décompose en deux sous-processus, la *vérification du code* et la *vérification de la solution* ;

2. American Society of Mechanical Engineer

- la validation est opérée suivant trois étapes pour mesurer l'adéquation du modèle :
  - par comparaison entre le résultat de la simulation et le résultat expérimental,
  - par utilisation du modèle pour anticiper ou extrapoler,
  - et par rapport aux exigences.

3.3.1 Vérification

La double nature du processus de vérification correspond d'une part à une vérification de la forme du modèle exécutable et d'autre part à la vérification du processus de production du code du modèle exécutable.

D'après Mary Jean Harrold, l'objectif de la vérification est d'aider les développeurs à produire du logiciel de qualité par la détection précoce des erreurs [Har00]. Dans [OR10], il est précisé que l'objectif principal de la vérification du code est de garantir que les algorithmes sont correctement mis en œuvre et qu'ils fonctionnent comme prévu. La vérification doit prendre en considération les problèmes de compatibilité vis-à-vis des systèmes d'exploitation, des compilateurs, des librairies utilisées,... De ce point de vue, le processus de vérification est empirique, basé sur les tests unitaires, l'observation et l'analyse des résultats d'exécution et les méthodes formelles.

La vérification de la solution vise à garantir que le modèle exécutable est produit correctement. C'est alors plus le processus de production du logiciel qui est évalué et non le logiciel lui-même. Conventionnellement, il s'agit de vérifier comment l'analyse est effectuée mais aussi la gestion du code et des versions, la planification, le processus de développement et de maintenance, les aspects administratifs et les besoins en documentation. La vérification s'apparente donc à un processus d'assurance qualité qui vise à limiter les erreurs humaines pour la production du modèle exécutable.

3.3.2 Validation

La première étape de la validation est opérée sur la base d'une comparaison entre des résultats de simulation et des résultats expérimentaux. L'adéquation est calculée via l'application d'un opérateur de validation qui doit être prédéfini. Cette évaluation est reproductible dans des conditions strictement équivalentes. Les hypothèses sur les données de validation doivent décrire précisément les types des données, les bornes, la précision attendue. Pour la validation, le lien avec l'expression du besoin et le modèle conceptuel est ainsi explicitement établi.

La seconde étape consiste à mesurer si le modèle résiste à l'extrapolation. Il s'agit d'utiliser le modèle pour des simulations pour lesquelles on n'a pas de résultat d'expérimentation ou en aveugle par rapport à des résultats connus. Il n'y a pas comparaison comme c'est le cas dans la première étape mais évaluation de la vraisemblance des résultats vis-à-vis du domaine et de l'intention de l'application.

La troisième étape procède par utilisation du modèle exécutable et évaluation du simulateur dans un certain contexte et vis-à-vis des besoins et des conditions d'utilisation. Typiquement, il peut s'agir de l'évaluation de la rapidité du simulateur, de sa facilité d'utilisation par rapport aux utilisateurs (expérimentés ou novices). L'évaluation est menée au regard de l'expression des exigences et des attentes des utilisateurs.

3.4 Vérification et validation dans le cadre de l'IDM

Le processus de vérification et de validation présenté dans le chapitre précédent a été élaboré et est adapté aux sciences expérimentales pour lesquels des protocoles expérimentaux peuvent être établis. Les aspects qualités de l'IDM héritent directement des travaux menés dans le cadre du développement logiciel et des méthodes éprouvées dans le cadre de la simulation pour les sciences expérimentales. Les définitions données pour les termes de vérification et de validation sont en effet très proches [Mac05] :

- La *vérification* est un processus qui permet d'assurer que les modèles sont bien formés et développés correctement suivant les bonnes pratiques. Les pratiques peuvent être issues de la technique de modélisation, de la communauté des utilisateurs du langage ou de l'atelier mais aussi du domaine et du type d'application pour lequel le modèle est développé.
- La *validation* permet d'assurer que le modèle est cohérent par rapport à son intention en termes de méthodes utilisées et du résultat obtenu ; le but ultime est d'obtenir un modèle utile, c'est-à-dire un modèle qui adresse le bon problème, qui fournit des informations précises sur le système modélisé et que le modèle est finalement réellement utilisé.

L'un des fondements de l'IDM est de considérer le modèle comme l'artéfact pivot pendant tout le cycle de vie. Considérer les modèles tout au long du cycle de vie et non plus uniquement du point de vue du modèle exécutable pose des problèmes spécifiques. Notamment, comment le modèle peut il être validé si [Mac05] :

- des expériences contrôlées ne peuvent être menées sur le système, par exemple si les données expérimentales ne peuvent être collectées qu'une seule fois,
- le système à modéliser n'existe pas,
- le modèle est non déterministe.

Dans [MD], Parastoo Mohagheghi et Vegard Dehlen étudient les caractéristiques particulières de l'IDM au regard de la validation. En substance, ces auteurs précisent que :

- les modèles sont utilisés à toutes les étapes du développement logiciel, des phases amont (modèle des exigences) aux phases de test, de simulation ou de génération de code ; les modèles sont souvent incomplets et imprécis pour les phases amont ; ils sont progressivement précisés et complétés ;
- les modèles peuvent être ou non exécutables ;
- le même système est modélisé suivant plusieurs points de vue ; par exemple, un modèle structurel et modèle comportemental, ou encore tous les modèles possibles avec UML ; chaque vue et chaque niveau d'abstraction peut impliquer des critères de qualité et des métriques spécifiques ;
- les modèles subissent des transformations automatisées ; plus généralement, les modèles sont utilisés comme entrée ou bien sont produits par des outils ; pour une transformation, des informations non présentes dans un modèle source peuvent être composées avec le modèle source ; c'est typiquement le cas dans l'approche MDA de l'OMG ; il est donc nécessaire d'évaluer les modèles à toutes les étapes, le code final n'est plus le seul artefact contrôlé.

Dans ce cadre, la notion de qualité est difficile à définir du fait de sa subjectivité par rapport à la nature du modèle, au niveau d'abstraction, par rapport à la place du modèle dans le cycle

de vie et au processus de développement logiciel. Les contours des activités de vérification et validation ne sont pas aisément discernables. Ils varient suivant les auteurs et le contexte dans lequel le modèle est créé ou utilisé.

3.4.1 Objectifs de la qualité

Comme l'indiquent Parastoo Mohagheghi et al dans [Par08], l'énoncé des objectifs de la qualité dans le contexte de l'IDM varie sensiblement suivant les auteurs, la nature des modèles ou l'espace technique. L'étude décrite dans [Par08] propose une revue des travaux importants au regard des objectifs du contrôle de la qualité pour l'IDM. Les auteurs identifient six classes d'objectifs : la *correction*, la *complétude*, la *cohérence*, la *compréhensibilité*, le *confinement* et l'*adaptabilité*.

3.4.1.1 Correction

La correction est le premier objectif de qualité du modèle. La correction implique premièrement que le modèle est constitué d'éléments et de relations en éléments valides par rapport à la syntaxe du langage utilisé mais aussi par rapport au domaine. Deuxièmement, les règles et conventions de modélisation doivent être respectées.

L'objectif de correction concerne donc au prime abord des aspects syntaxiques et des règles de présentation mais aussi à des aspects sémantiques, liés au domaine et à la compréhension du domaine. Dans [Lin94], Lindland et al parlent d'objectifs *pragmatiques*. Dans [Unh05] Unhelkar parle d'*esthétique*. La notion de règle de présentation se rapporte à celle de style. La description des styles peut faire l'objet de guides de modélisation spécifiques au langage ou au domaine. Un modèle peut être valide syntaxiquement et sémantiquement mais invalide par rapport à un style. La notion de style peut être purement syntaxique, c'est à dire liée à la présentation même du modèle[3]. Elle peut aussi être liée à la manière dont les éléments de modèle sont agencés et organisés entre eux. Par exemple, pour un modèle objet, il peut s'agir du respect de patron de conception (par exemple les patrons de conception objet [Ga96]), ou de la pratique de modélisation particulières liées à l'outillage, au langage (comme par exemple [Pal95]) ou au domaine d'application (par exemple, pour le développement de modèles [VB04]).

3.4.1.2 Complétude

La complétude implique que le modèle comprend tous les éléments nécessaires, suffisamment détaillés par rapport à l'objectif du modèle. D'un point de vue fonctionnel, cela peut signifier que tous les comportements sont spécifiés suffisamment précisément pour permettre la description de tests unitaires associés au modèle. D'un point de vue architectural, un modèle complet reflètera intégralement l'architecture finale de l'application avec tous ses composants et toutes leurs connections.

3.4.1.3 Cohérence

La cohérence implique que les éléments de modèles ou les modèles ne sont pas contradictoires entre eux. Les caractéristiques du modèle doivent être en cohérence par exemple par rapport au niveau d'abstraction ou à la phase de développement. La cohérence doit aussi

3. Par exemple `https://www.securecoding.cert.org` ou encore `http://java.sun.com/docs/codeconv/`

être préservée entre plusieurs niveaux d'abstraction et différentes phases du développement. Notamment, la cohérence doit être préservée par les transformations de modèle. La cohérence concerne des aspects syntaxiques comme par exemple le nommage ou les règles d'association entre éléments. La cohérence sémantique concerne l'interprétation des éléments par rapport au domaine et à l'application modélisée.

3.4.1.4 Compréhensibilité

La compréhensibilité implique que le modèle est exploitable manuellement par un utilisateur ou un lecteur ou bien automatiquement par un outil. La compréhensibilité et fortement liée aux aspects esthétiques ou pragmatique de l'objectif de correction. La compréhensibilité concerne l'organisation du modèle, sa simplicité, le choix des termes en cohérence par rapport au domaine.

3.4.1.5 Confinement

Le confinement implique que le modèle est en cohérence par rapport à l'objectif de la modélisation pour l'application et par rapport au type de système développé. Cet objectif concerne le choix de la nature du modèle, du bon niveau d'abstraction. Le modèle ne doit pas comprendre des informations inutiles et être aussi simple que possible par rapport au niveau d'abstraction et à la phase dans le cycle de développement.

3.4.1.6 Adaptabilité

L'adaptabilité est la capacité du modèle à accepter les changements et les améliorations. L'adaptabilité est indispensable pour permettre au système d'évoluer au fur et à mesure qu'évoluent la compréhension du domaine, les besoins des utilisateurs et les conditions techniques de mise en œuvre. L'adaptabilité est un objectif crucial pour la maintenance d'un système. Cet objectif doit être autorisé par le langage de modélisation mais aussi par les outils utilisés.

3.4.2 Techniques de contrôle de la qualité

Les techniques de contrôle peuvent être manuelles ou automatisées : le *contrôle manuel* est effectué directement par un individu ou un groupe d'individu par relecture, observation, manipulation et discussion du ou des modèles. Le contrôle manuel peut être outillé pour questionner, représenter le modèle de façon particulière ou encore simuler ou prototyper tout ou partie d'une application. Le *contrôle automatisé* procède lui de l'utilisation d'un outil automatiquement produit à partir du ou des modèles en cours de contrôle.

3.4.2.1 Le contrôle outillé

Le contrôle outillé s'appuie sur une ou des applications développées pour le contrôle même du modèle. Ces applications ne sont donc pas développées pour répondre au besoin de l'application mais pour faciliter le contrôle des modèles utilisés à une certaine étape du cycle de vie pour produire les modèles ou les artefacts de l'étape suivante.

Il peut s'agir d'outils de consultation ou de visualisation particulière. Ces outils sont construits spécifiquement. Ils peuvent être liés de manière horizontale au langage de modélisation utilisé ou de façon plus verticale, être liés au domaine et à l'application.

Les environnements et les ateliers de modélisation peuvent prévoir des infrastructures ou des bibliothèques facilitant ou automatisant la construction des outils. Par exemple, dans [Mor91, KMS92], les auteurs décrivent une infrastructure logicielle permettant de construire automatiquement un outil de visualisation de modèle STEP à partir de la spécification du schéma de données en EXPRESS. *Moose* [NDG05] est un environnement notamment dédié à la ré-ingénierie. Une part très importante de *Moose* comprend des outils dédiés à la représentation des modules d'une application et à la construction de butineurs de modèles spécifiques. Dans [LX05], les auteurs décrivent un outil et une méthode pour déverminer un programme. Cette méthode est basée sur la spécification de scénarios permettant de visualiser de façon particulière le comportement d'une application. Dans [PCH12], les auteurs décrivent un outil permettant de visualiser les tests existants pour aider à l'identification des tests manquants et ainsi améliorer le taux de couverture des tests.

Le prototypage est un contrôle outillé permettant de contrôler des exigences, des idées de conception, de fonctionnement d'une application ou encore de son apparence par le développement d'un exemplaire d'application simplifié qui se focalise sur les points à contrôler. Le prototype facilite les interactions humaines permettant la critique d'une solution [BL05].

Il y a trois catégories de prototypes [LHB10] : les prototypes de conception, les prototypes jetables et les prototypes évolutifs. Les prototypes de conception sont développés en phase amont pour explorer des idées ou une apparence de l'application. Le prototype jetable est développé pour explorer rapidement et très spécifiquement les fonctionnalités ou l'apparence de l'application cible. Le terme jetable signifie que le prototype est oublié lorsqu'il a permis de répondre aux questions que l'on se pose sur l'application. Dans ce cas, la qualité du prototype lui même importe peu. Par contre, un prototype évolutif est maintenu en continu tout au long du développement et de la maintenance d'une application. Il permet d'évaluer ou de simuler une partie de l'application et de vérifier et valider une évolution avant d'adapter l'application prototypée. La qualité du prototype, notamment, sa capacité à évoluer est un point crucial.

Certaines publications utilisent les termes de prototype *hautement fidèle* ou à l'inverse de *faiblement fidèle*. Le prototype hautement fidèle est un prototype calquant le système ou l'aspect du sous-système pour lequel le prototype est développé. Ce type de prototype est coûteux à mettre en œuvre. Il sert à valider très précisément le système prototypé. A l'inverse, le prototype faiblement fidèle est schématique et rapidement développé. Un tel prototype sert à montrer une orientation de la solution ou à consolider une idée concernant un aspect du système à produire. Un prototype évolutif peut être tout d'abord vu comme un prototype faiblement fidèle est être ensuite progressivement adapté et devenir hautement fidèle.

Un problème important, en particulier en ce qui concerne les prototypes évolutifs, est le maintien de la qualité et le surcroit d'activité que peut demander l'évolution des prototypes à chaque itération. Le fait qu'un prototype constitue une solution simplifié qui ne prend pas en considération toutes les contraintes du système à produire (en particulier certaines exigences non-fonctionnelles comme la performance ou la sécurité) peut aussi conduire à des prises de décision erronées.

3.4.2.2 Le contrôle automatisé

Le contrôle automatisé constitue une catégorie particulière de contrôle outillé. Les outils permettant ce type de contrôle sont automatiquement produits à partir du modèle pour lequel il est exploité. La spécificité du contrôle automatisé est qu'aucune mise en œuvre manuelle n'est nécessaire. Trois approches permettent le contrôle automatisé : les méthodes formelles, les tests basés sur le modèle et la vérification par traduction.

Méthodes formelles. La première approche de contrôle automatisé consiste en l'utilisation de méthodes formelles pour la spécification de la sémantique du langage de modélisation. Les principes de cette approche sont décrits dans les chapitres 2.2.5 et 3.2.4.

Tests basés sur le modèle. La seconde approche est le *test basé sur le modèle* [Bin99, UL10]. Pratiquement, les tests et un oracle permettant de décider du résultat des tests sont produits automatiquement à partir du modèle. Pour que cette approche soit possible, le modèle doit avoir les caractéristiques suivantes [Bin99] :

- la sémantique du modèle doit être clairement et précisément définie ;
- le modèle doit être complet (au sens de la complétude vue au chapitre 3.4.1.2) par rapport aux tests à produire ;
- il est suffisamment abstrait pour éviter un nombre trop important de tests redondants ;
- il est suffisamment concret pour permettre la production de tests utiles ;

Vérification par traduction. Dans certains cas, le modèle est indirectement testé par traduction vers un autre modèle qui lui peut être testé soit par une approche formelle soit par des tests basés sur le modèle.

3.5 Conclusion

Dans ce chapitre, nous avons rappelé pourquoi améliorer la vérification et la validation est un défi important et de plus en plus difficile. Nous avons ensuite introduit les notions importantes concernant la détection et la correction des erreurs. Nous avons ensuite décrit plus précisément les activités de vérification et de validation en nous appuyant sur les travaux menés pour les sciences expérimentales. Ces travaux ont abouti à la définition normalisée d'un processus orienté qualité basé sur la simulation et l'expérimentation. Ensuite, les spécificités de la qualité dans le cadre de l'IDM ont été introduites avec les objectifs principaux de la qualité et les techniques utilisées pour la vérification et la validation pour l'IDM.

Selon Mohagheghi et al [Par08], les efforts menés pour la qualité se regroupent suivant deux grandes catégories. La première concerne le processus de développement avec notamment sa formalisation et son automatisation et la seconde est plus liée à l'outillage. Dans la suite du rapport, nous faisons un point sur les pratiques actuelles concernant la vérification et la validation. L'adéquation des outils disponibles dans le cadre de l'IDM est notamment discutée.

Chapitre 4

Constat sur les pratiques et l'outillage de l'IDM

De nombreuses études s'intéressent aux problèmes liés à la qualité du logiciel. Un grand nombre de travaux a été publié notamment pendant les années 90. Les auteurs mettent en évidence la manière dont les erreurs sont réellement détectées et traitées avec un accent mis sur le fait que la détection des erreurs est pratiquée principalement via les tests fonctionnels, tardivement dans le cycle de développement.

On constate que des techniques rudimentaires de déverminage sont souvent employées. Par exemple, dans [Lie97], Henry Lieberman indique que majoritairement, les développeurs ajoutent manuellement l'affichage de traces dans le code pour comprendre et trouver les erreurs. De même, d'après Marc Eisenstadt dans [Eis97], l'affichage de traces et le raisonnement sont majoritairement utilisés (pour 80% des cas étudiés). Par corrélation avec le fait que cette technique est très souvent utilisée par les étudiants, les études scientifiques indiquent que la technique de déverminage évolue très peu et que l'amélioration des pratiques n'est pas une priorité dans l'industrie du logiciel. Dans [Sin98], Janice Singer étudie les pratiques de maintenance du logiciel en s'appuyant sur des interviews pratiquées sur un échantillon de 13 entreprises et organisations gouvernementales. Cette étude fait apparaître que le code du système est considéré très majoritairement comme la source la plus fiable pour comprendre le logiciel et effectuer les opérations de maintenance. Parallèlement, la documentation n'est pas considérée comme fiable. Enfin, la correction d'un problème est pratiquée par sa reproduction en exécutant le logiciel. La conséquence est qu'un problème non reproductible demeure généralement non résolu.

Comme l'attestent de nombreuses publications parues depuis la fin des années 90, l'IDM et les méta-ateliers pour l'IDM ont permis des avancées. Cependant, nous considérons que

le bilan est mitigé notamment à cause du manque d'agilité autorisé par l'outillage. Dans ce chapitre, nous présentons tout d'abord notre point de vue sur le bilan. Nous détaillons ensuite les problèmes que nous avons identifiés relatifs aux méta-ateliers pour l'IDM et à leur processus d'utilisation.

4.1 Apports des outils pour l'IDM : un bilan mitigé

Depuis la fin des années 90 jusqu'à maintenant, de très nombreuses publications attestent de l'intérêt de l'IDM pour la validation. Dans le domaine du nucléaire [IAE00] ou celui des systèmes temps-réel critiques des contrôles automatisés sont systématiquement pratiqués et l'IDM a permis des avancées tant au niveau méthode qu'au niveau des langages et des outils.

Par exemple, dans le domaine des systèmes temps-réel critiques, deux standards majeurs on été défini pour la spécification des systèmes, le profil UML/Marte [OMG07] et le langage AADL [AAD09]. Ces langages permettent des vérifications syntaxiques et un certain niveau de vérification sémantique des traitements et des architectures. Les systèmes modélisés sont validés formellement par traduction par exemple vers les réseaux de Pétri ou vers les automates temporisés. L'ordonnancement des systèmes peut être validé via par exemple, l'utilisation de l'environnement spécialisé Cheddar [SLNM04b]. Les tests fonctionnels peuvent être effectués sur des applications développées en Ada ou en Java. Le code généré peut-être contraint à respecter les spécifications du profil Ravenscar [BDV04]. Ce profil garanti que l'ordonnancement est, par construction, vérifiable statiquement. (voir par exemple les travaux de Gilles Lasnier [Las12] autour de la vérification et la validation des systèmes temps-réel critiques).

De manière plus générale, après plus de deux décennies de travaux et d'applications, l'impact de l'IDM dans l'industrie est toujours difficile à évaluer. Certains auteurs mettent en avant un bilan mitigé [MD08, Mel09, Amb03]. Le gain pouvant être obtenu en termes de qualité et sécurité du logiciel est un facteur déterminant, or, ce gain n'est pas clairement établi.

Concernant la validation, le processus doit prendre en considération le besoin de validation précoce des artefacts. D'un autre point de vue, la validation nécessite des spécifications relativement précises des contraintes du domaine et parfois même des contraintes matérielles.

De fait, une partie importante de la validation est pratiquée tardivement sur le système final. Donc, un compromis doit être trouvé entre haut niveau d'abstraction, séparation des préoccupations, validation précoce des modèles et un contrôle fonctionnel impliquant un plus bas niveau d'abstraction et une faible séparation des préoccupations.

En outre, la grande diversité des notations, les problèmes d'interopérabilité entre outils, la complexité d'intégration des artefacts impliquant souvent un haut niveau d'expertise spécifique sont autant de freins à l'adoption de l'IDM. Dès 1996, Sandra Slaughter dans [SB96] explique que l'utilisation de l'approche générative a des impacts significatifs sur la quantité d'efforts nécessaires à la maintenance. La génération de code peut avoir des conséquences sur l'organisation du code d'un logiciel impliquant plus de difficultés pour comprendre le nouveau code en cas de changement.

L'approche générative est très majoritairement utilisée dans l'IDM. Cette approche suppose de disposer de méta-modèles suffisamment précis et détaillés pour autoriser l'intégration des méta-modèles et la transformation des modèles.

Cependant, pour disposer de spécifications précises répondant au besoin, en amont, un travail d'élaboration, de vérification et de validation est nécessaire. Ce travail d'élaboration s'effectue par itération successives. Au cours de chaque itération, les méta-modèles sont

vérifiés et validés. Plus particulièrement lorsque le problème à résoudre est abordé, les vérifications et les validations ont pour conséquence des remises en cause des méta-modèles et leur modification est nécessaire.

L'approche par raffinement successifs prônée par l'IDM est justement adaptée à ce processus. Les différents projets et expérimentations et les méta-ateliers développés pour supporter cette approche tirent partie de la transformation de modèle et plus particulièrement de la traduction des spécifications vers un formalisme exécutable pour assurer une partie importante des validations. Le coût et le manque de souplesse de l'approche par transformation de modèle sont plus rarement mentionnés par les auteurs. Pratiquement, le coût d'une itération intégrant la génération de code est élevé. Le processus est pénalisé et manque d'agilité.

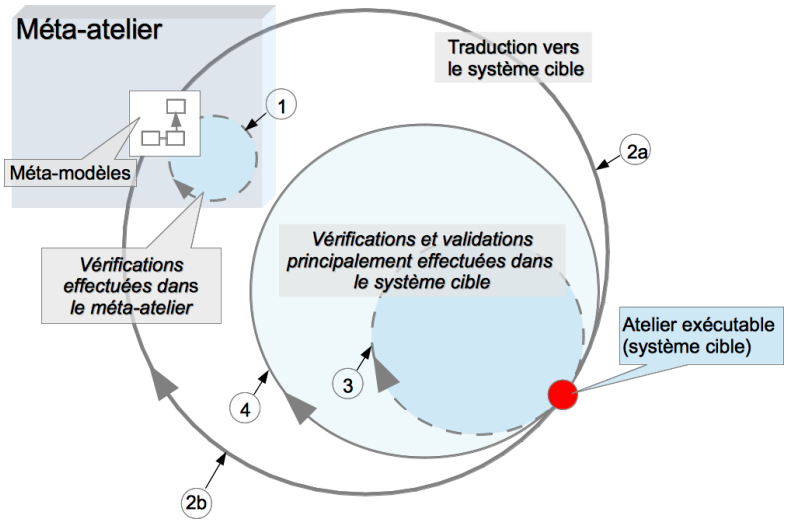

FIGURE 4.1 – Approche classique pour la vérification et la validation

La notion d'élaboration implique la possibilité d'itérer au sein d'un méta-atelier pour parvenir à des spécifications adéquates par rapport au besoin. Pour cette élaboration, le contrôle par vérification et validation est utilisé pour converger vers une solution satisfaisant a priori le besoin et permettre à l'utilisateur du méta-atelier d'apprécier l'adéquation de la solution vis-à-vis du besoin.

La figure 4.1 montre une représentation schématique des activités de méta-modélisation, de vérification, de validation et production du système cible telles qu'elles sont pratiquées classiquement. Le méta-atelier se constitue essentiellement d'un ensemble d'éditeurs pour produire la structure des méta-modèles (syntaxe abstraite), éditer les contraintes d'intégrité ou bien les pré ou post-conditions, déclarer les règles de transformation ou encore spécifier la syntaxe concrète et la relier à la syntaxe abstraite. Les méta-modèles sont élaborés dans le méta-atelier. Les vérifications des méta-modèles sont menées au sein du méta-atelier (1). Puis, les méta-modèles sont traduits par transformation de modèle pour être exploités au sein de la plateforme cible (2a). Des vérifications (3) et des validations (4) sont ensuite menées au sein de la plateforme d'exécution cible. Les anomalies éventuelles sont alors prises en compte pour corriger les méta-modèles (2b).

Les contrôles possibles au niveau d'un méta-atelier sont donc essentiellement des vérifications :

- pour chaque point de vue, des contrôles spécifiques peuvent être effectués par observation manuelle ou automatiquement, par les outils d'édition ou des outils spécifiques ;
- le méta-atelier joue le rôle d'intégrateur de ces spécifications ; la cohérence des spécifications entre elles peut aussi être vérifiée ;
- la transformation elle-même permet d'assurer un certain niveau de vérification (le modèle est transformable).

Au niveau du méta-atelier, la vérification est éventuellement outillée. Elle peut procéder d'une analyse statique des types et des associations ou procéder par traduction vers un formalisme vérifiable dynamiquement (par exemple vers un réseau de pétri).

Si un atelier exécutable est produit, il permet d'effectuer des contrôles par une mise à l'épreuve dans une plateforme cible. Si des modules ou des paquetages sont produits, alors ils peuvent être utilisés pour construire des prototypes ou être intégrés à un système cible.

La vérification et la validation sont en grande partie menées lorsque le système cible est opérationnel. Selon notre point de vue, la prévention des erreurs n'est pas optimale. Les conséquences négatives de cette détection tardive est un accroissement du coût du développement et un risque de baisse du niveau de qualité du système cible. En effet, cette approche est coûteuse en termes de temps. Le cycle édition-validation peut être long voire très long du fait des manipulations à effectuer et du fait de la discontinuité entre les méta-modèles et les modèles conformes. Enfin, l'utilisation de langages statiquement typés, justifiée dans le cadre de la transformation de modèle, diminue l'agilité de l'édition, des validations et du prototypage.

Au regard des méta-ateliers existants, nous identifions donc trois problèmes :

- le premier problème est lié au processus d'utilisation des méta-ateliers et plus particulièrement à l'utilisation intensive de la génération de code ;
- le deuxième problème est lié à la nature des langages de méta-modélisation et notamment, en ce qui concerne les approches mixtes, l'utilisation systématique du typage statique ;
- le troisième problème concerne la mise en œuvre des méta-ateliers et notamment celle du maintient du lien de causalité.

4.2 Problèmes liés à l'approche générative

L'approche générative est souvent indispensable. Cependant nous identifions deux problèmes important liés à l'approche générative :

- la mise au point d'un langage peut nécessiter beaucoup d'itérations et il se pose alors le problème du coût d'une itération,
- le code généré peut-être modifié ce qui augmente la complexité technique du processus

4.2.1 Problème du coût d'une itération

Les contrôles effectués après transformation sont tardifs et une approche purement générative est problématique en termes de vélocité des cycles de développement car la rétro-influence n'est possible que tardivement et le développement ne peut pas être incrémental. En cas de comportement non conforme, il faut :

- découvrir l'origine de l'anomalie,

- corriger les méta-modèles
- vérifier localement les méta-modèles,
- générer le code de l'atelier cible,
- compiler le code,
- exécuter l'atelier cible pour valider la correction, ce qui implique :
  - d'initialiser et de configurer l'atelier cible exécutable pour parvenir au point de déclenchement de l'anomalie,
  - de contrôler ensuite l'exécution de l'atelier cible.

De plus, les générateurs de code peuvent présenter des anomalies. Les transformations de modèles sont difficiles à écrire, vérifier et maintenir. Pour pallier ces difficultés, des outils spécifiques doivent être développés [VP04, MD, Ams10].

Il est constaté que l'activité de correction s'appuie sur l'exécution et la reproduction des problèmes [Sin98]. Le processus par transformation de modèle peut nécessiter plusieurs itérations pour une seule anomalie. Deux conséquences sont observées :

- le cycle de validation peut être très coûteux et le risque est qu'une solution comportant des anomalies ou ne satisfaisant pas les besoins soit finalement conservée [RFBlO01]
- le grain des évolutions est d'autant plus important qu'il est coûteux de les prendre en compte et de les contrôler ; déterminer l'origine d'une anomalie est alors plus difficile.

Eclipse-EMF est très utilisé. C'est un représentant typique de l'approche par génération de code. Le langage de niveau $M3$ est ECore. Un méta-modèle ECore se constitue d'un graphe de concepts. Encodé en XML, un méta-modèle se présente comme une description inerte principalement structurelle et la génération de code est indispensable pour évaluer opérationnellement les méta-modèles. Par exemple, Eclipse-EMF est utilisé pour le projet *TOPCASED Model Simulation*[1] qui propose un méta-modèle pour la simulation de machines à états de UML2.0. A partir de ce méta-modèle, un atelier est construit autour des classes java produites automatiquement à partir du méta-modèle. L'atelier construit permet de spécifier des modèles comprenant des états et des transitions. Pour activer un tel modèle, il est nécessaire de produire le code java correspondant aux états et aux évènements modélisés et d'intégrer le code généré à un moteur de simulation générique. Ainsi, pour valider le méta-modèle ECore, deux niveaux de génération de code sont nécessaires.

De nombreuses publications relèvent le problème du manque de vélocité en mettant en avant le besoin de disposer de méta-modèles exécutables [DG06, MFJ05, Pai04, PBO04, RFBlO01, NDR09, SK03, DAS11]. Considérer qu'un méta-modèle doit être exécutable revient à considérer la méta-modélisation comme une activité de programmation à haut niveau d'abstraction. Cette activité doit alors s'appuyer :

- sur des outils qui s'apparentent à des environnements de programmation,
- sur une machine virtuelle pour l'exécution des méta-modèles.

ConceptBase, GME/MetaGME et MetaEdit+ permettent de vérifier des règles d'intégrités. Avec ConceptBase il est possible d'animer plus spécifiquement des modèles mais au prix de spécification formelles et au sein d'une architecture client-serveur (Modèle des bases de

1. https://gforge.enseeiht.fr/projects/topcased-ms

données actives fondée sur la réponse à des évènements et les exceptions). Pour le prototypage ne respectant pas ce cadre formel, la génération de code doit être utilisée et repose sur la programmation en Prolog.

AMMA/KM3 [JB06] intègre une machine virtuelle pour interpréter les transformations de modèle spécifiées avec ATL. Cependant l'animation et le prototypage demeurent dépendants de la transformation de modèles.

MetaEdit+ [Met] permet de mettre en œuvre des éditeurs de modèles graphiques à partir de la spécification de langages spécifiques à un domaine. L'éditeur est conçu comme un interpréteur de méta-modèle et son exécution ne nécessite pas de génération de code. MetaEdit+ permet aussi l'interprétation de règles OCL et l'interprétation de transformation de modèles. Le guide d'utilisation de MetaEdit+ indique que les étapes préliminaires de la spécification d'un atelier de modélisation consistent en l'identification et la définition des types d'objets, des types de relations, des rôles et de leurs propriétés. MetaEdit+ permet ces spécifications à l'aide d'interfaces graphiques. Mais l'identification de ces différents éléments de méta-modèles nécessite une connaissance a priori du domaine et du besoin. Le prototypage permettant de mettre à l'épreuve le méta-modèle n'est pas directement possible. Il est nécessaire de traduire des modèles à l'aide du langage de script *Merle* et d'utiliser le résultat de la traduction dans des développements exploratoires.

Kermeta, Dome et Moose permettent la programmation de prototypes directement au sein du méta-atelier. En particulier, Kermeta est une extension d'EMOF 2.0 et permet de spécifier du comportement impératif aux entités du méta-modèle. Le prototypage et la transformation de modèles est ainsi possible directement dans le méta-atelier. Pratiquement, les capacités en termes d'interprétation sont utilisées essentiellement pour la validation précoce des modèles (interprétation de pré et post conditions et évaluation des contraintes) et l'exécution des générateurs de code. La construction de prototypes repose pour beaucoup sur la génération de code.

Dome est un méta-atelier intégré à un environnement Smalltalk. Dome permet la définition de méta-modèles et de leur comportement. La spécification peut être interprétée ou directement compilée en Smalltalk. La structure des méta-modèles est spécifiée graphiquement et le comportement est programmé à l'aide d'une variante du langage *Scheme*. L'exécution peut être assurée soit directement en Scheme soit en Smalltalk. Étant donné sont intégration dans un environnement Smalltalk, les prototypes sont construits et exécuté directement dans le méta-atelier, sans génération de code et les modifications apportées au méta-modèle sont directement prise en compte par les éditeurs et les prototypes.

Moose est aussi un environnement intégré dans un système Smalltalk. L'objectif premier de Moose n'est pas de définir un langage particulier mais d'aider à l'analyse de composants logiciels notamment par l'utilisation d'outils préexistants ou la méta-programmation d'outils de visualisation ou par le calcul de métriques. Les outils disponibles sont génériques et peuvent être adaptés à des besoins spécifiques. Tout comme pour Dome, les outils sont directement exécutés au sein du méta-atelier, sans génération de code.

4.2.2 Problème de la modification du code généré

L'intérêt de la méta-modélisation associée à la génération de code est de masquer les spécificités de la plate-forme d'exécution cible dans les méta-modèles. Les générateurs de code sont des méta-outils par nature génériques souvent destinés à être réutilisés pour des domaines différents. En conséquence, le code généré est de nature générique donc non spécifique aux fonctionnalités du domaine. Il est par exemple souvent possible de générer le squelette des

classes, des composants d'accès aux données ou des composants de validation ou de tests basés sur les types décrits dans les méta-modèles.

L'utilisation des langages spécifiques au domaine permet de pallier à ce problème. Une partie du code domaine peut aussi être automatiquement produit. Cependant, un modèle étant une représentation simplifiée d'un système, il est très rare de pouvoir générer 100% d'un système cible.

Le code produit est donc incomplet par rapport aux besoins de l'application et les développeurs doivent enrichir les méta-modèles avec des spécifications concernant la plate-forme d'exécution et aussi compléter manuellement le code généré de façon à implanter le comportement spécifique pour le domaine, pour l'application ou pour contrôler les aspects non fonctionnels.

La modification directe du code généré est bien sur à proscrire. Plusieurs techniques permettent d'adapter le code généré pour étendre ou modifier le comportement du système. Dans [Kle08], cinq techniques classiques sont présentées :

- *programmation évènementielle.* Par le biais d'un abonnement à un type d'évènement particulier, le comportement additionnel peut être automatiquement invoqué lors de la réception d'une exception ou d'un évènement ;
- *framework.* La modification ou l'ajout de fonctionnalités peut intervenir au niveau de fonctions non générées mais systématiquement appelées ; ces fonctions non générées peuvent être vide par défaut et être complétées suivant le besoin ;
- *Classes partielles.* C# dispose de la possibilité de construire des classes par portions ; une portion de classe peut être générée et d'autres peuvent être produite manuellement ;
- *Sous-classes vides.* Lorsque le langage cible est orienté objet, des sous-classes vides peuvent être générées et systématiquement utilisées ; ainsi, pour ajouter ou modifier un comportement, le code est programmé dans la sous-classe ; les sous-classes vides ne sont générées qu'une seule fois ou à la demande ;
- *Portion de code annoté.* le code ajouté ou le code généré peuvent être marqués ou encadrés par des annotations ; le code généré est ainsi discernable par le générateur de code qui peut ainsi préserver le code manuellement ajouté.

Quelque-soit la technique employée, l'implantation de code spécifique peut être très coûteuse. Dans [Amb03], Scott W. Ambler indique que dans certains cas, 90% du temps est consacré à compléter le code généré. Le code peut être intégré manuellement à des fins de validation du système. Le code ajouté manuellement doit être préservé par les régénérations successives. l'hétérogénéité des artefacts à intégrer implique une complexité plus importante du processus, de la gestion des configurations et des outils.

4.3 Problèmes liés au typage statique des méta-modèles

L'approche générative nécessite que les méta-modèles soient typés statiquement. En effet, un processus de génération de code est dirigé par les types déclarés. MOF, ECore, Java, Eiffel ou des langages spécifiques comme Kermeta [Fle06] ou KM3 [JB06] peuvent être utilisés comme langage de niveau $M3$. L'approche est déclarative et/ou impérative. Outre la nécessité du typage pour la transformation de modèle, les raisons de l'utilisation du typage statique sont multiples. L'avantage majeur du typage statique est qu'il permet un contrôle précoce des spécifications. Ce bénéfice est donc en accord avec le besoin de vérification précoce des artéfacts. Le typage statique constitue aussi une documentation de qualité d'autant plus

qu'elle est fiable et directement insérée dans les spécifications. Le typage statique autorise des analyses fines, des optimisations et des transformations automatisées des méta-modèles. Par exemple, les recherches menées dans le cadre du projet Kermeta mettent en avant des possibilités d'utilisation des types pour la reconnaissance de patrons et la transformation automatique de méta-modèles basés sur ces patrons [Fle06]. Enfin, le typage statique permet aux éditeurs et aux outils du méta-atelier de se comporter spécifiquement (complétion automatique ou présentation de menus contextuels déduits des types déclarés).

Par exemple, Ada est un exemple typique de langage à typage statique fort. La notion de type fort consiste à rendre impossible les opérations entre expressions de types différents. Dans [MCSH10] les auteurs précisent que deux expressions sont compatibles si et seulement si leur type est identique. L'identité de type se concrétise dans le langage par une égalité du nom du type. Ainsi, deux types numériques entiers portant un nom différent sont incompatibles. Dans un tel langage, un type statiquement déclaré représente une contrainte prédéfinie. Cependant, dans le cadre d'une approche semi-formelle, le typage statique ne suffit pas à garantir que les associations et que le comportement réel sont valides [TW07]. Le typage statique a trois inconvénients sensibles plus particulièrement au regard de la nature de la phase d'ingénierie et du besoin de souplesse :

- le premier inconvénient est relatif au confort et à l'efficacité de l'édition des méta-modèles ; le typage statique introduit une certaine lourdeur dans l'écriture des spécifications ;
- le typage statique implique que l'utilisateur du méta-atelier a une idée a priori précise quant aux types à utiliser et à leurs rôles dans le système, or l'objectif de la phase d'ingénierie est de découvrir la forme adéquate du méta-modèle et notamment les types des méta-données, des associations et des cardinalités.
- le typage statique a tendance à rigidifier les spécifications du méta-modèle [NBD+05] alors que pendant la phase d'ingénierie, une grande souplesse est nécessaire.

Pour mettre au point un système ou un langage complexe, un niveau élevé de souplesse est indispensable. Les étapes préliminaires, et dans une moindre mesure, la préparation d'une évolution d'un système peuvent être caractérisées de la façon suivante :

- la conceptualisation est dirigée par un besoin qui peut être plus ou moins précisément spécifié,
- les concepts manipulés sont tout d'abord identifiés,
- les associations entre concepts et les compositions, notamment les types précis des concepts associés et les cardinalités sont plus difficiles à établir ; les choix effectués sont souvent remis en cause,
- la manipulation des concepts par des prototypes et des simulations permet de préciser les associations.

Pendant la phase d'ingénierie, des prototypes ou des simulations peuvent être mis en œuvre. Ces développements sont particulièrement mouvants et instables car l'utilisateur cherche à cerner une forme adéquate du méta-modèle par rapport au besoin. Les modifications sont pratiquées par des changements locaux. L'effet d'un changement, doit être très précocement observable de façon à éviter les régressions et valider le changement opéré. Les types statiques empêchent les changements locaux et les vérifications immédiates après changement car le méta-modèle et les composants logiciels qui l'utilisent doivent être intégralement valides du point de vue du typage [NBD+05]. Par exemple, modifier la définition

du type d'une information dans une structure Ada implique la revue de toutes les fonctions et les paquetages qui utilisent cette structure.

Le typage statique diminue le niveau d'abstraction autorisé par le langage puisque les types introduisent des contraintes supplémentaires. Pour converger précocement vers une solution satisfaisante au besoin, le typage statique peut-être très contraignant pour la spécification et l'évolution des prototypes.

Le typage statique et le typage dynamique ont chacun leur utilité et se complètent [MD04]. Disposer à la fois du typage dynamique et du typage statique est souhaitable pour permettre la souplesse et le besoin de précision. Une spécification dans un langage dynamique peut être complétée par des informations de type. On parle de typage optionnel [TW07].

Dans [Spi01], Diomidis Spinellis décrit des patterns pour la mise en œuvre de langages spécifiques. En particulier, les patterns *pipeline*, *Langage Extension* et *Langage Specialization* sont explicités. Concernant *pipeline*, il est possible d'appliquer ce pattern lorsque plusieurs langages sont utilisés et que le résultat du traitement d'un modèle exprimé dans un langage particulier sert d'entrée à un traitement exprimé dans un autre langage. Au cours de cette chaine de traitements, le modèle original en entrée peut-être transformé notamment par des informations de type. Ainsi, un même élément peut être décoré par différentes annotations. Chaque type d'annotation étant ajouté pour un aspect particulier. Plusieurs systèmes de types statiques peuvent ainsi cohabiter dans une même spécification. De plus, le modèle d'origine peut lui ne pas être typé ou annoté. L'application des patterns *Langage Extension* et *Langage Specialization* peut aussi impliquer la cohabitation de plusieurs systèmes de type dans un même modèle.

4.4 Problèmes liés à la mise en œuvre du lien de causalité

Dans le cadre d'une approche générative, un méta-atelier peut-être considéré comme un méta-système (voir chapitre 2.1.5.2) et le lien de causalité est maintenu par transformation de modèle. Du fait de l'approche générative, une discontinuité entre les méta-modèles et les modèles conformes est introduite. La transformation de modèle entretient une forme statique du lien de causalité. Les modifications structurelles et comportementales opérées au niveau d'un méta-modèle sont suivies d'une évolution des modèles cibles issus de la transformation de modèles sources. Le lien de causalité entre le niveau du méta-modèle et celui des modèles conformes n'est pas dynamique du fait de la séparation entre la spécification la mise en œuvre [RFBlO01]. Ainsi, une modification d'un méta-modèle n'est pas immédiatement et automatiquement suivie d'une adaptation des modèles conformes.

Par exemple, il n'est pas possible, pendant l'exécution, de modifier la structure d'un méta-modèle ou son comportement et de reprendre directement l'exécution. La traçabilité entre les méta-modèles et les artefacts manipulés n'est pas directe et est plus difficilement observables puisqu'elle repose sur la transformation de modèle.

L'organisation en fichiers exploités séparément, pour la spécification des méta-modèles (par exemple dans un fichier XML), pour le résultat de leur traduction vers un formalisme exécutable (par exemple en paquetages Java) et pour le stockage des modèles conformes (par exemple dans des fichiers XML) induit un processus transactionnel qui se caractérise par des étapes successives : une modification est suivie d'une étape d'adaptation des modèles conformes. L'automatisation est plus difficile et n'est pas dynamique. De plus, ces différentes étapes se concrétisent par des manipulations supplémentaires devant être opérées par l'utilisateur.

Pour que le lien de causalité soit maintenu dynamiquement, deux conditions sont nécessaires :

- les mécanismes du méta-atelier doivent être réflexifs
- les modèles, les méta-modèles et le méta-méta-modèle doivent être maintenus en cohérence au sein d'un repository persistant.

La mise en œuvre de mécanismes réflexifs n'est pas théoriquement impossible avec un langage statiquement typé, elle est néanmoins très difficile. Les langages dynamiques disposent plus facilement de cette capacité [TW07]. À notre connaissance, parmi les méta-ateliers étudiés, ConceptBase, MetaEdit+ et Dome maintiennent le lien de causalité de façon dynamique.

4.5 Conclusion

Dans ce chapitre nous avons dressé un constat concernant l'utilisation de l'IDM et des méta-ateliers propres à l'IDM au regard de la vérification et de la validation. Des avancées importantes ont été décrites par de nombreuses publications. La vérification et la validation ont pu être améliorées plus particulièrement dans certains domaines et nous citons notamment les travaux menés dans le cadre des systèmes temps réel critiques. Cependant, le bilan est mitigé et nous avons identifiés trois problèmes importants liés d'une part (1), à l'approche générative et d'autre part (2), à l'utilisation des langages typés statiquement pour la spécification des méta-modèles et (3) liés à la mise en œuvre du maintient du lien de causalité au sein des méta-ateliers. La suite de ce rapport présente notre contribution concernant ces problèmes menés dans le cadre du projet Platypus.

Deuxième partie

Le projet Platypus

Dans cette partie, nous présentons nos travaux dans le cadre de l'IDM et plus spécifiquement dans le cadre de la méta-modélisation pour la construction d'outils à des fins de vérification et de validation des méta-modèles. L'objectif et d'autoriser plus de vérification et de validation précoce. Nous regardons comment faciliter l'identification et la correction des anomalies directement depuis l'outillage au sein duquel sont produits les méta-modèles. L'effort de recherche intègre à la fois la proposition d'un processus de méta-modélisation facilitant la validation mais aussi un environnement pour l'élaboration des méta-modèles et pour la validation des méta-modèles et des modèles conformes.

La particularité de la méthode est de considérer qu'un processus de méta-modélisation adapté doit non seulement permettre de construire les méta-modèles pour spécifier le système ou le langage à mettre au point mais aussi favoriser la compréhension de l'ingénieur vis-à-vis du problème. Une meilleure compréhension acquise plus tôt pendant le cycle de développement doit permettre à l'ingénieur de maximiser les possibilités de vérification et de validation avant de produire le système cible.

Chapitre 5

Une approche agile pour la spécification des méta-modèles

Parallèlement à l'intérêt porté à la méthode et aux processus de développement logiciel prônés par l'IDM, on observe un intérêt important pour les méthodes agiles [Bec99, Amb02, Agi, BBB+09]. Nous constatons que les méta-ateliers existants favorisent l'approche générative. Avec cette approche, la durée d'une itération augmente le coût d'une itération en termes de temps et rend plus difficile le contrôle du système produit. La détection précoce des anomalies n'est pas favorisée. L'approche générative implique un risque important de non adaptation de l'atelier et de présence d'anomalies.

Nous pensons qu'il est possible d'améliorer la vélocité de la vérification, de la validation et du remodelage des méta-modèles. Nous proposons donc d'exploiter une approche agile et un outillage dédié qui favorisent des itérations courtes et incrémentales au sein même du méta-atelier.

Dans la suite de ce chapitre, après avoir présenté globalement la méthode et ses objectifs, nous expliquons pourquoi l'exploitation d'une méthode agile est possible pour la méta-modélisation. Nous expliquons ensuite pourquoi un processus agile est adapté à la méta-

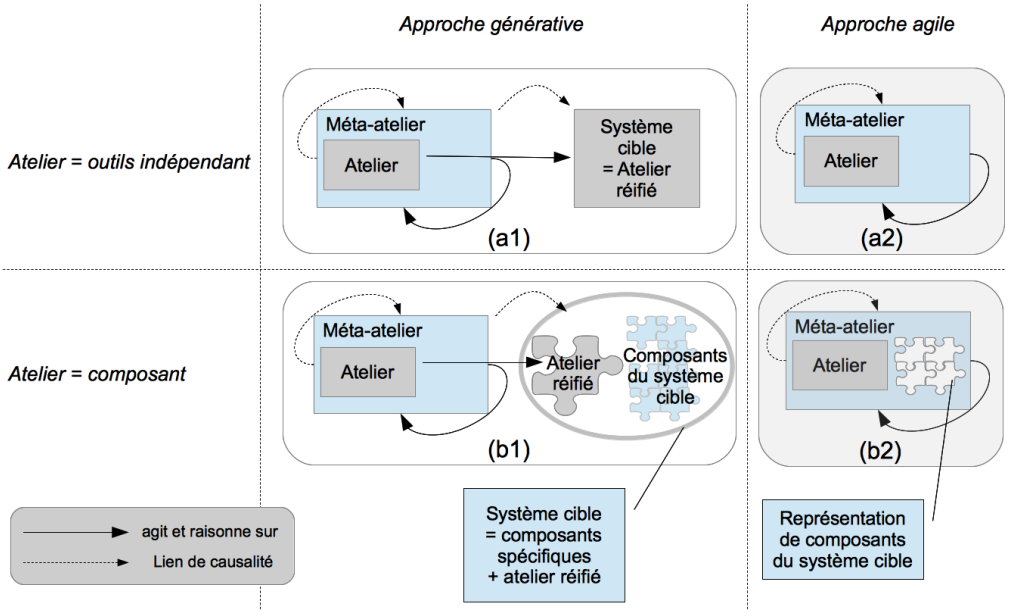

FIGURE 5.1 – Approche générative versus approche agile

modélisation en nous appuyant sur les caractéristiques des itérations et les besoins au niveau des spécifications. Nous expliquons ensuite comment le méta-atelier et le système cible sont intégrés pour permettre leur utilisation en synergie pour la validation. Nous concluons en précisant les caractéristiques désirables du méta-atelier pour supporter cette approche agile.

5.1 Présentation générale de la méthode

Les utilisations possibles des ateliers produits sont très diverses. Par exemple, il peut s'agir d'un outil utilisé pour éditer, vérifier, valider des modèles conformes aux méta-modèles sur la base desquels l'atelier est construit. Il peut aussi s'agir d'un composant intégré à une application. Notre propos est de faciliter la production et la validation d'un atelier. La figure 5.1 montre deux cas typiques : un atelier peut être un outil indépendant (ligne du haut) ou bien constituer un ou plusieurs composants d'un système (ligne du bas). Si l'atelier est un outil indépendant, alors, le système cible est constitué de l'atelier, sinon, le système cible est constitué de l'atelier et de composants spécifiques.

Dans le cadre de l'approche agile (colonne de droite), le méta-atelier constitue une plateforme d'exécution. Si l'atelier est un outil indépendant, alors il est directement exécuté au sein du méta-atelier (cas a2 de la figure 5.1). Si l'atelier constitue un ou plusieurs composants du système cible, alors, des composants spécifiques additionnels sont éventuellement intégrés au méta-atelier pour pouvoir exécuter le système cible au sein du méta-atelier (cas b2 de la figure 5.1). Pour le cas b2, on parlera de simulation du système cible au sein du méta-atelier. Dans les cas b1 et b2, des outils additionnels utiles à la vérification et à la validation précoce de l'atelier peuvent être développés et exécutés au sein du méta-atelier. Pour toutes nos expérimentations, la plateforme d'exécution cible est différente de celle du méta-atelier. L'approche générative est alors indispensable pour produire le résultat final, soit un outil

indépendant ou bien une partie d'un système cible répondant aux besoins des utilisateurs.

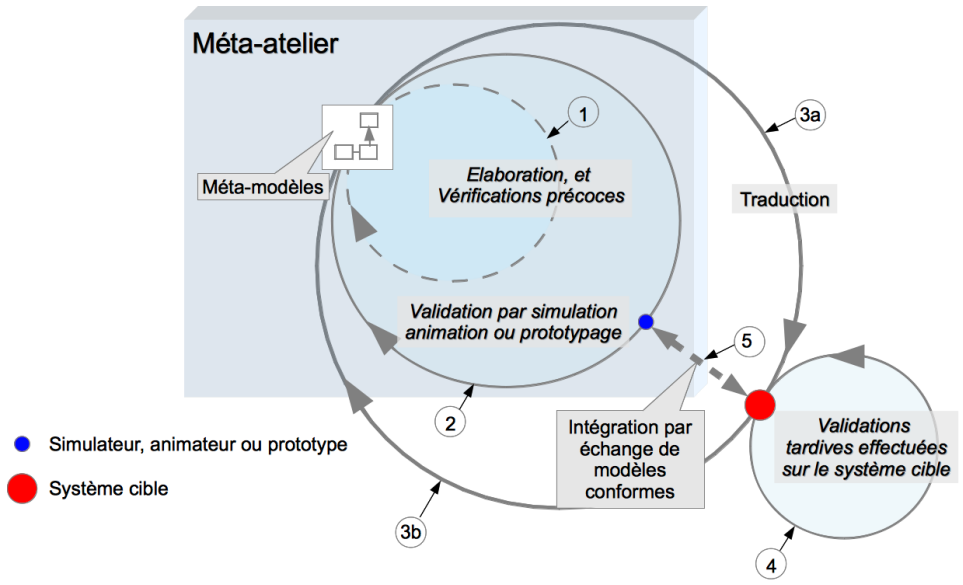

FIGURE 5.2 – Point de vue synthétique de notre approche pour maximiser les validations au sein du méta-atelier

La figure 5.2 donne un point de vue synthétique de notre approche. L'objectif est de maximiser les vérifications et surtout les validations possibles au sein du méta-atelier (1) et (2). Le système cible demeure un élément important pour la validation. De la même façon que pour l'approche classique présentée dans le chapitre 4.1, tout ou partie du système cible est produit automatiquement à partir du méta-atelier (3a). Par contre, la part des vérifications et des validations tardives (4) est diminuée par rapport à l'approche classique.

Le système cible et le méta-atelier sont intégrés pour permettre l'échange de modèles conformes aux méta-modèles mis au point dans le méta-atelier (5). Cette intégration permet notamment de bénéficier et d'exploiter, au sein du méta-atelier, de résultats de validations tardives effectuées au niveau du système cible. La rétro-influence (3b) peut ainsi être facilitée.

Pour couvrir le plus largement possible les besoins en terme de vérification et de validation, il doit être possible à la fois de favoriser l'acquisition des connaissances concernant le domaine et l'application. La convergence progressive vers une solution doit aussi être facilitée par la manipulation des méta-modèles tout d'abord à haut niveau d'abstraction puis spécifiés suffisamment précisément pour autoriser la production du système cible. Enfin, le résultat des validations effectuées au sein du système cible doivent aussi pouvoir être exploitées dans le méta-atelier.

5.2 Adéquation entre la méta-modélisation et les méthodes agiles

Une première idée forte du projet Platypus est de tirer profit de certains aspects des méthodes agiles en faveur de la qualité et de l'adéquation des ateliers produits par rapport au besoin. Sur la base du manifeste de la modélisation agile [BBB+09], les principes qui nous

intéressent plus particulièrement sont [1] :

- la priorité est de satisfaire les besoins ; le client fait partie intégrante de l'équipe de développement et il intervient en continu pour la validation ;
- l'évolution du projet est mesuré par rapport au fonctionnement du système et le système est livré fréquemment ;
- la simplicité est essentielle.

Ces principes impliquent les pratiques suivantes :

- le code est considéré comme l'artefact de référence ; en ce qui concerne la méta-programmation, l'artefact de référence est le méta-modèle du langage en cours de mise en œuvre ;
- le grain des évolutions est fin ; le code est construit de manière itérative et incrémentale et les itérations sont courtes ; le gain escompté est la détection et la correction précoce des anomalies ;
- les tests et les prototypes sont développés en amont, en relation avec le besoin exprimé par l'utilisateur ; les activités de vérification, de validation et de prototypage peuvent être menées en continu à chaque itération ;
- la résolution de problèmes complexes procède par composition et assemblage d'éléments simples sur la base des interfaces des éléments [Hic11].

Pour que ces pratiques puissent être appliquées, certaines conditions organisationnelles et techniques doivent pouvoir être satisfaites. Dans [TFR05], Daniel Turk et al discutent des hypothèses favorables et défavorables à l'application des méthodes agiles. Du point de vue des caractéristiques organisationnelles (points 1 et 2 de la liste ci-dessous) et du point de vue de la nature de l'application développée (points 3 à 5 de la liste ci-dessous), les méthodes agiles sont considérées comme difficilement applicables si :

1. les développements sont physiquement répartis sur des sites différents (plusieurs agences, off-shore,...),
2. de nombreux développeurs interviennent (entre 10 et 20),
3. le système est de nature critique,
4. l'application est de taille importante,
5. l'objectif est de construire des composants réutilisables.

Concernant la méta-modélisation et la construction d'ateliers spécifiques, nous considérons que les conditions sont plutôt favorables à l'utilisation d'une méthode agile. Notre point de vue est fondé sur les observations suivantes :

- l'organisation du développement peut s'articuler autour d'un nombre limité d'acteurs (voir chapitre 2.3.4) ; par rapport à la méthode agile, l'expert du domaine est le client ; il interagit ainsi naturellement avec le ou les fournisseurs que sont les experts en méthode, en méta-modélisation et ceux ou celui de la plateforme d'exécution (élimination des points 1 et 2) ;
- un atelier pour un langage spécifique est un outil non critique bien qu'il puisse être utilisé pour valider ou développer un système qui lui peut être critique (élimination du point 3) ;

1. Nous ne retenons pas les principes faisant appel à des notions beaucoup plus difficilement mesurables telles que la motivation des développeurs, la confiance au sein du projet, l'auto-organisation des équipes et l'excellence technique.

– un langage spécifique est par nature un ”petit” langage et les développements ne sont pas de taille importante [VDK98, MHS05] ; un langage spécifique fait partie d'un système plus large ; les contours sont bien identifiés [Spi01] ; dans [TMC99], les auteurs parlent de *niveau de restriction* pour signifier la portée restreinte d'un langage spécifique (élimination du point 4) ;
– dans le cadre d'une méthode agile, l'objectif est de satisfaire un besoin spécifique le plus simplement possible ; la spécificité par rapport au domaine est en contradiction avec le développement de composants réutilisables pour plusieurs domaines et plusieurs applications (élimination du point 5).

5.3 Agilitée des itérations

Dans le chapitre 2.3.2 nous expliquons les relations qui s'établissent pendant la conception entre le modèle mental, les modèles conceptuels et le système finalement produit. La rétro-influence qui s'exerce par observation du système produit induit des mécanismes dynamiques et itératifs. Dans cette partie, nous précisons la notion d'itération suivant les deux points de vue suivant :

– le premier point de vue concerne l'objectif double d'une itération qui a pour but à la fois de préciser la connaissance du concepteur et de produire un modèle ; on parle alors de deux dimensions de l'itération, la *dimension cognitive* et la *dimension de la représentation* ;
– le second point de vue considère globalement l'activité de conception comme une succession d'itérations, partant de l'expression d'un besoin vers la production du système et d'une version d'un système.

Ces deux points de vue nous permettent d'expliquer en quoi l'agilité est indispensable au niveau du méta-atelier pour répondre au besoin de vérification et de validation précoce.

5.3.1 Dimension cognitive et dimension de la représentation

La nature de l'activité de conception est double :

– d'une part, le point de vue est tourné vers les concepteurs et il s'agit d'affiner le modèle mental des différents experts qui interviennent afin d'augmenter leur niveau d'expertise vis-à-vis du problème et de comprendre quels sont les outils utiles à la résolution du problème ;
– d'autre part, le point de vue est orienté vers le système ou l'objet à produire ;

La double nature des activités de (méta-)modélisation est identifiée dans [BL05] comme deux dimensions des itérations : *la dimension cognitive* et *la dimension de la représentation*. Ces deux dimensions sont toujours présentes lors d'une activité d'apprentissage. La dimension cognitive correspond au travail et aux représentations produites dans le but de comprendre le domaine, le problème et les solutions pour répondre au besoin. La dimension de la représentation correspond au travail et aux représentations produites pour mettre en œuvre les solutions. La figure 5.3 schématise les relations entre ces deux dimensions pendant une itération. En début d'itération, il s'agit essentiellement d'acquérir des connaissances. L'activité du concepteur est alors orientée vers la dimension cognitive. Concevoir consiste ensuite à traduire les connaissances acquises en des représentations utiles à la solution. La part de la dimension de la représentation augmente pour être prépondérante en fin d'itération.

Une différence fondamentale entre les deux dimensions se situe par rapport à leur relation à l'erreur :

- pour la dimension cognitive, l'erreur peut être bénéfique puisque sa détection et sa correction permettent de préciser la connaissance ;
- pour la dimension de la représentation, l'erreur n'est pas bénéfique et doit être évitée ;

On note aussi l'effet de la rétro-influence décrite dans le chapitre 2.3.2, qui participe à l'acquisition des connaissances (construction du modèle mental) et qui donc favorise la dimension cognitive.

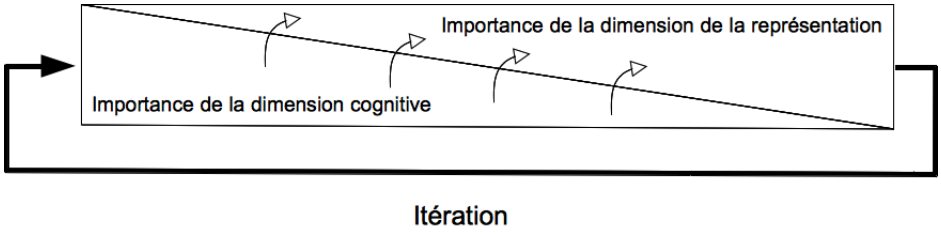

FIGURE 5.3 – La dimension cognitive et la dimension de la représentation au sein d'une itération

Dans le domaine de l'IDM, la dimension cognitive est souvent éludée au profit de la dimension de la représentation. Par exemple, le MDA [OMG03] développé par l'OMG considère différentes catégories de modèles suivant le niveau d'abstraction et l'étape du processus. Le point de vue est tourné vers les modèles et finalement vers les systèmes à produire par la prise en compte progressive des aspects de plus en plus concrets.

La dimension cognitive s'appuie tout d'abord sur des connaissances, des consensus sociaux et des notions implicites. L'idée est proche de la notion d'*élicitation*. Dans le domaine de l'ingénierie du besoin, le processus d'élicitation comprend des activités qui permettent la compréhension des objectifs, des besoins, des contraintes et qui motivent la proposition d'un système. Dans le domaine de l'ingénierie du besoin, la majorité des travaux concernant l'élicitation se concentrent sur les techniques qui permettent la précision de l'expression du besoin et l'exploitation formelle de l'expression des besoins [CA07]. Les techniques favorisant l'élicitation sont la simulation, l'animation et le prototypage. Dans les phases amont, ces techniques se concrétisent par des modèles informels, des descriptions, des tableaux ou encore des dessins. Une part très importante du travail de conception s'effectue à l'aide de ces modèles. Pour les simulations, les différents acteurs peuvent procéder par jeu de rôle. Le prototypage et la simulation sont ensuite utilisés lorsque les modèles peuvent être représentés de façon formelle et que l'automatisation est possible. Les modèles ou les prototypes créés dans un objectif d'élicitation peuvent ne pas persister. Les modèles sont considérés comme jetables [Amb02]. L'objectif est en effet de préciser le modèle mental et non de produire une description persistante.

La figure 5.4 montre les relations entre les dimensions de la représentation, la nature des représentations exploitées et le type de validation privilégié. La représentation informelle est de nature intellectuelle. Au fur et à mesure de l'acquisition des connaissances, des représentations formelles sont utilisées. Tout d'abord des tableaux, des cartes sémantiques, des schémas et des dessins. La validation est alors visuelle et repose sur la communication entre concepteurs. Puis, les représentations informatisées peuvent être utilisées et la validation dynamique, programmée par des animations, des prototypes et des simulations est exploitée.

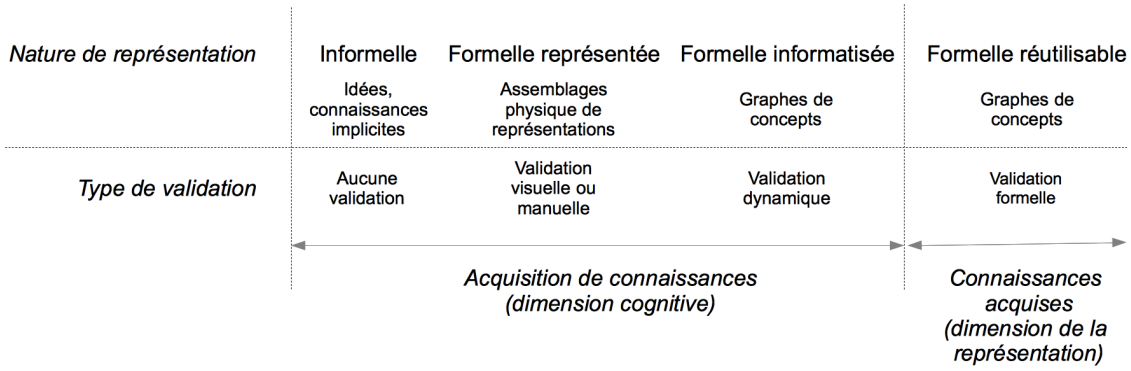

FIGURE 5.4 – Nature de la représentation et de la validation par rapport aux deux dimensions de la représentation

La représentation réutilisable est une représentation informatisée qui est exploitée lorsque la connaissance est considérée comme acquise. La représentation est rigidifiée notamment par le typage statique. Elle est considérée comme réutilisable car elle peut être validée formellement (validation basée sur les types ou une théorie mathématique par exemple). La notion d'élicitation est ici représentée au niveau de l'acquisition des connaissances.

L'élicitation est d'autant plus efficace que les outils utilisés sont adaptés. Une caractéristique importante de la dimension cognitive concerne cette activité d'adaptation de l'outillage. Il s'agit aussi de comprendre et de définir les outils nécessaires à la compréhension et à la résolution du problème. Ce point est relevé par Bill Buxton dans [Bux07] concernant le design et constitue aussi une des caractéristiques importantes des méthodes agiles [Amb02]. Le design comprend des phases de recherche de solutions en utilisant des outils de modélisation mais aussi des phases pendant lesquelles on cherche à définir et adapter les outils eux-mêmes.

5.3.2 Itérations en cours de conception

Dans [Bux07], Bill Buxton explique que la conception est un processus de création et que la création se caractérise par deux phases :

- la phase d'élaboration des options possibles ;
- la phase de réduction ou de choix parmi les options trouvées ;

Dans la figure 5.5, le point à gauche (D) représente le point de départ absolu de la conception. Le point à droite (A) représente le point d'arrivée absolu. Au niveau de D, le problème est posé et la conception commence par une élaboration de la connaissance. Au cours de la conception, une itération commence à partir d'un certain point entre D et A par une phase d'élaboration. Ensuite, la phase de réduction de l'itération permet la sélection des choix, certains sont éliminés et d'autres sont confirmés. Une étape de conception est vue comme une phase de création pendant laquelle on élabore des alternatives dans un premier temps, certaines sont finalement écartées dans un second temps car non conformes au besoin, trop complexes, trop générales,... Le fait d'éliminer des options fait appel à une notion de choix et d'erreur. Le fait de faire des erreurs, de s'en apercevoir et de finalement les éviter est une caractéristique importante de la conception et toute méthode de conception doit intégrer cette caractéristique.

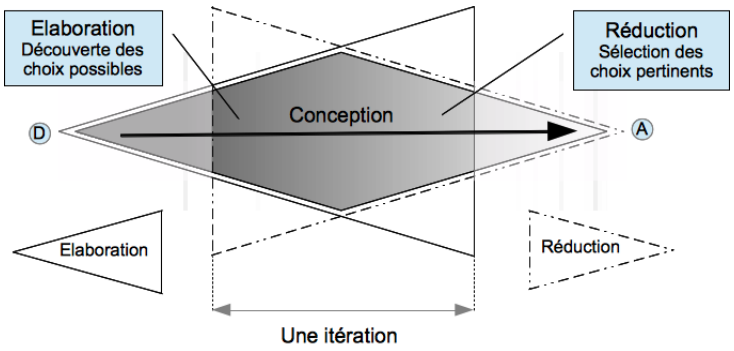

FIGURE 5.5 – Élaboration et réduction lors d'une itération [Bux07]

Un projet informatique est constitué d'une succession d'itérations majeures. Chaque itération débute par une prise en compte de besoins initiaux et abouti à la production d'une version d'un système. Une itération majeure se compose elle même d'itérations mineures. Une itération mineure comprend les deux phases d'élaboration et de réduction présentées précédemment.

Pour une itération mineure n, le fait d'éliminer des choix pendant la phase de réduction ouvre naturellement d'autres perspectives et donc permet l'élaboration de l'itération mineure $n+1$. Intuitivement, d'itération en itération, on rejette plus qu'on ne garde [Bux07] : la part de la phase de réduction à tendance à augmenter alors que la part de la phase d'élaboration a tendance à diminuer. Globalement, pour une itération majeure, le processus de conception se traduit par une convergence vers une solution. Ce point de vue global est montré par la figure 5.6.

En début d'itération majeure, le niveau d'abstraction est élevé, les hypothèses et les choix possibles sont nombreux, les besoins initiaux et le domaine doivent être compris. La première itération abouti à une ébauche de solution. Cette première solution est éventuellement représentée sans utiliser l'outil informatique (on parle aussi de phase d'exploration [Bec99]). Pendant cette phase d'exploration, la dimension cognitive est prépondérante. Les modèles sont utiles comme moyen de communication entre les différents acteurs mais ne sont pas fondamentaux en tant qu'artefact. Par contre, la pratique de la modélisation est importante ainsi que la recherche des représentations adéquates (schéma de données pour la syntaxe abstraite, vocabulaire et syntaxe, modèle de comportement, définition du domaine sémantique, ébauche de plan de traduction, dessin des interfaces homme-machine,...). Cette étape d'exploration, et donc la dimension cognitive, est plus ou moins importante suivant qu'il s'agisse du démarrage ou d'une évolution d'un système existant.

Après une ou plusieurs itérations, les outils adaptés sont connus, les modèles peuvent être précisés et traduits dans des représentations formelles exploitables par l'outil informatique pour des vérifications et des validations. La dimension de la représentation devient de plus en plus importante. L'utilisation de l'outil informatique pendant le processus de conception est montré par les surfaces hachurées de la figure 5.6. On remarque que la proportion des surfaces hachurées par rapport à la surface totale de la phase de réduction augmente au cours d'une itération majeure. En effet, la vérification et la validation peuvent être de plus en plus précisément menées au fur et à mesure que la conception converge vers une solution.

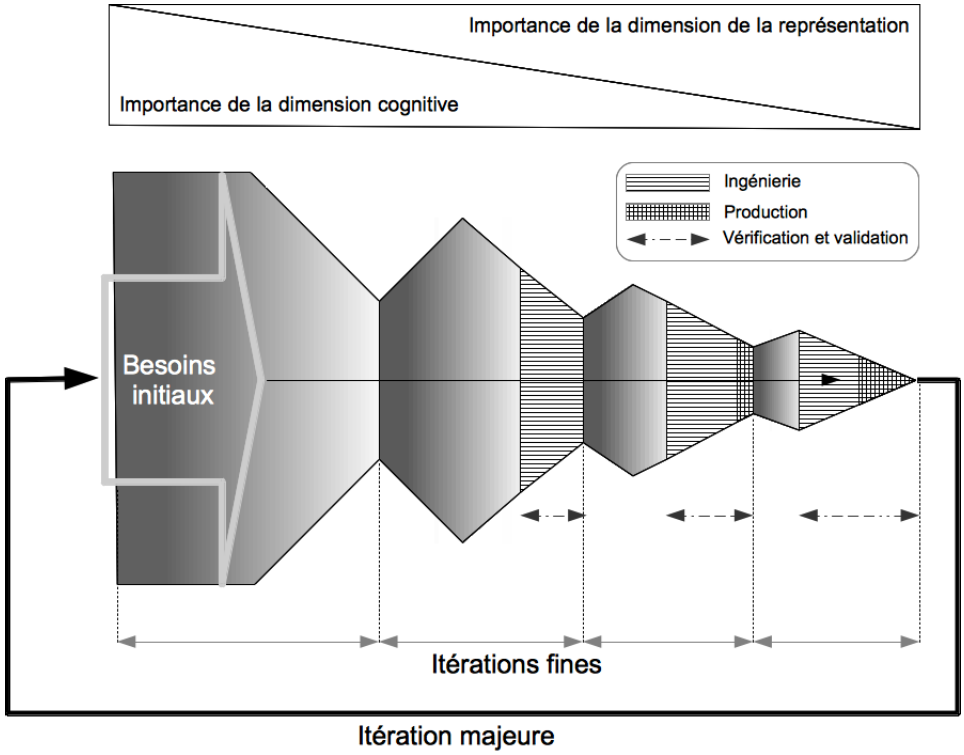

FIGURE 5.6 – Point de vue global du processus de conception inspiré de [Bux07]

5.3.3 Itérations au sein d'un méta-atelier

Si on considère la notion d'itération telle que décrite par les deux points de vue introduits précédemment, le processus doit alors répondre à deux besoins :

- besoin de haut niveau d'abstraction : le niveau d'abstraction exploitable doit être suffisamment élevé pour autoriser l'élaboration et favoriser la création et la manipulation agile des choix ;
- besoin de précision : la transformation de modèles, la vérification et la validation de plus en plus précise doivent être possibles sans limiter l'élaboration menée à haut niveau d'abstraction ;

Ces préoccupations sont sous-jacentes à l'introduction des itérations dans le cycle de vie [JBR99] et l'évolution des méthodes de développement [Ber06]. Par exemple, selon le processus unifié, une itération se compose de deux phases : celle d'*ingénierie* et celle de *production*. Dans la suite, nous utilisons le vocabulaire employé pour décrire une itération du processus unifié. Plus particulièrement, nous utilisons les termes illustrés par la figure 5.7.

Chaque itération du processus unifié correspond à une itération majeure telle que présentée précédemment. Chaque phase se compose de deux étapes. La phase d'ingénierie se compose d'une étape initiale et d'une étape d'élaboration. Cette phase correspond à l'élicitation et à l'élaboration des choix envisageables plus particulièrement importants en début d'itération majeure. La phase de production se compose d'une étape de construction et d'une étape de transition vers une solution reprise par l'itération majeure suivante. L'étape de construction correspond plus particulièrement à la production d'une solution de plus en plus mature au

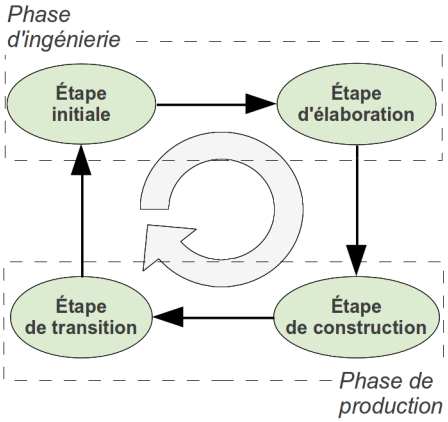

FIGURE 5.7 – Les différentes étapes du processus unifié [JBR99]

fur et à mesure de l'évolution d'une itération majeure et l'étape de transition à la production d'une nouvelle version.

La phase d'ingénierie bénéficie d'un haut niveau d'abstraction pour permettre l'élaboration tout en disposant d'un maximum d'agilité. Notamment, une modification locale doit avoir globalement le moins d'impacts possibles. Ce haut niveau d'abstraction est adapté à l'élicitation et à la réduction des choix par l'expérimentation. Le développement de prototypes, d'animations des modèles et de simulations sont facilités. Pendant la phase d'ingénierie, le méta-atelier peut éventuellement être adapté au besoin par ajout ou modification des outils dont il dispose.

Le besoin de précision est plus sensible pendant la phase de production. La précision est notamment apportée par le typage précis des éléments de méta-modèles. Ce plus bas niveau d'abstraction est orienté vers la vérification et la validation basée sur les types déclarés. Les transformations de modèles plus particulièrement utilisées pendant l'étape de transition s'appuient sur le typage précis des méta-modèles.

En reprenant les termes employés dans le cadre du processus unifié, les phases d'ingénierie et de production sont effectuées au sein du méta-atelier (voir figure 5.8). Nous considérons que ces deux phases sont elle-même itératives (itérations mineures présentées précédemment). L'étape de transition consiste plus particulièrement à produire l'atelier pour la plateforme d'exécution cible par génération de code.

La phase d'ingénierie permet d'élaborer les méta-modèles pour la spécification du langage. Elle se déroule à un haut niveau d'abstraction. La phase de production consiste à affiner la spécification du langage pour autoriser la traduction de l'atelier vers un système cible. Les opérations de vérification et de validation peuvent être effectuées au sein du méta-atelier ou sur le système cible. Dans la figure 5.6, une surface hachurée horizontalement correspond à la phase d'ingénierie et une surface quadrillée correspond à la phase de production.

5.4 Intégration entre un atelier et le système cible

Le système cible est un élément important pour la vérification et la validation de l'atelier. Dans la pratique, une grande partie de la validation est opérée par exécution du système cible (voir le chapitre 4). Cependant, la découverte des anomalies est tardive et il est plus difficile

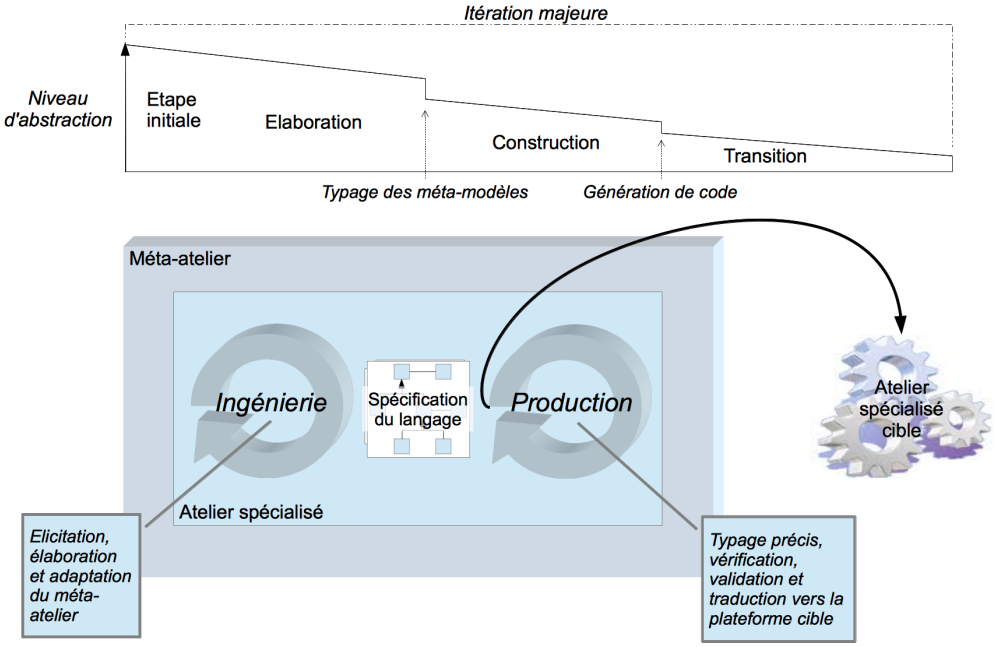

FIGURE 5.8 – Les étapes d'ingénierie et de production dans un méta-atelier

de localiser leurs causes. En effet, les différents aspects fonctionnels et non fonctionnels sont plus difficilement dissociables et manipulables au sein du système cible.

Ces aspects peuvent être explicitement séparés dans le méta-atelier. La validation y est plus intéressante en termes de localisation précoce des anomalies. De plus, des outils d'analyse peuvent être développés dans le méta-atelier. Idéalement, toutes les vérifications et validations devraient pouvoir être effectuées dans le méta-atelier. La transition produirait un système cible qui serait considéré comme validé à priori. Concrètement, on tente de maximiser la détection précoce des anomalies par du *model checking*, des tests dynamiques (simulations, animations, prototypage ou analyse de résultats obtenus par exécution du système,...). Le système cible obtenu après transition demeure un élément fondamental notamment pour les tests dynamiques.

Un atelier est conçu à partir de l'élaboration de méta-modèles. Les tests dynamiques consistent à produire et analyser les modèles conformes à ces méta-modèles :

1. l'effort nécessaire pour la production de ces modèles conformes peut-être très coûteux, même dans le méta-atelier car il peut être nécessaire d'écrire des compilateurs ou des interfaces de saisie particulières ;
2. l'exécution de l'atelier dans le méta-atelier peut être très longue et ne pas couvrir tous les cas possibles ;
3. dans certaines conditions, les exécutions dans le méta-atelier ne peuvent être que des simulations (c'est le cas, par exemple, lorsque la plateforme d'exécution cible se constitue d'un dispositif matériel particulier comme un réseau de capteurs ou un matériel embarqué,...)
4. la mise au point de simulateurs est très coûteuse, par exemple, pour les systèmes embarqués, une part très importante de la validation est dédiée à la mise au point des

simulateurs [MS10].

Nous considérons que c'est toujours le système cible qui constitue la référence et que c'est par ce dernier que des mises à l'épreuve réalistes peuvent être évaluées. Le méta-atelier est par contre toujours plus approprié pour exploiter les modèles conformes à des fins de vérification et de validation.

Comme le rappelle la figure 5.9, une itération comprend une part de création, du méta-atelier vers le système cible représentée notamment par l'étape de transition de la phase de production. Elle comprend aussi une part de rétro-influence, du système cible vers le méta-atelier et en définitive, vers le modèle mental du concepteur (voir chapitre 2.3.2).

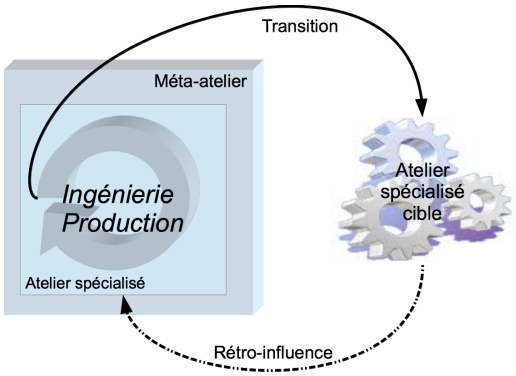

FIGURE 5.9 – Vue simplifiée d'une itération entre l'atelier dans le méta-atelier et le système cible

La rétro-influence est fondée sur l'observation et l'analyse. Pour exploiter efficacement les observations effectuées, au sein du méta-atelier ou de la plateforme d'exécution cible, un haut niveau d'expertise de l'observateur est primordial. Cependant, ce niveau élevé d'expertise n'est pas suffisant. L'observateur doit aussi bénéficier d'informations et de retours sur les exécutions pour améliorer la qualité de la maintenance et des prédictions possibles [JS02]. Les informations collectées concernent les aspects fonctionnels et les aspects non fonctionnels. Les aspects fonctionnels sont représentés par l'état du système. Les aspects non fonctionnels comme les consommations de ressources ou les temps d'exécution, sont calculées par le système en cours d'exécution ou par observation externe du système et de la plateforme d'exécution.

La suite de ce chapitre explique comment un atelier dans le méta-atelier et le système cible sont exploités en synergie pour la collecte des données et pour maximiser les possibilités de validation au sein du méta-atelier.

5.4.1 Représentation et collecte des informations

La représentation des informations peut être en partie automatisée en utilisant les techniques de l'orienté donnée très largement exploitées pour l'instrumentation du langage de niveau $M3$ d'un espace technique (voir chapitre 2.3.4.1). A partir de la spécification d'un méta-modèle, un langage permettant la représentation concrète des modèles conformes peut être automatiquement produit. Par exemple, si le MOF est utilisé pour méta-modéliser, les modèles conformes peuvent être représentés et échangés sous la forme de modèles XML. Les fonctionnalités de sérialisation et de matérialisation de modèles conformes vers et depuis XML sont automatiquement produite à partir du méta-modèle.

Les informations permettant de tracer et d'analyser les aspects fonctionnels se constituent typiquement de l'état des objets manipulés par l'atelier. Cet état interne est décrit dans les méta-modèles par la spécification des objets métiers. Cette spécification comprend les types d'entités spécifiques. Chaque type d'entité comprend un ensemble d'attributs. Par construction, imposée par les règles de transformation de modèles, l'état d'un objet est représenté par les valeurs associées aux attributs du type d'entité de l'objet. La collecte est automatiquement effectuée par l'invocation d'une opération de sérialisation dont le résultat est une représentation conforme au langage de représentation des modèles. L'opération inverse de matérialisation permet de reconstituer les objets à partir d'une représentation conforme au langage de représentation des modèles.

Pour les aspects non fonctionnels, les informations collectées peuvent aussi être spécifiées par les méta-modèles. Cette spécification doit comprendre non seulement le type des données mais aussi le calcul des valeurs. Des types particuliers d'entité peuvent être spécifiés pour représenter les aspects non-fonctionnels de façon à faciliter la collecte des informations. Cependant, le calcul des valeurs demeure spécifique. Il peut être spécifié dans les méta-modèles et traduit automatiquement dans le système cible ou encore implanté directement dans le système cible.

5.4.2 Synergie entre le méta-atelier et le système cible

L'atelier et le système cible peuvent être finement intégrés et interopérables par les données (voir les chapitres 2.3.4.2 et 2.3.4.3). L'interopérabilité par échange ou partage de données est particulièrement bien adaptée puisque le méta-modèle est partagé par, d'un coté, l'atelier au sein du méta-atelier et, de l'autre coté, par l'atelier réifié au sein de la plateforme d'exécution cible.

La capacité du système cible à produire des modèles conformes réalistes peut ainsi être exploitée pour alimenter le méta-atelier. Pour que les modèles produits par le système cible soient assimilables au niveau du méta-atelier, l'interopérabilité entre le système cible et le méta-atelier doit être maintenue. Pour ce faire, les composants d'échange de modèles sont intégrés au système cible par génération de code à partir du méta-modèle lors d'une étape de transition. Ainsi, un système cible de version N peut fournir des modèles conformes pour les évaluations internes effectuées pour la version $N + 1$. En fin d'itération majeure, les modèles conformes peuvent être conservés pour servir de base pour des tests de régression.

Pour la validation, un atelier au sein du méta-atelier et au sein de la plateforme d'exécution cible sont exploités en synergie. Cette synergie est permise par l'intégration par les méta-données. La suite de ce chapitre explique dans un premier temps les mécanismes utilisés pour l'intégration et dans un deuxième temps, décrit un exemple typique d'intégration.

5.4.3 Intégration par les méta-données

A tout moment, des modèles conformes représentant l'état de l'atelier ou issus de la collecte d'informations reflétant des caractéristiques non fonctionnels peuvent être échangés. Le protocole d'échange est standardisé entre l'atelier dans le méta-atelier et le système cible. Les méta-modèles représentent la partie variable. Les mécanismes d'encodage et de décodage permettant respectivement la sérialisation et la matérialisation sont fixes et décrit par un standard ou spécifiquement pour le domaine et l'application. Les composants qui mettent en œuvre l'échange (sérialisation et matérialisation) peuvent être génériques ou bien spécialisés et générés automatiquement (voir chapitre 2.3.5.4).

Il est du ressort du concepteur de décider des points de production et de récupération des méta-données dans le flot de contrôle d'un outil de vérification, d'un simulateur, d'un prototype.... Ce point de production peut se matérialiser par un dialogue prévu dans l'interface utilisateur (un bouton ou une option de menu, une ligne de commande particulière) ou être systématisé, lors de la réception d'un évènement ou tout simplement explicitement spécifié dans le code.

Le sens le plus souvent exploité des échanges de méta-données est du système cible vers l'atelier dans le méta-atelier. En effet, les méta-données sont exploitées pour les vérifications et validations qui sont effectuées de manière préférentielle au sein du méta-atelier. L'échange du méta-atelier vers le système cible peut être utilisé pour la configuration et l'initialisation du système cible.

5.5 Trois exemples d'utilisation d'un méta-atelier pour la vérification et la validation d'un système cible

Nous identifions trois possibilités typiques d'intégration entre l'atelier et le système cible. Ces trois possibilités se distinguent suivant les spécificités du système cible. Le premier point de vue concerne la validation sans contrainte particulière concernant le matériel constituant la plateforme d'exécution cible. Les deux points de vue suivants concernent des systèmes cible particuliers, constitués de dispositifs matériels qui contraignent plus ou moins fortement la validation et les possibilités d'intégration entre l'atelier et le système cible.

5.5.1 Vérification et validation en séquence

La validation en séquence représente un cas typique d'utilisation de la méthode Platypus. Dans cet exemple, on cherche à mettre au point un système cible [2]. Ce processus de validation en séquence est présenté par la figure 5.10. Ce cycle comprend potentiellement deux points d'échange de méta-données entre le méta-atelier et le système cible (points 4 et 8 sur la figure 5.10).

En début d'itération majeure, les méta-modèles sont supposés immatures. Le cycle s'effectue alors uniquement dans le méta-atelier (Partie bleue, points 1, 2 et 3). Quand les méta-modèles sont jugés suffisamment matures, ils peuvent être typés statiquement. Après un ou plusieurs cycles de validation exploitant le typage statique, le système cible peut être produit. L'atelier peut alors être exécuté à la fois dans le méta-atelier et dans la plateforme cible.

En fin d'itération majeure, une itération mineure débute dans le méta-atelier (1) par l'édition ou la réutilisation de modèles conformes produits pendant les itérations précédentes. Les vérifications et les validations sont pratiquées (2) sur la base de ces modèles conformes. Si la validation fait apparaître une anomalie, une imprécision ou un manque (3), des corrections sont éventuellement apportées au méta-modèle dans le méta-atelier. Si les résultats des validations sont acceptables (4), alors l'atelier dans la plateforme cible est éventuellement mis à jour. Les modèles conformes peuvent être sérialisés pour être réutilisés dans la plateforme cible (4), sinon, des modèles conformes sont créés ou réutilisés et les validations sont menées dans la plateforme d'exécution cible (5). Si une anomalie est constatée (6) et localisée, alors le méta-modèle peut éventuellement être corrigé (9) (l'anomalie peut aussi provenir d'une

2. Mettre au point un atelier dans le méta-atelier par rapport à un système cible est aussi possible par exemple pour l'élaboration d'une simulation

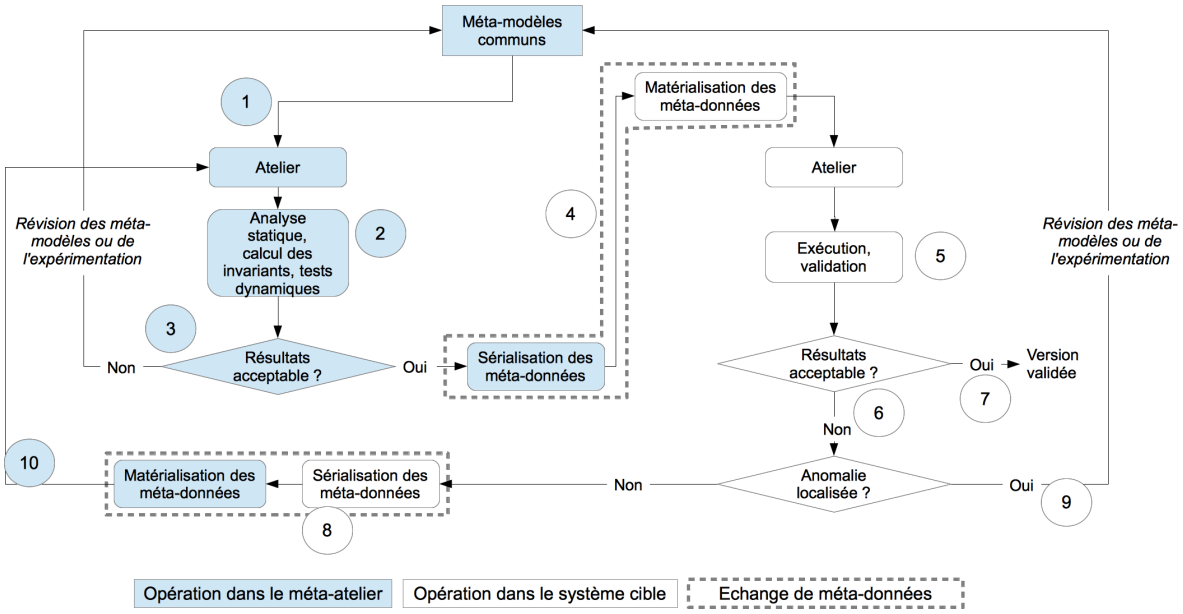

FIGURE 5.10 – Validation en séquence dans le méta-atelier puis dans le système cible

partie de la mise en œuvre non générée de l'atelier). Si une anomalie est constatée (6) mais non localisée, alors les méta-données sont sérialisées et matérialisées (8) dans le méta-atelier. Le modèle conforme produit par la matérialisation des méta-données (10) est exploité pour les opérations de validation (2).

5.5.2 Vérification et validation d'une simulation

La simulation est utilisée pour prévoir et étudier le comportement du système sans avoir à l'exécuter. Le méta-atelier peut être utilisé pour implanter un simulateur. Le simulateur est un prototype qui vise à imiter le comportement d'un système qui préexiste ou qui est développé en parallèle. Nous faisons l'hypothèse que le système simulé est un logiciel implantés dans un matériel (un automate dans un système embarqué ou un robot par exemple).

L'objectif ultime est donc de permettre de valider un programme avant sont installation et son exécution dans le système cible. La simulation est un élément fondamental de la vérification et de la validation dans les sciences expérimentales et le processus d'utilisation de Platypus montré par la figure 5.11 s'inspire du processus de vérification et de validation présenté dans le chapitre 3.3.

Pour sa mise au point, le simulateur doit être étalonné. L'étalonnage est un processus qui vise à s'assurer que le simulateur est conforme par rapport au besoin et au système que l'on cherche à simuler. Pour l'étalonnage, le simulateur et le système sont exécutés conjointement sur la base des mêmes hypothèses (données en entrée). Les résultats obtenus sont comparés et le simulateur dans le méta-atelier est éventuellement corrigé ou adapté si les résultats obtenus dans le simulateur ne sont pas conformes aux résultats du système.

La notion de validation précoce avant transition vers le système cible est alors discutable puisque le système cible est systématiquement exploité en parallèle du simulateur. L'agilité réside essentiellement dans la capacité à récupérer efficacement les informations issues du

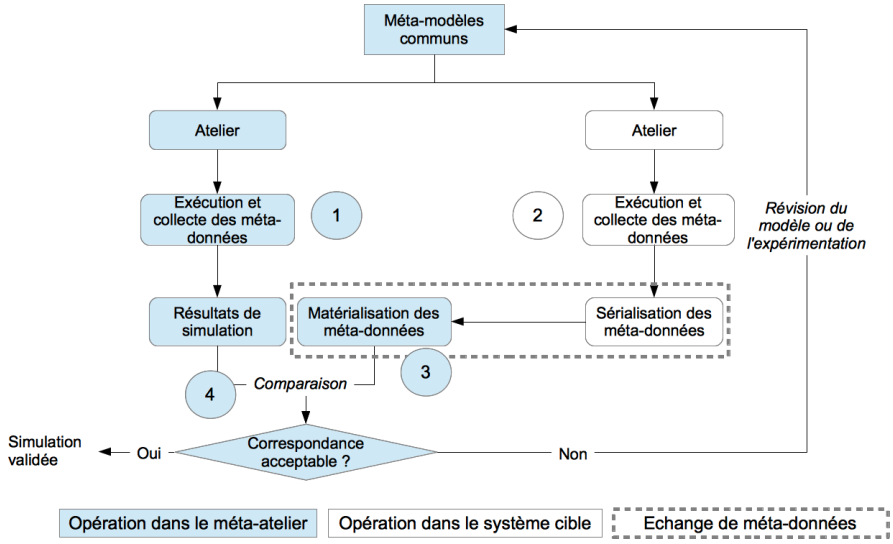

FIGURE 5.11 – Validation d'une simulation

système cible et à outiller l'atelier (comprenant le simulateur) pour détecter les anomalies, et adapter ou corriger le simulateur. Par exemple, dans [BHH09], les auteurs décrivent un atelier agile pour la validation de programmes pour des robots *NXT lego*. Cet atelier permet de spécifier des programmes et de les mettre au point par simulation avant leur installation. La machine virtuelle installée dans le robot est maitrisée intégralement (une machine virtuelle a été mise en œuvre) de sorte que la mise au point des programmes au sein du simulateur et au sein du robot peuvent être menées en utilisant les mêmes outils de déverminage.

Le simulateur et le système cible sont développés sur la base des mêmes méta-modèles de sorte notamment qu'ils soient interopérables par les données. La collecte et la sérialisation des données doit être possible sur le système cible. La collecte est prévue dans les programmes installés. Pour la validation, les exécutions sont menées d'une part dans le méta-atelier et d'autre part, dans la plateforme cible, (1) et (2) dans la figure 5.11. En fin d'exécution, les données collectées sont récupérées et matérialisées dans l'atelier (3). Les données correspondantes obtenues par simulation dans l'atelier sont comparées aux données produites par le système cible (4).

5.5.3 Vérification et validation par monitoring

Dans le cas de matériel très spécifiques, par exemple des unités matérielles miniaturisées ou des systèmes embarqués, il est possible que le système cible ne puisse pas intégrer des composants logiciels pour la matérialisation et la sérialisation des données. La mémoire peut être insuffisante, les capacités en termes de puissance de calcul peuvent être limitées ainsi que l'énergie disponible.

L'intégration peut alors s'effectuer par communication directe avec le système cible est récupération de l'état de ses composants. Le schéma d'intégration présenté par la figure 5.12 montre que l'intégration entre l'atelier et le système cible est opérée depuis l'atelier par observation du système cible. Les composants de l'atelier qui mettent en œuvre le protocole de communication des données peut être produit à partir de la description des données

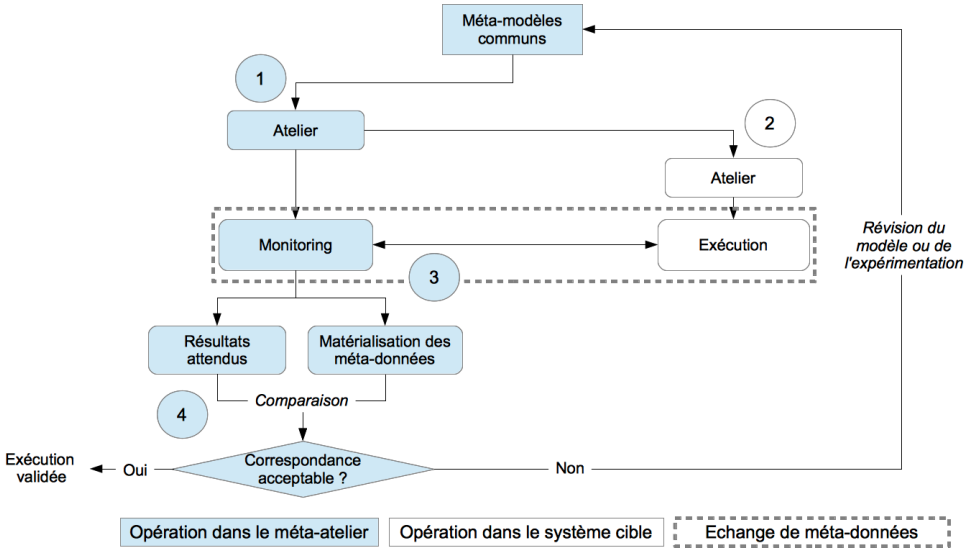

FIGURE 5.12 – Validation par monitoring

échangées. Concrètement, la mise en œuvre est souvent très spécifique puisqu'il dépend de la nature et même des marques de matériels utilisés. Le système cible est démarré depuis l'atelier, (1) et (2) dans la figure 5.12. La récupération des données en provenance du système cible est orchestrée depuis l'atelier pendant l'exécution du système cible. Ces données sont ensuite matérialisées, et validées par rapport à des résultats attendus.

5.6 Conclusion

Dans ce chapitre, nous avons tout d'abord montré qu'il est possible d'exploiter une méthode agile pour la méta-modélisation. Nous avons ensuite décrit suivant deux points de vue les caractéristiques des itérations.

Le premier point de vue décrit une itération comme un processus double qui permet à la fois d'apprendre (dimension cognitive) et de produire progressivement une solution adéquate (dimension de la représentation). L'approche agile et le méta-atelier qui la supporte doivent donc permettre de couvrir au maximum les besoins pour ces deux dimensions d'une itération. Le prototypage et l'animation de modèles doivent être possibles au niveau du méta-atelier sans nécessiter de génération de code. De plus, le méta-atelier doit pouvoir être adapté pendant la phase d'apprentissage. De nouveaux outils spécifiques pour le domaine et le problème doivent pouvoir être construits pour spécialiser le méta-atelier.

Suivant le second point de vue, le méta-atelier doit permettre l'élaboration des méta-modèles à haut niveau d'abstraction. Les méta-modèles sont progressivement vérifiés et validés par rapport au besoin. Quand un certain niveau de confiance vis-à-vis de la solution produite au niveau du méta-atelier est atteint, tout ou partie du système cible peut être automatiquement construit.

Nous considérons que le système cible est l'artéfact final de référence du point de vue des vérifications et des validations. Nous avons montré comment Platypus autorise l'exploitation des résultats de l'utilisation du système cible pour la validation des aspects fonctionnels et

dans une certaine mesure des aspects non fonctionnels. L'intégration par échange de modèles conformes permet cette exploitation des résultats par leur instrumentation au sein du méta-atelier.

Pratiquement, l'application d'une méthode agile pour la méta-modélisation implique qu'il doit être possible, au niveau du méta-atelier, de vérifier, de valider, et d'animer les méta-modèles par des spécifications exécutables.

Le haut niveau d'abstraction des spécifications est utile pour, notamment, favoriser l'élicitation. Un plus bas niveau d'abstraction, permettant la définition précise des types et des contraintes des éléments de méta-modèle est aussi nécessaire. Le méta-atelier doit autoriser la méta-modélisation pour ces deux niveaux d'abstraction.

Ainsi, nous privilégions l'utilisation d'un méta-atelier mettant en œuvre un langage dynamique (haut niveau d'abstraction), autorisant un grain fin pour les évolutions, permettant de bénéficier du maintient dynamique du lien de causalité au sein du méta-atelier et autorisant le typage optionnel.

Chapitre 6

Validation de l'approche par l'outillage

La méta-modélisation est une activité de programmation à haut niveau d'abstraction. Le méta-atelier doit être un environnement de programmation disposant d'un interpréteur ou d'un compilateur pour permettre l'exécution des spécifications et des outils de validation afférents. Nous avons vu dans le chapitre précédent que le méta-atelier doit pouvoir être utilisé à haut niveau d'abstraction pour permettre d'outiller l'élaboration et de faciliter l'élicitation. Nous avons aussi vu que le méta-atelier doit parallèlement permettre d'assurer un maximum de vérifications, de mettre en œuvre des validations et des transformations de modèles. Les spécifications doivent donc pouvoir être précisément typées.

Dans ce chapitre, nous présentons notre méta-atelier Platypus qui permet de satisfaire ces besoins. Nous présentons Platypus comme un environnement STEP [ISO94a] intégré à un

système Smalltalk [GR83]. Smalltalk permet de supporter l'agilité pour la mise en œuvre des spécifications exécutables. La suite de ce chapitre décrit tout d'abord globalement Platypus. Nous présentons ensuite les éléments techniques de Smalltalk et de la norme STEP. Enfin nous présentons un exemple simple d'utilisation de Platypus.

6.1 Présentation générale du méta-atelier Platypus

Platypus a été tout d'abord conçu pour permettre la méta-modélisation métier par les données [PR06a]. Nous disposions d'un bon niveau d'expertise concernant STEP. Nos expérimentations antérieures (travaux de thèse) ont permis de montrer en particulier l'adéquation d'EXPRESS pour la modélisation métier, la méta-modélisation et la spécification des générateurs de code. Pousser l'outillage de ce standard a été notre première motivation. Smalltalk a été choisi pour ses capacités en termes de développement et d'instrumentation rapide.

Lors d'une première expérimentation, nous avons rapidement compris que la validation précoce est un point clé pour faciliter le développement d'outils. En particulier, pour éviter l'inconvénient majeur de la lourdeur du processus de génération de code, les méta-modèles doivent pouvoir être validés par interprétation, directement en cours d'édition. Platypus c'est alors enrichi d'un interpréteur de contraintes EXPRESS ainsi que d'un éditeur générique de modèles permettant la validation précoce des méta-modèles par des exemplaires de modèles conformes [PR05].

Dans ce chapitre, nous décrivons l'architecture logicielle de notre méta-atelier, puis nous expliquons brièvement son processus d'utilisation.

6.1.1 Architecture logicielle

Nous avons vu que tous les méta-ateliers sont construits sur la base d'une même architecture générique (voir chapitre 2.3.5). Cette architecture s'articule autour d'un méta-méta-modèle et de mécanismes. Dans une telle architecture un ensemble de méta-modèles conformes au méta-méta-modèle, et, pour chaque méta-modèle, un ensemble de modèles conformes sont maintenus en cohérence. Le maintient en cohérence peut être dynamique si tous ces modèles coexistent au sein d'un même environnement réflexif mettant en œuvre un MOP. Pour outiller Platypus, nous utilisons un système Smalltalk [GR83] enrichi d'un composant permettant le typage statique des méta-modèles :

- le système Smalltalk choisi est Pharo [Pha], une mise en œuvre libre dont la communauté est très active ;
- le typage des méta-modèles est optionnel et pour sa spécification, nous avons implanté un composant dédié, basé sur une mise en œuvre des outils du standard ISO STEP [ISO94a], plus particulièrement, le langage de modélisation de données EXPRESS [ISO94b] pour la spécification intentionnelle du domaine, et le langage de représentation des modèles de STEP [ISO94c] pour la représentation directe des modèles conformes.

Les mécanismes sont mis en œuvre par le MOP de Smalltalk. L'ensemble des outils transversaux se constitue au minimum des outils standards et génériques d'un système Smalltalk classique (Browser pour visualiser et éditer les méta-modèles, le dévermineur et les inspecteurs standards pour contrôler, visualiser et éditer dynamiquement les modèles conformes).

Cependant, Platypus ne peut pas être décrit par l'extension en énumérant tous ces composants. Son noyau peut être complété suivant les besoins, la nature du langage à mettre

en œuvre et celle du domaine. En effet, de manière non exhaustive, cet ensemble peut être enrichi par des composants additionnels comme, par exemple, un compilateur de compilateurs si une syntaxe concrète textuelle particulière est nécessaire, une couche logicielle pour le développement web ou encore une interface vers une base de données objet ou relationnelle. Les outils transversaux comprennent aussi des composants pour la construction d'éditeurs ou d'interfaces graphiques, des animations, des visualisations particulières et des prototypes.

Dans Platypus, le niveau $M3$ est fixe. Un méta-modèle particulier se compose de classes Smalltalk standards. Cette homogénéité avec le système standard permet bénéficier du maintient de causalité et de garantir la réutilisation des outils transversaux et enfin permet d'assurer un niveau acceptable de compatibilité avec d'autres systèmes Smalltalk.

6.1.2 Processus d'utilisation

Nous avons vu qu'en début d'itération majeure, l'activité de méta-modélisation se traduit par des étapes d'acquisition des connaissances et d'adaptation de l'outillage pour répondre aux besoins de représentation des modèles découverts pendant l'élaboration (voir chapitre 5.3).

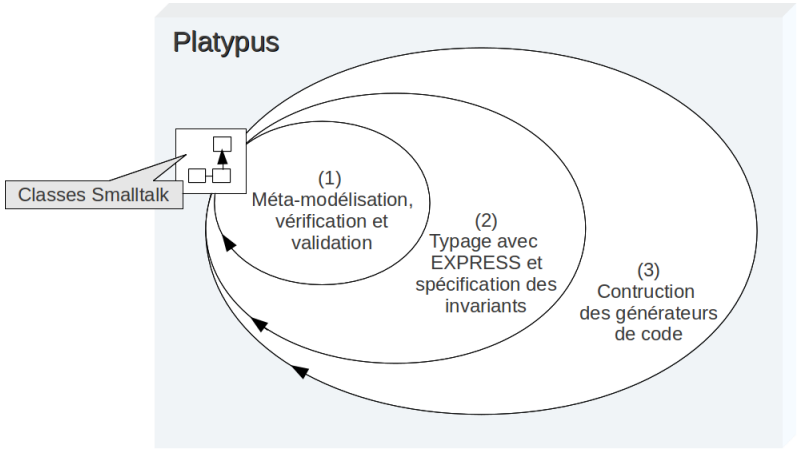

FIGURE 6.1 – Les cycles complémentaires d'utilisation de Platypus

La figure 6.1 montre les trois cycles complémentaires liés à l'utilisation de Platypus :

1. le premier cycle consiste en une méta-modélisation à haut niveau d'abstraction ; les éléments des méta-modèles sont des classes Smalltalk ; des prototypes peuvent être construits pour valider le méta-modèle suivant l'expression du besoin ;
2. le type des propriétés, les cardinalités des associations et les invariants peuvent être explicités à l'aide d'un modèle de données EXPRESS ;
3. Les outils de vérification et de validation s'appuyant sur les types déclarés en EXPRESS ainsi que les générateurs de code peuvent ensuite être mis en œuvre.

Platypus n'impose pas de point d'entrée particulier. Selon le cas, un atelier peut être construit en commençant par la méta-modélisation en Smalltalk ou bien par la spécification des types statiques à l'aide d'un schéma EXPRESS. En effet, si le domaine est bien connu ou si un existant doit être prise en compte et qu'une phase exploratoire du domaine n'est pas nécessaire, il est possible de spécifier directement les entités du domaine en EXPRESS.

Le choix de Smalltalk nous rapproche des solutions telles que Dome [EK00] ou Moose[NDG05, DG06], le typage optionnel positionne plus particulièrement Platypus comme proche de Moose.

6.2 Smalltalk

La première version du langage, développée au Xerox Parc, date de 1972. Toutes les versions actuelles sont très directement des mises en œuvre de la révision de 1982 (Smalltalk-80 [GR83]). Un noyau standard constitué des éléments les plus utilisés est normalisé par l'ANSI de sorte que les différentes variantes commerciales ou non sont raisonnablement compatibles entre elles [1].

Smalltalk est un langage orienté objet très largement influencé par Lisp et Simula. Les aspects influencés par Simula concernent l'approche objet avec l'abstraction de données, la possibilité de surcharge des opérations et le polymorphisme. L'influence de Lisp se localise au niveau mise en œuvre du langage basé sur le typage dynamique, les concepts d'identité et celui de référence qui constitue l'unique moyen d'accéder à un objet pour l'invocation d'un service (envoi de message). L'influence de Lisp concerne aussi la gestion transparente et dynamique de la mémoire avec notamment plusieurs niveaux de ramasse-miette.

Comme le montre le tableau de la figure 6.2, la syntaxe, basée sur quelques mots clés, est très simple. Il n'y a pas d'instruction ni de structure de contrôle, tout est basé sur l'envoi de message et les fermetures littérales qui sont aussi des objets, manipulés par référence. Historiquement, c'est le premier langage basé sur le "tout-objet". Tout ce qui est manipulable l'est sous la forme d'un objet. Les classes, les méthodes, la pile d'exécution, l'environnement de programmation, le compilateur ... sont des entités de première classe. Ainsi, il est possible d'accéder à leur représentation, de les référencer et éventuellement de les modifier de manière très régulière et uniforme dans le système.

Smalltalk est un langage dynamique [TW07] : le type d'un objet peut être connu et manipulé pendant l'exécution. Le type d'un objet est un objet représenté par la classe qui l'a créé. La réflexivité est en partie basée sur le concept de méta-classe. Une méta-classe est une classe gérée par le système comme le type d'une classe. Chaque classe est décrite par sa propre méta-classe. Ainsi, un système Smalltalk est basé sur deux hiérarchies de classes : la hiérarchie des classes "métier" et celle des méta-classes.

6.2.1 Les mécanismes dans Smalltalk

Un système Smalltalk met en œuvre un MOP [Tan09, KR91] qu'il est possible de manipuler et d'adapter. Smalltalk est un système réflexif [Tan09] (voir le chapitre 2.1.5.2) qui permet uniformément la méta-programmation par l'introspection et l'intercession. Une partie importante des mécanismes et des capacités réflexives de Smalltalk est inhérente à son modèle objet et au concept de méta-classe.

Dans [DPZ01], Mohamed Dahchour et al décrivent le modèle objet et concept de méta-classe dans Smalltalk de la façon suivante. Une application se constitue d'une collection d'objets et de classes. Les objets représentent des entités du "monde réel" (objets "métiers") et une classe décrit un ensemble d'objets similaires (classe "métier"). Une classe décrit des

1. Par exemple, Platypus est développé en utilisant Pharo [Pha] mais une version pour VisualWorks traduite automatiquement à partir de la version sous Pharo est aussi disponible

"Commentaire"	Commentaires entre deux double-quote.
.	Le point sépare les instructions.
;	Sépare deux messages pour des envois de message en cascade au même receveur.
:=	Affectation.
^ a	Pour sortir d'une méthode en retournant a.
true false	Mots clés booléens.
nil	Mot clé pour une valeur indéfinie.
self	Mot clé pour désigner le receveur courant.
super	Mot clé pour accéder à la méthode surchargée depuis le corps d'une méthode.
thisContext	Mot clé pour accéder à l'objet qui décrit le contexte d'exécution courant
'Une chaine de caractères'	Représentation des chaines de caractères.
$a	Représentation du caractère 'a'.
#Symb	Le symbole 'Symb', chaine de caractères dont il n'existe qu'une seule instance.
#(a b)	Tableau statique contenant a et b.
{ 1+2. 4 }	Tableau dynamique contenant 3 et 4.
| tmpVar1 tmpVar2 |	Déclaration de deux variables locales.
[des instructions]	Un bloc d'instructions (Fermeture littérale).
[:x :y | 2* y * x]	Bloc avec deux arguments.

FIGURE 6.2 – Les éléments lexicaux et syntaxiques de Smalltalk extrait de [DGKR08]

propriétés structurelles et comportementales. Dans un objet, l'ensemble des valeurs d'attributs représentent l'état de l'objet. Cet état peut être accédé et modifié par envoi de messages. Dans un tel système deux niveaux d'abstraction sont donc présents : le niveau des classes et celui des objets (instances des classes). Cette organisation permet la manipulation des objets mais pas des classes en dehors de leurs instances. Par exemple, il peut être nécessaire de requérir le nom d'une classe, la liste de ses attributs et méthodes, d'ajouter ou de modifier dynamiquement un comportement... Pour autoriser ces opérations, Smalltalk permet de manipuler les classes elle-même comme des objets dont la classe est une méta-classe. Une méta-classe exprime donc la structure et le comportement d'une classe de telle sorte qu'un message peut être envoyé à une classe comme c'est le cas au niveau des instances.

Dans Smalltalk, toutes les classes "métier" héritent directement ou indirectement de la classe *Object*, toutes les méta-classes héritent directement ou indirectement de la classe *MetaClass*. *MetaClass* est auto-descriptive : la classe de *MetaClass* est décrite par *MetaClass*. Dans un système en cours d'exécution, tout objet est donc typé par un autre objet au sein du même système. L'objet qui représente le type accessible et manipulable de manière uniforme par envoi de message à l'objet typé.

6.2.2 Maintient dynamique du lien de causalité

Un système Smalltalk ne consiste pas seulement en une bibliothèque de classes et de méta-classes. Il s'agit d'un système interactif en cours d'exécution : schématiquement, quatre processus sont continuellement actifs pour, (1) la gestion des entrées-sorties, (2) celle des interfaces utilisateurs, (3) celle de la mémoire et (4) la gestion du des processus et des interruptions. Considérée très simplement, la mémoire est une table d'objets. Tous les objets du système sont maintenus dans cette table d'objets. Ainsi, les classes, les méta-classes et leurs instances sont maintenues actives et dynamiquement manipulables en permanence. Le système est persistant : l'ensemble des objets du système peuvent être à tout moment sauvegardé

dans un état cohérent dans un fichier image.

La modification d'une classe entraine automatiquement l'adaptation de toutes les instances de la classe. L'ajout, la suppression ou la modification d'une méthode sont dynamiquement pris en compte par le système. Ainsi, il est possible de modifier, supprimer ou même ajouter une méthode manquante depuis un débogueur ou par méta-programmation sans remise en cause globale. Les animations et les prototypes peuvent être corrigés à la volée. Les tests dynamiques peuvent être exécutés en continu pour la détection précoce des anomalies et des régressions.

6.2.3 Créativité et élicitation

Favoriser la créativité et servir de médium de communication entre individus pour exprimer explicitement des idées sont les deux objectifs majeurs qui ont motivés les concepteurs du langage [Ing81]. Les spécificités techniques, tout particulièrement, la simplicité et l'homogénéité du langage, l'interactivité et la forte capacité en termes de réutilisation et d'adaptation de l'environnement servent ces objectifs. Par rapport aux caractéristiques désirables du méta-atelier, la créativité est possible du fait de l'agilité et de l'interactivité du système. Les auteurs mettent particulièrement en avant le haut niveau d'abstraction et les capacités du système en terme d'évolutivité et d'adaptabilité des spécifications et du système.

Smalltalk permet de couvrir les besoins en termes de représentation de la connaissance qui peut être de nature informatisée, validée dynamiquement et de nature réutilisable validée formellement. La validation dynamique s'effectue via le prototypage (voir la figure 5.4 du chapitre 5.3.1). La validation formelle peut s'appuyer sur des types ajoutés sous la forme d'objets ou d'annotations (appelées *Pragma* dans Pharo). Les annotations sont accessibles par envoi de message. Le système peut réagir dynamiquement lors de l'ajout, de la modification ou de la suppression des annotations. Des spécifications en Smalltalk peuvent ainsi naturellement supporter un typage optionnel et la vérifications automatisée des types.

6.3 Les outils de la norme STEP

La norme *STEP* (STandard for the Exchange of Product model data), référencée comme le standard *ISO 10303* [ISO94a], a pour objectif de produire des modèles de données standards par métier, appelés *protocole d'application*. Les grandes sociétés industrielles comme *General Motors* ou *Dassault* sont à l'origine des travaux de normalisation pour résoudre leurs problèmes d'échange de données (voir par exemple [GAL] concernant l'utilisation de *STEP* dans le secteur automobile). Le problème de l'échange de données adressant des questions techniques et méthodiques, la norme s'est dotée d'une part, d'un socle technologique de description et de mise en œuvre des données et d'autre part, d'un procédé d'élaboration de modèles métiers, fondé sur l'usage de ces données dans les applications.

La norme STEP définit non seulement une méthode de description essentiellement basée sur le langage de modélisation de données EXPRESS [ISO94b] mais aussi des méthodes de mise en œuvre pour l'exploitation automatisée des modèles EXPRESS. Les objectifs principaux de ces méthodes de mise en œuvre sont de décrire les protocoles d'échange et de partage des instances des schémas entre systèmes hétérogènes. L'échange de données s'effectue par fichiers d'échanges standards [ISO94c] et le partage de l'information par une interface logicielle standard dite *Interface Standard d'Accès aux Données* ou *SDAI* [ISO94d]. Les évolutions plus récentes de cette technologie tendent vers un rapprochement avec les outils de l'*OMG*,

avec la standardisation de passerelles conceptuelles et techniques entre *EXPRESS-STEP* et *UML-XMI*.

6.3.1 Le langage *EXPRESS*

Le langage EXPRESS permet la spécification de modèles de données. Du point de vue de la spécification des entités d'un modèle, EXPRESS est très proche de KM3 [JB06] ou encore du langage de modélisation de Kermeta [MFJ05]. Son originalité est de permettre la spécification des contraintes locales aux types ou aux entités mais aussi des contraintes globales.

```
1  ENTITY cartesian_point SUBTYPE OF ( point );
2    coordinates  : LIST [ 1  : 3 ] OF REAL;
3  DERIVE
4    dim  : dimension_count  := dimension_of ( SELF );
5  END_ENTITY;
6
7  ENTITY direction SUBTYPE OF ( geometric_representation_item );
8   direction_ratios  : LIST [ 2  : 3 ] OF REAL;
9  WHERE
10  wr1  : (SIZEOF(QUERY( tmp <* direction_ratios | (tmp <> 0))) > 0);
11 END_ENTITY;
12
13 ENTITY axis2_placement_3d SUBTYPE OF (geometric_representation_item);
14  location  : cartesian_point;
15  axis  : OPTIONAL direction;
16  ref_direction  : OPTIONAL direction;
17 DERIVE
18  p  : LIST [3 :3] OF direction  := build_axes(axis,ref_direction);
19 WHERE
20  wr1  : (SELF\placement.location.dim = 3);
21  wr2  : ((NOT EXISTS (axis)) OR (axis.dim = 3));
22  wr3  : ((NOT EXISTS (ref_direction)) OR (ref_direction.dim = 3));
23  wr4  : ((NOT EXISTS (axis)) OR (NOT EXISTS (ref_direction)) OR
24              (cross_product (axis,ref_direction).magnitude > 0));
25 END_ENTITY;
```

FIGURE 6.3 – Entités EXPRESS extraites du protocole d'application *config control design*

Les *protocoles d'application* de la norme STEP sont spécifiés avec *EXPRESS*. Ce langage est à la fois textuel et graphique. Pour Platypus, nous nous intéressons plus particulièrement à la représentation textuelle des modèles. La syntaxe textuelle repose sur une grammaire de type LL(1). Bien qu'il intègre certaines caractéristiques des langages à objets typés comme *C++* ou *Java* et de langages de définition et de manipulation de données comme *SQL*, il est spécifié indépendamment d'un système ou d'un langage cible particulier :

- un schéma se compose de types d'entités et de contraintes globales ;
- une entité décrit un ensemble d'objets qui partagent des propriétés et des contraintes communes ; ainsi, la spécification d'une entité comprend des attributs explicite, dérivés, inverses et la définition de contraintes locales ou de règles d'unicité ;

  - un attribut explicite est une propriété dont la valeur est stockée (échangée au sens *STEP*) ; sa valeur est obligatoire par défaut ;
  - un attribut dérivé comprend une expression qui permet de calculer sa valeur ;
  - un attribut inverse permet d'expliciter les cardinalités et de naviguer dans le sens inverse de l'association ;
  - une contrainte permet de spécifier une règle d'intégrité locale à l'entité ;
  - une règle d'unicité permet de spécifier une liste d'attributs dont la combinaison des valeurs doit être unique pour toutes les instances de l'entité ;
- une contrainte globale permet de contraindre simultanément toutes les instances d'une ou plusieurs entités.

EXPRESS autorise l'héritage multiple. Les ambiguïtés concernant les accès à la valeur d'un attribut sont réglées par l'expression explicite du chemin d'accès dans l'arbre d'héritage.

La figure 6.3 montre un exemple de déclaration en EXPRESS. Cet exemple comprend trois entités extraites du méta-modèle standard AP203 utilisé notamment pour la modélisation d'objets en 3D. Ces deux entités spécifient le concept de placement d'un axe en 3D utile à la définition d'une surface élémentaire (comme un plan, un cylindre, un cône ou une sphère). Une direction est un point en deux ou trois dimensions, qui est l'extrémité du vecteur de direction (l'origine du vecteur étant l'origine de l'espace). Pour être valide, une instance de *direction* doit respecter la contrainte d'intégrité suivante : la requête calculant si une des coordonnées de la direction est différente de 0 doit avoir au moins une ligne résultat, autrement dit, l'extrémité de la direction ne peut pas être l'origine. Un placement est constitué d'un point (*location*), éventuellement complété d'un axe (*axis*) et d'une direction (*ref_direction*). Les trois directions de l'axe sont données par l'attribut *p* dont la valeur est calculée par la fonction *build_axes*. Pour être valide, une instance doit respecter quatre contraintes d'intégrité (à titre d'exemple la première contrainte, nommée *wr1*, indique que le point de référence doit être en trois dimensions).

6.3.2 Représentation des instances

```
#11=DIRECTION('dir1',(1.0,0.0,0.0));
#10=DIRECTION('dir10',(0.0,0.0,1.0));
#13=CARTESIAN_POINT('cp13',(0.75,0.5,−0.25));
#14=AXIS2_PLACEMENT_3D('ap5',#13,#10,#11);
```

FIGURE 6.4 – Extrait d'instances en correspondance interne

Dans le contexte d'un lot d'instances, chaque instance possède un identifiant numérique entier unique, le nom de l'entité correspondante et une liste de valeurs d'attributs. Un lot d'instances est stocké dans un fichier et un fichier ne peut contenir qu'un seul lot d'instances. Seules les valeurs des attributs explicites sont échangées. Les valeurs des attributs dérivés, les contraintes et les algorithmes décrits dans le schéma EXPRESS ne sont pas contenus dans un fichier d'échange STEP. Par contre, le lot d'instances encodé dans un fichier représente le contexte utilisé pour le calcul des contraintes et des attributs dérivés spécifiés dans le schéma EXPRESS correspondant au lot d'instances.

```
#11=(
  GEOMETRIC_REPRESENTATION_ITEM('dir1')
  DIRECTION((1.0,0.0,0.0)) ;
#10=(
  GEOMETRIC_REPRESENTATION_ITEM('dir10')
  DIRECTION((0.0,0.0,1.0)) ;
#13=(
  POINT('cp13')
  CARTESIAN_POINT((0.75,0.5,−0.25)) ;
#14=(
  GEOMETRIC_REPRESENTATION_ITEM('ap5')
  AXIS2_PLACEMENT_3D(#13,#10,#11)) ;
```

FIGURE 6.5 – Extrait d'instances en correspondance externe

Par rapport aux termes employés dans l'IDM, un schéma EXPRESS permet la spécification d'un méta-modèle et un lot d'instances valide pour un schéma représente un modèle conforme à ce méta-modèle.

La liste de valeurs d'attributs est construite dans l'ordre défini par celui des attributs de l'entité EXPRESS correspondante. Le type des valeurs est implicitement exprimé par le format textuel de la valeur. Dans le cas d'une valeur dont l'attribut correspondant est une union de types (en EXPRESS un type *select*), il est possible de qualifier explicitement le type de la valeur. Ceci permet d'assurer dans tous les cas un typage précis.

La syntaxe du langage d'instanciation prévoit deux algorithmes pour la création de la représentation d'une instance, la *correspondance interne* et la *correspondance externe* :

- la *correspondance interne* est utilisée s'il n'y a qu'une seule possibilité pour l'ordre des valeurs d'attributs ; dans ce cas, la liste des valeurs de l'entité sous-type est concaténée à celle de ses super-types ; l'ordre est donné par la relation d'héritage [ISO94c] ; La figure 6.4 présente l'encodage *STEP* en correspondance interne d'une instance valide de *axis2_placement_3d* ainsi que les instances avec lesquelles elle est en relation ;
- la *correspondance externe* est utilisée s'il existe plusieurs possibilités pour l'ordre d'insertion des valeurs d'attributs ; La figure 6.5 présente l'encodage *STEP* en correspondance externe pour les mêmes instances de la figure 6.4.

La correspondance interne est la plus naturellement adoptée pour les instances dont le type entité est simplement une racine dans une hiérarchie ou dont le type entité n'hérite que d'un seul autre type entité. En EXPRESS, il est possible de spécifier des types entités *complexes* permettant d'exprimer des compositions d'instances. Un type entité peut être déclaré super type d'autres types entités et composer ses sous-types en utilisant les opérateurs *AND*, *ANDOR* et *ONEOF*. Si A, B et C sont des types entités et que B et C héritent de A, alors, les compositions possibles sont :

- *A **SUPERTYPE OF** (B **AND** C)* spécifie 2 cas valides : instance de A ou de A&B&C ;
- *A **SUPERTYPE OF** (B **ANDOR** C)* spécifie 4 cas valides : instance de A, de A&B, de A&C ou de A&B&C ;
- *A **SUPERTYPE OF ONEOF**(B ,C)* spécifie 3 cas valides : instance de A, de A&B ou de A&C ;

La composition par défaut est *ANDOR*. L'algorithme de calcul des compositions valides est donné dans la norme [ISO94b]. Comme nous l'expliquons dans le chapitre 6.4.6.3, la spécification des opérations de transformation de modèle peuvent bénéficier de la composition d'instance notamment pour découpler le domaine source du domaine cible et ainsi faciliter la réutilisation des transformations.

6.3.3 Intégration par échange des données

Une des méthodes les plus couramment employées pour interagir entre systèmes est l'échange d'informations par fichier. Cette méthode est en effet souvent très satisfaisante car les données sont persistantes et peuvent être manipulées par des systèmes et langages hétérogènes. Le langage EXPRESS est utilisé pour décrire les informations transportées. Chaque fichier d'échange constitue une instanciation d'un schéma EXPRESS.

Deux systèmes communicants mettent chacun en œuvre un composant d'import/export de données. Le schéma EXPRESS qui décrit les données échangées est partagé par les deux systèmes communicants. Il représente un consensus sur la définition des données échangées.

Étant donné le schéma EXPRESS, l'encodage des données est explicitement décrit par la norme [ISO94c]. De plus, les contraintes décrites dans le schéma EXPRESS peuvent être vérifiées par les composants d'import/export. Ainsi, à partir d'un schéma EXPRESS spécifiant un méta-modèlele, il est en possible d'automatiser la vérification de la conformité des modèles.

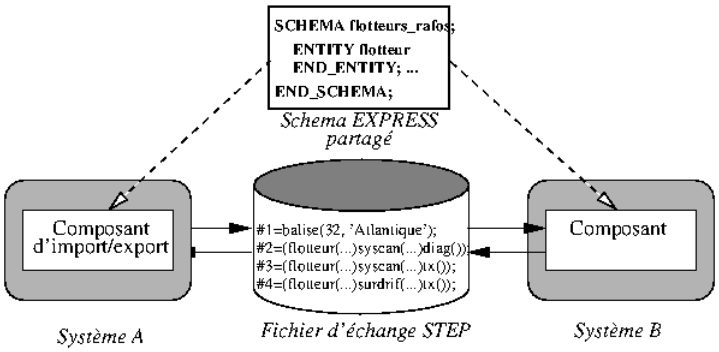

FIGURE 6.6 – Les processus d'échange par fichier STEP

6.3.4 Intégration par partage des données

L'objectif de la norme STEP est de pouvoir décrire et exploiter les données de produits pendant tout le cycle de vie. A chaque phase du cycle de vie correspond un ensemble de données, manipulées par des applications et/ou des composants logiciels spécifiques. Les informations mises à jour au cours d'une phase du cycle de vie peuvent être réutilisées directement ou indirectement au cours d'une autre étape.

La norme STEP prévoit un protocole de partage de données entre les différentes phases du cycle de vie. Ce protocole est décrit par la partie 22 [ISO94d] qui spécifie une interface et le protocole d'accès aux données stockées dans un dépôt de données (par exemple, une base

de données). Cette interface logicielle est communément appelée SDAI[2]. Cette spécification est donnée indépendamment des langages et des systèmes : le langage EXPRESS est utilisé pour la description des entités structurelles de la SDAI et le comportement de ces entités est décrit informellement en langue anglaise.

Les mises en œuvre de cette spécification dans des langages de programmation particuliers tel que le C++ [ISO95a], C [ISO95b] ou IDL [ISO95c], sont aussi décrites. Cependant, la SDAI est dite neutre car, quelque-soit le langage et le système physique de stockage, les données manipulées par les applications sont accédées au travers de la SDAI, telle qu'elles sont décrites par les schémas EXPRESS qui spécifient ces données. La SDAI définit une interface fonctionnelle entre les applications et l'environnement de stockage des données de sorte qu'elle permet l'indépendance des applications vis-à-vis des spécificités liées au stockage des données.

La norme STEP prévoit deux possibilités pour la mise en œuvre d'une SDAI. Elle peut être soit spécialisée, soit indépendante du schéma EXPRESS qui décrit les données manipulées (ou schéma EXPRESS source) :

- une mise en œuvre indépendante est dite SDAI générique (ou tardive), car elle permet l'accès aux données, quelque-soit le schéma EXPRESS source ;
- une mise en œuvre spécialisée (ou précoce) est dépendante du schéma EXPRESS source et peut être automatiquement produite à partir du schéma EXPRESS source.

Conformément aux deux approches pour la mise en œuvre d'outils, présentées dans le chapitre 2.3.5.4, la SDAI générique constitue un exemple d'application de l'approche par interprétation alors que la SDAI spécialisée constitue un exemple d'application de l'approche par génération de code.

C'est la SDAI qui permet l'échange de modèles conformes entre le système cible et le méta-atelier. La SDAI spécifie les composants qui permettent de manipuler les données au sein du méta-atelier et du système cible. Dans Platypus, la SDAI mise en œuvre dans le méta-atelier est générique alors qu'elle est de nature spécialisée au sein du système cible.

6.4 Platypus sur un exemple

Le besoin consiste à gérer un ensemble de courriers électroniques (exemple repris de [PSGR11]). L'objectif et de concevoir un système permettant l'envoi de courriers électroniques avec, pour chaque courrier, un émetteur, un destinataire, un sujet et un contenu de courrier. Le système cible est constitué par une application Java permettant la saisie et l'envoi des courriers électroniques. Dans ce chapitre, nous montrons comment :

- déclarer et manipuler un méta-modèle en construisant un prototype ;
- associer une syntaxe concrète particulière au méta-modèle pour implanter un langage spécifique ;
- déclarer les types et les contraintes (invariants) en EXPRESS ;
- manipuler les types déclarés ;
- vérifier la conformité d'un modèle en exploitant la déclaration des types et des invariants ;
- mettre en œuvre les transformations de modèles ;
- mettre en œuvre l'intégration avec le système cible.

2. Standard Data Acces Interface

6.4.1 Déclarer et manipuler un méta-modèle

Cet exemple est très simple, nous spécifions directement en Smalltalk une première version du méta-modèle *MbxMailBox* avec une classe *MbxMail* pour modéliser un courrier électronique et une classe *MbxMailBox* pour une boite à courriers.

6.4.1.1 La déclaration du méta-modèle en Smalltalk

La déclaration de la classe *MbxMailBox* est montrée par la figure 6.7. Un mail comprend quatre attributs : les attributs *sender* et *recipient* pour respectivement l'émetteur et le destinataire, et les attributs *subject* et *content* pour respectivement le sujet et le contenu du mail. Les accesseurs sont mis en œuvre automatiquement par l'éditeur de Pharo. Les accesseurs pour l'attribut *sender* de la classe *MbxMail* sont montrés par la figure 6.8. Il est important de remarquer qu'aucun type n'est précisé pour les attributs. Dans l'accesseur en écriture de la figure 6.7, *aString* est juste un nom de paramètre formel.

```
1  Object subclass : #MbxMail
2    instanceVariableNames : 'sender recipient subject content'
3    classVariableNames : ''
4    poolDictionaries : ''
5    category : 'MbxMailBox'
6
7  Object subclass : #MbxMailBox
8    instanceVariableNames : 'mails'
9    classVariableNames : ''
10   poolDictionaries : ''
11   category : 'MbxMailBox'
```

FIGURE 6.7 – Première version du méta-modèle *MailBox*

```
1  MbxMail>>sender
2    ^sender
3  MbxMail>>sender : aString
4    sender  := aString
```

FIGURE 6.8 – Les accesseurs pour l'attribut *sender*

6.4.1.2 Instrumentation du méta-modèle sous Pharo

Mettre à l'épreuve une version de méta-modèle consiste à le mettre en œuvre pour pouvoir l'instancier. Le système cible est mis en œuvre en Java. Nous disposons d'un générateur d'EXPRESS vers Java. Cependant, modéliser en EXPRESS puis générer en Java, assembler le système cible de façon être en mesure d'instancier le méta-modèle pour le tester est relativement coûteux.

Une des fonctions principales de notre système cible est l'envoi de courriers électroniques. Nous désirons atteindre un certain niveau de confiance quant à l'adéquation entre le besoin et

la spécification du méta-modèle. Avant de produire le code Java, nous voulons par exemple, nous assurer que l'envoi de courriers est possible et que le méta-modèle est satisfaisant pour cette fonction.

Un premier script de test. Dans Pharo, une couche réseau et des services réseau de base sont disponibles. Par exemple, un envoi de courrier électronique peut s'effectuer à l'aide des classes *MailMessage* et *MailSender*. Pour évaluer notre méta-modèle, nous pouvons mettre en œuvre le script présenté dans la figure 6.9. Ce script effectue les opérations suivantes :

- construction d'un tableau dynamique contenant des courriers (instances de la classe *MbxMail*),
- itération sur le tableau, pour chaque instance de *MbxMail*, construire une instance de *MailMessage* et utiliser *MailSender* pour envoyer le courrier électronique.

```
| mails |
mails := {
   MbxMail new sender : 'ap@br.fr' ; recipient : 'fs@br.fr' ; subject : 'Trip' ; content : 'Ready ?'.
   MbxMail new ...}.
mails
   do : [:m | | mes mime |
      mes := MailMessage empty.
      mes setField : 'from' toString : m sender.
      mes setField : 'to' toString : m recipient.
      mes setField : 'subject' toString : m subject.
      mime := MIMEDocument contentType : 'text/plain' content : m content.
      mes body : mime.
      MailSender sendMessage : mes].
```

FIGURE 6.9 – Un contrôle précoce du méta-modèle *MailBox* en Smalltalk sous Pharo

En exécutant ce script, nous observons que l'envoi de courrier est possible. Étant donnée la simplicité de cet exemple, notre connaissance déjà acquise et notre pratique de l'utilisation du courrier électronique, nous constatons immédiatement, à la simple lecture, que le méta-modèle n'est pas satisfaisant. Par exemple, le concept d'adresse électronique n'est pas explicitement représenté, un courrier ne prend en compte qu'un seul émetteur et qu'un seul destinataire, la mise en copie d'un courrier est impossible,... Pour un problème complexe dont la solution n'est pas connue au départ et pour lequel il n'y a pas de patron de conception, l'élaboration du méta-modèle peut être longue. Des erreurs de conception sont inévitablement introduites, puis détectées et corrigées. A ce niveau, c'est la mise en action et donc l'interaction entre le concepteur et son méta-modèle qui facilite la convergence vers une solution satisfaisante.

6.4.2 Déclarer une syntaxe concrète

Pour un système cible plus complexe, il serait utile d'enrichir les tests dynamiques pour accroître notre expérience et notre pratique au sein même du méta-atelier et mettre à l'épreuve le méta-modèle. Les tests interactifs sont appropriés dans le cas de cet exemple. Nous pouvons construire une interface Homme-Machine graphique ou associer une syntaxe textuelle à notre méta-modèle pour construire un langage spécifique complet. L'intérêt du langage est triple :

– il permet d'utiliser directement les éléments de méta-modèle, ce qui peut influer sur son élaboration ;
– il permet à des non experts de Smalltalk de se mettre d'accord sur un formalisme commun pour expérimenter le méta-modèle ;
– il simplifie l'écriture des tests et encapsule le savoir faire domaine.

Sous Pharo, plusieurs compilateurs de compilateurs sont disponibles. Pour cet exemple, nous utilisons *PetitParser* [Ren10]. Pour la spécification du langage, une classe particulière est nécessaire. Pour notre langage, elle se constitue de six méthodes. Chaque méthode comprend deux expressions. La première expression (1) spécifie la règle syntaxique et la seconde (2), exprime le lien avec les éléments du domaine. Par exemple, la figure 6.10 montre la méthode *mailProd* de la classe *PPMbxLang* qui construit une instance de courrier à partir d'une phrase du langage.

```
1 PPMbxLang>>mailProd
2   ∧(from, to, subject, content, semicolon) "(1) Production syntaxique"
3   ==> [:arr |
4         MbxMail new "(2) Instanciation d'un courrier"
5           sender : arr first ;
6           recipient : arr second ;
7           subject : arr third ;
8           content : arr fourth]
```

FIGURE 6.10 – Traitement du mot clé *from* du langage spécifique pour le courrier électronique

En utilisant notre langage spécifique, le script de test de la figure 6.9 est simplifié. La nouvelle version est montrée par la figure 6.11. Pratiquement, le méta-atelier permet de programmer directement l'éditeur avec la colorisation des mots clés.

```
1 from : 'ap@br.fr'
2 to : 'fs@br.fr'
3 subject : 'Trip'
4 Ready ? ;
5 from :...
```

FIGURE 6.11 – Un script utilisant le langage spécifique *MbxLang*

6.4.3 Déclaration des types en EXPRESS

Pour mettre en œuvre des vérifications basées sur les types dans le méta-atelier ou pour générer le code Java correspondant au méta-modèle, nous devons préciser les types des éléments de notre méta-modèle. Les types sont déclarés séparément par un modèle de données en EXPRESS. Des vérifications supplémentaires, basées sur les types et les invariants peuvent alors être implantées.

Le schéma EXPRESS est montré par la figure 6.12. Le lien avec le méta-modèle déjà implanté sous Smalltalk est directement établi par les noms, modulo le préfixe *Mbx* qui est précisé la première fois que le schéma EXPRESS est intégré.

```
SCHEMA MailBox;

  ENTITY Mail;
    sender  : STRING;
    recipient  : STRING;
    subject  : OPTIONAL STRING;
    content  : STRING;
  WHERE
    not_empty_keys  : -- subject et content ne peuvent pas etre vides tous les deux
      (EXISTS(subject) AND (LENGTH (subject) > 0))
        OR (LENGTH (content) > 0);
    well_formed_addrs  :: -- sender et recipient doivent etre des adresses courrier valides
      self sender isValidMailAddress
        and : [self recipient isValidMailAddress];
  END_ENTITY;

  ENTITY MailBox;
    mails  : LIST OF Mail;
  END_ENTITY;

  RULE mail_not_empty_rule FOR (Mail);
  WHERE
    not_empty_keys  : SIZEOF ( QUERY ( m <* Mail |
      ((NOT EXISTS(m.subject)) OR (LENGTH (m.subject) = 0))
        AND (LENGTH (m.content) = 0))) = 0;
  END_RULE;

END_SCHEMA;
```

FIGURE 6.12 – Déclaration des types en EXPRESS pour le méta-modèle *MailBox*

Ce méta-modèle comprend les deux entités *Mail* et *MailBox*. Les invariants sont déclarés sous la forme de contraintes locales ou globales. L'entité *Mail* spécifie quatre attributs explicites et deux contraintes locales.

La contrainte *not_empty_keys* spécifie qu'une instance de *Mail* doit avoir un sujet non vide ou un contenu non vide.

La contrainte *well_formed_addrs* permet de contrôler que les adresses courrier des attributs *sender* et *recipient* sont correctement formées. Cette contrainte est directement exprimée en Smalltalk [3].

La contrainte globale *mail_not_empty_rule* est données ici à titre d'exemple. Elle permet aussi de contrôler que toutes les instances de *Mail* ont un sujet non vide ou un contenu non vide. Le calcul est ensembliste. Il consiste à spécifier le calcul d'un ensemble d'instances et à tester le nombre d'instances de l'ensemble. Une instance de *Mail* appartient à l'ensemble si la valeur de l'expression exprimée par la requête est la valeur booléenne *true*. Dans la règle *not_empty_keys* de la contrainte globale *mail_not_empty_rule*, l'ensemble calculé doit être vide. Cette règle globale est redondante par rapport à la règle *not_empty_keys* déclarée

3. Depuis 2011, une nouvelle version du compilateur de Platypus permet d'utiliser directement Smalltalk pour la spécification des contraintes et des attributs dérivés ;

localement dans l'entité *Mail*.

6.4.4 Manipulation des types

La spécification du domaine en EXPRESS complète notre atelier spécialisé avec les descriptions des types et des contraintes. Ces descriptions peuvent être manipulées sous la forme d'objets. Ici encore, les outils standards de Smalltalk sont utilisés pour naviguer au sein du graphe d'objets décrivant le méta-modèle (par exemple à l'aide d'un inspecteur).

Des scripts peuvent aussi être mis en œuvre pour manipuler la description des types. Ces scripts peuvent être développés pour intégrer des visualisations particulières des modèles (par exemple produire une représentation de type UML) ou encore écrire des transformations de modèles.

```
| attr |
attr  := MbxMail platypusMetaData explicitAttributes
         detect : [:a | a name = 'sender'].
Transcript show : attr domain asClearText.
```

FIGURE 6.13 – Accès à la description d'un attribut déclaré en EXPRESS

Par exemple, le script de la figure 6.13 affiche le type de l'attribut *sender* pour le type entité *Mail*. Pour ce faire :

- l'objet qui décrit le type entité *Mail* est récupéré (envoi du message *platypusMetaData* à la classe *MbxMail*) ;
- la liste des attributs explicites est récupérée (envoi du message *explicitAttributes*) ;
- l'objet qui décrit l'attribut *sender* est ensuite récupéré par son nom ;
- la dernière ligne récupère le type de l'attribut (envoi du message *domain* à la description de l'attribut *sender* référencé par la variable locale *attr*) ;
- le domaine est aussi représenté par un objet qui est converti en une chaine de caractères finalement affichée sur une console.

6.4.5 Vérification des invariants

L'intégration d'un méta-modèle EXPRESS enrichi notre méta-atelier avec les définitions complètes des types en créant de nouvelles classes Smalltalk ou en enrichissant les classes existantes implantées en phase d'élaboration. Les outils standards de Smalltalk peuvent être utilisés pour manipuler les instances du méta-modèle et évaluer les contraintes. Par exemple, une série de tests *sUnit* peuvent être mis en œuvre pour vérifier dynamiquement des instanciations de notre méta-modèle. Le code de la méthode *testMailsConstraints* de la classe *MailBoxTest* est présentée dans la figure 6.14. Cette classe comprend la variable d'instance *mailbox* qui est initialisée (via des ressources de tests [BDN+]) avec une instance de *MailBox*. Pour chaque instance de *Mail* contenue dans *mailbox*, les contraintes spécifiées par l'entité *Mail* sont vérifiées :

- contraintes implicites : par défaut, un attribut explicite doit être obligatoirement valué, donc les valeurs des attributs *sender*, *recipient* et *content* ne doivent pas être *nil*

```
MailBoxTest>>testMailsConstraints
   mailbox mails do : [:m |
      self assert : m sender notNil.
      self assert : m recipient notNil.
      self assert : m content notNil.
      self assert : m not_empty_keys.
      self assert : m well_formed_addrs].
```

FIGURE 6.14 – Un test *sUnit* pour une instance de *MailBox*

(message *notNil* envoyé à chacune des valeurs d'attribut) ; l'attribut *subject* n'est pas contrôlé ici car il est explicitement qualifié comme pouvant être *nil* ;

– contraintes explicites : *not_empty* et *well_formed_addrs* sont aussi évaluées.

Des tests dynamiques de cette nature sont en fait générés automatiquement [4]

La visualisation avec colorisation, l'édition ou la vérification des invariants peut aussi s'effectuer directement au travers d'un outil spécialisé. Cet outil permet de se concentrer uniquement sur la forme EXPRESS pour l'édition des méta-modèles et sur le format STEP pour l'encodage, l'import et l'export de modèles conformes et la vérification interactive, locale ou globale, des instances de modèles conformes.

6.4.6 Transformation de modèles

Des transformations de modèles conformes peuvent s'appuyer sur les types déclarés en EXPRESS de deux façons différentes. Soit les transformations sont écrites en Smalltalk soit directement en EXPRESS.

Supposons que l'on veuille transformer des instances de *Mail* en une représentation textuelle de type *mbox*. Pour un mail, le résultat d'une transformation est montré par la figure 6.15.

```
From : ap@br.fr
To : fs@br.fr
Subject : Trip
<html>
 <body text="#000000" bgcolor="#FFFFFF">
     Ready ? <br>
 </body>
</html>
```

FIGURE 6.15 – Représentation de type *mbox* d'un courrier électronique

6.4.6.1 Spécification d'une transformation en Smalltalk

Cette transformation peut s'implanter à l'aide d'une extension de la classe Smalltalk *Mail* (le code des extensions est conservé en cas de re-génération d'une classe) ou à l'aide d'une

4. La génération automatique des tests est implantée depuis 2011 dans une version récente de Platypus

autre classe, comme par exemple, *MailtoMbox* dont la méthode principale *mboxFrom* : (abrégée) est montrée dans la figure 6.16. Cette méthode exploite le paramètre formel *aMail* qui référence une instance de *Mail*. La méthode retourne une chaine de caractère contenant le code de type *mbox* construit à partir de l'instance passée en paramètre. La chaine de caractères retournée est construite tout simplement par concaténation.

```
1 MailtoMbox>>mboxFrom : aMail
2   ^'From : ', aMail sender, ... , 'text= "' , textColor , ... ,'</html>'
```

FIGURE 6.16 – Transformation d'une instance de *Mail* mise en œuvre directement en Smalltalk

6.4.6.2 Spécification et évaluation d'une transformation en EXPRESS

La spécification des transformations en Smalltalk est avantageuse du point de vue de l'efficacité du processus de développement. Par contre, la spécification des transformations n'est pas aisément partageable avec d'autres systèmes que Platypus.

Il est possible de spécifier les transformations en EXPRESS par un schéma qui réutilise le schéma du domaine. L'avantage est que la spécification en EXPRESS est plus facilement réutilisable comme documentation et pour l'interopérabilité avec d'autres systèmes. Il est par exemple possible en utilisant Platypus, de traduire la spécification des transformations de modèle en un composant Java pouvant être réutilisé dans l'application cible.

La figure 6.17 montre l'entité *MailtoMbox* dont l'attribut dérivé *toMbox* spécifie la construction d'une représentation de type *mbox*, sous la forme d'une chaine de caractères, à partir d'une instance de *Mail*.

En cours de mise au point ou pour utiliser la traduction dans un test ou un prototype, on dispose de deux possibilités :

- une traduction particulière peut-être directement évaluée en EXPRESS, la figure 6.18 montre une expression comprenant la déclaration d'une instance de *Mail* utilisée pour construire une instance de *MailtoMbox* ; le résultat de la traduction est obtenu par lecture de l'attribut dérivé *toMbox* ;
- la même traduction peut être exprimée en Smalltalk ; c'est ce que montre la figure 6.19.

```
1 ENTITY MailtoMbox;
2   textColor  : CBinary;
3   backgroundColor  : CBinary;
4   mail  : Mail;
5 DERIVE
6   toMbox  : STRING  := 'From : ' + mail.sender + ... + 'text= "' + textColor + ... +'</html>';
7 END_ENTITY; ...
```

FIGURE 6.17 – Spécification en EXPRESS de la transformation d'une instance de *Mail*

Dans cette version de transformation, les informations nécessaires sont récupérées depuis une instance de *Mail* et le chemin d'accès à ces informations est explicitement déclaré. Le

```
(MailtoMbox('000000', 'FFFFFF', Mail('ap@br.fr', 'fs@br.fr', 'Trip', 'Ready ?') ) ).toMbox
```

FIGURE 6.18 – Evaluation d'une transformation d'une instance de *Mail* en EXPRESS

```
(MbxMailtoMbox new
 textColor : '000000';
 backgroundColor : 'FFFFFF';
 mail : (MbxMail new sender : 'ap@br.fr'; recipient : 'fs@br.fr'; subject : 'Trip'; content : 'Ready ?')) toMbox
```

FIGURE 6.19 – Evaluation d'une transformation d'une instance de *Mail* en Smalltalk

couplage entre *Mail* et *MailtoMbox* est fort puisque le type entité *Mail* est utilisé comme type de l'attribut *mail* de *MailtoMbox*.

La transformation est paramétrée non seulement par une instance de *Mail*, provenant d'un modèle source à transformer, mais aussi avec des informations concernant le modèle cible de transformation, soit le résultat à produire. Ces informations sont ici les couleurs de texte et de fond. On remarque qu'il est impossible de distinguer ce qui est lié au modèle source de ce qui est lié au modèle cible. Les différents aspects du modèle cible ne sont pas correctement différenciés.

Le couplage fort entre composants rend impossible la réutilisation indépendantes de *MailtoMbox* sans le type entité *Mail* et la non-différenciation des aspects rend plus difficile la compréhension et aussi la réutilisation des spécifications.

Les transformations de modèle sont coûteuses et peuvent aussi être difficiles à mettre en œuvre. Il convient de faciliter au maximum leur réutilisation. Pour découpler les deux types entités, nous utilisons la composition d'instance autorisée par EXPRESS.

6.4.6.3 Découplage d'une transformation à l'aide des instances complexes en EXPRESS

Dans le chapitre 6.3.2, nous avons vu qu'une instance peut être construite par composition de parties d'instances. Chaque partie d'instance complexe est décrite par un type entité. En EXPRESS standard, tous ces types entités sont contraints d'avoir un type entité commun, accessible par héritage. Les compositions valides sont exprimées au niveau de la définition du type entité commun. Par défaut, toutes les combinaisons de partie d'instance correspondant aux sous types du type commun sont autorisées.

Dans Platypus, pour pouvoir composer des parties d'instances, le type entité commun n'est pas obligatoire. Nous élargissons ainsi les possibilités de composition. Ainsi, dans Platypus, il est possible de composer n'importe quelle partie d'instance, même si les types entités correspondants ne sont pas liés par héritage.

Il s'agit donc d'un non-respect du standard. Cependant, nous disposons d'une possibilité de découplage complète : les types entités qui décrivent les parties des instances complexes peuvent être complètement indépendants.

La figure 6.20 montre la spécification d'une transformation découplée du modèle source. En effet, l'instance de mail utilisée comme partie de modèle source à transformer n'est pas explicitement spécifiée par un attribut. Le couplage est faible et le lien est dynamique : il

```
ENTITY MailtoMbox;
   textColor  : CBinary;
   backgroundColor  : CBinary;
DERIVE
   toMbox  : STRING  := 'From : ' + self.sender + ... + 'text= "' + textColor + ... +'</html>';
WHERE
   senderType  : 'STRING' IN TYPEOF(self.sender);
END_ENTITY; ...
```

FIGURE 6.20 – Version découplée d'une spécification en EXPRESS de la transformation d'une instance de *Mail*

est impossible de connaitre statiquement la provenance de l'attribut *sender* et son type. Ce n'est qu'au moment de la lecture de l'attribut dérivé *toMbox* que ces informations devront être accessibles et valides par rapport à leur utilisation dans l'expression.

Le modèle peut être enrichi d'une règle qui contraint la valeur provenant de *sender*. Par exemple, dans la figure 6.20, la contrainte *senderType* précise que le type de *sender* doit être compatible avec le type *STRING*. Pour aider à la mise au point, cette contrainte peut être évaluée dynamiquement, avant lecture des attributs dérivés. Cette contrainte est aussi très utile pour la compréhension du modèle.

Une instance de *MailtoMbox* peut être composée dans une instance complexe si les valeurs de *sender*, *recipient*, *subject* et de *content* peuvent être fournies par les autres instances de l'instance complexe. Aucune hypothèse n'est indiquée concernant comment ces valeurs sont obtenues ce qui rend la transformation spécifiés par *MailtoMbox* découplée et réutilisable. L'idée est proche de celle des *traits* [SDNB03, DNS+06]. Un trait représente une unité de comportement réutilisable. Un trait est spécifié sous la forme d'une pseudo-classe qui ne peut pas être instanciée. La spécification d'un trait doit être associée à la définition d'une classe. Les instances de la classe peuvent alors utiliser les services apportés par le trait. La différence avec notre approche est que dans Platypus, la réutilisation peut s'effectuer dynamiquement, au niveau des instances. Les utilisations d'une instance complexe en EXPRESS et en Smalltalk sont montrées respectivement par les figures 6.21 et 6.22.

```
(
 MailtoMbox('000000','FFFFFF')
 || -- Operateur de composition d'instance en EXPRESS
 Mail('ap@br.fr', 'fs@br.fr', 'Trip', 'Ready ?') ).toMbox
```

FIGURE 6.21 – Utilisation d'une instance complexe pour l'évaluation d'une transformation d'une instance de *Mail* en EXPRESS

6.4.7 Intégration par échange de données

Lorsque le système cible est opérationnel, des modèles conformes peuvent être échangés entre l'atelier dans le méta-atelier et l'atelier au sein du système cible (voir le chapitre 5.4).

Supposons qu'un système cible soit programmé en Java autour de composants générés

```
(
 (MbxMailtoMbox new  textColor : '000000'; backgroundColor : 'FFFFFF'))
 \\ "Message pour la composition d'instances en Smalltalk (extension de Platypus)"
 (MbxMail new sender : 'ap@br.fr'; recipient : 'fs@br.fr'; subject : 'Trip'; content : 'Ready  ?') ) toMbox
```

FIGURE 6.22 – Utilisation d'une instance complexe pour l'évaluation d'une transformation d'une instance de *Mail* en Smalltalk

à partir du méta-modèle *MailBox*. Suivant notre méthode, les composants Java générés comprennent des paquetages pour la gestion des courriers mais aussi des paquetages pour la sérialisation et la matérialisation des instances vers/depuis le format standard d'échange de modèles conformes (le format neutre de STEP en ce qui concerne la version actuelle de Platypus).

```
import step.core.*; // Import du framework STEP
import MbxMailBox.*; // import pour le schema MailBox
class ExportDemo {
   public static void main(String [] args) throws Exception {
      // Creation du repository
      MbxMailBox_StepRepository repo = new MbxMailBox_StepRepository();
      MbxMail m = new MbxMail(); // Creation d'une instance de Mail
      m.setSender("ap@br.fr");
      m.setRecipient("fs@br.fr");
      m.setSubject("Trip");
      m.setContent("Ready  ?");
      repo.record(m); // Enregistrement de l'instance dans le depot
      repo.writeFileNamed("m.step"); // Serialisation dans un fichier d'echange STEP
   }
}
```

FIGURE 6.23 – Construction et sérialisation d'un dépôt avec une instance de Mail en Java

Pendant sont exécution, le système cible gère ses propres instances de *Mail* et de *MailBox*. Des fichiers d'échange peuvent être produits depuis le système cible et matérialisés au sein de l'atelier spécialisé construit avec Platypus. Un générateur d'EXPRESS vers Java a été mis en œuvre avec Platypus dans le cadre du projet Morpheus. Le script de la figure 6.23 montre une utilisation des composants Java générés pour notre exemple. Ce script montre la création d'un dépôt de données, la création et l'enregistrement d'un courrier puis la sérialisation du dépôt dans un fichier d'échange. Le script de la figure 6.24 montre comment un fichier d'échange peut-être matérialisés et ses instances utilisées dans Platypus.

6.5 Conclusion

Dans ce chapitre, nous avons tout d'abord présenté l'architecture logicielle de notre méta-atelier Platypus, un environnement STEP [ISO94a] intégré à un système Smalltalk [GR83]. Platypus implante notre méthode pour la spécification, la vérification et la validation précoce

```
| repo |
repo := MbxMailBoxRepository new stepFileIn : 'm.step'. " Materialization depuis le fichier 'm.step' "
repo allInstancesOf : MbxMail do : [:m | | mes mime | " Pour tous les courriers, faire : "
   mes := MailMessage empty. " Creation, initialisation et envoi d'un MailMessage "
   mes setField : 'from' toString : m sender.
   mes setField : 'to' toString : m recipient.
   mes setField : 'subject' toString : m subject.
   mime := MIMEDocument contentType : 'text/plain' content : m content.
   mes body : mime.
   MailSender sendMessage : mes].
```

FIGURE 6.24 – Matérialisation et utilisation d'instances stockées dans un fichier d'échange

des méta-modèles. Smalltalk permet de d'assurer un bon niveau d'agilité pour la mise en œuvre des spécifications exécutables à haut niveau d'abstraction. Les outils de la norme STEP permettent d'enrichir les spécifications avec les types et les contraintes vis à vis du domaine et ainsi d'assurer le besoin de précision des spécifications. Sur la base de l'exemple simple d'un éditeur de courriers électroniques, nous avons ensuite très concrètement expliqué comment Platypus peut être utilisé.

Chapitre 7

Mise en œuvre de Platypus

Pour produire tout ou partie d'un système cible à partir de la spécification des méta-modèles, il est nécessaire de développer des transformations de modèles. En l'occurrence, il s'agit de transformer les méta-modèles spécifiés dans le méta-atelier vers une représentation qu'il est possible d'exploiter pour mettre en œuvre le système au sein de la plateforme d'exécution cible. Pour notre exemple du chapitre 6.4, il s'agit de générer du code Java à partir du modèle EXPRESS spécifié, vérifié et validé dans Platypus pour construire un éditeur de courrier en Java. Cette transformation peut être programmée de manière homogène, en spécialisant Platypus, car son méta-méta-modèle est décrit en EXPRESS et est utilisé pour sa mise en œuvre. Dans la suite de ce chapitre, nous décrivons comment Platypus est mis en œuvre autour de son méta-méta-modèle et comment son outillage est développé.

7.1 Le méta-méta-modèle de Platypus

Le méta-méta-modèle de Platypus est lui-même décrit et géré en EXPRESS. Il décrit tous les concepts manipulables pour la spécification et la mise en œuvre STEP des méta-modèles. Le méta-méta-modèle de Platypus comprend la définition complète du langage EXPRESS, incluant notamment la définition des expressions et des instructions. Il comprend aussi la spécification des instances et des gestionnaires de lots d'instances. Ainsi, on dispose d'une représentation très fine et exécutable des méta-modèles. Une grande partie de l'infrastructure de Platypus est produite automatiquement à partir de son méta-méta-modèle. L'auto-génération facilite la validation et la traduction de Platypus lui-même.

7.1.1 Compilation d'un méta-modèle EXPRESS

Pour chaque élément d'un méta-modèle EXPRESS, une instance du type entité correspondant du méta-méta-modèle de Platypus est créée et liée à la classe Smalltalk qui met en œuvre l'élément du méta-modèle EXPRESS. L'envoi du message *platypusMetaData* à cette classe retourne l'instance du méta-méta-modèle de Platypus qui la décrit. Ce lien est établi par le compilateur de schéma EXPRESS.

Par exemple, lorsqu'un type entité est spécifié, une instance de la méta-méta-entité *PltEntityDefinition* est créée. Cette classe décrit la structure et le comportement d'un type entité. Le type entité *entity_definition* est l'entité du méta-méta-modèle de Platypus correspondant à la classe *PltEntityDefinition*. Ce type entité est donné dans la figure 7.1.

```
1  ENTITY named_type
2    ABSTRACT SUPERTYPE OF ( ONEOF ( entity_definition, defined_type ) )
3    SUBTYPE OF ( dictionary_instance );
4    name  : STRING;
5    where_rules  : LIST OF where_rule;
6    SELF\entity_instance.owner  : context_definition;
7  END_ENTITY;
8
9  ENTITY entity_definition
10   SUBTYPE OF ( named_type );
11   supertype_constraint  : OPTIONAL supertype_constraint;
12   supertypes  : LIST OF UNIQUE entity_definition_reference;
13   attributes  : LIST OF UNIQUE attribute;
14   uniqueness_rules  : LIST OF UNIQUE uniqueness_rule;
15   instantiable  : BOOLEAN;
16 END_ENTITY;
```

FIGURE 7.1 – Les types entité *named_type* et *entity_definition* du méta-modèle de Platypus

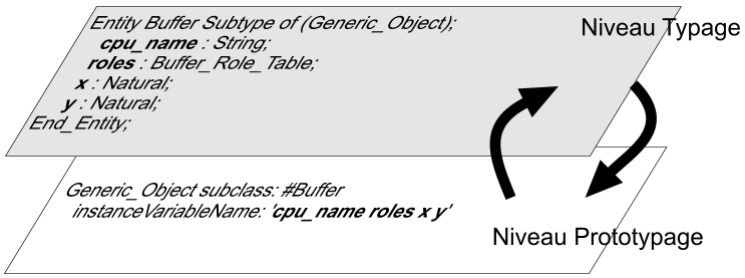

FIGURE 7.2 – Les niveaux *Prototypage* et *Typage* de Platypus

7.1.2 Mise en œuvre du double niveau de méta-modélisation

Platypus exploite un typage optionnel des méta-modèles. Il en résulte une méta-modélisation à deux niveaux de spécification qui se superposent et se complètent :

- le niveau *prototypage* est constitué des classes Smalltalk résultat de la phase d'ingénierie ;
- le niveau typage est constitué des méta-modèles EXPRESS qui précisent les types des éléments de méta-modèles.

La figure 7.2 illustre ces deux niveaux. Ces deux niveaux reflètent le même méta-modèle : lorsqu'un schéma EXPRESS est intégré, la représentation en Smalltalk est automatiquement créé ou modifiée. Le modèle, les schémas et les entités mais aussi les fonctions, les procédures et les règles globales sont intégrés sous la forme de classes Smalltalk dont chacune est liée à sa propre description stockée comme une instance du méta-méta-modèle.

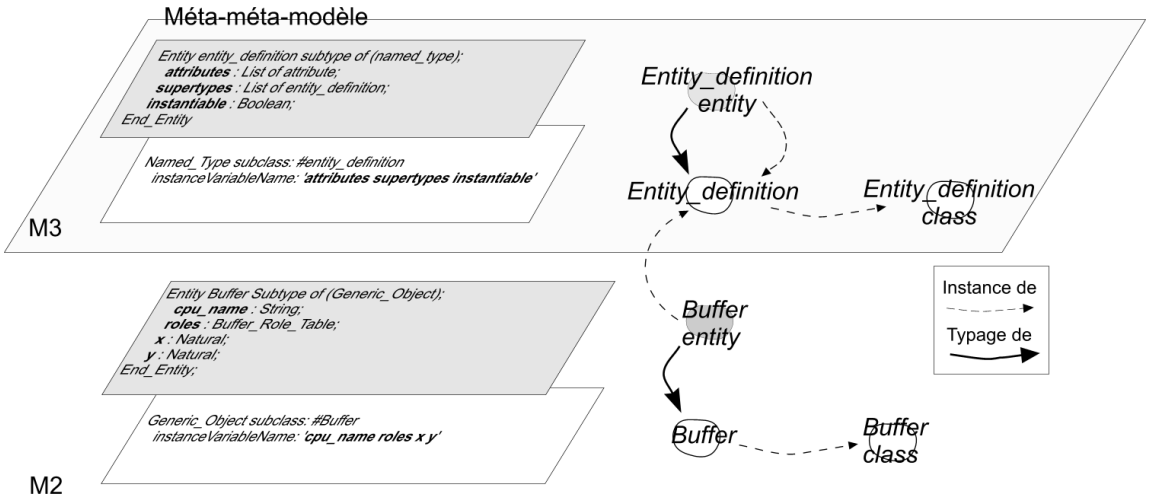

FIGURE 7.3 – Les couches M2 et M3 de la tour de méta-modélisation dans Platypus

Comme le montre la figure 7.3, cette méta-modélisation à deux niveaux est aussi implantée de façon régulière au niveau du méta-méta-modèle. L'intégration d'un méta-modèle EXPRESS produit ainsi non seulement le code Smalltalk du niveau prototypage mais produit aussi les instances conformes au méta-méta-modèle de Platypus. Ainsi, toutes les informations contenues dans le schéma EXPRESS d'un méta-modèle sont accessibles par navigation par le code Smalltalk du niveau prototypage sous la forme d'instances du méta-méta-modèle (introspection). Enfin, le méta-méta-modèle de Platypus peut lui-même être spécialisé. Cette spécialisation peut s'effectuer par ajout d'entités et d'invariants dans le méta-méta-modèle en EXPRESS, et peut s'effectuer aussi par enrichissement du code Smalltalk du niveau prototypage du méta-méta-modèle. Il en résulte la possibilité de spécialiser le méta-atelier lui-même et ce de manière homogène.

7.2 Extension et spécialisation de Platypus

Comme décrit dans le chapitre 6.4.6, la manipulation du méta-méta-modèle de Platypus s'effectue soit en Smalltalk, soit en EXPRESS. L'outillage de Platypus est implanté

directement par des extensions des classes du méta-méta-modèle (par exemple dans la classe $PltEntityDefinition$), par des visiteurs [Mar02] ou bien par extension en EXPRESS, par la spécification d'un complément au méta-méta-modèle de Platypus.

7.2.1 Extension du comportement en Smalltalk

Une extension du comportement peut être implantée dans les classes du niveau prototypage du méta-méta-modèle de Platypus. Le mécanisme d'extension est standard et s'effectue par ajout de méthodes directement dans les classes ou par utilisation d'un visiteur. Les différents éditeurs de Platypus, les analyseurs et certaines transformations de modèles sont intégrés de cette façon.

L'intérêt majeur est qu'il est possible d'utiliser l'environnement standard de Smalltalk pour le développement. Un autre avantage est que l'efficacité en termes de rapidité d'exécution est optimisée.

L'extension directement en Smalltalk est cependant problématique car

- on est limité par les concepts décrits par le méta-méta-modèle de Platypus ;
- une extension peut nécessiter l'adaptation du compilateur et des différents analyseurs ; une telle extension peut-être coûteuse et la compatibilité des ateliers mis en œuvre avant l'évolution peut être perdue ;
- alors que le méta-méta-modèle de Platypus peut lui-même servir de modèle source pour une transformation de modèle, par exemple pour porter Platypus, les extensions implantées directement en Smalltalk ne sont pas automatiquement prise en compte.

7.2.2 Extension structurelle en EXPRESS

La portée du méta-méta-modèle de Platypus est large puisqu'il comprend non seulement la description des types entités et de leurs relations mais aussi des aspects comportementaux (les contraintes locales et globales et les attributs dérivés). Les concepts de procédure, de fonction de variable, de paramètres,... sont spécifiés et utilisés pour la validation des méta-modèles. Cependant, il peut être nécessaire d'introduire de nouveaux concepts ou de spécialiser les concepts existants notamment pour décrire un autre langage qu'EXPRESS.

Des nouveaux concepts peuvent être déclarés pour enrichir le méta-méta-modèle et spécialiser Platypus. L'analyseur de modèle EXPRESS peut être aussi spécialisé pour automatiser la prise en compte des extensions.

7.2.2.1 Déclaration de nouveaux concepts

La déclaration de nouveaux concepts s'effectue par ajout d'un nouveau schéma EXPRESS et la réutilisation explicite du méta-méta-modèle de Platypus. Par exemple, dans le projet Cheddar, le langage de la plateforme cible est Ada. Ce langage intègre deux représentations possibles pour les classes. On dispose en effet des *record* et des *tagged record*. Ces concepts ne sont pas présents en EXPRESS mais peuvent être spécifiés comme des spécialisations du concept d'entité d'EXPRESS.

La figure 7.4 montre la spécification de *tagged_record* et de *record*. Ces déclarations sont effectuées dans le nouveau schéma *ada_dictionary_schema* qui réutilise le méta-méta-modèle de Platypus (schéma *platypus_dictionary_schema*). On remarque que l'entité *tagged_record* comprend l'attribut explicite *is_private*, un tagged record pouvant être privé ou public. Ces deux entités héritent de *ada_entity*. Les attributs dérivés *ads_code* et

```
SCHEMA ada_dictionary_schema;
USE FROM platypus_dictionary_schema; -- Reutilisation du meta-meta-modele de Platypus

  (* Description d'une entite Ada comme une specialisation d'une entite EXPRESS *)
  ENTITY ada_entity
    ABSTRACT SUPERTYPE
    SUBTYPE OF ( entity_definition );
  DERIVE
    ads_code : STRING := ?;
    adb_code : STRING := ?;
  END_ENTITY;

  (* Description d'un tagged record Ada *)
  ENTITY tagged_record
    SUBTYPE OF ( ada_entity );
    is_private : BOOLEAN;
  DERIVE
    SELF\ada_entity.ads_code : STRING := tagged_record_ads_code ( SELF, is_private );
    SELF\ada_entity.adb_code : STRING := tagged_record_adb_code ( SELF );
  END_ENTITY;

  (* Description d'un record Ada *)
  ENTITY record
    SUBTYPE OF ( ada_entity );
  DERIVE
    SELF\ada_entity.ads_code : STRING := record_ads_code ( SELF );
    SELF\ada_entity.adb_code : STRING := record_adb_code ( SELF );
  END_ENTITY;
```

FIGURE 7.4 – Spécialisation du méta-méta-modèle de Platypus pour décrire les concepts de *record* et de *tagged_record* du langage Ada

adb_code décrivent le calcul du code pour respectivement, la spécification et le corps d'un paquetage Ada.

7.2.2.2 Adaptation de l'analyseur

L'instanciation du méta-méta-modèle peut être effectuée directement en Smalltalk. Cependant, classiquement, l'instanciation d'un méta-méta-modèle est la responsabilité d'un analyseur de méta-modèles spécifiés dans une syntaxe concrète particulière. Dans Platypus, le langage par défaut pour spécifier les méta-modèles est EXPRESS.

L'analyseur de schéma EXPRESS est programmé en Smalltalk. Il n'est pas facile de faire évoluer un tel compilateur. De plus, une adaptation particulière introduit une nouvelle version dépendante du contexte d'utilisation ce qui induit une gestion de configuration très complexe, voire impossible à maintenir.

L'analyseur de schéma EXPRESS peut être adapté pour permettre l'utilisation de nouveaux concepts introduits par spécialisation du méta-méta-modèle. Pour son adaptation, nous considérons l'analyseur de schéma EXPRESS comme un aiguilleur constitué d'un ensemble d'aiguillages valides.

Dans le cas général d'un analyseur pour un langage L, à partir d'une expression particulière du langage L, l'analyseur sait construire une et une seule représentation interne valide. Donc, habituellement, l'ensemble des aiguillages dont l'analyseur est constitué est figé. Cet ensemble constitue un élément de la partie fixe de la mise en œuvre de l'analyseur.

Pour autoriser l'adaptation de l'analyseur de schéma EXPRESS, nous réifions le concept d'aiguillage et nous autorisons le paramétrage de l'analyseur de schéma EXPRESS avec l'ensemble des aiguillages valides. L'ensemble des aiguillages autorisés est un élément de la partir variable de la mise en œuvre de notre analyseur de schéma EXPRESS.

Dans Platypus, pour adapter l'analyseur de schéma EXPRESS un langage spécifique a été implanté. Une déclaration dans ce langage consiste en la spécification d'un aiguillage.

Pour notre exemple, la spécification des aiguillages permet d'indiquer à l'analyseur de schéma EXPRESS dans quel cas instancier les types entité *tagged_record* et *record* au lieu d'instancier directement *entity_definition* et avec quelle valeur pour l'attribut *is_private* de *tagged_record*. Une telle déclaration est montrée par la figure 7.5.

Le schéma *Cheddar_Buffers* constitue une partie du méta-modèle du système cible. Il comprend la définition d'un type tampon (type d'élément d'une architecture temps réel utilisé dans Cheddar). Le système cible est mis en œuvre en Ada. Dans le système cible, le concept de *Buffer* est mis en œuvre par un *tagged record*.

```
1  SCHEMA Cheddar_Buffers;
2  USE FROM Cheddar_Object;
3
4    ENTITY Buffer
5      SUBTYPE OF ( Generic_Object );
6      cpu_name  : STRING;
7      address_space_name  : STRING;
8      queueing_system_type  : Queueing_Systems_Type;
9      size  : Natural;
10     roles  : Buffer_Roles_Table;
11   END_ENTITY; ...
12 END_SCHEMA;
13
14 SCHEMA Cheddar_Mapper;
15 META FROM ada_dictionary_schema;
16 USE FROM Cheddar_Buffers;
17
18   MAP TO ada_entity ( ) || tagged_record ( false );
19     Generic_Object;
20     Buffer;
21   END_MAP; ...
```

FIGURE 7.5 – Adaptation de l'analyseur de schéma EXPRESS

Le schéma *Cheddar_Mapper* spécifie les aiguillages que l'analyseur peut utiliser. En particulier les déclarations entre les mots clé MAP et END_MAP. Dans l'aiguillage du schéma *Cheddar_Mapper* de la figure 7.5, l'expression

$$ada\_entity() || tagged\_record(false)$$

indique que les entités *Generic_Object* et *Buffer* doivent être considérées, par l'analyseur de schéma EXPRESS, comme des *tagged_record* publics. Ainsi, pour les entités *Generic_Object* et *Buffer* du méta-modèle décrit par le schéma $Cheddar\_Buffers$, une instance de *tagged_record* (méta méta entité décrite dans le schéma *ada_dictionary_schema*) est crée. La valeur affectée à la propriété *is_private* de la méta méta entité *tagged_record* est $false$.

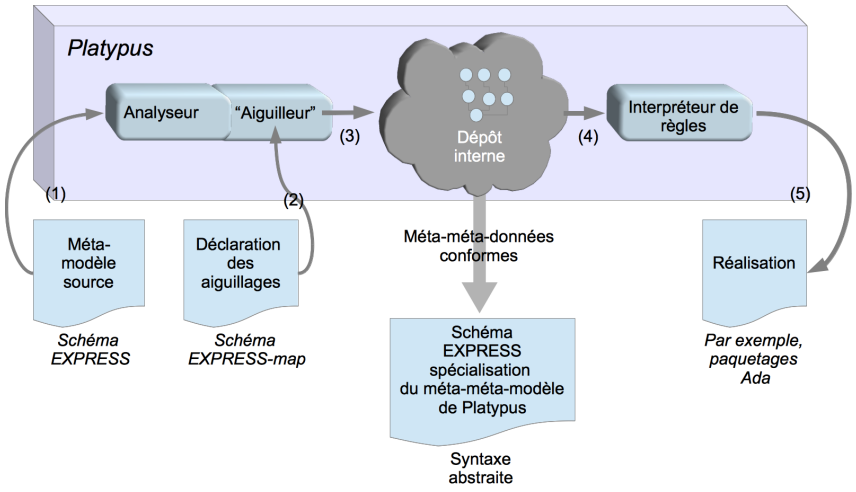

FIGURE 7.6 – Processus d'analyse et d'aiguillage mis en œuvre pour une transformation de modèle

La figure 7.6 illustre le principe de l'aiguillage et son utilisation dans un processus de traduction d'un schéma EXPRESS. Cette figure comprend cinq points :

1. le méta-modèle source est lu par l'analyseur de schéma EXPRESS,
2. la déclaration des aiguillages est analysé, le résultat est composé avec celui de l'analyse du méta-modèle,
3. le méta-méta-modèle de Platypus étendu avec les définitions spécifiques au langage cible (ici Ada) est instancié, les méta-méta-données sont gérées dans un dépôt,
4. les méta-méta-données sont lues depuis le dépôt
5. les règles de traductions associées aux méta-méta-données déclarées dans l'extension du méta-méta-modèle de Platypus (par exemple *ads_code* et *adb_code* du type entité *tagged_record* dans l'exemple de la figure 7.4), sont interprétées pour produire la réalisation (ici, un ensemble de paquetages Ada).

7.3 Conclusion

Pour traduire un méta-modèle vers une représentation utile au système cible, il est nécessaire de pouvoir manipuler le méta-méta-modèle de Platypus. Ce méta-méta-modèle est mis en œuvre de façon homogène dans Platypus. On dispose dans Platypus, du niveau prototypage et du niveau typage pour le méta-méta-modèle de Platypus lui même. Il ainsi possible

de spécialiser Platypus pour implanter, par exemple, des générateurs de code tout en bénéficiant de manière régulière, du méta-atelier Platypus lui même et de son agilité pour spécifier, vérifier et valider ses extensions.

Dans ce chapitre, nous avons décrit comment la réflexivité de Platypus est mis en œuvre autour de son méta-méta-modèle et comment cette mise en œuvre peut être réutilisée pour le spécialiser.

Une extension de Platypus peut notamment être implantée par spécialisation de son méta-méta-modèle en Smalltalk ou bien en EXPRESS. Nous avons aussi expliqué comment l'analyseur de schéma EXPRESS peut être adapté pour exploiter des extensions de Platypus lorsqu'elles sont spécifiées en EXPRESS.

Chapitre 8

Expérimentations

Platypus et la méthode de méta-modélisation qu'il permet de mettre en œuvre ont été très concrètement exploités. Les applications ont permis des améliorations techniques du méta-atelier et une clarification progressive de la méthode. Trois projets ont directement bénéficié de Platypus :

- Dans *Premecs II*, un projet Européen de l'Ifremer, *Platypus* a été utilisé pour la spécification et le prototypage d'un langage permettant la définition de la structure et des propriétés de talons de chaluts ;
- Dans le cadre du projet Européen *Morpheus*, *Madeo*, est un environnement pour la spécification, la validation et l'implantation de circuits reconfigurables, *Platypus* a été utilisé pour la ré-ingénierie du langage de spécification des circuits et la synthèse automatique des composants d'échange des circuits ;
- Dans le cadre du projet Cheddar, l'environnement *Cheddar* permet la validation de l'ordonnancement des architectures temps-réel ; *Platypus* est utilisé pour la production

semi-automatique de Cheddar et est actuellement au cœur d'un contrat de transfert technologique avec la société Ellidiss Technologies ; ce contrat a permis notamment le cofinancement d'une thèse régionale.

Pour ces trois projets, je suis principalement intervenu comme expert en méthode et en langage (voir le chapitre 2.3.4). J'ai pu travailler dans trois contextes différents avec, pour chaque projet des experts du domaine et de la plateforme d'exécution cible.

8.1 2005-2006 Premecs II

Le besoin de spécifier et d'élaborer un méta-modèle en Smalltalk indépendamment d'un typage statique en EXPRESS est apparu comme très bénéfique pour un projet exploratoire consistant à évaluer les capacités de Platypus pour la ré-ingénierie d'un langage de modélisation des talons de chalut [PR06b]. Cette application a été développée dans le cadre du projet Européen PREMECS II [Pla05] dont IFREMER Brest est membre coordinateur. PREMECS II développe un modèle prédictif de sélectivité des talons de chaluts de pêche. L'objectif de PREMECS II est de prévoir la sélectivité de chaluts commerciaux. La sélectivité est défini comme la capacité de garder certaines espèces et tailles de poissons. Dans ce cadre, une chaîne de traitements permettant la modélisation des talons de chaluts en 2D puis leur visualisation en 3D a été développée par Daniel Priour de l'IFREMER.

Concernant Platypus, cette étude a pour objectif d'évaluer sur une partie de la mise en œuvre de cette chaine de traitements une méthode de ré-usinage de logiciels principalement basées sur la modélisation des données et la génération de code.

Une partie importante de l'étude consistait à élaborer un langage pour exprimer des propriétés structurelles, physiques et dynamiques de talons de chaluts. Un atelier, construit autour de la définition d'un tel langage, permettant l'édition textuelle et graphique de modèles de talons de chalut a été développé avec Platypus.

8.1.1 Contexte à IFREMER

Le sujet de l'étude est le logiciel FEM, développé dans le cadre de PREMECS par Daniel Priour, ingénieur-chercheur à IFREMER. Ce logiciel scientifique permet :

- la modélisation et la visualisation de structures (par exemple des talons de chaluts),
- et le calcul et la visualisation de modèles 3D de structures

La modélisation de structures s'effectue par la spécification d'éléments de base comme des panneaux ou des câbles. Ces éléments sont décorés par des valeurs de propriétés mécaniques qui servent à la documentation de la structure mais aussi aux calculs effectués pour la visualisation 3D et pour la validation des caractéristiques mécaniques d'une structure.

Les logiciels développés par Daniel Priour sont écrits en C. Hormis l'utilisation de la librairie *libsx* pour les visualisations et d'un composant open-source pour la triangulation des polygones, les programmes sont de nature monolithique, très fortement centrés sur la mise à jour et la consultation d'un modèle de données complexe.

8.1.2 Problématiques

Il s'agit d'un problème de ré-ingénierie. Le problème est double. (1) Le domaine n'est pas explicitement décrit et (2) le contexte est celui d'un laboratoire de recherche, les logiciels ont un objectif exploratoire et sont en continuelle évolution.

8.1.2.1 Définition du domaine

Bien que fonctionnel, le logiciel FEM s'avère très difficilement maintenable. De plus, il apparaît comme peu réutilisable, en particulier parce que les contours du domaine ne sont pas discernables. Il est difficile de s'appuyer sur le code existant pour déterminer les objets du domaine et leurs propriétés :

- les structures de données sont complexes, elles spécifient les ensembles de données fondamentaux correspondant aux concepts manipulés (les chaluts, les panneaux, les nœuds ...) mais aussi des données spécifiques à certains calculs ;
- certaines structures ou éléments de structures sont obsolètes ;
- on est en présence d'un problème de type "code spaghetti" : il est difficile de définir clairement les fonctionnalités et les données qui sont manipulées via des variables globales ;

On dispose d'exemples de définitions de structures de chaluts, exploitées dans PREMECS (les descriptions de chalut stockées dans les fichiers « .don », fichier ASCII dont le texte de la figure 8.1 en présente un extrait). A partir de ces descriptions, il nous est possible de déterminer le type des données manipulées. Bien que non explicites les informations de types, les associations entre un élément et ses propriétés mécaniques, peuvent être exploitées pour produire une description explicite et formelle du modèle sous-jacent.

```
1  Panneau : 1
2  nombre de points du contour : 3
3  points du contour dans l ordre du contour no x y z U V no_type no_type_suivant :
4  1 −4.3 0 9.5 0.0 49.0 2 2
5  2 −2.7 0 10.8 40.0 0.0 2 2
6  3 −4.3 0 10.8 40.0 49.0 2 2
7  Raideur traction (N) : 7.800000e+04
8  Raideur compression (N) : 1
9  Raideur ouverture (N.m/rad) : 0.000000
10 Maille au repos (m) : 0.075000
11 Masse volumique (kg/m3) : 968.000000
12 Diametre hydrodynamique (m) : 0.003200
13 largeur noeud : 0.006400
14 Cd normal : 1.200000
15 Cd tangentiel : 0.080000
16 Pas du maillage (m) : 0.75
17 type des noeuds interieurs : 2
18 type de maillage : 2
```

FIGURE 8.1 – Un exemple de déclaration de panneau décrit dans le format utilisé à l'origine par la chaine de traitement de PREMECS

8.1.2.2 Évolutivité

Les logiciels existants évoluent très souvent ce qui implique une revue continuelle du code. La cause de cette instabilité est double :

- pour un même type d'application (le type d'application premier étant l'étude de la sélectivité des talons de chalut), différentes façons de modéliser sont testées ; les variations entre modèles concernent essentiellement les caractéristiques ou attributs des éléments des modèles étudiés et les algorithmes qui les utilisent ; on peut ajouter, retirer ou modifier une caractéristique ;
- bien que construits pour répondre aux besoins des projets de PREMECS, les logiciels développés ont déjà été ou bien seront exploités pour d'autres applications similaires comme la modélisation de structures pour l'aquaculture ou encore les barrages flottants ; toutes ces applications ont en commun la manipulation de structures mais se différencient par la terminologie et les caractéristiques des composants utilisés pour la modélisation des structures.

Régulièrement, une nouvelle version d'application peut être développée pour évaluer l'impact ou l'intérêt de nouveaux paramètres ou de nouveaux modes de calcul.

L'existant opérationnel peut cependant être utilisé par plusieurs intervenants du projet Européen PREMECS. Les versions publiées et stabilisées ne doivent donc pas être remises en cause par de nouvelles investigations logicielles.

On a d'une part un besoin de stabilisation d'une application, répondant à la nécessité de partage entre les différents membres de PREMECS, et d'autre part un besoin de forte capacité d'évolution permettant de disposer d'un plan de travail pour des investigations scientifiques.

Du point de vue de l'ingénierie du logiciel, ces deux besoins sont contradictoires. Bien que l'utilisation des technologies objets permette d'améliorer substantiellement la qualité d'un tel développement, un procédé classique de mise en œuvre considérant un ensemble de composants maintenus en configuration autour d'une application unique ne permet pas la stabilisation complète d'une version et la satisfaction du besoin d'évolution continue.

8.1.3 Les travaux effectués

La démarche adoptée consiste à développer un méta-atelier spécialisé pour la mise en œuvre d'une famille d'applications ayant en commun le domaine de la manipulation de structures. Chaque application de cette famille constitue un atelier spécifique. Un atelier met en œuvre un sous-domaine. Le développement d'un atelier s'effectue après expertise et définition claire du sous-domaine cible. La définition du sous-domaine est conduite à partir de son instrumentation par et pour les utilisateurs. Nous proposons un processus de ré-ingénierie en deux grandes étapes :

1. la première étape a pour objectif la construction d'un prototype d'atelier permettant de remplacer la solution actuelle en se bornant strictement aux besoins du sous-domaine du projet PREMECS (modélisation et visualisation 2D et 3D de talons de chaluts) ;
2. la seconde étape consiste à mettre en place le méta-atelier spécialisé pour la manipulation de structures qui, à partir d'une spécification d'un sous-domaine produit un atelier dédié de type PREMECS ; à l'issue de cette étape, on doit pouvoir produire non seulement l'exemplaire programmé manuellement lors de la première étape mais aussi des variantes ou des ateliers pour d'autres sous-domaines.

Les travaux effectués pour ce projet ont permis le développement d'un prototype d'atelier (première étape) et d'étudier les caractéristiques du méta-atelier exploitable pour plusieurs sous-domaines.

8.1.3.1 Prototypage d'un atelier pour les talons de chalut

La mise en œuvre procède tout d'abord de la modélisation du sous-domaine concernant les talons de chalut. Comme le montre la figure 8.2, le processus de modélisation est cyclique. La convergence vers une solution procède de l'utilisation d'un ensemble d'outils construits spécifiquement : un éditeur structuré, un éditeur 2D et un module d'export vers le langage de présentation *VRML* pour les visualisations 3D.

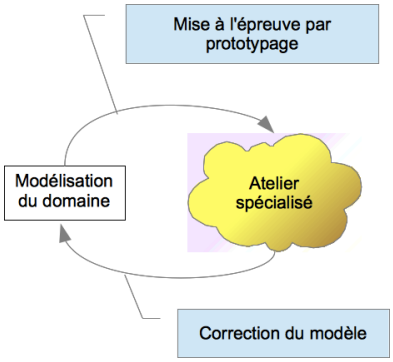

FIGURE 8.2 – Les cycles de développement du prototype

La première version considérée comme suffisamment stable constitue un exemplaire d'atelier pour le sous-domaine des talons de chaluts. Pour la suite, cet atelier est considéré comme l'exemplaire de référence (appelé *Phobos*). La figure 8.3 montre une capture d'écran avec, en fond, l'éditeur de modèle textuel de talon de chaluts, et en avant plan, l'éditeur de modèles en 2D. Cet atelier spécialisé a été mis en œuvre sous Squeak [Squ], le système Smalltalk qui a été utilisé pour le développement des premières versions de Platypus. Ces prototypes ont essentiellement servi à comprendre le domaine et à valider les caractéristiques désirables du langage de modélisation des talons de chalut en présence de l'expert du domaine à l'IFREMER.

8.1.3.2 Le méta-atelier spécialisé

L'objectif de la seconde étape est de répondre au fort besoin d'évolution suivant les deux niveaux décrits dans le chapitre précédant. L'idée est de définir les contours d'un méta-atelier permettant de générer automatiquement des exemplaires d'application de type *Phobos*. Comme la figure 8.4 le montre, le méta-atelier a pour rôle de produire des applications appelées les exemplaires. Un exemplaire correspond à une version d'applications pour un sous-domaine particulier. Les différents exemplaires d'un même sous-domaine constituent une ligne d'exemplaires. Les exemplaires d'application générés se différencient donc suivant deux axes :

- le premier axe est celui déterminé par le sous-domaine avec des applications différentes par sous-domaine,
- le second axe correspond aux différentes versions qu'on peut produire pour un sous-domaine particulier par la modification de la modélisation spécifique au sous-domaine (ajout ou modification des éléments de structure, ajout ou modification des propriétés relatives à un élément de structur, ...).

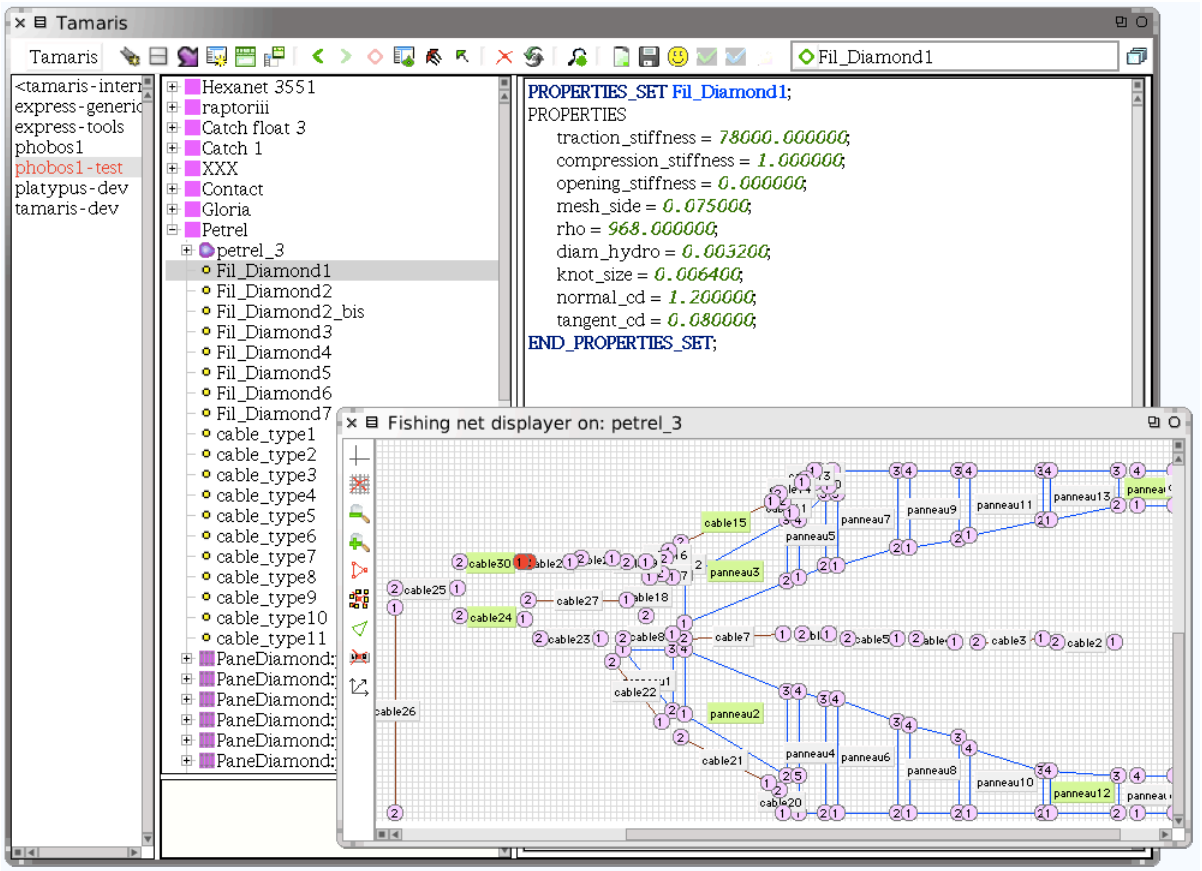

FIGURE 8.3 – Les éditeurs de talons de chalut de Phobos

Tous les exemplaires sont constitués d'une partie commune (méta-modèles pour les structures et mécanismes associés). Cette partie commune est identique d'un exemplaire à un autre et est gérée au sein du méta-atelier.

Les ateliers sont tout d'abord vérifiés et validés au sein du méta-atelier, par l'utilisation d'éditeurs et d'outils spécifiques (par exemple les outils montrés dans la figure 8.3). Des méta-modèles communs peuvent être réutilisés et spécialisés pour un sous domaines. Les outils associés aux méta-modèles communs sont réutilisables pour permettre la vérification et la validation au sein du méta-atelier, avant génération de l'atelier spécialisé pour la plateforme d'exécution cible. La grammaire de la syntaxe du langage textuel pour la spécification d'une structure et l'analyseur produisant une représentation interne à partir d'une spécification de structure sont automatiquement déduits des méta-modèles décrivant la partie spécifique du sous-domaine. Il en est de même pour l'éditeur 2D dont la mise en œuvre au sein du méta-atelier peut être automatiquement synthétisée.

Pour la génération de code, le méta-atelier prend en paramètre une partie variable, qui correspond à ce qui varie d'un exemplaire à un autre. Cette partie variable est déclarée dans des méta-modèles spécifiques aux sous-domaines au sein du méta-environnement.

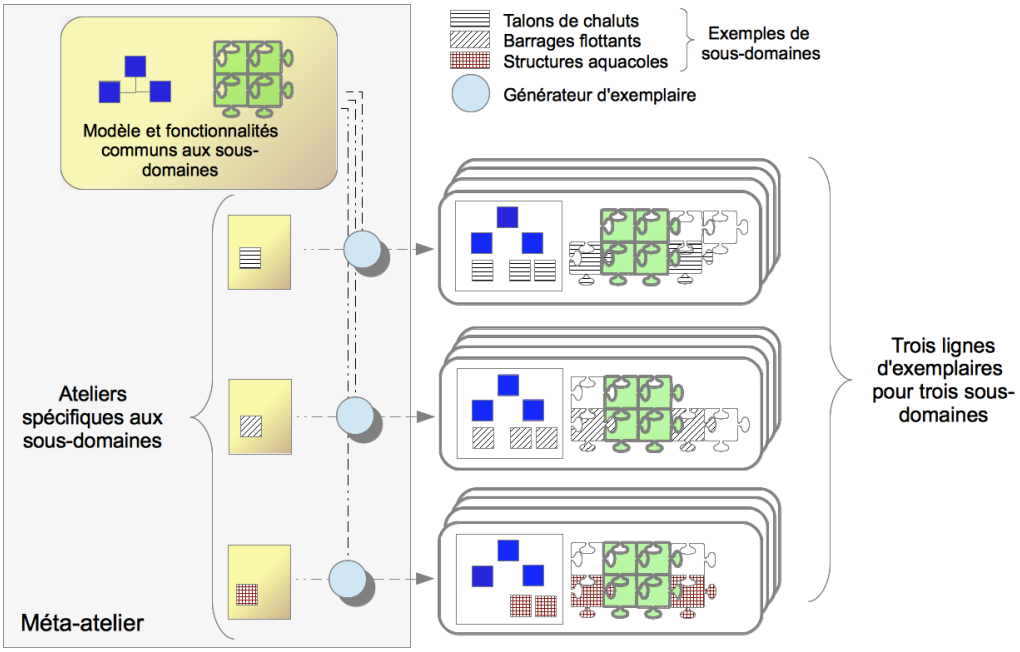

FIGURE 8.4 – Premecs : proposition de méta-atelier pour les structures encordées

8.1.4 Bilan

Pendant ce projet certaines bases importantes de la méthode Platypus ont pu être établies :

- la manipulation concrète du langage au travers de prototypes est intéressante pour faciliter la convergence vers une solution satisfaisant les besoins du domaine ;
- alors que la plateforme cible visée est une station Unix et que le langage finale est le C, le prototypage au sein du méta-atelier avant génération de code a constitué un réel atout pour minimiser l'effort de conception ;
- la possibilité d'utiliser des modèles existants, issus de cas réels construits dans le système cible constitue un atout pour l'élaboration et la validation des méta-modèles.

8.1.4.1 Typage optionnel

Pour ce projet, nous avons tout d'abord procédé classiquement par une spécification de méta-modèles en EXPRESS et l'utilisation des classes Smalltalk correspondantes pour développer un composant de reprise des données. Cependant, les premières versions ont été très rapidement remises en cause. Maintenir la cohérence entre d'une part une représentation textuelle typée en EXPRESS et les méta-modèles exécutables s'est avéré coûteux, même au sein du méta-atelier. De plus, le typage EXPRESS réifié en Smalltalk n'était pas utilisé. Les évolutions ont donc été ensuite appliquées directement sur les méta-modèles exécutables sans tenir compte du typage statique. A la fin du projet, nous avons synchronisé le typage EXPRESS avec les classes et les associations Smalltalk pour documenter le langage, préciser les invariants et expérimenter la génération de code vers le langage VRDL pour la représentation 3D des culs de chalut.

8.1.4.2 Importance du système cible pour la validation

La transformation de modèle est intervenue en deux points du processus d'élaboration du langage. En particulier, pendant la phase d'acquisition des connaissances, nous avons construit une passerelle de reprise des données, permettant de traduire les modèles de talons de chaluts existants vers le langage de STEP exploité par le méta-atelier Platypus. Ces modèles issus de cas réels ont permis de constituer très précocement un repository de modèles. Disposer de ces modèles réels s'est avéré très favorable à la compréhension du domaine et du besoin fonctionnel.

8.2 2006-2007 Morpheus

Parallèlement, Platypus a été exploité dans le cadre du projet Européen *Morpheus*. Dans ce contexte, l'atelier *Madeo* était utilisé pour spécifier et valider des descriptions de traitements pour des architectures reconfigurables. L'atelier Madeo a été tout d'abord développé à l'UBO en Smalltalk par Loïc Lagadec, indépendamment de Platypus. Une première version a été stabilisée et rigidifiée par un typage statique avec EXPRESS à l'aide de Platypus [BLPL07]. Des vérifications supplémentaires utilisant le typage ont pu être développées ainsi qu'un générateur de code vers Java [PTV+07]. Pour ce projet, Platypus a été automatiquement porté vers VisualWorks, le système Smalltalk utilisé pour *Madeo*.

8.2.1 Le projet Morpheus

Le déploiement accru de systèmes embarqués implique de nouveau besoins en termes de performances de calcul, de coût de conception et de validation des systèmes, de consommation d'énergie, ... La conséquence est un accroissement de la complexité des solutions et les outils disponibles ne sont plus adaptés.

Le projet Morpheus vise à apporter une solution intégrant d'une part, le matériel reconfigurable et d'autre part, l'outillage logiciel. L'ambition est de construire des plateformes spécifiques au domaine et flexibles, positionnées entre les technologies pour le matériel reconfigurable et les technologies pour le développement des processeurs généralistes.

Le projet est mené par un consortium de 22 intervenants industriels (Thales, la branche Allemande de Thomson, Lucent Technologies Network Systems, STMicroelectronics, Critical Blue, Alcatel-Lucent...) et académiques (universités de Karlsruhe, de Delft, de Bretagne Occidentale...). La durée du projet est de 45 mois et le budget est de 8,4 millions d'Euros.

8.2.2 Madeo et Platypus

Une approche pour réduire le coût de conception d'un circuit intégré consiste à réutiliser des portions déjà définies dans un autre cadre et à les interconnecter. Les portions réutilisées sont qualifiées d'IPs [1]. Les principales difficultés viennent d'une part de la validation et d'autre part de la constitution d'un cadre logiciel permettant l'exploitation du circuit (compilateur, système, ...).

L'emploi des technologies reconfigurables permet d'amortir les coûts non récurrents sur un volume de vente supérieur et s'impose de ce fait comme une solution attrayante. Dans ce

1. Intellectual Property

contexte, constituer un SoC reconfigurable (RSoC) consiste à y intégrer des IPs reconfigurables. L'hétérogénéité des IPs reconfigurables accroît le spectre des applications cibles sans compromis sur la taille du circuit, mais au coût d'une conception complexifiée du logiciel.

En particulier, chaque vendeur d'IP possède son propre environnement de programmation, et une pleine exploitation des ressources passe par une gestion système à même de partitionner finement les tâches, pour favoriser une allocation efficace des ressources matérielles.

La structuration de la mémoire est également critique, avec une efficacité maximale atteinte sur des mémoires locales aux IPs.

S'il n'apparaît pas réaliste de proposer un flot de conception applicatif capable d'adresser toute IP, il est en revanche possible d'offrir un support d'expression décorrélé de ces dernières et sachant interfacer leurs outils constructeurs. Cela favorise la mise en place d'une vision de niveau système, avec à la clé une plus grande latitude d'implémentation des traitements (concurrence, relocalisation de tâches, ...) et potentiellement des outils de déverminage.

8.2.2.1 Intégration au sein d'une chaine de traitements

Comme l'illustre la figure 8.5, le développement d'une chaîne de programmation pour des architectures reconfigurables et hétérogènes requiert la mise en œuvre d'outils communiquants à deux niveaux :

1. un niveau abstrait, ou couche haute, permettant, indépendamment de toute cible matérielle, de modéliser une représentation algorithmique des traitements avec le flot de contrôle, le flot de données ainsi que la structure du programme source ;
2. un niveau d'implantation ou couche basse, qui assure la configuration d'une cible matérielle et le transfert des données à partir d'une représentation algorithmique des traitements.

La couche haute est constituée d'une chaîne logicielle : chaque outil de la chaîne intervient spécifiquement sur la définition de la représentation algorithmique des traitements. Les traitements peuvent être par exemple, des filtres numériques ou des algorithmes de traitement d'image. Cette chaîne logicielle est vue comme un ensemble de processus interopérables qui interviennent tour à tour sur une définition de représentation algorithmique des traitements.

Chaque outil de la couche basse a pour rôle la traduction d'une représentation algorithmique des traitements vers une cible matérielle particulière.

Entre la couche haute et la couche basse s'intercalent des outils de niveau intermédiaire spécifiquement dédiés à la maintenance des définitions de traitements et à l'aide au développement. Ces outils permettent notamment, la vérification sémantique des architectures logiques, leur visualisation ou encore, l'import ou l'export depuis ou vers d'autres plateformes.

Dans notre chaîne de synthèse d'application, l'articulation entre d'une part, les différents outils de la couche haute entre eux, et d'autre part, l'articulation de la couche haute vers la couche basse et vers la couche intermédiaire est considérée comme un problème d'échange de données dont la spécification structurelle et sémantique de la représentation algorithmique des traitements est l'élément pivot : quel que soit leur niveau, la mise en œuvre des différents outils s'appuie sur une spécification normalisée et consensuelle. Un mécanisme standard de production automatique des interfaces de lecture et d'écriture des représentations des traitements assure au système sa portabilité et son évolutivité.

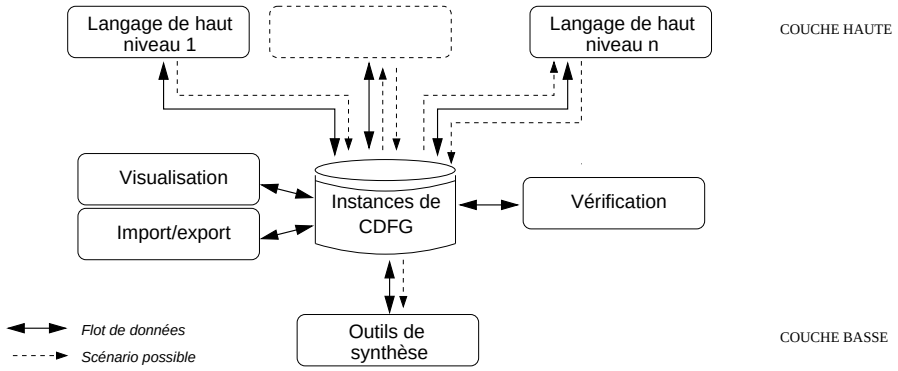

FIGURE 8.5 – Echanges de données entre les différentes couches

8.2.2.2 Intégration de Platypus

Madeo est un atelier pour la mise au point et la validation de traitements pour des architecture reconfigurable [Lag08]. Madeo a été développé par Loïc Lagadec sous VisualWorks, le système Smalltalk de Cincom [Vis].

L'approche est descendante et à haut niveau d'abstraction, par composition d'éléments d'architecture. Les architectures sont validées par simulation. En effet, à l'aide d'un langage spécifique au domaine, des traitements sont directement spécifiés et exécutés dans Madeo. Après validation, une cible matérielle (comme par exemple un FPGA) peut être exploitée pour produire un circuit par synthèse du traitement.

Pour permettre la synthèse des traitements et la production de circuits, une spécification précise du langage domaine utilisé pour représenter les traitements est indispensable. Ce langage a tout d'abord été élaboré par la mise en œuvre de Madeo : cet atelier est construit autour du méta-modèle du langage spécifié sous Smalltalk. Le langage spécifique étant partagé entre Madeo et plusieurs chaines d'outils (couche haute et couche basse), une spécification fine des types des éléments du langage spécifiques au domaine c'est avéré nécessaire pour l'intégration entre les différentes chaines d'outils et le développement des outils de vérification et de déverminage propres à Madeo. A titre d'exemple, l'interface utilisateur d'un outil de navigation dans la définition d'un traitement est montré dans la figure 8.6. Cet outil présente deux points de vue complémentaires : la vue du haut présente la hiérarchie des éléments de traitement et la vue du bas présente précisément toutes les caractéristiques des nœuds du graphe représentant le traitement.

8.2.3 Les travaux effectués

Nous sommes intervenus à deux niveaux : (1) pour la spécification et la mise en œuvre du langage spécifique et des outils afférents et (2) pour l'intégration des chaines d'outils entre les différents acteurs du projet Morpheus.

8.2.3.1 Développement du langage spécifique pour les traitements

Du point de vue de Platypus, Madeo est un atelier spécialisé dont la mise en œuvre s'articule autour d'un langage spécifique au domaine. Le cycle de développement de Madeo s'est avéré conforme à la méthode Platypus avec un travail d'élaboration préliminaire du

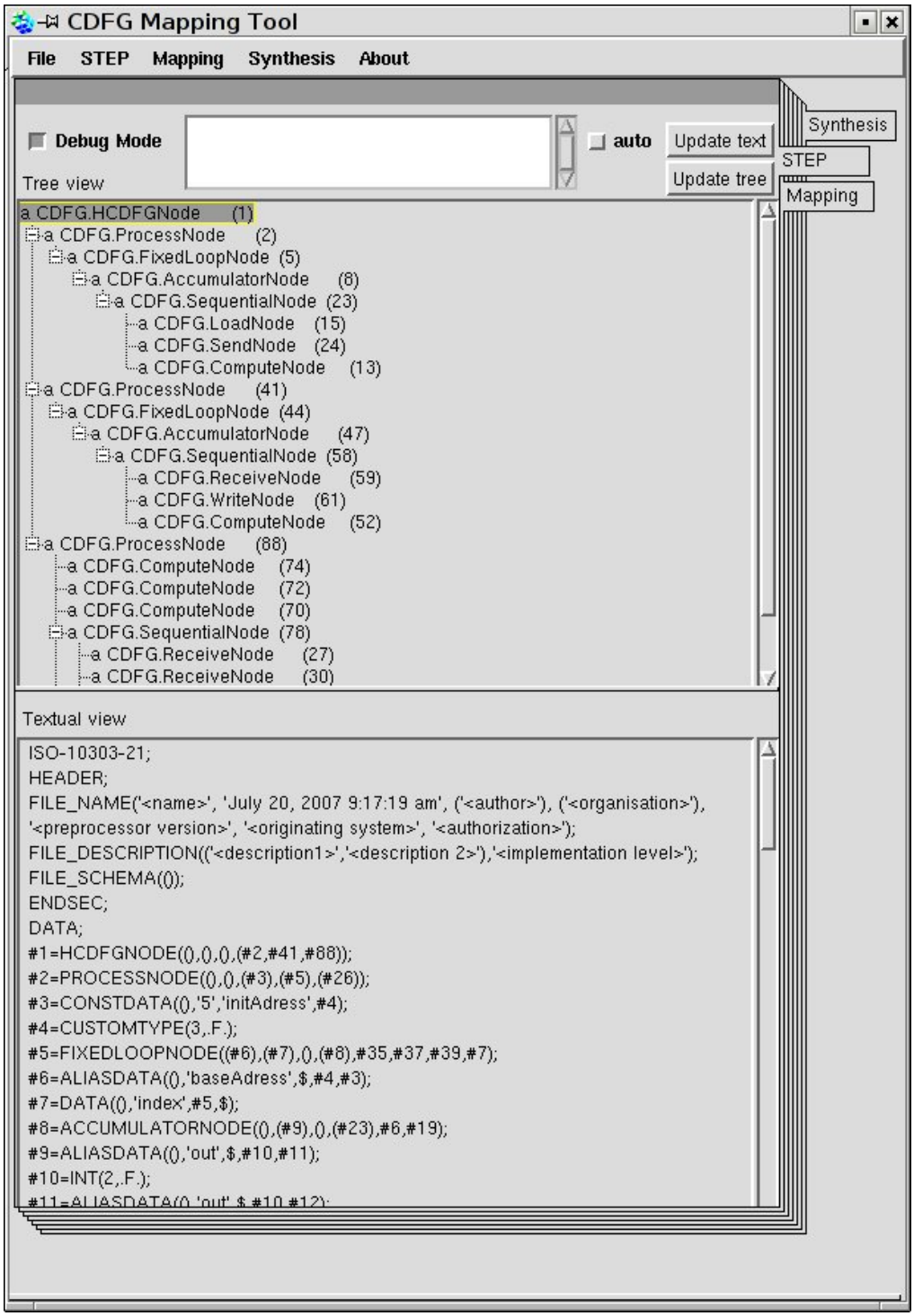

FIGURE 8.6 – Un outil de navigation dans une structure de traitement

langage spécifique et une seconde étape de typage et d'intégration des chaines d'outils. L'utilisation du méta-atelier Platypus a concerné cette seconde étape. Nous avons tout d'abord

méta-modélisé en EXPRESS le langage spécifique et intégré les composants principaux de Platypus dans Madeo.

L'intégration de Platypus a permis les développements basés sur les types du méta-modèle des traitements. Ainsi, au sein de Madeo, la définition même du langage spécifique est utilisée pour méta-programmer des vérifications, des visiteurs spécifiques permettant le parcours des graphes représentant les traitements et des outils de synthèse vers des cibles matérielles particulières.

8.2.3.2 Intégration par les données dans Morpheus

Pour le passage à l'échelle, afin de valider globalement l'approche Morpheus pour la production de circuits, une vision d'ensemble est nécessaire et l'intégration entre les différents partenaires de Morpheus est indispensable. Pour ce problème, la définition du langage spécifique joue un rôle de pivot entre les différents acteurs (voir figure 8.7).

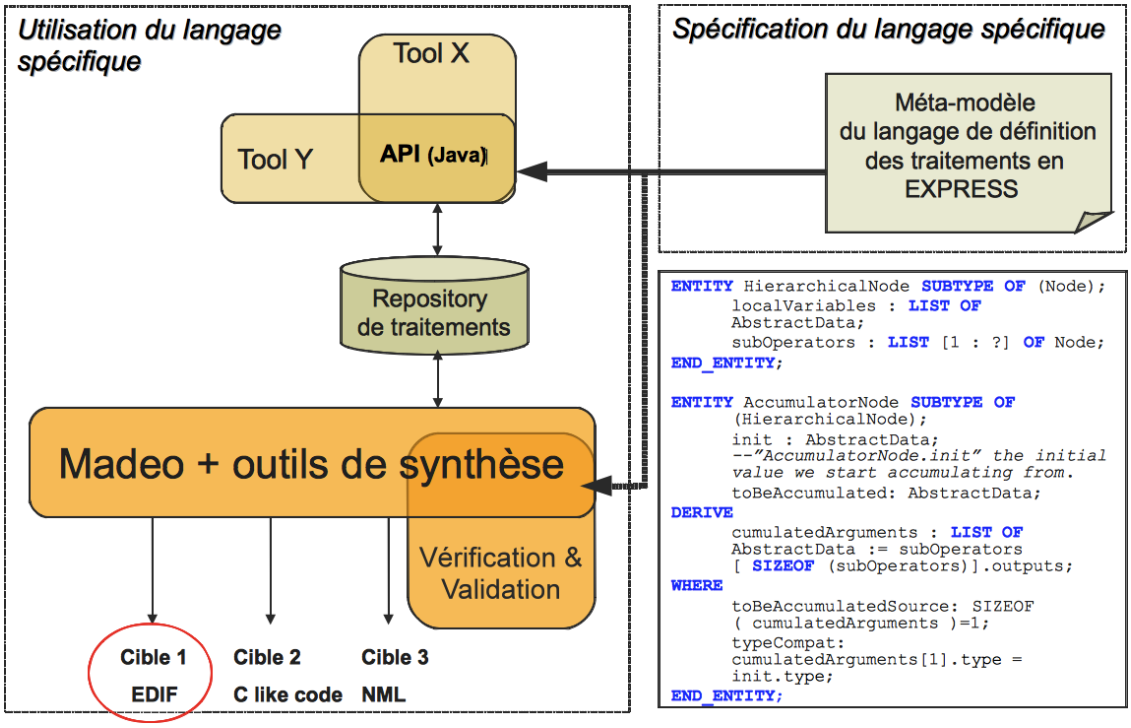

FIGURE 8.7 – Intégration entre les différentes chaines d'outils de Morpheus

Des traitements spécifiés à l'aide du langage spécifiques doivent pouvoir être produits et réutilisés entre les différents intervenants de Morpheus. Notamment, les outils de la couche haute sont développés par des partenaires du projet à l'aide d'environnements basés sur Java. Pour assurer l'interopérabilité entre les différentes chaines d'outils, notre travail a consisté à produire les composants logiciels Java nécessaires au partage des définitions de traitement. Ainsi, nous avons développé par spécialisation de Platypus, un générateur de composants Java pour l'échange et le partage de définition de traitements.

8.2.4 Bilan

L'utilisation de Platypus dans Morpheus a permis de confirmer les acquis permis par le projet PREMECS concernant le processus d'élaboration et de production. La définition d'un langage s'effectue tout d'abord par élaboration de son méta-modèle à haut niveau d'abstraction et par la manipulation de modèles conformes au travers d'outils. La définition précise des types des éléments d'un langage peut être effectuée dans un second temps, lorsque l'expérimentation démontre un certain niveau de maturité du méta-modèle. Mon intervention dans le projet a tout d'abord consisté en la spécification du méta-modèle du langage de définition des traitements, en coopération avec l'expert du domaine, Loïc Lagadec. Je suis ensuite intervenu en support concernant l'utilisation de Platypus pour la méta-programmation des tests des traitements et des différents outils basés sur la définition du langage spécifique.

Le projet Morpheus comprend des acteurs agissant à différents niveaux des traitements et de plus, les sites sont géographiquement répartis dans toute l'Europe. L'intégration entre ces différents intervenants est un facteur déterminant pour la réussite d'un tel projet. La définition du domaine effectuée dans Platypus et la génération automatique des composants logiciels permettant très concrètement l'intégration par partage et échange des définitions de traitements ont été des facteurs très favorables au projet. Mon intervention a consisté à mettre en œuvre les transformations de modèle avec Platypus pour produire les composants Java permettant l'intégration par les données entre Madeo et les chaines d'outils de la couche haute.

Par rapport à l'intégration entre les différents acteurs du projet, un problème important est la convergence vers un consensus pour la représentation commune des traitements. Du point de vue de Platypus, plusieurs systèmes cibles sont intégrés avec l'atelier Madeo. Le retour des définitions de traitements, des systèmes cibles vers Madeo est un facteur déterminant pour permettre la correction du méta-modèle. Dès la première étape de transition, nous avons en effet pu bénéficier des retours pour détecter plusieurs problèmes liés à la définition même du langage. Ce point confirme donc l'importance du retour des systèmes cibles vers l'atelier. Par contre, il ne nous a pas été possible de réitérer et de prendre en compte les corrections des anomalies par une seconde étape de transition. La répartition géographique des différents intervenants, la taille du système, son hétérogénéité et la contrainte temps sont clairement des freins à la mise en place effective d'un cycle itératif et agile et renforce notre position sur l'importance de la qualité des vérifications et des validations effectuées avant l'étape de transition.

8.3 Depuis 2005 Cheddar

Depuis fin 2005, Platypus est utilisé pour la mise en œuvre de l'atelier *Cheddar*. Une première version de cet atelier a été développée en Ada par Frank Singhoff. Il permet la vérification des propriétés temporelles des architectures temps-réel par l'analyse et la simulation de l'ordonnancement. Concernant l'utilisation de Platypus, il s'agissait tout d'abord de disposer d'un point de vue simplifié de l'architecture logicielle de Cheddar afin d'en faciliter la ré-ingénierie. Les méta-modèles de Cheddar ont été directement modélisés en EXPRESS. En effet, le travail d'élaboration était déjà effectués et les méta-modèles Ada considérés comme suffisamment matures. Par transformation de modèle, des versions successives améliorées de Cheddar ont été générées [PS06]. Ensuite, le langage de Cheddar pour la programmation d'ordonnanceurs à été intégré dans Platypus pour permettre la réification automatique des

ordonnanceurs spécifiés par un utilisateur de Cheddar [SP07, SPDL09].

Depuis 2010, nous travaillons en collaboration avec la société Ellidiss Technologies, sur l'utilisation de patrons de conception métier, pour la validation des architectures temps-réel. La méthode de validation procède par utilisation de la théorie de l'ordonnancement. Cette théorie définit un ensemble de tests de faisabilités qui permettent de garantir formellement l'ordonnançabilité des architectures temps-réel. Ces tests de faisabilités ne sont malheureusement pas applicables dans tous les cas. Des hypothèses d'applicabilité doivent en effet être vérifiées pour déterminer quels sont les tests de faisabilité qui peuvent être appliqués. Nous utilisons Platypus pour exprimer les hypothèses d'applicabilité sous la forme de règles qui contraignent les méta-modèles de Cheddar [PSDL10]. Pendant sa thèse, Vincent Gaudel[2] a développé un prototype d'outil d'aide à la décision à partir des spécifications dans Platypus. Le prototype est développé manuellement en Ada mais l'idée est de le générer depuis Platypus pour permettre automatiquement la prise en compte de nouvelles contraintes et de nouveaux patrons de conception [GSP+11].

8.3.1 Le projet Cheddar

La théorie de l'ordonnancement fournit des méthodes algébriques et algorithmiques permettant de vérifier le comportement temporel d'une application temps réel. Les bases de cette théorie ont été proposées par Lui et Layland dans les années 1970 [LL73a] et ont depuis, fait l'objet de très nombreuses recherches et expérimentations [KRP+94, GRS96, CDKM02]. Dans de très nombreux cas toutefois, la théorie de l'ordonnancement temps réel reste inappliquée par le monde industriel, même si souvent, le monde industriel montre un intérêt certain pour ce type de méthodes. Plusieurs hypothèses peuvent être avancées pour expliquer cet état de fait : l'inadéquation de cette approche dans certains contextes tels que les systèmes réparties où peu de résultats théoriques ont été proposés, l'absence d'outil flexible et ouvert, le faible couplage avec les méthodes de conception ou les processus d'ingénierie, ce qui requiert de la part des ingénieurs une maîtrise significative de cette théorie pour pouvoir l'employer, ...

Le projet Cheddar se propose d'évaluer l'applicabilité de la théorie de l'ordonnancement temps réel en étudiant certaines de ces hypothèses. De nombreuses applications industrielles ne peuvent pas être analysées par les méthodes analytiques de la théorie de l'ordonnancement : la théorie de l'ordonnancement temps réel ne fournit pas nécessairement de tests de faisabilité pour les ordonnanceurs et les modèles de tâches employés par le monde industriel. D'autre part, élaborer ces tests de faisabilité est une tâche coûteuse et généralement difficile. Les systèmes à analyser dans ce contexte sont généralement de grande taille et il peut-être difficile d'employer des méthodes d'analyse statique.

La simulation demeure une solution utilisée dans l'industrie. Il existe de nombreux modèles à événements discrets qui offrent des outils de simulation permettant de réaliser ces études de performances. C'est notamment le cas des Réseaux de Pétri [Pet81] et d'outils tels que CPN Tools [Wel06]. Une autre solution consiste à développer des outils de simulations ad-hoc. Cette solution est coûteuse et n'offre, en général, qu'un faible niveau de réutilisation du logiciel.

L'atelier Cheddar propose une alternative par la mise à disposition d'un langage spécifique ainsi que des outils associés (interpréteur, compilateur, ...). Ce langage permet de modéliser un ordonnanceur temps réel et d'en étudier le comportement par simulation. Nous proposons un processus d'utilisation associé à ce langage afin que l'utilisateur puisse aisément concevoir et

2. thèse financée par la région Bretagne et par Ellidiss Technologies

tester son ordonnanceur spécifique. Par la suite, grâce à l'utilisation de Platypus, l'utilisateur peut générer son outil de simulation, qui, après compilation, est capable de conduire des simulations sur des modèles de taille importante.

8.3.2 L'atelier Cheddar

Cheddar est un canevas constitué d'un ensemble de paquetages Ada et dont l'objectif est d'offrir à l'utilisateur des outils pour l'analyse de performance d'applications temps réel [SLNM04a]. Ce canevas implante les outils analytiques et les algorithmes d'ordonnancement temps réel classiques. Les systèmes à analyser peuvent être décrit, soit dans un formalisme propriétaire, soit par une spécification AADL (Architecture Analysis and Design Language). AADL est un langage graphique et textuel normalisé par la SAE sous le standard AS-5506 [SAE04]. Ce langage permet la description de l'architecture matérielle et logicielle d'un système temps réel embarqué.

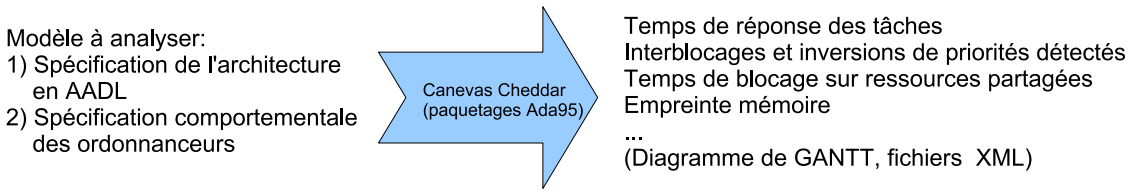

FIGURE 8.8 – Fonctionnement général de l'atelier Cheddar

La figure 8.8 décrit succinctement l'utilisation de Cheddar lorsque l'on souhaite effectuer une analyse de performance :

1. Il est d'abord nécessaire de modéliser l'architecture que l'on souhaite analyser. Cette spécification peut être exprimée en AADL. Il s'agit alors de décrire le système en termes de composants tels que des processeurs, tâches, ou ressources partagées. Chaque composant est décoré de propriétés décrivant son comportement logique et temporel. Ainsi, il est possible de spécifier la période d'une tâche, sa capacité, le type d'ordonnanceur associé à un processeur donné. Bien qu'il existe des propriétés standardisées par AADL, Cheddar propose un certain nombre d'extensions autorisant l'application des méthodes d'analyse proposées par la théorie de l'ordonnancement temps réel [CDKM02]. Cheddar offre un éditeur simplifié permettant de décrire l'architecture à analyser. Néanmoins, la démarche naturelle consiste à utiliser un outil de modélisation pour l'édition d'un tel modèle. A ce jour, des possibilités d'interopérabilité entre Cheddar et des outils de modélisation tels que STOOD existent.
2. L'utilisateur peut alors utiliser deux types de services : la simulation ou le calcul de tests de faisabilités. Le choix entre l'un, l'autre ou les deux types d'outils dépend des caractéristiques du modèle à analyser. Après analyse, les résultats peuvent être soit affichés dans l'éditeur de Cheddar, soit exportés vers d'autres outils via des fichiers XML.

D'autre part, l'atelier Cheddar offre un langage spécifique ainsi qu'un ensemble d'outils associés pour la modélisation et la mise en œuvre d'algorithmes d'ordonnancement temps réel. L'utilisation de ce langage permet à l'utilisateur de définir un algorithme et de le tester

rapidement grâce à l'environnement Cheddar, avant d'exploiter son modèle pour effectuer une analyse de performances.

Il existe donc plusieurs niveaux d'utilisation de l'atelier Cheddar :

- Lorsque le modèle AADL à analyser comporte des algorithmes d'ordonnancement classiques et déjà implantés dans le canevas, l'utilisateur exploite l'interface homme-machine pour éditer son modèle et en effectuer une analyse de performances. Cette alternative ne demande pas d'expertise particulière de la part de l'utilisateur concernant l'utilisation de Cheddar.
- Si l'algorithme n'existe pas dans le canevas Cheddar, mais qu'il peut être modélisé par le langage spécifique de Cheddar, l'utilisateur doit décrire son algorithme d'ordonnancement en plus du modèle AADL à analyser. L'utilisateur doit alors maitriser le langage de modélisation d'ordonnanceurs temps réel.
- Enfin, si l'algorithme et/ou le système à analyser sont trop spécifiques, l'utilisateur doit modifier le canevas Cheddar afin d'implanter ses algorithmes d'ordonnancement temps réel. Cette dernière alternative est la plus coûteuse pour l'utilisateur car il lui est alors nécessaire de comprendre et modifier l'architecture et la mise en œuvre du canevas. L'existence du langage spécifique à pour objet d'éviter, tant que faire se peut, la modification directe du canevas par l'utilisateur afin que l'utilisation du canevas reste la plus simple possible.

8.3.3 Un langage pour la modélisation d'ordonnanceurs temps réel

Le langage spécifique de Cheddar doit permettre de modéliser le comportement d'un ordonnanceur temps réel. Modéliser le comportement d'un ordonnanceur temps réel consiste à modéliser :

1. Les opérations arithmétiques et logiques de l'ordonnanceur, qui décrivent, par exemple, comment calculer la priorité des tâches. Il s'agit essentiellement de traitement à effectuer sur les attributs de certaines entités tels que les tâches, les processeurs ou les ressources partagées du système à analyser.
2. Les contraintes temporelles et les synchronisations entre des entités telles que les tâches et les ordonnanceurs de l'architecture à analyser. Il s'agit ici d'exprimer comment et quand ces entités doivent collaborer afin de partager les différentes ressources.

Le langage Cheddar est donc défini par un sous ensemble d'Ada pour la partie algorithmique et par un modèle d'automates temporisés pour les contraintes temporelles et de synchronisation.

8.3.4 Processus d'utilisation d'un programme Cheddar

Si l'architecture à analyser comporte des ordonnanceurs non encore implantés dans l'atelier Cheddar, l'utilisateur peut utiliser le processus décrit par la figure 8.9 pour la mise en œuvre de ses simulations :

1. L'utilisateur doit d'abord décrire avec le langage de Cheddar les ordonnanceurs à implanter. Il lui est alors possible de mettre au point ses modèles d'ordonnanceur grâce à un interpréteur implanté dans l'atelier Cheddar et qui lui permet de tester rapidement le comportement de son programme par simulations.

2. Une fois son ordonnanceur au point, grâce à Platypus, il peut transformer son programme Cheddar en un ensemble de paquetages Ada pouvant être automatiquement intégré au canevas. Dans la phase de transformation du modèle vers le code Ada, il est possible de spécifier des contraintes afin d'adapter le code généré aux futures simulations (ex : empreinte mémoire et temps de réponse des simulations selon les caractéristiques de l'architecture à analyser).
3. Les paquetages Ada générés peuvent enfin être compilés et intégrés dans Cheddar afin de produire une nouvelle version du canevas. Par la suite, l'utilisateur peut exécuter des simulations de grande taille comme si son ordonnanceur avait été implanté manuellement dans Cheddar.

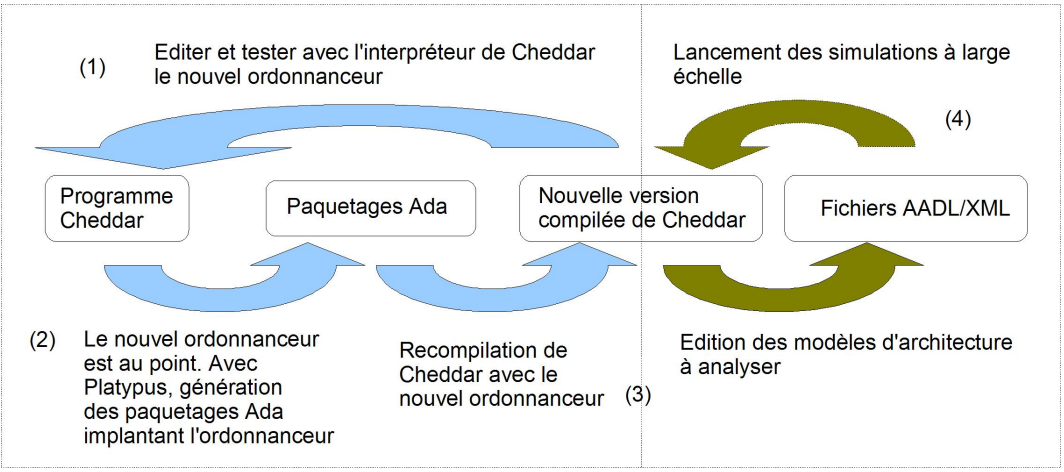

FIGURE 8.9 – Processus de simulation d'un ordonnanceur avec Cheddar

8.3.5 Les composants architecturaux de Cheddar

Comme le montre la figure 8.10, Cheddar est constitué de six composants principaux. Ces composants se répartissent classiquement en deux couches logicielles :

- La couche basse, pour la gestion des données ; elle est directement dépendante des types de données manipulées et des contraintes associées en termes de règles structurelles et sémantiques ; cette couche de nature générique est fortement réutilisable et évolutive.
- La couche haute est liée au domaine sous-jacent de la simulation de systèmes temps réel ; bien que cette couche soit de nature très spécifique, elle est cependant évolutive de part l'utilisation d'un procédé de mise en œuvre dirigé par les modèles qui se concrétise par l'utilisation d'un générateur de code spécifique.

8.3.5.1 La couche basse

Cheddar comprend un dépôt centralisé pour le stockage des données et des méta-données ; les tâches et les processeurs sont des exemples de données, la définition d'une transition d'un automate ou une instruction sont elles des méta-données. Il s'agit d'un constituant abstrait

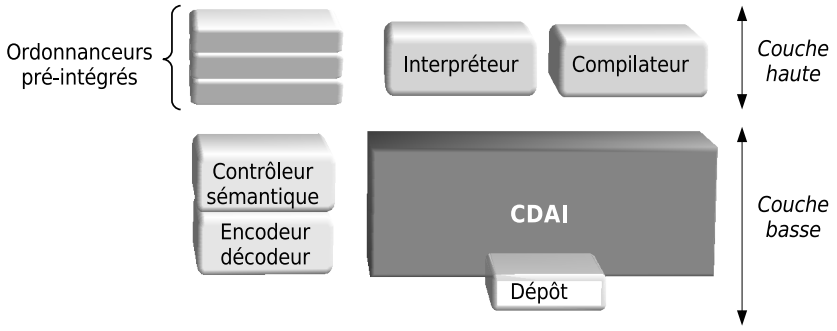

FIGURE 8.10 – Architecture logicielle à deux couches de Cheddar

qui représente un composant physique de stockage de données. Concrètement, il peut s'agir de la mémoire ou d'une base de données.

Le dépôt est encapsulé dans un composant d'accès aux données (appelé *CDAI* pour ≪*Cheddar Data Acces Interface*≫) de sorte que tous les accès en lecture ou écriture de données ou méta-données s'effectuent au travers de ce composant. Le composant d'accès aux données est l'élément architectural central autour duquel s'articulent tous les autres composants de Cheddar.

La couche basse met en œuvre des composants additionnels, spécifiquement liés à la définition des données manipulées au travers de la CDAI :

- Le contrôleur sémantique permet la validation des données et des méta-données ;
- Le composant d'encodage et de décodage assure à Cheddar une interopérabilité par échange de données ; ce composant met en œuvre un protocole standard pour l'encodage et le décodage des données et des méta-données vers *XML*.

8.3.5.2 La couche haute

La couche haute permet la simulation de systèmes temps réel à deux niveaux :

1. Cheddar met en œuvre des algorithmes d'ordonnancement reconnus comme classiques par la communauté. Ces algorithmes d'ordonnancement peuvent être directement utilisés pour un modèle de tâches définit par l'utilisateur. Ces ordonnanceurs sont pré-intégrés sous la forme de paquetages dédiés (un par ordonnanceur) : un ordonnanceur est représenté par des classes pour l'automate et les algorithmes d'ordonnancement
2. Des ordonnanceurs spécifiques et non intégrés peuvent être programmés à l'aide du langage de Cheddar. Ces programmes sont analysés par le compilateur qui, à partir d'un programme, en produit une représentation interne stockée dans le dépôt via la CDAI. L'interpréteur peut alors exploiter la représentation interne pour la simulation proprement dite.

8.3.6 Ingénierie de Cheddar avec Platypus

Cheddar a tout d'abord été développé manuellement par Frank Singhoff. Étant donné la complexité du domaine, après plusieurs versions, il n'est pas étonnant que son évolution

soit devenue coûteuse et de plus en plus difficile. Notamment, il est devenu difficile d'appréhender le rôle des éléments des méta-modèles mis en œuvre. Les différents aspects et leurs interdépendances étaient de plus en plus difficiles à comprendre et à maintenir.

Pour l'utilisation de Platypus, l'idée première est de clarifier et de vérifier la structure des méta-modèles par un processus de rétro-ingénierie. Nous avons donc spécifié les méta-modèles de Cheddar en EXPRESS dans Platypus. La possibilité de visualiser la structure nous a permis de corriger les méta-modèles de Cheddar pour faciliter l'introduction des automates et la spécification du langage spécifique de Cheddar.

Nous avons ensuite spécialisé Platypus pour intégrer les concepts liés au langage Ada et au style de programmation de Cheddar. Des générateurs ont été implantés pour automatiser la transition entre Platypus et la mise en œuvre Ada. Les deux couches logicielles présentées précédemment sont ainsi en partie automatiquement produites par un procédé dirigé par les modèles. Le procédé est illustré par la figure 8.11.

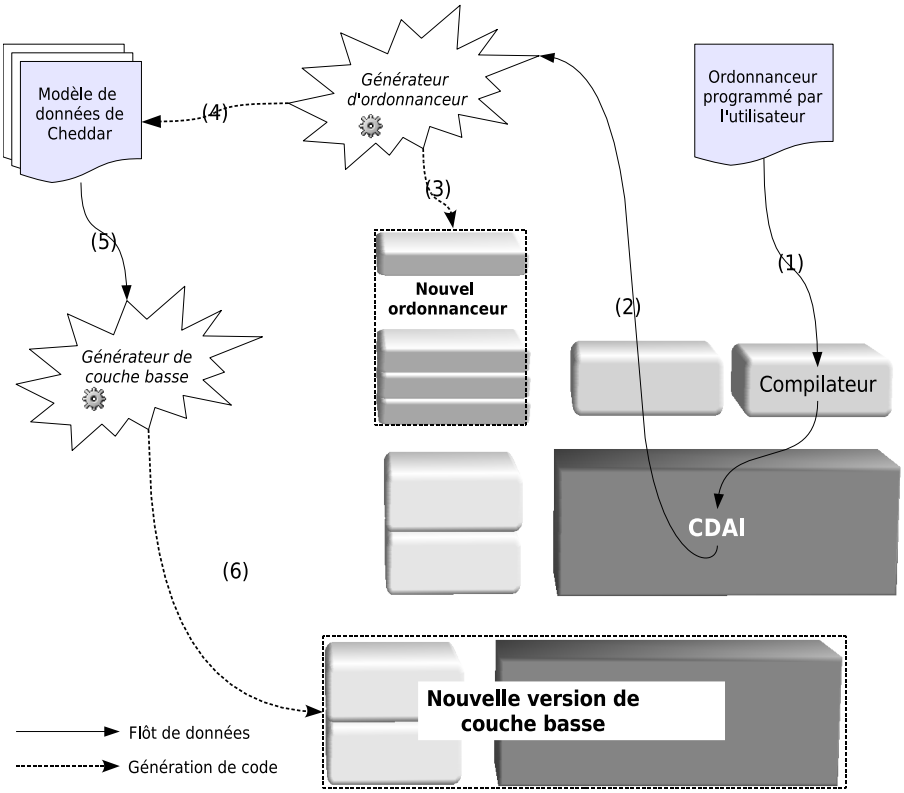

FIGURE 8.11 – Evolution incrémentale de Cheddar

8.3.6.1 Mise en œuvre de la couche basse

Pour la couche basse, les objets manipulés, comprenant les données et les méta-données, sont spécifiés par le modèle de données de Cheddar. Ce modèle consiste en un ensemble de schémas EXPRESS. Il intègre la définition structurelle des hiérarchies et la définition des contraintes des entités du système. A une version particulière de ce modèle correspond une version de la couche basse puisque tous les constituants de cette couche sont automatiquement

produits à partir de ce modèle par *le générateur de couche basse* qui produit les constituants Ada de cette couche.

8.3.6.2 Mise en œuvre de la couche haute

Lorsqu'aucun ordonnanceur intégré n'est satisfaisant pour une simulation particulière, l'utilisateur peut programmer un ordonnanceur spécifique avec le langage de Cheddar (voir le chapitre 8.3.4). Un programme Cheddar est analysé par le compilateur qui produit une représentation interne équivalente au programme sous la forme d'instances de la syntaxe abstraite du langage de Cheddar.

Pour un programme donné, *le générateur d'ordonnanceur* permet de traduire la représentation interne du programme en un ordonnanceur sous la forme d'un paquetage spécifique intégré à Cheddar. Cette traduction met en jeux deux processus de génération. Tout d'abord, le code de l'ordonnanceur, comprenant notamment la mise en œuvre de l'automate et les calculs associés aux états, constitue l'ordonnanceur proprement dit. Ce paquetage est produit, dans la couche haute, sous la forme d'un ordonnanceur intégré, par le générateur d'ordonnanceur. En complément, le nouvel ordonnanceur doit être déclaré dans le modèle de données de Cheddar ce qui implique la génération d'un schéma complémentaire. Le processus de production automatique (voir chapitre 8.3.6.1) est alors exploité pour la mise à jour du code de la couche basse. Après compilation de l'ensemble, le nouvel ordonnanceur fait partie intégrante de Cheddar et il peut être directement exécuté.

L'intégration d'un ordonnanceur spécifique représente un incrément. A chaque incrément, une nouvelle version de Cheddar spécifiquement adaptée au contexte d'utilisation est ainsi produite.

8.3.7 Aide automatisée pour la sélection de tests de faisabilité

Un système temps réel est dit critique si son dysfonctionnement peut avoir pour conséquence d'entraîner des dégâts sur des personnes ou pour l'environnement. La validation d'un tel système est donc très importante. La théorie de l'ordonnancement fournit des outils pour la validation des architectures temps réel avec notamment des méthodes analytiques appelées *tests de faisabilité*. Cependant, leur utilisation implique une expertise approfondie : en effet, les tests de faisabilité ne sont utilisables que dans certaines conditions bien précises. Suivant le critère de performance évalué et suivant les caractéristiques de l'architecture, un tel test peut ne pas être applicable. Ainsi, déterminer quels tests de faisabilité sont applicables est une tâche complexe et coûteuse. En définitive, on observe que la théorie de l'ordonnancement est peu utilisée.

Nous proposons une caractérisation des architectures par le biais de patrons de conception qui permettent d'automatiser la sélection des tests de faisabilité. Nous avons défini cinq patrons basés sur les protocoles de communication et de synchronisation entre les tâches. Les patrons de conception sont modélisés par des ensembles de conditions sur les propriétés des modèles d'architectures. Nous analysons les modèles d'architecture pour vérifier le respect de ces conditions : on vérifie que les modèles sont conformes aux patrons. Dans le cas où la conformité à un patron de conception est confirmée, nous sommes capables de déterminer une liste de tests de faisabilité applicables. Cette liste est dépendante du patron de conception utilisé ainsi que de contraintes sur les propriétés de l'architecture.

Nous avons réalisé un prototype capable de sélectionner les tests de faisabilité applicables à un modèle d'architecture AADL conforme à un patron de conception. Ce prototype a été

conçu comme une extension de l'atelier Cheddar.

Ce travail est effectué par Vincent Gaudel en thèse à l'UBO. La thèse est encadrée par Frank Singhoff et moi même. Elle est financée par le Conseil Régional de Bretagne et par la société Ellidiss Technologies.

8.3.7.1 Le problème de la vérification des systèmes critiques

La théorie de l'ordonnancement temps réel permet au concepteur d'un système d'analyser le comportement temporel d'un ensemble de tâches avec des méthodes algébriques appelées tests de faisabilités. Par exemple, Liu et Layland [LL73b] définissent des tâches réalisant périodiquement un traitement par trois paramètres : le délai critique (D_i), la période d'activation (P_i), la capacité (C_i). à chaque fois qu'un tâche i est activée, elle doit réaliser un travail dont le temps d'exécution est borné par C_i. Ce travail doit être complété avant D_i unités de temps après l'instant d'activation de la tâche. De ces paramètres ont été conçu différents types de tests de faisabilité : des tests basés sur le facteur d'utilisation processeur, sur le temps de réponse pire cas, etc [LL73b]. Par exemple, Liu *et al.* proposent un moyen de calculer le facteur d'utilisation processeur par :

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$$

La somme des rapports de la capacité des tâches sur leur période doit être inférieure ou égale à 1. Dans le cas où cette somme est supérieure à 1, la charge de calcul est trop importante vis-à-vis de la capacité du processeur. Le système ne peut donc pas être ordonnancé de façon à respecter les contraintes temporelles des tâches.

Cependant, ce test n'est applicable que dans un contexte bien précis : les tâches doivent être périodiques et indépendantes et le protocole d'ordonnancement doit être *Earliest Deadline First* préemptif.

L'application d'un test de faisabilité à un système requiert donc que ce dernier possède des propriétés architecturales données. Nous nommerons ces contraintes *"contraintes d'applicabilité"* dans la suite de ce rapport. Elles caractérisent les propriétés des entités inhérentes aux systèmes temps réel comme la périodicité des tâches, le protocole d'ordonnancement utilisé, le protocole de communication, ...

8.3.7.2 Définition des patrons de conception

Comme nous le précisions précédemment, la classification des différents systèmes analysables est réalisée par le biais de cinq patrons de conception identifiés par les noms suivants : *Synchronous data-flow*, *Ravenscar*, *Blackboard*, *Queued buffer* et *Unplugged*. Chaque patron de conception propose une solution architecturale à un problème de synchronisation entre les tâches en définissant un protocole de communication inter-tâches [DS08, PSDL10].

Synchronous data-flow est le patron de conception le plus simple. La communication entre les tâches s'effectue par zones de mémoire. Les tâches lisent ces zones à leur réveil, et écrivent des données sur ces mêmes zones lors de leur terminaison. Les réveils et terminaisons sont disjoints, il n'y a donc pas de besoin de synchronisation. Pour *Ravenscar*, les données sont partagées de façon asynchrone, sous le contrôle du protocole d'héritage de priorités. Ravenscar permet aux tâches de partager des données protégées par des sémaphores. Ces derniers peuvent être utilisés pour construire différents protocoles de synchronisation tels que les sections critiques, lecteurs-écrivains, producteurs-consommateurs, etc. *Blackboard* implante le

patron de conception lecteurs-écrivains : seule la dernière donnée produite est accessible aux tâches. *Queued buffer* implante le patron de conception producteurs-consommateurs. L'accès aux messages est géré par un protocole FIFO. Le dernier patron, *Unplugged*, modélise l'absence de communication entre les tâches.

Il existe bien sûr de nombreux autres paradigmes possibles de synchronisation qui puissent justifier une telle spécification. Dans un premier temps, nous estimons que ces cinq patrons sont suffisants pour valider notre approche [SPDL09].

8.3.7.3 La méthode du point de vue concepteur

La figure 8.12 décrit le cycle de conception de modèles d'architectures temps réel incluant l'outil de sélection de tests. Le concepteur propose un modèle d'architecture en AADL (1). L'outil de sélection analyse le modèle d'architecture et vérifie sa conformité à un patron conception (2) (les cinq patrons sont vérifiés un à un). L'outil propose alors une liste de tests de faisabilité spécifique au système (4), avant de les exécuter (6). Lors de chacune des étapes 2, 4 et 6, le concepteur est susceptible de revenir sur la phase de conception.

Dans un premier temps lors de la vérification de la conformité du modèle d'architecture à un patron : si le système n'est pas conforme à un patron, l'outil n'est donc pas en mesure de fournir une liste de tests de faisabilité, et propose à la place un ensemble de métriques pour assister le concepteur (3).

Dans un second temps, lors de l'étape (4) : si la liste de tests de faisabilité proposée ne le satisfait pas (besoin d'appliquer un test en particulier par exemple). L'outil propose alors un ensemble de tests faisabilité différents et leurs contraintes d'applicabilité non-respectées (5).

Enfin, après l'analyse d'ordonnançabilité : dans le cas où le système n'est pas ordonnançable ou si les tests faisabilité ne sont pas en mesure de valider l'ordonnancement (7).

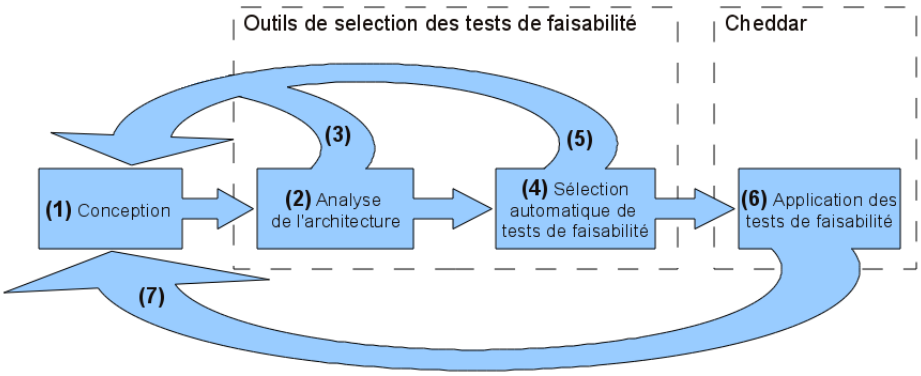

FIGURE 8.12 – Schéma des étapes de la méthode du point de vue concepteur.

8.3.7.4 Modélisation des patrons de conception

Avec pour objectif la génération automatique du code du prototype, nous proposons une modélisation des contraintes à évaluer pour déterminer les tests de faisabilité applicables aux modèles d'architectures. Dans cette partie, nous présentons les entités modélisées puis la façon dont nous en déduisons les tests de faisabilité applicables.

Déterminer les tests applicables. Nous avons défini, pour chacun des patrons, un nombre de cas d'application de tests de faisabilité (126 pour *Synchronous data-flow*). Chacun de ces cas d'application correspond à une liste de tests de faisabilité applicables.

Pour déterminer le cas d'application auquel correspond un modèle d'architecture, nous vérifions successivement les contraintes relatives aux environnements de déploiement, aux patrons de conception et aux cas d'application de tests de faisabilité.

Structure de la modélisation. Le méta modèle de Cheddar nous fournit tous les outils pour réaliser la modélisation de nos patrons. La figure 8.13 donne un extrait du méta modèle de Cheddar. Une tâche générique est définie par sa capacité, son échéance, son instant d'activation et sa priorité, entre autres. Une tâche périodique est définie comme une tâche générique pour laquelle on précise la période et le jitter.

```
1  SCHEMA Tasks;
2
3    ENTITY Generic_Task
4    ... Capacity  : Natural;
5        Deadline  : Natural;
6        Start_Time  : Natural;
7        Priority  : Priority_Range; ...
8    END_ENTITY; ...
9
10   ENTITY Periodic_Task SUBTYPE OF ( Generic_Task );
11       Period  : Natural_type;
12       Jitter  : Natural_type;
13   END_ENTITY; ...
14
15 END_SCHEMA;
```

FIGURE 8.13 – Extrait du méta modèle de Cheddar : la définition d'une tâche générique et d'une tâche périodique.

```
1  RULE all_tasks_are_periodic FOR ( generic_task );
2  WHERE
3   R1  : SIZEOF(QUERY( t <* generic_task | NOT ('TASKS.PERIODIC_TASK' IN TYPEOF (t)))) = 0;
4  END_RULE;
```

FIGURE 8.14 – Extrait de la spécification de *Synchronous data-flow* en EXPRESS

Pour chacune des entités présentées dans la partie 8.3.7.4, nous exprimons à l'aide du méta modèle de Cheddar l'ensemble de ses contraintes.

Dans le cas de Synchronous data-flow, par exemple, les contraintes sont les suivantes : (1) toutes les tâches sont périodiques ; (2) le système ne contient ni tampon, ni données partagées ; (3) le protocole de partage de données entre les tâches ne peut être que soit *immediate connexion*, soit *sampled connexion* soit *delayed connexion*[3] ; (4) il existe un unique

3. Protocoles de communication AADL

espace d'adresse par processeur. La modélisation de ces contraintes est réalisée en EXPRESS. La figure 8.14 contient le modèle de la contrainte numéro (1), extraite de la spécification de Synchronous data-flow en EXPRESS. Elle est exprimée sous la forme d'une règle EXPRESS vérifiant que parmi toutes les tâches, le nombre de tâches non périodiques est égal à zéro.

8.3.8 Bilan

L'atelier Cheddar est maintenant très intimement lié à Platypus et les évolutions de Cheddar procèdent de l'adaptation de Platypus et des méta-modèles de Cheddar dans Platypus.

Le premier impact positif de l'utilisation de Platypus, lié à l'activité de méta-modélisation et la ré-ingénierie, a été de documenter la conception de l'atelier Cheddar.

La génération des composants par méta-programmation dans Platypus a permis d'imposer un style de mise en œuvre. La conséquence est un gain important quant à l'homogénéité du code et à son intelligibilité.

Le processus de maintenance a aussi été grandement facilité grâce à la génération de code puisqu'une partie importante du code de l'atelier Cheddar est automatiquement produit. Actuellement, environ 30% du code de l'atelier est généré automatiquement. Il s'agit plus particulièrement des couches basses.

Le gain en termes de conceptualisation mais aussi en termes d'intelligibilité et d'homogénéité du code a été augmenté par la mise en œuvre du langage spécifique de Cheddar à l'aide de Platypus. Le framework de Cheddar a été revu pour permettre la production automatique des ordonnanceurs. Par contre, leur génération est restée à l'état de prototype et le passage à l'échelle n'est pas encore mesuré.

Platypus sert aussi d'atelier pour la spécification des contraintes pour l'identification automatisée des tests de faisabilité applicables. Grâce au prototype développé par Vincent Gaudel, les méta-modèles sont spécifiés et le plan de génération vers Ada pour l'implantation automatique est en place.

Nous sommes actuellement en mesure d'augmenter les impacts de Platypus quant à la vérification et la validation des architectures, les architectures et les ordonnanceurs pouvant être manipulés en amont dans Platypus.

Différents projets utilisent l'atelier Cheddar et bénéficient directement de l'effort de méta-modélisation. Les différents méta-modèles servent comme base documentaire à jour par rapport à la mise en œuvre. Ce point permet l'interopérabilité avec d'autres environnements et facilite les interactions entre nos différents projets et les utilisateurs de l'atelier Cheddar. Par exemple, Shuai Li, actuellement en thèse chez Thalès, utilise directement les méta-modèles de Cheddar pour la validation d'architectures temps-réel [LSRB12]. Le code des applications expérimentales est écrit en Java et nous avons utilisé notre générateur de paquetage Java à partir d'EXPRESS pour fournir à Thalès le composant permettant l'interopérabilité avec Cheddar.

8.4 Conclusion

Les différents projets pour lesquels Platypus a été exploité montrent les bénéfices de l'approche et l'intérêt pratique du méta-atelier Platypus. Il est intéressant de constater que l'intégration de Platypus s'est effectuée de trois façons lors de trois étapes différentes du processus :

– Platypus est pris en compte dès le départ du projet PREMECS ;

– pour Morpheus, Platypus a été intégré après méta-modélisation et construction d'une partie de l'atelier Madeo mais avant son utilisation par des acteurs de Morpheus externes à l'UBO ; de plus, les concepteurs des premières versions du méta-atelier Madeo n'avait pas de connaissance a priori du mécanisme de typage introduit par Platypus ;
– pour Cheddar, Platypus a été utilisé après que les premières versions fonctionnelles du système cible (l'atelier Cheddar développé en Ada) soient développées et en partie validées.

Pour les projets Premecs et Morpheus, des simulateurs fonctionnels et des outils d'édition et de vérification basés sur le langage spécifié dans Platypus ont pu être développés et l'intérêt du typage optionnel a été observé.

Pour Premecs, nous avons directement bénéficié d'une part, de l'agilité permise par Pharo et d'autre part du typage optionnel. L'expérimentation n'a pu être menée à son terme. Le système cible n'a pas été produit et exploité par IFREMER. Toutefois, l'infrastructure est en place et nous avons montré la faisabilité du projet de méta-atelier spécialisé pour le domaine des structures immergées.

Pour Morpheus, l'atelier Madeo a tout d'abord été développé indépendamment de Platypus. Platypus a pu ensuite être intégré dans le processus de développement. Des outils additionnels de vérification et de validation basés sur le typage ont été ajoutés. De plus, l'effet de rétro-influence a été très concrètement observé. Le retour des modèles conformes depuis les systèmes cibles (chaines de la couche haute de Morpheus) ont pu être instrumentés et exploités. Le passage à l'échelle a été démontré de part l'intégration opérationnelle obtenue entre Madeo et les chaines d'outils tiers (plus particulièrement les chaines de la couche haute).

En l'état actuel, concernant le projet Cheddar, le bilan est plus difficile à établir. Les gains obtenus au niveau de la mise en œuvre, de la documentation et de maintenabilité sont essentiellement dus à la mise en place d'un processus d'ingénierie par les modèles. La génération de code demeure le facteur de gain le plus important. Étant donné que l'atelier Cheddar était fonctionnel lorsque nous avons introduit l'utilisation de Platypus dans le processus de mise en œuvre, l'étape d'élaboration n'a pas pu être expérimentée. Par contre, toute l'infrastructure est en place et sera utilisée pour élaborer les évolutions de l'atelier Cheddar. En effet, la génération des ordonnanceurs demeure un sujet important pour Cheddar. L'automatisation de la validation précoce des ordonnanceurs spécifiés par les utilisateurs est un point déterminant pour le passage à l'échelle et la spécialisation de l'atelier Cheddar à des applications spécifiques.

Par ailleurs, l'approche peut-être améliorée, notamment en ce qui concerne le lien maintenu entre une application cible et le méta-atelier Platypus. Dans la suite de ce rapport nous donnons une conclusion générale et les prolongements que nous envisageons à nos travaux.

Chapitre 9

Conclusion et perspectives

Depuis mon intégration à l'UBO en 2000, mes travaux de recherche contribuent au domaine de la vérification et de la validation des langages dans le contexte de l'ingénierie dirigée par les modèles (IDM). L'IDM permet de spécifier des langages à haut niveau d'abstraction. Les méthodes liées à l'IDM procèdent par raffinement successifs de spécifications de haut niveau d'abstraction vers des réalisations de plus bas niveau d'abstraction. La prise en compte des contraintes de réalisation est effectuée par enrichissement des méta-modèles, par combinaison de différents méta-modèles recouvrant des aspects différents et par transformation de modèles. L'approche est mise en œuvre par des environnements qui exploitent la transformation de modèles et principalement la génération de code pour la réification des méta-modèles. Cette approche générative implique qu'une grande partie des vérifications et surtout des validations sont opérées au sein de la plateforme d'exécution cible. Le problème majeur est celui du coût du processus. De nombreuses publications attestent d'un intérêt de l'industrie pour l'IDM. Cependant, le coût élevé du processus est une des causes de la sous-utilisation de l'IDM dans l'industrie notamment en ce qui concerne la validation.

Nous proposons une méthode et le méta-atelier Platypus dont l'objectif est de maximiser les possibilités de vérification et de validation d'un langage avant la production d'un système cible par transformation de modèle. La méthode de spécification des méta-modèles est itérative et incrémentale. Les vérifications et les validations peuvent être progressivement ajoutées. Le typage des méta-modèles est optionnel, il peut être introduit en cours de processus de méta-modélisation pour renforcer les vérifications et les validations et pour implanter les règles de transformation de modèles.

Une fois construit, le système cible constitue aussi un élément important pour les vérifications et les validations. Les modèles conformes réalistes constituent des cas d'utilisation dont il est possible de tirer partie pour à nouveau renforcer l'adaptation des méta-modèles

aux besoins, au domaine et à l'application. Les systèmes cibles et le méta-atelier sont fortement intégrés et interopérables. Cette intégration permet l'échange de modèles conformes et l'exploitation par le concepteur du langage de retours d'informations concernant l'utilisation du langage.

Trois projets ont directement bénéficié de Platypus. Le travail de méta-modélisation et les résultats obtenus en termes de vérification et de validation montrent un réel bénéfice. Cependant, la méthode et le méta-atelier peuvent être améliorés. La première partie de ce chapitre décrit quelques perspectives d'utilisation et d'amélioration de Platypus.

Platypus est actuellement exploité dans le cadre du projet Cheddar pour la vérification et la validation des architectures temps réel. L'atelier Cheddar permet la vérification et la validation des contraintes temporelles d'une architecture temps réel soit par la simulation, soit par une validation basée sur la théorie de l'ordonnancement. Une partie significative de Cheddar est produite automatiquement par méta-modélisation dans Platypus. Le méta-atelier Platypus et Cheddar sont finement intégrés. Cette intégration ouvre des perspectives supplémentaires pour l'évolution de Cheddar, l'amélioration de sa mise en œuvre et de ses capacités fonctionnelles. La seconde partie de ce chapitre décrit brièvement ces perspectives.

9.1 Perspectives liées à la méthode et au méta-atelier

9.1.1 Évolution des méta-modèles

Le système cible demeure un élément fondamental de notre approche. Il constitue la référence en termes de validation pour l'utilisateur final. Le méta-atelier tire partie de l'interopérabilité établie avec le système cible pour intégrer les modèles réalistes indispensables à la validation et au passage à l'échelle. Cependant, en cours d'une itération, les modifications apportées aux méta-modèles sources au sein du méta-atelier peuvent altérer le lien avec le système cible et rendre impossible l'interopérabilité avec ce dernier du moins jusqu'à la prochaine étape de transition. Les modèles conformes pour une version N peuvent ne plus être conformes lorsqu'une nouvelle version est en cours d'élaboration au sein du méta-atelier. Il s'agit d'une limite importante de l'approche.

Une perspective de nos travaux consiste à prendre en considération les modifications des méta-modèles sources opérées dans le méta-atelier pour maintenir l'interopérabilité avec le système cible. L'infrastructure offerte par Pharo permet déjà de tracer tous les changements effectués sur une version. Ces traces sont représentées par des objets. Une trace est exécutable de sorte qu'il est possible de la rejouer. En cas d'arrêt non prévu ou forcée par l'utilisateur, il est ainsi possible de requérir les actions effectuées depuis la dernière version cohérente du système et de les exécuter afin de retrouver le système dans son état avant l'arrêt non anticipé de la machine virtuelle.

Ce système de trace est opérationnel mais n'est cependant pas satisfaisant pour la méta-modélisation dans Platypus. Dans Platypus, le méta-méta-modèle, les méta-modèles et les modèles conformes sont maintenus en cohérence grâce au maintient dynamique du lien de causalité. Les dépôts de méta-données et de données sont en effet gérés par les mécanismes réflexifs (voir le chapitre 2.1.5.2). Une évolution d'un élément de méta-modèle peut-être comportementale ou structurelle.

Un ajout, une modification ou une suppression d'un comportement peut être effectué "à chaud", sans perte de cohérence entre les différents dépôts de méta-données et de données. Les outils disponibles permettent d'adapter les développements spécifiques effectués pour la

vérification et la validation.

Par contre, les mécanismes sont inadaptés à l'activité de méta-modélisation en ce qui concerne les aspects structurels. La suppression d'un attribut d'une entité suivi d'un ajout du même attribut dans la même entité est une opération courante en cours d'élaboration d'un méta-modèle. Par exemple, supposons que les deux opérations suivantes sont effectués l'une après l'autre :

- suppression de l'attribut a dans le type entité E ;
- ajout de l'attribut a dans le type entité E.

A cause du maintient dynamique du lien de causalité, lors de la suppression de l'attribut a, toutes les instances de l'entité modifiée E sont automatiquement adaptées à la nouvelle spécification de l'entité. La conséquence est la perte des valeurs de l'attribut correspondant au sein de tous les objets conformes à l'entité altérée E. L'opération suivante permet de reconstituer structurellement la définition de E. Cependant, tous les attributs a des objets conformes à E auront la valeur nil. Le bilan est neutre du point de vue de la spécification de la structure, par contre, le résultat est une perte d'information au niveau des objets conformes.

Il s'agit d'un problème d'évolution de modèle et de migration de données connus dans le domaine des bases de données qui suscite actuellement beaucoup d'intérêt notamment pour les systèmes d'informations collaboratifs sur internet, les ontologies,... L'évolution des schémas et un problème important du fait du risque de perte d'information et du coût élevé des évolutions de schéma. Les problèmes à résoudre sont de prévoir et d'évaluer les effets des évolutions de schéma, de modifier effectivement les schémas et de migrer les données tout en minimisant les pertes d'information.

L'approche souvent explorée consiste en l'utilisation d'un langage spécifique. Par exemple, dans le cadre du projet PRISM [CMZ08, CTZ08], un langage spécifique pour les évolutions de schéma est proposé. Ce langage permet l'expression de scripts comprenant des séries d'opérations élémentaires. L'environnement PRISM permet de valider les scripts et de les exécuter. Les auteurs proposent de plus un modèle formel pour l'évolution des bases de données.

Dans le cadre de la méta-modélisation, ce type de système n'est pas satisfaisant notamment à cause de la lourdeur des opérations de maintenance. L'évolution des méta-modèles doit pouvoir être effectuée de manière agile et les opérations élémentaires automatiquement déduites des modifications des éléments d'un méta-modèle. Cette problématique constitue aussi un sujet de recherche au *Lab-STICC*, notamment Jean-Philippe Babau et Mickaël Kerboeuf ont développé un ensemble d'opérateurs pour des méta-modèles objets dans le cadre de la réutilisation d'outils pour des domaines et des applications connexes [BK11, KB11].

9.1.2 Estimation des impacts sur le système cible

Les méta-modèles sont des éléments clés de l'IDM et toute modification apportée à ces derniers peut avoir pour conséquence des impacts importants sur le système cible. Ainsi, une modification possible au sein du méta-atelier peut au final s'avérer problématique voire impossible au niveau du système cible. Ce problème est d'autant plus sensible si le système cible est maintenu en co-ingénierie par des équipes différentes. Pour orienter les évolutions opérées sur les méta-modèles au sein du méta-atelier, il convient dans un premier temps de pouvoir connaître l'impact d'une modification et dans un deuxième temps de pouvoir la mesurer. C'est un problème difficile inhérent à l'IDM car au niveau des méta-modèles, on ne dispose que d'une vue partielle du système cible.

Une solution possible serait de réifier la description du système cible et du code généré. Le méta-atelier devrait alors gérer une dépendance entre les éléments de méta-modèle et les objets qui représentent le système cible. Cette dépendance pourrait être prise en compte par les mécanismes du méta-atelier. Un système permettant l'évolution avec migration des objets conformes pourrait prendre en compte la dépendance vis-à-vis du système cible pour permettre la validation et la prédiction des impacts sur le système cible.

Les questions qui se posent sont :

- comment décrire le système cible et manipuler cette description ?
- en particulier, comment associer un ou des éléments de méta-modèle à un ou plusieurs éléments décrivant le système cible ?
- comment maintenir le lien de causalité et gérer ces nouvelles dépendances ?
- comment exploiter les descriptions du système cible et les dépendances vis-à-vis du méta-modèle pour prévoir les impacts d'une évolution sur le système cible ?

9.1.3 Applications distribuées et réseaux de capteurs

Depuis une dizaine d'années, nous sommes témoins d'une révolution quant à la nature des applications et celle des plateformes d'exécution. Les applications sont distribuées, le réseau constitue non seulement le support pour les communications mais aussi pour le stockage des informations. Les applications deviennent collaboratives, c'est à dire que les décisions peuvent être prises en collaboration par plusieurs nœuds. Parmi les évolutions remarquables on peut noter la révolution des réseaux de capteurs sans fil [1] et l'émergence des plateformes mobiles. La miniaturisation du matériel et notamment le développement des réseaux de capteurs sans fil ouvre des perspectives quant aux applications en matière d'écologie, de développement durable, de surveillance des biens ou du milieu naturel, de services aux personnes dépendantes,...

Les applications développées pour ces plateformes d'exécution sont mobiles, communicantes. Un grand nombres d'unités de calcul miniaturisées (>1000) peuvent être déployées pour une même application. Le milieu naturel, le contexte environnemental, la topologie du réseau (fixe ou mouvant), les communications entre plateformes d'exécution et le calcul collaboratif sont autant de nouveaux paramètres à prendre en compte pour le développement et la validation des systèmes. Le contexte économique doit aussi être pris en compte et on observe une évolution vers des applications très spécialisées et développées rapidement. Les applications sont de moins en moins onéreuses voire gratuites. Les infrastructures doivent être adaptées à l'évolution en continu des applications, à leur spécialisation et à leur mise au point [Men12].

Dans ce contexte, j'envisage deux projets complémentaires : le premier concerne un travail sur l'infrastructure mise en œuvre par le méta-atelier et le second concerne des applications dans le domaine des réseaux de capteurs sans fil.

9.1.3.1 Évolution de l'infrastructure de Pharo

L'élaboration, le prototypage et plus généralement la validation doivent s'adapter à la miniaturisation et à la mobilité. L'infrastructure doit aussi répondre au besoin de communication des données et aux contraintes de sécurité. Les points difficiles qui m'intéressent plus particulièrement sont :

1. Voir par exemple http://wsn.univ-brest.fr/

- la définition et la manipulation de l'environnement spécifique de développement et de déploiement des applications ; comment spécifier le contexte, le réifier dynamiquement et l'utiliser pour les développements, le déploiement et la gestion des versions ?
- l'aide au prototypage et à la validation d'une application qui doit être à la fois distribuée et contrôlée localement ; comment contrôler dynamiquement, détecter et corriger les anomalies dans un contexte distribué ?
- l'interactivité du système est un point fort de Pharo, cependant les outils souffrent d'une certaines rigidité, par exemple, il n'est pas aisé de construire un outils fonctionnel par composition d'éléments de base ou réutilisation de parties d'outils existants ; comment spécifier et réutiliser une interface Homme-Machine qui encapsule une présentation, la gestion des interactions avec les utilisateurs et un comportement vis-à-vis du contexte d'exécution tout en minimisant la programmation de la "glu" entre les composants ?

Ces travaux pourront s'appuyer sur les évolutions en préparation du système Pharo [DDP12].

9.1.3.2 Réseaux de capteurs sans fil

Les réseaux de capteurs ont maintenant des applications à l'échelle de plusieurs milliers de systèmes permettant de représenter et traiter des états distribués de très grande taille pour les besoins d'une application. Au Lab-STICC, les travaux du projet WSN ont pour objectif le développement d'une chaîne de conception intégrant les spécifications de déploiement, la simulation concurrente du comportement distribué, et le conditionnement de ce comportement dans les systèmes locaux par synthèse de code. L'approche s'appuie sur le modèle de communication synchrone, et sur la spécification du comportement global en termes de processus communicants.

La spécification du comportement est basée sur une représentation abstraite des réseaux et inclut un modèle d'exécution fixe comprenant trois phases : (1) une phase pour l'émission vers les partenaires adjacents, (2) une phase pour la réception depuis les partenaires adjacents et enfin (3) une phase pour le calcul de changement de l'état des nœuds. L'exécution distribuée peut être spécifiée par un programme Occam. La chaine de conception permet la spécification du réseau à l'aide de l'outil NetGen [Fai10], l'association du comportement spécifié en Occam et la simulation par compilation des sources Occam. Lorsque le code des capteurs varie peu, il est possible de simuler le comportement en utilisant une machine SIMD (par exemple, simulation sur GPU Nvidia en utilisant la chaîne de développement CUDA).

La chaine de conception du projet WSN utilise Occam pour la spécification et la simulation des applications distribuées. L'objectif est à terme d'être en mesure de spécifier une application à haut niveau d'abstraction et de la déployer sur des réseaux réels. La faisabilité du développement des applications a été démontrée par plusieurs projets. Cependant, le développement et la validation d'une application demeure très coûteux. Par exemple, la partie comportementale des nœuds est décrite manuellement en Occam. Le déverminage des applications distribuées est aussi un point très dur.

Dans le cadre du projet WSN, ma perspective de contribution est double :

- la première contribution serait d'accroitre le niveau d'abstraction pour la spécification du comportement à l'aide d'un langage spécifique ; le bénéfice serait de faciliter la définition du comportement, de vérifier et valider en amont les programmes et de permettre la synthèse des programmes vers des langages de plus bas niveau pour lesquels on dispose de compilateurs (comme Occam) ;

– en 2011, j'ai spécifié un méta-modèle pour Occam et un analyseur complet de programme Occam à l'aide de Platypus ; les perspectives de ce travail est la maitrise des spécifications de bout en bout pour la vérification et la validation des spécifications mais aussi pour la synthèse des programmes vers des cibles matérielles spécifiques, pour le conditionnement des données et le contrôle de la communication des informations.

9.2 Renforcer la vérification et la validation dans Cheddar

9.2.1 Validation des architectures temps réel par les patrons de conception

Avec Frank Singhoff, nous encadrons Vincent Gaudel en thèse à l'UBO. La société Ellidiss finance en partie la thèse et nos travaux actuels en collaboration avec Ellidiss concernent notamment l'utilisation des patrons de conception pour faciliter la sélection des tests de faisabilité. Nous avons identifiés et décrit cinq patrons de conception. Le premier objectif est de couvrir tous les patrons de conception que nous avons spécifiés et d'automatiser la spécialisation de Cheddar en utilisant Platypus. La perspective immédiate est de permettre aussi la détection des patrons lorsqu'ils sont combinés au sein d'une même architecture.

La limitation pratique de cette approche est qu'elle n'est applicable que pour des architectures conformes à un ou plusieurs patrons combinés. En seconde perspective, il s'agit de permettre l'utilisation de l'outillage pour des architectures non conformes au patron mais pouvant être modifiées afin qu'elles le deviennent. Pour ces travaux, les questions posées sont :

– comment modéliser et évaluer la différence entre deux modèles d'architectures ?
– comment exploiter les différences trouvées entre une architecture non conforme et une architecture conforme à un ou plusieurs patrons de conception pour en calculer les transformations possibles afin de parvenir à une architecture conforme ?

9.2.2 Validation des ordonnanceurs spécifiques

Pour la simulation et la validation des contraintes temporelles, Cheddar permet d'utiliser plusieurs types d'ordonnanceurs parmi les plus utilisés dans la communauté. Dans le cadre de certains projets industriels utilisant Cheddar, des ordonnanceurs spécifiques sont toutefois nécessaires. Ces ordonnanceurs sont alors programmés à l'aide du langage spécifique de Cheddar. Un fois mis au point, ils sont interprétés par Cheddar.

Le langage de Cheddar, est à l'origine prévu pour des ordonnanceurs simples. Cependant, les ordonnanceurs développés dans le cadre des projets industriels se sont révélés beaucoup plus complexes que prévu. Deux problèmes importants se posent :

– *temps d'exécution* : l'interprétation pour la simulation d'un ordonnancement peut prendre plusieurs heures ;
– *vérification et validation* : les ordonnanceurs spécifiques sont difficile à mettre au point.

Pour le passage à l'échelle et gagner en termes de temps d'exécution, nous avons mis en place un processus de génération des ordonnanceurs spécifiques en Ada. Cependant, la génération d'une nouvelle version de Cheddar est coûteuse et le processus n'est pas exploitable sans vérification et validation des ordonnanceurs en amont avant leur réification dans le canevas de Cheddar. La vérification et la validation doivent être effectuées à deux niveaux :

– au niveau des types déclarés pour la spécification des ordonnanceurs et sur les contraintes exprimées dans le méta-modèle pour vérifier la cohérence sémantique du programme ;

– au niveau de la nature des éléments de l'architecture temps-réel exploités par les programmes.

Le premier niveau consiste classiquement à garantir la cohérence par analyse statique. Le second niveau est plus complexe car il s'agit de tenir compte du type d'architecture pour lequel l'ordonnanceur est programmé (architecture mono/multi-cœurs, ordonnancement hiérarchique à 1, 2 ou n niveaux).

Ces problèmes seront notamment abordés par le projet *SMART* (Projet pôle image et réseau, financement régional en collaboration avec les sociétés Ellidiss et Virtualis) et par un projet *Egide*/PESSOA que nous poursuivons en collaboration avec l'université de Lisbonne.

9.2.3 Ingénierie de Cheddar pour augmenter la portée de la méta-modélisation

Concernant l'utilisation de Platypus pour Cheddar, nous désirons accroitre la portée de la méta-modélisation et augmenter les capacités en termes de génération de code. Les couches basses de Cheddar sont actuellement automatiquement générées à partir de Platypus. Lorsque les méta-modèles de Cheddar évoluent, il est nécessaire d'adapter manuellement la mise en œuvre des interfaces Homme-Machine. Ce travail est très coûteux. Nous désirons aussi bénéficier de la génération de code pour la mise en œuvre des interfaces Homme-Machine de Cheddar.

Les travaux à mener concernent l'enrichissement des méta-modèles pour intégrer les contrôles sémantiques des architectures et la construction de générateurs de code pour produire les interfaces Homme-Machine de Cheddar. Les gains obtenus seront principalement une plus grande efficacité des évolutions et une intégration automatique des contrôles sémantiques spécifiés dans les méta-modèles directement dans les interfaces de saisie et les passerelles logicielles permettant l'échange des architectures temps-réel. Ce travail d'ingénierie est notamment en partie effectué à l'UBO dans le cadre d'un post-doctorat.

Bibliographie

[610] I. 610.12. IEEE Standard Glossary of Software Engineering Terminology.

[AAD09] AADL. *Architecture Analysis and Design Language (AADL)*. Aerospace Standard SAE AS5506A, aadl v2 (final draft) edition, 2009.

[AAFZ92] D. S. Arnon, I. Attali, and P. Franchi-Zannettacci. Language-based document processing. Technical Report 1731, Institut National de Recherche en Informatique et en Automatique, Sophia Antipolis, http ://www.inria.fr/, 1992.

[ABE10] M. V. Amstel, M. V. D. Brand, and L. Engelen. An exercise in iterative domain-specific language design. In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, IWPSE-EVOL '10, pp. 48–57, New York, NY, USA, 2010. ACM.

[Agi] Agile Modeling. http ://www.agilemodeling.com/.

[Amb02] S. Ambler. *Agile Modeling : Effective Practices for eXtreme Programming and the Unified Process*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[Amb03] S. W. Ambler. Agile model driven development is good enough. *IEEE Software*, 20(5) :71–73, 2003.

[Ams10] M. F. V. Amstel. The Right Tool for the Right Job : Assessing Model Transformation Quality. In *Proceedings of the 34th Annual IEEE Computer Software and Applications Conference Workshops*, pp. 69–74, 2010.

[ASM06] *Guide for Verification and Validation in Computational Solid Mechanics*. ASME, 2006.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers : Principles, Techniques and Tools*. 1986.

[BB01] B. Boehm and V. R. Basili. Software defect reduction top 10 list. *Computer*, 34 :135–137, 2001.

[BBB+09] K. Beck, M. Beedle, A. V. Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, and J. Sutherland. Manifesto for agile software development. http ://agilemanifesto.org/, 2009.

[BBF03] A. Belangour, J. Bézivin, and M. Fredj. Towards platform independence : a MDA organization. In *Workshop on Information Technology, Rabat, Morocco, March 17-19*, 2003.

[BBM03] F. Budinsky, S. A. Brodsky, and E. Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[BCD+88] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR : the system. In *ACM SIGSOFT'88, Third annual symposium on software development environment*, 1988.

[BDN+] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. Pharo by Example. http ://pharobyexample.org.

[BDV04] A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the ada ravenscar profile in high integrity systems. *Ada Letters*, June 2004.

[Bec99] K. Beck. Embracing change with extreme programming. *Computer*, 32 :70–77, October 1999.

[Ber06] Bernd-Bruegge. Software Lifecycles Models, Software Engineering WS 2006/2007, lecture 17. http ://ebookbrowse.com/l17-softwarelifecyclemodels-pdf-d24854882, 2006.

[BG01] J. Bézivin and O. Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *Automated Software Engineering (ASE 2001)*, pp. 273–282, Los Alamitos CA, 2001. IEEE Computer Society.

[BHH09] M. Beck, M. Haupt, and R. Hirschfeld. Nxtalk : dynamic object-oriented programming in a constrained environment. In *Proceedings of the International Workshop on Smalltalk Technologies*, IWST '09, pp. 38–49, New York, NY, USA, 2009. ACM.

[Bin99] R. V. Binder. *Testing Object-Oriented Systems - Models, Patterns, and Tools.* Addisol-Wesley, 1999.

[BK11] J.-p. Babau and M. Kerboeuf. Domain specific language modeling facilities. In *5th MoDELS workshop on Models and Evolution*, Oct. 2011.

[BL05] N. Berente and K. Lyytinen. Iteration in Systems Analysis and Design : Cognitive Processes and Representational Artifacts. *Sprouts : Working Papers on Information Systems*, 5(23) :5–23, 2005.

[Boo87] G. Booch. *Ingénierie du logiciel avec ADA*. Addison Wesley, 1987.

[Bro04] A. Brown. An introduction to Model Driven Architecture - Part I : MDA and today's systems. http ://www-106.ibm.com/developerworks/rational/library, 12 January 2004.

[Bux07] B. Buxton. *Sketching User Experiences : Getting the Design Right and the Right Design.* Morgan Kaufmann, 2007.

[Bé05] J. Bézivin. On the unification power of models. *Software and System Modeling*, 4(2) :171–188, 2005.

[CA07] B. H. C. Cheng and J. M. Atlee. Research directions in requirements engineering. In *2007 Future of Software Engineering*, FOSE '07, pp. 285–303, Washington, DC, USA, 2007. IEEE Computer Society.

[CDKM02] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Scheduling in Real Time Systems.* John Wiley and Sons Ltd editors, 2002.

[CE00] K. Czarnecki and U. Eisenecker. *Generative Programming : Methods, Tools, and Applications.* Addison Wesley, Reading, Massachusetts, USA, 1st edition, June 2000.

[CH03] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Proceedings of the Workshop on Generative Techniques in the Context of Model-Driven Architecture, OOPSLA'03 Workshop*, 2003.

[CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3) :621–645, 2006.

[Che76] P. Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1) :9–36, 1976.

[Cle88] J. C. Cleaveland. Building Application Generators. *IEEE Software*, July 1988.

[CM00] J. Claxton and P. A. McDougall. Measuring the quality of models. http ://www.tdan.com/i014ht03.htm, 2000.

[CMZ08] C. A. Curino, H. J. Moon, and C. Zaniolo. Graceful database schema evolution : the prism workbench. In *Very Large Data Base (VLDB)*, 2008.

[Com08] B. Combemale. *Approche de métamodélisation pour la simulation et la vérification de modèle - Application à l'ingénierie des procédés.* PhD thesis, INPT ENSEEIHT, July 2008.

[CRC+06] B. Combemale, S. Rougemaille, X. Crégut, F. Migeon, M. Pantel, C. Maurel, and B. Coulette. Towards a rigorous metamodeling. In *Proceedings of the 2nd International Workshop on Model-Driven Enterprise Information Systems (MDEIS), Paphos, Cyprus*, pp. 5–14. INSTICC press, 2006.

[CTZ08] C. A. Curino, L. Tanca, and C. Zaniolo. Information systems integration and evolution : Ontologies at rescue. In *International Workshop on Semantic Technologies in System Maintenance (STSM)*, 2008.

[DAS11] S. M. N. Damak, S. ASSAR, and C. Souveyet. Pour une perspective comportementale dans les méta-modèles de processus. pp. 351–366, Lille, France, May 2011.

[Dav03] J. Davis. Gme : the generic modeling environment. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pp. 82–83, New York, NY, USA, 2003. ACM.

[DDP12] S. Ducasse, M. Denker, and D. Pollet. Pharo's vision goals, processes, and development effort. http ://www.pharo-project.org, 2012.

[Des94] P. Desfray. *Object engineering : the fourth dimension.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[DG06] S. Ducasse and T. Girba. Using smalltalk as a reflective executable metalanguage. In *International Conference on Model Driven Engineering Languages and Systems (MODELS/UML 2006). volume 4199 of LNCS*, pp. 604–618. Springer-Verlag, 2006.

[DGKR08] S. Ducasse, T. Girba, A. Kuhn, and L. Renggli. Meta-environment and executable meta-language using smalltalk : an experience report. *Softw Syst Model, Springer-Verlag*, 2008.

[DHR+07] M. B. Dwyer, J. Hatcliff, R. Robby, C. S. Pasareanu, and W. Visser. Formal software analysis emerging trends in software model checking. In *2007 Future of Software Engineering*, FOSE '07, pp. 120–136, Washington, DC, USA, 2007. IEEE Computer Society.

[DKV00] Deursen, Klint, and Visser. Domain-specific languages : an annotated bibliography. *SIGPLAN Not.*, 35(6) :26–36, 2000.

[DNS+06] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits : A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2) :331–388, 2006.

[DPZ01] M. Dahchour, A. Pirotte, and E. Zimànyi. Definition and application of metaclasses. In H. Mayr, J. Lazansky, G. Quirchmayr, and P. Vogel, editors, *12th International Conférence on Database and Expert Systems Applications (DEXA'01)*, pp. 32–41. LNCS 2113, Springer-Verlag, 2001.

[DS08] P. Dissaux and F. Singhoff. Stood and Cheddar : AADL as a Pivot Language for Analysing Performances of Real Time Architectures. Proceedings of the European Real Time System conference. Toulouse, France, Jan. 2008.

[Eis97] M. Eisenstadt. My hairiest bug war stories. *Communications of the ACM*, 40(4) :30–37, 1997.

[EK00] E. Engstrom and J. Krueger. Building and rapidly evolving domain-specific tools with dome. In *Proceedings of IEEE International Symposium on Computer-Aided Control System Design (CACSD*, pp. 83–88, 2000.

[Eng00] V. Englebert. *A Smart Meta-CASE. Towards an Integrated Solution.* PhD thesis, University of Namur, 2000.

[Eva01] W. M. Evanco. Prediction models for software fault correction effort. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, CSMR '01, IEEE Computer Society*, pp. 114–120, 2001.

[Fai10] T. Failler. Netgen : un générateur de code pour CUDA, principes, implémentation et performances. Technical report, Lab-STICC, UBO/WSN, 2010.

[Fav04] J.-M. Favre. Towards a Basic Theory to Model Model Driven Engineering. In *Procs. of the 3rd Int. Workshop in Software Model Engineering, (WiSME'04)*, 2004. WiSME 2004.

[FEBF06] J.-M. Favre, J. Establier, and M. Blay-Fornarino, editors. *L'ingénierie dirigée par les modèles : au-delà du MDA.* Hermes-Lavoisier, Cachan, France, Feb. 2006.

[FGS11] T. Frey, M. Gelhausen, and G. Saake. Categorization of Concerns - A Categorical Program Comprehension Model. In *Proceedings of the Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU) at the ACM Onward! and SPLASH Conferences, Portland, Oregon, USA*, October 2011.

[Fla02] R. Flatscher. Metamodeling in EIA/CDIF - meta-metamodel and metamodels. *ACM Trans. Model. Comput. Simul*, 12(04), 2002.

[Fle06] F. Fleurey. *Langage et méthode pour une ingénierie des modèles fiable.* PhD thesis, Université de Rennes 1, Equipe Triskell - IRISA, 2006.

[Foo92] B. Foote. Objects, Reflection, and Open Languages. Workshop on Object-Oriented Reflection and Metalevel Architectures (ECOOP '92) - www.laputan.org, 1992.

[FR07] R. France and B. Rumpe. Model-driven development of complex software : A research roadmap. In *2007 Future of Software Engineering*, FOSE '07, pp. 37–54, Washington, DC, USA, 2007. IEEE Computer Society.

[Ga96] E. Gamma and al. *Design patterns.* Addison-Wesley, 1996.

[GAL] LE STANDARD INTERNATIONAL STEP ET SON PROTOCOLE D'APPLICATION 214. http ://www.galia.com/.

[GGD07] O. Greevy, T. Girba, and S. Ducasse. How Developpers Develop Features. In *European Conference on Software Maintenance and Reengineering (CSMR'07)*, 2007.

[Gir10] T. Girba. Humane assessment. http ://www.humane-assessment.com/, 2010.

[GLN05] M. Gälli, M. Lanza, and O. Nierstrasz. Towards a taxonomy of sunit tests. In *ESUG 2005 Research track*, 2005.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80 : The Language and its Implementation.* 1983.

[Gro04] O. M. Group. Meta object facility (MOF) 2.0 core final adopted specification. Technical report, Object Management Group, 2004.

[GRS96] L. George, N. Rivierre, and M. Spuri. Preemptive and Non-Preemptive Real-time Uni-processor Scheduling. INRIA Technical report number 2966, 1996.

[GSCK04] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories : Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley, 2004.

[Har00] M. J. Harrold. Testing : a roadmap. In ACM, editor, *ICSE'00 Proceedings of the Conference on The Future of Software Engineering*, pp. 61–72, 2000.

[Hic11] R. Hickey. Simple made easy. StrangeLoop 2011 conference talk, http ://www.infoq.com/presentations/Simple-Made-Easy/, Oct 2011.

[HR00] D. Harel and B. Rumpe. Modeling Languages : Syntax, Semantics and All That Stuff, Part I. Technical report, Mathematics & Computer Science, Weizmann Institute Of Science, Weizmann Rehovot, Israel, August 2000.

[HR04] D. Harel and B. Rumpe. Meaningful Modeling : What's the Semantics of "Semantics" ? *IEEE Computer*, 37 :64– 72, october 2004.

[IAE00] IAEA. Effective handling of software anomalies in computer based systems at nuclear power plants. *International Working Group on Nuclear Power Plant Control and Instrumentation*, 2000.

[Ing81] D. H. H. Ingalls. Design Principles Behind Smalltalk. *BYTE Magazine*, August 1981.

[ISO94a] ISO 10303-1. *STEP Part 1 : Overview and fundamental principles*, 1994.

[ISO94b] ISO 10303-11. *STEP Part 11 : EXPRESS Language Reference Manual*, 1994.

[ISO94c] ISO 10303-21. *Part 21 : Clear Text Encoding of the Exchange Structure*, 1994.

[ISO94d] ISO 10303-22. *Part 22 : Standard Data Access Interface*, 1994.

[ISO95a] ISO 10303-23. *Part 23 : C++ Programming Language Binding to the Standard Data Access Interface Specification*, 1995.

[ISO95b] ISO 10303-23. *Part 24 : C Programming Language Binding to the Standard Data Access Interface Specification*, 1995.

[ISO95c] ISO 10303-23. *Part 24 : IDL Programming Language Binding to the Standard Data Access Interface Specification*, 1995.

[JB06] F. Jouault and J. Bézivin. KM3 : a DSL for Metamodel Specification. In *FMOODS'06*, volume 4037, pp. 171–185, 2006.

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The unified software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[JE97] I. U. Jürgen Ebert, Roger Süttenbach. Meta-case in practice : A case for kogge. In *ADVANCED INFORMATION SYSTEMS ENGINEERING*, volume 1250. Lecture Notes in Computer Science, 1997.

[Jeu] M. Jeusfeld. Conceptbase web site. http ://conceptbase.sourceforge.net/.

[JGJ+95] M. Jarke, R. Gallersdörferand, M. A. Jeusfeld, M. Staudt, and S. Eherer. Conceptbase - a deductive object base for meta data management. *Journal of Intelligent Information Systems, SpringerLink*, pp. 167–192, 1995.

[Jon08] C. Jones. Software quality in 2008 : Survey of the state of the art (Presentation). http ://www.scribd.com/doc/7758538/ Capers-Jones-Software-Quality-in-2008, 2008.

[JPAA06] S. Jean, G. Pierra, and Y. Ait-Ameur. Domain Ontologies : A DataBase-Oriented Analysis. In *Proceedings of Web Information Systems and Technologies (WEBIST'06)*, 2006.

[JS02] M. Jorgensen and D. I. K. Sjoberg. Impact of experience on maintenance skills. *Journal of Software Maintenance*, 14(2) :123–146, 2002.

[Kai89] G. E. Kaiser. Incremental Dynamic Semantics for language-Based Programming Environments. In *ACM Transaction on Programming Languages and Systems*, volume 11, pp. 169–193, April 1989.

[KB89] B. Krieg-Brückner. Algebraic specification and fundamentals for transformational program and meta program development. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 2 : Advanced Seminar on Foundations of Innovative Software Development II and Colloquium on Current Issues in Programming Languages*, TAPSOFT '89, pp. 36–59, London, UK, UK, 1989. Springer-Verlag.

[KB11] M. Kerboeuf and J.-p. Babau. A dsml for reversible transformations. In *11th OOPSLA Workshop on Domain-Specific Modeling*, Oct. 2011.

[KBJV06] I. Kurtev, J. Bézivin, F. Jouault, and P. Valduriez. Model-based DSL frameworks. In *OOPSLA '06 : Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pp. 602–616, New York, NY, USA, 2006. ACM.

[Ken02] S. Kent. Model driven engineering. In *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, pp. 286–298, London, UK, UK, 2002. Springer-Verlag.

[KK02] D. Karagiannis and H. Kühn. Metamodelling Platforms. In *Proceedings of the third International Conference EC-Web 2002 - Dexa 2002*, pp. 182. LNCS 2455, Springer-Verlag, 2002.

[Kle07] A. Kleppe. A Language is More than a Metamodel. In *Procs. of the 4th Int. MoDELS Workshop on Software Language Engineering (ATEM 2007), Nashville (USA)*, 2007.

[Kle08] A. Kleppe. *Software Language Engineering : Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 2008.

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Trans. Softw. Eng. Methodol.*, 2 :176–201, April 1993.

[KM08] P. Kelsen and Q. Ma. A Lightweight Approach for Defining the Formal Semantics of a Modeling Language. In *MoDELS 2008, LNCS 5301*, pp. 690–704, 2008.

[KMS92] T. R. Kramer, K. C. Morris, and D. A. Sauder. *A Structural EXPRESS Editor*. NISTIR 4903, 1992.

[KR91] G. Kiczales and J. D. Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[KRP+94] M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A Practitioner's Handbook for Real Time Analysis*. Kluwer Academic Publishers, 1994.

[Kru92] C. W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2), June 1992.

[KT08] S. Kelly and J.-P. Tolvanen. *Domain Specific Modelling*. John Wiley, 2008.

[Kü05] T. Kühne. What is a model ? In J. Bezivin and R. Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[Kü06] T. Kühne. Matters of (Meta-)Modeling. *Software and Systems Modeling*, 5(4) :369–385, Dec. 2006.

[Lag08] L. Lagadec. MADEO : Object Oriented Programming, Modelization, and Tools for FPGAS. In *European Smalltalk User Group*, Amsterdam Pays-Bas, 2008.

[Las12] G. Lasnier. *Une Approche Intégrée pour la Validation et la Génération de Systèmes Critiques par Raffinement Incrémental de Modèles Architecturaux*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, France, 2012.

[LHB10] W. Lidwell, K. Holden, and J. Butler. *Universal Principles of Design*. Rockport Publishers, 2010.

[Lie97] H. Lieberman. The debugging scandal and what to do about it. *Communications of the ACM*, 40(4) :26–29, 1997.

[Lin94] Understanding quality in conceptual modelling. *IEEE Software*, 11(2) :42–49, 1994.

[LL73a] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environnment. *Journal of the Association for Computing Machinery*, 20(1) :46–61, Jan. 1973.

[LL73b] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1) :46–61, 1973.

[LSRB12] S. Li, F. Singhoff, S. Rubini, and M. Bourdellès. Applicability of rt schedulability analysis on a software radio protocol. *ACM SIGAda Ada Letters, ISSN :1094-3641*, 32(3) :61–68, 2012.

[LX05] D. Liang and K. Xu. Debugging object-oriented programs with behavior views. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, AADEBUG'05, pp. 133–142, New York, NY, USA, 2005. ACM.

[MA07] P. Mohagheghi and J. Aagedal. Evaluating quality in model-driven engineering. In *Workshop on Modelling in Software Engineering (MISE'07), Proc. of ICSE'07*, pp. 6, 2007.

[Mac05] C. M. Macal. Model verification and validation. In *Workshop on Threat Anticipation : Social Science Methods and Models, Chicago*, 2005.

[Mae87] P. Maes. Concepts and experiments in computational reflection. *SIGPLAN Not.*, 22 :147–155, December 1987.

[Mar02] R. C. Martin. Visitor. http ://objectmentor.com/resources/articles/visitor.pdf, 2002.

[MCSH10] J. Mc Cormick, F. Singhoff, and J. Hugues. *Building Parallel, Embedded, and Real-Time Applications with Ada*. Cambridge University Press, UK, 365 pages. ISBN-13 : 9780521197168., July 2010.

[MD] P. Mohagheghi and V. Dehlen. An overview of quality frameworks in model-driven engineering and observations on transformation quality. http ://citeseerx.ist.psu.edu/viewdoc/download ?doi=10.1.1.96.8653.

[MD04] E. Meijer and P. Drayton. Static typing where possible, dynamic typing when needed : The end of the cold war between programming languages. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

[MD07] P. Mohagheghi and V. Dehlen. An overview of quality frameworks in model-driven engineering and observations on transformation quality. In M. S. L. Kuzniarz, J. L. Sourrouille, editor, *Proceedings of the 2nd Workshop on Quality in Modeling*, MoDELS 2007, pp. 3–17, Nashville, TN, USA, 2007.

[MD08] P. Mohagheghi and V. Dehlen. Where is the proof ? - a review of experiences from applying mde in industry. In *Proceedings of the 4th European conference on Model Driven Architecture : Foundations and Applications*, ECMDA-FA '08, pp. 432–443, Berlin, Heidelberg, 2008. Springer-Verlag.

[Mel09] S. J. Mellor. Models. models. models. so what ? In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, MODELS '09, pp. 1–1, Berlin, Heidelberg, 2009. Springer-Verlag.

[Men12] Mendix. DON'T GET LOST IN THE APP JUNGLE - 10 Things Every IT Leader Should Know to Survive and Thrive in the App Revo. http ://ww2.mendix.com, 2012.

[Met] MetaEdit+ Technical Summary. http ://www.metacase.com/papers/index.html.

[MFBC10] P.-A. Muller, F. Fondement, B. Baudry, and B. Combemale. Modeling modeling modeling. In *in : International Conference on Model Driven Engineering Languages and Systems, (MoDELS)*. Springer, 2010.

[MFJ05] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In *in : International Conference on Model Driven Engineering Languages and Systems (MoDELS), LNCS 3713 (2005*, pp. 264–278. Springer, 2005.

[MHS05] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4) :316–344, Dec. 2005.

[Mil09] R. Milner. *From semantics to Computer Science*. Cambridge University Press, 2009.

[MJD96] J. Malenfant, M. Jacques, and F.-N. Demers. A tutorial on behavioral reflection and its implementation. In *Proceedings of the first International Conference on Metalevel Architectures and Reflection (Reflection 96)*, pp. 1–20, 1996.

[Moo] Moose. http ://www.moosetechnology.org.

[Mor91] K. C. Morris. Architecture for the Validation Testing System Software. Technical Report NISTIR 4742, National Institute of Standards and Technology, Gaithersburg, Maryland, 1991.

[MPAA03] M. E.-H. Mimoune, G. Pierra, and Y. Ait-Ameur. An ontology-based aproach for exchanging data between heterogeneous database systems. In *ICEIS 2003 : Proceedings of the 5th International Conference On Enterprise Information Systems*, Angers - France, 2003. École Supérieure d' Électronique de l' Ouest.

[MPGK00] J. Magee, N. Pryce, D. Giannakopoulou, and J. Kramer. Graphical animation of behavior models. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pp. 499–508, New York, NY, USA, 2000. ACM.

[MS10] N. Mellegård and M. Staron. Characterizing model usage in embedded software engineering : a case study. In *Proceedings of the Fourth European Conference on Software Architecture : Companion Volume*, ECSA '10, pp. 245–252, New York, NY, USA, 2010. ACM.

[MWD+05] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in Software Evolution. In O. R. O. T. O. U. repository of research publications and other research outputs, editors, *8th International Workshop on Principles of Software Evolution*, IWPSE, 2005.

[NBD+05] O. Nierstrasz, A. Bergel, M. Denker, S. Ducasse, M. Gaëlli, and R. Wuyts. On the revival of dynamic languages. In LNCS, editor, *Proc. Software Composition 2005*, volume 3628, pp. 1–13, 2005.

[NDG05] O. Nierstrasz, S. Ducasse, and T. Gĭrba. The story of moose : an agile reengineering environment. *SIGSOFT Softw. Eng. Notes*, 30 :1–10, September 2005.

[NDR09] O. Nierstrasz, M. Denker, and L. Renggli. Model-Centric, Context-Aware Software Adaptation. In B. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pp. 128–145. Springer, 07 2009.

[OFT97] OFTA, editor. *Application des techniques formelles au logiciel*. Série ARAGO. Observatoire Français des Techniques Avancées, 5, rue Descartes, 75005 Paris, 1997.

[OMG92] The Common Object Request Broquer : Architecture and Specification. Technical report, Object Management Group, 1992.

[OMG03] OMG. Model Driven Architecture. http ://www.omg.org/mda, 2003.

[OMG07] OMG. UML profile for Marte. http ://www.omg.org/, 2007.

[OR10] W. L. Oberkampf and C. J. Roy. *Verification and Validation in Scientific Computing*. Cambridge University Press, 2010.

[Ost96] L. Osterweil. Strategic directions in software quality. *ACM Comput. Surv.*, 28 :738–750, December 1996.

[Pai04] R. F. Paige. Specification-driven development of an executable metamodel. In *In Proc. Workshop in Software Model Engineering 2004, co-located with UML*, 2004.

[Pal95] M. Palmer. Guidelines for the development and approval of STEP application protocols. Technical report, ISO TC184/SC4/WG4 N511, 1995.

[Par08] Parastoo Mohagheghi and Vegard Dehlen and Tor Neple. Towards a tool-supported quality model for model-driven engineering. In *Proceedings of the 3rd Workshop on Quality in Modelling (QiM'08), MODELS 2008*, 2008.

[PBO04] R. Paige, P. Brooke, and J. Ostroff. Agile development of a metamodel in eiffel. In *Proc. Fifteenth IEEE International Symposium on Software Reliability Engineering 2004*, 2004.

[PCH12] M. Perscheid, D. Cassou, and R. Hirschfeld. Test Quality Feedback - Improving Effectivity and Efficiency of Unit Testing. In *Proceedings of the 10th International Conference on Creating, Connecting and Collaborating through Computing*, 2012.

[Pet81] J. L. Peterson. *Petri Net theory and the Modelling of Systems*. Prentice Hall, 1981.

[Pha] The Pharo project. http ://www.pharo-project.org.

[PSDL10] A. Plantec, F. Singhoff, P. Dissaux, and J. Legrand. Enforcing applicability of real-time scheduling theory feasibility tests with the use of design-patterns. In *Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation-Volume Part I*, pp. 4–17. Springer-Verlag, 2010.

[RBP+91] J. Rumbaugh, M. Blaha, M. Premerlani, W. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[Ren10] L. Renggli. *Dynamic Language Embedding With Homogeneous Tool Support*. PhD thesis, University of Bern, October 2010.

[Res02] Research Triangle Institute. The Economic Impacts of Inadequate Infrastructure for Software Testing, 2002. Sponsored by the Department of Commerce's National Institute of Standards and Technology.

[RFBlO01] D. Riehle, S. Fraleigh, D. Bucka-lassen, and N. Omorogbe. The Architecture Of A UML Virtual Machine. In *OOPSLA'01 Proceedings*. ACM Press, 2001.

[RG02] J. Ruiz-Garcia. *Contribution à la validation des systèmes réflexifs tolérants aux fautes : stratégie de test de protocoles à métaobjets*. PhD thesis, 2002.

[RH] N. Rossiter and M. Heather. Four-level Architecture for Closure in Interoperability. Technical report.

[RSS00] S. Ramanujan, R. W. Scamell, and J. R. Shah. An experimental investigation of the impact of individual, program, and organizational characteristics on software maintenance effort. *Journal of Systems and Software*, 54 :137–157, 2000.

[SA04] M. Suzana and M. Amoretti. Categorisation process and conceptual maps. In F. M. G. A. J. Cañas, J. D. Novak, editor, *Concept Maps : Theory, Methodology, Technology, Proc. of the First Int. Conference on Concept Mapping, Pamplona, Spain*, 2004.

[SAE04] SAE. Architecture Analysis and Design Language (AADL) AS 5506. Technical report, The Engineering Society For Advancing Mobility Land Sea Air and Space, Aerospace Information Report, Version 1.0, Nov. 2004.

[SB96] S. Slaughter and R. D. Banker. A study of the effects of software development practices on software maintenance effort. In *Proceedings of the International Conference on Software Maintenance, ICSM '96, IEEE Computer Society*, pp. 197–205, 1996.

[Sch06] D. C. Schmidt. Model-driven engineering. *IEEE Computer*, 39(2), February 2006.

[SCS12] J. Sappidi, B. Curtis, and A. Szynkarski. The CRASH Report - 2011/2012. Technical report, CAST Research Labs, 2012.

[SDNB03] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits : Composable Units of Behaviour. In L. N. in Computer Science, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2743, pp. 248–274. Springer-Verlag, 2003.

[Sin98] J. Singer. Practices of software maintenance. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pp. 139–, Washington, DC, USA, 1998. IEEE Computer Society.

[SK03] S. Sendall and W. Kozaczynski. Model Transformation : The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5) :42–45, 2003.

[SLNM04a] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar : a Flexible Real Time Scheduling Framework. *ACM SIGAda Ada Letters, ACM Press, New York, USA*, 24(4) :1–8, Dec. 2004.

[SLNM04b] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar : a flexible real time scheduling framework. In ACM, editor, *SIGAda international conference on Ada*, pp. 1–8, 2004.

[SPDL09] F. Singhoff, A. Plantec, P. Dissaux, and J. Legrand. Investigating the usability of real-time scheduling theory with the Cheddar project. *Real-Time Systems*, 43(3) :259–295, 2009.

[Spi92] J. M. Spivey. *The Z Notation : A reference manual.* 2nd edition, 1992.

[Spi01] D. Spinellis. Notable design patterns for domain specific languages. *Journal of Systems and Software*, 56(1) :91–99, Feb. 2001.

[Squ] The Squeak web site. http ://www.squeak.org.

[Tan09] E. Tanter. Reflection and open implementations. Technical report, DCC, University of Chile, Avenida Blanco Encalada 2120, Santiago, Chile, 2009.

[TFR05] D. E. Turk, R. B. France, and B. Rumpe. Assumptions underlying agile software-development processes. *J. Database Manag.*, 16(4) :62–87, 2005.

[TMC99] S. A. Thibault, R. Marlet, and C. Consel. Domain-specific languages : From design to implementation application to video device drivers generation. *IEEE Trans. Softw. Eng.*, 25(3) :363–377, May 1999.

[TSL+05] M.-N. Terrasse, M. Savonnet, E. Leclercq, T. Grison, and G. Becker. Points de vue croisés sur les notions de modèle et métamodèle. In *1ères journées sur l'Ingénierie Dirigée par les Modèles*, pp. 17–28, June 2005.

[TW07] L. Tratt and R. Wuyts. Guest Editors' Introduction : Dynamically Typed Languages. *IEEE Software*, 24(5) :28–30, 2007.

[UL10] M. Utting and B. Legeard. *Practical Model-Based Testing*. Morgan Kaufmann, 2010.

[Unh05] B. Unhelkar. *VERIFICATION AND VALIDATION FOR QUALITY OF UML 2.0 MODELS*. Wiley-Interscience, 2005.

[Usc03] M. Uschold. Where are the semantics in the semantic web ? *AI Mag.*, 24(3) :25–36, 2003.

[VB04] M. Völter and J. Bettin. Patterns for model-driven development, http ://www.voelter.de/services/mdsd.html, 2004.

[VDK98] A. Van Deursen and P. Klint. Little languages : little maintenance. *Journal of Software Maintenance*, 10(2) :75–92, Mar. 1998.

[vHTVMP04] van Hung Tran, A. L. Van, P. Massonet, and C. Ponsard. Goal-oriented requirements animation. In *Proceedings of the Requirements Engineering Conference, 12th IEEE International*, RE '04, pp. 218–228, Washington, DC, USA, 2004. IEEE Computer Society.

[Vis] The VisualWorks web site. http ://www.cincomsmalltalk.com/.

[VP04] D. Varro and A. Pataricza. Generic and meta-transformations for model transformation engineering. In Array, editor, *7th International Conference on the Unified Modeling Language (UML 2004)*, pp. 290–304, Lisbon, Portugal, 2004. Springer-Verlag. LNCS 3273.

[Weg96] P. Wegner. Interoperability. *ACM Computing Surveys*, 28(1), 1996.

[Wei11] S. M. Weinschenk. *100 Things Every Designer Needs to Know About People*. New Riders Press, 2011.

[Wel06] L. Wells. Performance Analysis using CPN Tools. Proceedings of the First International Conference on Performance Evaluation Methodologies and Tools 2006. ACM Press, ValueTools'06, 2006.

[wika] wikipedia. http ://en.wikipedia.org/wiki/Mental_model.

[wikb] wikipedia. http ://en.wikipedia.org/wiki/Abductive_reasoning.

[WWM+07] T. Weigert, F. Weil, K. Marth, P. Baker, C. Jervis, P. Dietz, Y. Gui, A. Van Den Berg, K. Fleer, D. Nelson, M. Wells, and B. Mastenbrook. Experiences in deploying model-driven engineering. In *Proceedings of the 13th international SDL Forum conference on Design for dependable systems*, SDL'07, pp. 35–53, Berlin, Heidelberg, 2007. Springer-Verlag.

[ZWN+06] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. On the Value of Static Analysis for Fault Detection in Software. *IEEE Transactions on Software Engineering*, 32(4), April 2006.

Troisième partie

Annexes

Curriculum Vitæ

État civil

Né le 22 janvier 1964 à Vernon (Eure).

Adresse Professionnelle
Département d'Informatique
U.F.R. Sciences et Techniques
Université de Bretagne Occidentale
20, avenue Le Gorgeu, 29233 Brest
Tél : +(33) 2 98 01 81 50
Fax : +(33) 2 98 01 80 11
E-mail : alain.plantec@univ-brest.fr
Web : http://www.labsticc.univ-brest.fr/pages_perso/plantec

Adresse personnelle
22 rue Béranger,
29200, Brest

- **Domaine de recherche** : Vérification et validation par simulation et prototypage dans le contexte de l'ingénierie dirigée par les modèles.
- **Mots clés** : Ingénierie Dirigée par les Modèles, Vérification, Validation, Prototypage, Langages Dynamiques, *Smalltalk*, Interopérabilité par les données, Norme STEP.

Parcours professionnel

2011/—- Membre du conseil du département Informatique de l'UBO,

2000/—- Maître de conférences à l'Université de Bretagne Occidentale,
UFR des sciences et techniques.
Membre du Laboratoire en sciences et techniques de l'information, de la communication et de la connaissance (Lab-STICC / CNRS UMR 6285)

1994/2000 Ingénieur chez Thales-IS pendant ma thèse CIFRE

1988/1990 Analyste programmeur chez un éditeur de logiciels de gestion pour PME

Encadrement doctoral et scientifique

- Je co-encadre avec M. Frank Singhoff le post-doc de Christian Fotsing à Brest, dans le cadre du projet SMART (Projet pôle image et réseau, financement régional en collaboration avec les sociétés Ellidiss et Virtualis).
- Encadrement de la thèse de Vincent Gaudel (octobre 2010-novembre 2013). J'encadre cette thèse à 50% (M. Frank Singhoff en assure la direction et le reste de l'encadrement).

Vincent Gaudel étudie des patrons de conception temps réel qui doivent permettre d'automatiser l'utilisation d'outils d'analyse de l'ordonnancement tels que Cheddar. La méta-modélisation est un élément important de notre approche. Cette thèse est co-financée par le conseil régional du Finistère et la société Ellidiss Technologies.
5 publications acceptées et présentées : [GSP+11, CSR+12, PSGR11, DLP+11, GSP+11]. Soutenance prévue en novembre 2013.

– Encadrement de la thèse de Maxime Louvel (octobre 2008-novembre 2011). J'ai encadré cette thèse à 50% (M. Jean-Philippe Babau en a assuré la direction et le reste de l'encadrement). Maxime développe un modèle et une méthode pour garantir la qualité de service pour l'exécution d'applications multimédia dans un environnement domotique. Ce travail implique une méta-modélisation et le développement d'outils pour la gestion de ressources hétérogènes (CPU, réseau, ..). La méthode s'appuie sur une validation a priori de la disponibilité des ressources. L'originalité du travail est de prendre en considération les aspects distribution dans une architecture répartie et l'hétérogénéité des éléments d'un environnement domotique. Cette thèse est financée par une convention CIFRE avec France Telecom R&D.
3 publications acceptées et présentées : [LPPB11, LBPB11, LPPB10].
Un article de revue en cours de soumission.
Thèse soutenue le 17 novembre 2011.

– J'ai participé à 3 jurys de thèse de 2009 à 2011 en tant qu'examinateur

Rayonnement

Rayonnement par les logiciels que nous diffusons

– Depuis 2004, je suis le responsable du projet Platypus dans lequel nous développons et valorisons l'outil de même nom. Le méta-atelier Platypus met en œuvre une approche mixte basée sur l'implantation d'un langage de modélisation de données dans un environnement agile pour la méta-modélisation, la vérification et la validation des méta-modèles. Au sein de Platypus, les méta-modèles sont exécutables et peuvent être évalués très précocement avant la génération de code vers un système cible. Platypus a été valorisé par son implication importante dans trois projets :
  – Dans le cadre du projet Européen PREMECS de l'Ifremer, pour la spécification et le prototypage d'un langage permettant la définition de la structure et des propriétés de talons de chaluts ;
  – Dans le cadre du projet Européen Morpheus, pour la méta-modélisation des traitements et l'implantation des composants d'accueil des modèles de traitements en Java et Smalltalk (Intégration par l'IDM pour des systèmes hétérogènes).
  – Dans le cadre du projet Cheddar, pour la ré ingénierie des méta-modèles de Cheddar, l'implantation du langage de modélisation des ordonnanceurs (interpréteur et compilateur d'ordonnanceurs spécifiques) et actuellement pour la reconnaissance de patrons de conception temps-réel dans le cadre de la thèse de Vincent Gaudel.
  – J'ai développé un analyseur complet d'Occam à l'aide de Platypus ; cet analyseur est disponible sous Pharo et VisualWorks ; l'objectif est de contribuer au développement de la chaine de conception, de simulation et de déploiement d'applications pour réseaux de capteurs sans fil de l'équipe WSN du lab-STICC.

- Depuis 2006, je suis très fortement impliqué dans l'utilisation des méta-modèles et du méta-atelier Platypus pour le développement de Cheddar, une plateforme de vérification d'architecture temps réel. Nous continuons à faire évoluer Cheddar qui est aujourd'hui employé par de nombreuses équipes nationales et internationales :
  - Dans l'industrie, par les entreprises suivantes : Ellidiss Technologies (qui offre un support industriel pour nos outils), par Airbus (qui a intégré nos outils au sein d'outils de simulateur de vol) et par l'Agence Spatiale Européenne où notre outil a été intégré dans leur plate-forme de développement TASTE.
  - Pour des activités de recherche : dans le cadre de projets européens (IST ASSERT) ou de projets universitaires (à l'Université de Madrid, l'Université de Lisbonne, Yeungnam University, Banaras Hindu University, Télécom-Paris Tech, l'IRIT, le CNES, INSA-Toulouse, …),
  - Pour des activités d'enseignement : à Télécom Paris-Tech (France), University of Rhode Island (USA), University of Monash (Australie), Universitat Politécnica de Catalunya (Espagne), University of the West Indies (Indes), Université de Lisbonne, l'ISAE, le CERT-ONERA,...

Participation à des comités de programme

- *Smalltalk Directions 2012*, Biloxi, Mississippi, atelier de Smalltalk Industry Conference (STIC 2012),
- *International Workshop on Smalltalk Technologies*, IWST'09 à Brest (atelier de ESUG'09),
- Atelier *Motif pour la méta-modélisation*, MP'07, atelier de IDM'07.

Responsabilités éditoriales et directions de programme

- Co-guest editor pour la revue *Science of Computer Programming : Methods of Software Design : Techniques and Applications* (special issue "Smalltalk Based Systems" de Elsevier, 2012),
- Co-responsable de programme de *International Workshop on Smalltalk Technologies*, IWST'12 à Gent, (atelier de ESUG'12),
- Responsable de l'édition des actes pour ETR'11,
- Co-guest editor pour la revue *Software : Practice and Experience* (special issue, articles pré-sélectionnés du workshop IWST'11).
- Co-responsable de programme de *International Workshop on Smalltalk Technologies*, IWST'11 à Edimbourg (atelier de ESUG'11),

Relecture d'articles

J'ai relu des articles pour les revues et conférences internationales suivantes :

- Conférences internationales : TOOLS 2010, Models 2010, ECOOP 2010, workshop UML and AADL 2010
- Revues : Data and Knowledge Engineering (ELSEVIER), Science of Computer Programming (ELSEVIER), Journal of Computer Systems Science and Engineering (CSSE), Computer, Information and Software Technology (ELSEVIER)

Collaborations internationales, projets et contrats

- Je suis responsable du projet *Enhancement of the Adele AADL diagram editor* financé par l'US-Army et en collaboration avec la société Ellidiss Technologies (Montant 50K Euros, durée 10 mois, de Janvier à Octobre 2013),
- Je participe à un projet *Egide*/PESSOA que nous poursuivons en collaboration avec l'université de Lisbonne et visant à accroître les capacité d'analyse de Cheddar pour les architectures multi-cœur et les architectures temps réel hiérarchiques.
- Je participe au projet SMART (Projet pôle image et réseau, financement régional en collaboration avec les sociétés Ellidiss et Virtualis).
- Depuis 2009, je participe à un contrat de transfert technologique avec la société Ellidiss Technologies dans le cadre du projet Cheddar.
- Depuis 2009, je suis membre du comité de direction d'ESUG, l'Association Européenne des Utilisateurs de Smalltalk, (`www.esug.org`). Nous avons environ 600 membres en Europe. Cette association organise (1) une conférence annuelle accueillant chaque année 120 à 150 membres académiques ou industriels venant du monde entier, (2) des actions de sponsoring comme SummerTalk ou Google Summer of Code, pour les étudiants ou l'envoi de livres dans les bibliothèques. Nous aidons aussi financièrement les projets innovants par l'attribution de prix ou de subventions. Je suis plus particulièrement responsable du suivi des Summertalk depuis 2011.
- Depuis 2008, je participe régulièrement au comité de normalisation international AS-2C de la SAE (Society of Automotive Engineers). Ce comité élabore les standards associés au langage d'architecture AADL. Depuis 2008, nos travaux sont régulièrement présentés dans ce comité (cf. liste des publications, présentations [GSP+11, PGR+11, DLPS08]). Cette collaboration a notamment abouti cette année au contrat financé par l'US Army et l'entreprise Ellidiss Technologies, deux des principaux participants au comité AS-2C (cf. premier point de cette partie).
- Depuis 2008, je participe au projet Pharo (`www.pharo-project.org`) qui vise à développer un système Smalltalk libre et innovant. Dans le cadre de ce projet, je participe très régulièrement à des sessions de ré-ingénierie notamment lors des conférences d'ESUG.

Responsabilités scientifiques

Responsabilités pour l'organisation de congrès :

- Co-program chair de IWST'12 à Gent, Belgique (événement en coopération avec ACM).
- Co-program chair de IWST'11 à Edimbourg (événement en coopération avec ACM).
- Vice-président du comité d'organisation de l'école d'été temps réel 2011 (environ 60 participants)
- Co-organisateur (local chair) pour la 14th international conference on Reliable Software Technologies - Ada Europe, Juin 2009 (environ 120 participants). Conférence internationale de rang A dont les actes sont publiés par LNCS/Springer-Verlag.
- Co-responsable de l'organisation de International Smalltalk Conference, ESUG en 2010 , 2011 et 2012 (150 participants environ par an)
- Co-organisateur local de la conférence annuelle d'ESUG en 2009 à Brest (150 participants)

– Membre du comité de sélection UBO en 2010.

– Membre suppléant de la commission de spécialistes section 27 de septembre 2006 à juillet 2008, élu assesseur de cette commission en 2008.

Responsabilités de Master :

– De 2004 à 2010 et depuis octobre 2011, je suis président de jury du Master 2 Technologies de l'information de Brest. J'ai été porteur du projet au démarrage du Master 2 en 2004, puis, en 2010, avec Philippe Saliou, j'ai adapté cette formation à un dispositif d'alternance (25/30 contrats de professionnalisation/d'alternance par an).

– Directeur adjoint de la filière Ingénierie à Brest depuis la rentrée 2011 (environ 110 étudiants). Cette filière comporte trois parcours et deux spécialités de Master.

– De 2004 à 2010 et depuis octobre 2011, membre du conseil de filière Ingénierie.

Brevets ou licences :

– Logiciel Cheddar. Dépôt auprès de l'Agence de Protection des Programmes sous le numéro IDDN.FR.001.260026.000.S.P.2008.000.10600. Novembre 2008.

Vulgarisation

– En 2010, j'ai participé au projet exploratoire *Recapa* porté l'équipe WSN du Labsticc. Ce projet a réunit plusieurs disciplines pour chercher des directions de développement et préparer des projets communs, du capteur au système d'information. Le but était d'associer des compétences disciplinaires complémentaires pour trouver des applications des réseaux de capteurs sans fil, et les connaissances fondamentales permettant de les mettre en œuvre pratiquement. Dans le cadre de ce projet, j'ai participé à la conception d'une série de poster pour le salon du développement durable qui s'est tenu à Brest en 2010 [2] et invité des acteurs académiques et industriels du domaine à présenter leurs travaux et applications au cours de plusieurs séminaires.

– En 2007 et 2008, j'ai écrit plusieurs articles sur la programmation Smalltalk sous Squeak (programmation XML et interfaces graphiques) [3].

2. http://wsn.univ-brest.fr/ForumDurable/wsn.univ-brest.fr_ForumDurable/ForumDurable.html
3. http://community.ofset.org/index.php/Programmer_avec_Squeak