Langage et méthode pour une ingénierie des modèles fiable

Franck Fleurey

▶ **To cite this version:**

Franck Fleurey. Langage et méthode pour une ingénierie des modèles fiable. Génie logiciel [cs.SE]. Université Rennes 1, 2006. Français. NNT : . tel-00538288

HAL Id: tel-00538288
https://theses.hal.science/tel-00538288

Submitted on 22 Nov 2010

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

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

N° d'ordre: 3409

THÈSE

présentée

devant l'Université de Rennes 1

pour obtenir

le grade de : Docteur de l'Université de Rennes 1
Mention Informatique

par

Franck Fleurey

Équipe d'accueil : Triskell - IRISA
École Doctorale : Matisse
Composante universitaire : IFSIC

Titre de la thèse :

Langage et méthode pour
une ingénierie des modèles fiable

À soutenir le 9 octobre 2006 devant la commission d'examen

M. :	Jean-Pierre	Banâtre	Président
MM. :	Stéphane	Ducasse	Rapporteurs
	Michel	Riveill	
MM. :	Benoit	Baudry	Examinateurs
	Jean-Marc	Jézéquel	
	Pierre-Alain	Muller	
	Yves	Le Traon	

Remerciements

Je voudrais tout d'abord remercier les membres du jury. Merci à Stéphane Ducasse et Michel Riveill d'avoir accepté de rapporter cette thèse. J'ai apprécié la lecture en profondeur que vous avez faite du document ainsi que vos commentaires qui m'ont permis d'améliorer le document. Je tiens également à remercier Jean-Pierre Banâtre de m'avoir fait l'honneur de présider le jury.

Je remercie Jean-Marc Jézéquel, responsable de l'équipe Triskell qui m'a accueilli dans son équipe tout d'abord en stage lors de ma scolarité à l'IFSIC, puis en DEA et enfin en thèse. Je remercie chaleureusement Yves Le Traon de m'avoir fait découvrir la recherche en m'offrant mon premier stage dans ce domaine et d'avoir dirigé mon travail pendant ces trois années de thèse. J'ai beaucoup apprécié la confiance et l'autonomie qu'il m'a ainsi accordé.

Je remercie tout particulièrement Benoit Baudry qui, au delà de l'ami, du collègue et du voisin de bureau, a été, et ce, tout au long de ma thèse, un encadrant très rigoureux. Merci pour les discussions, les plans et plannings de travail, ainsi que les relectures et corrections détaillées du manuscrit et des articles. Je tiens également à remercier Pierre-Alain Muller pour le regard extérieur qu'il a apporté à son arrivée dans l'équipe, ses encouragements et son enthousiasme concernant les travaux sur Kermeta.

Je souhaite remercier Didier Vojtisek pour son important travail de coordination et de développement autour du projet Kermeta. Je remercie également tous les developpeurs ayant participés a Kermeta et en particulier Zoé Drey dont l'importante contribution a fait de Kermeta plus qu'un simple prototype.

Je remercie également mes collègues doctorants Jacques Klein et Jim Steel pour les discussions enrichissantes que nous avons pu avoir. De plus, je remercie tous les doctorants "passé et présent" de l'équipe Triskell. Les aînés Clémentine Nebut, Tewfik Ziadi et Damien Pollet nous ont montré le chemin et aidé à surmonter les obstacles ; et la relève est brillamment assurée par Franck Chauvel, Sebastien Saudrai, Jean-Marie Mottu, Erwan Brottier et Romain Delamare.

Sur un plan plus personnel, je remercie mon oncle Claude de m'avoir fait découvrir très tôt le domaine de l'informatique et d'avoir représenté la famille lors de la soutenance. Je remercie Marion d'avoir assuré la mission de restaurer l'ensemble des personnes ayant assisté à la soutenance. Enfin je remercie tout spécialement Valentine pour le soutien constant qu'elle m'a apporté au cours de ces trois années et pour la patience dont elle a fait preuve pour corriger mes innombrables fautes d'orthographe.

Table des matières

Chapitre 1

Introduction

Depuis les débuts du génie logiciel, la taille et la complexité des logiciels développés augmentent de plus en plus rapidement alors que les contraintes de temps de développement, de qualité, de maintenance et d'évolution sont toujours plus fortes. Dans ce contexte, les techniques de génie logiciel sont contraintes à évoluer sans cesse pour permettre de gérer la complexité et d'assurer la qualité du logiciel produit. Cette évolution se fait de manière continue depuis la création d'assembleurs et de compilateurs dans les années 1950 jusqu'à l'apparition de plateformes de programmation orienté-objets à la fin des années 1990. Cette évolution se traduit par une augmentation progressive du niveau d'abstraction auquel sont développés les systèmes logiciels. Dans la continuité, les grandes tendances actuelles du génie logiciel sont l'utilisation de langages dédiés, les approches orientées-aspects et les approches à base de composants. Un des points communs à ces approches est l'utilisation de modèles pour permettre une montée en abstraction par rapport aux langages de programmation.

Ces techniques sont regroupées sous l'intitulé Ingénierie Dirigée par les Modèles (IDM) ou Model-Driven Development (MDD). Ce terme reflète l'évolution du processus de développement logiciel d'une utilisation "contemplative" à une utilisation "productive" des modèles. Là où les modèles étaient utilisés comme des éléments de conception, de discussion ou de documentation, l'idée de l'IDM est de les exploiter comme entrée du processus de développement. En pratique, cela nécessite de formaliser les modèles pour les rendre exploitables par la machine et de réaliser des programmes permettant de traiter des modèles. Dans la terminologie de l'IDM, ces programmes sont regroupés sous le terme de *transformations de modèles*. Les deux originalités de l'IDM sont donc d'une part des modèles plus formels, et d'autre part des programmes de transformations de modèles. Afin d'assurer la qualité du processus de développement, et donc la qualité du logiciel produit, il faut assurer la qualité des modèles et la correction des transformations utilisées.

Pour chaque nouvelle technique de développement, il est nécessaire de proposer des techniques et un processus de fiabilisation adaptés. Dans les années 1990, un effort important a été porté à la définition de techniques de conception [RBP+91, Jéz96, GHJV95] et de test adaptées au paradigme orienté-objets [Bin99]. Ces techniques ont

permis une diffusion et une utilisation massive de la programmation orientée-objets. De la même manière, nous pensons que pour une large adoption de l'IDM, il est nécessaire de proposer de nouvelles approches pour sa fiabilisation. En pratique, cela signifie définir des méthodes de conception, des langages spécifiques, des techniques de vérification et de validation et des outils adaptés. Dans cette thèse nous nous concentrons sur *la définition d'un langage spécifique et de techniques de test pour l'IDM*. Dans les paragraphes suivants, nous détaillons le contexte, la problématique et les contributions de cette thèse.

Afin de préciser le contexte de cette thèse, nous représentons sur la figure 1.1 les trois principaux acteurs indispensables à tout développement logiciel utilisant des modèles. La partie droite de la figure schématise un processus d'ingénierie dirigée par les modèles (IDM). Les parties centrale et gauche de la figure détaillent les pré-requis de ce processus de développement : les environnements et outils indispensables à l'exploitation productive des modèles. Dans ce document, nous désignons sous le terme d'*ingénierie des modèles* l'ensemble des activités liées à cette exploitation productive des modèles. Les paragraphes suivants détaillent le rôle et les relations qui existent entre chacun des acteurs représentés sur la figure 1.1.

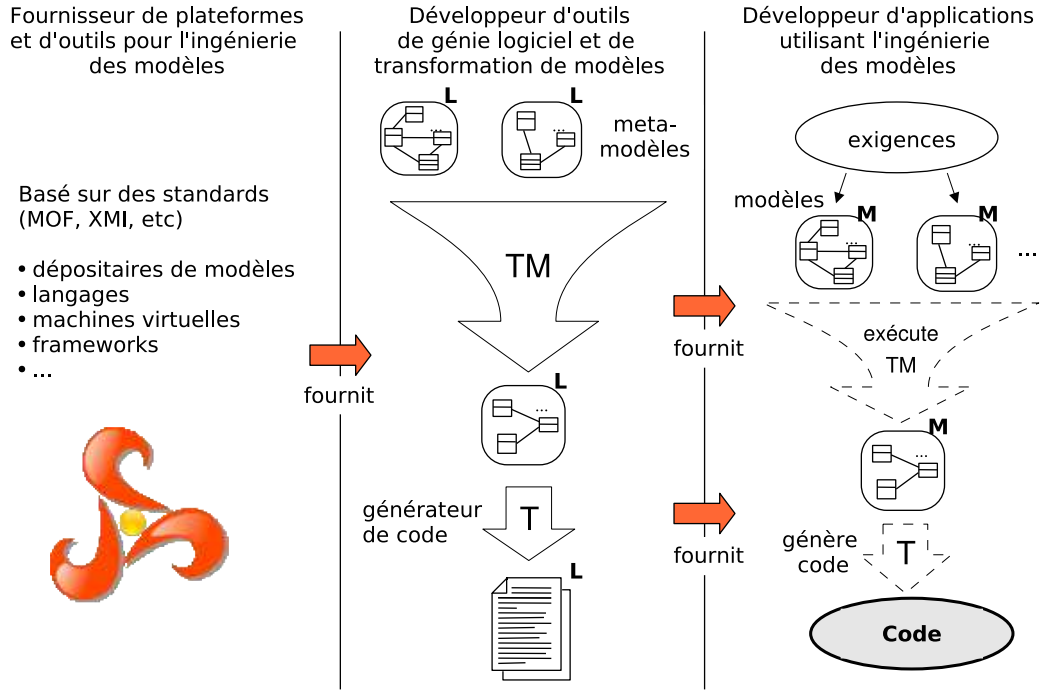


FIG. 1.1 – Trois principaux acteurs de l'ingénierie des modèles.

Ingénierie dirigée par les modèles La partie droite de la figure 1.1 schématise un processus de développement utilisant l'ingénierie des modèles. L'idée est que le développeur commence par modéliser son système à partir des exigences. Lors

de cette phase de modélisation, il utilise plusieurs langages (ou méta-modèles), fournis par le développeur d'outils de génie logiciel, afin de capturer au mieux les différents aspects du système. Pour passer de ces modèles d'analyse à un modèle de conception, le développeur édite les modèles et utilise des transformations sur étagère qui permettent de tisser, de composer ou de raffiner les différents modèles de l'application. Il peut ensuite, par exemple, utiliser un générateur de code afin de produire tout ou partie de l'implantation de l'application.

Outils de génie logiciel La partie centrale de la figure 1.1 présente les développeurs d'outils de génie logiciel et de transformations de modèles. Ces développeurs utilisent les normes et les plateformes génériques d'ingénierie des modèles pour créer des outils de génie logiciel destinés à un domaine d'application particulier. Le rôle de ces développeurs est de définir des langages ou méta-modèles spécifiques, de proposer des éditeurs permettant d'éditer les modèles correspondants et de rendre ces modèles exploitables en fournissant des transformations de modèles permettant, par exemple, de générer du code. A titre d'exemple, un certain nombre d'ateliers de génie logiciel s'appuient sur le langage UML et ses différents diagrammes, proposent des modeleurs UML et des transformations de modèles permettant d'appliquer des patrons de conception ou de générer du code.

Plateformes de modélisation La partie gauche de la figure 1.1 représente les communautés, les organismes de normalisation et les développeurs de plateformes qui sont à la base de l'ingénierie des modèles. Leur rôle est de définir les briques de base des techniques fondées sur les modèles. Ainsi, ils doivent, par exemple, définir précisément ce que l'on entend par modèle, méta-modèle, transformation, et les relations qui existent entre ces éléments. Une fois les éléments de base, leurs sémantiques et leurs relations clairement définies, par exemple dans des normes, l'objectif est de proposer des outils et des plateformes qui implantent ces normes. Le *langage Kermeta*, présenté dans le chapitre 3 est un exemple d'une telle plateforme.

L'objectif premier de l'ingénierie des modèles est d'améliorer la productivité du développement logiciel en s'appuyant sur l'utilisation de langages dédiés et de transformations de modèles. Implicitement, on attend donc de l'ingénierie des modèles qu'elle garantisse des niveaux de qualité satisfaisant et au moins équivalents à ce que permet la pratique actuelle. Sur la figure 1.1 cela signifie que l'objectif est d'assurer la qualité du code produit par le développeur d'applications (en bas à droite de la figure). Pour cela il faut fiabiliser le processus global, ce qui consiste à :

- fiabiliser les modèles et programmes produits par ce développeur. Ce premier aspect doit être assuré par l'utilisation de techniques de conception orienté-objet et de test à partir des modèles.
- fiabiliser les transformations de modèles et les outils de génie logiciel utilisés.

Dans ce document nous nous intéressons au dernier point. La difficulté principale tient à une différence majeure entre les techniques de développements classiques et les techniques d'IDM. Dans le premier cas, on utilise généralement un seul langage et les seuls outils susceptibles d'introduire des erreurs dans les programmes sont le compila-

teur et l'environnement d'exécution de ce langage. Dans le cas de l'IDM, les langages utilisés sont multiples et spécifiques à chaque domaine d'application. Afin d'assurer la qualité des applications produites, il est nécessaire d'assurer la fiabilité des outils de génie logiciel utilisés (éditeurs, transformations de modèles, etc.). En effet, la simple défaillance d'un de ces outils peut suffire à fragiliser le processus de développement dans son ensemble. Il est donc nécessaire de s'intéresser (1) à des techniques permettant d'assurer la qualité des langages de modélisation et (2) à la validation des programmes de transformations de modèles.

Un environnement pour l'ingénierie des modèles (à gauche sur la figure 1.1) doit permettre de définir des langages de modélisation et des transformations de modèles. Dans la pratique actuelle, des formalismes hétérogènes sont utilisés pour la définition de la structure et de la sémantique des langages et des transformations de modèles. L'hétérogénéité des formalismes utilisés est un obstacle à la fiabilité du processus de développement car la non interopérabilité entre ces formalismes rend difficile toute vérification relative à la cohérence entre les différents artefacts d'un projet. La première contribution de cette thèse est un langage dédié à l'ingénierie des modèles qui permet d'exprimer de façon homogène les différents artefacts d'un projet. Ce langage, baptisé Kermeta [MFJ05], est au coeur d'une plateforme d'ingénierie des modèles développée dans l'équipe Triskell. Kermeta a été construit par extension du langage de méta-données EMOF et se présente sous la forme d'un langage orienté-objets doté de bonnes caratéristiques pour la qualité (typage statique, conception par contrats, framework de test, etc.). En plus de constructions classiques des langages orienté-objets comme les classes, l'héritage multiple ou la liaison dynamique, le langage Kermeta inclut des constructions spécifiques à l'ingénierie des modèles comme les associations. Au cours de cette thèse, le langage Kermeta a été validé dans différents contextes [MFV$^+$05, KF06, RFG$^+$05, NF05, NFLTJ06].

Pour la validation de programmes de transformation de modèles (au milieu sur la figure 1.1), il est nécessaire d'utiliser toutes les techniques de vérification et de test disponibles. Bien que les transformations de modèles soient des programmes, les techniques existantes de test de programmes sont difficiles à appliquer et insuffisantes en raison de la complexité des données manipulées. De plus, les transformations de modèles ont des caractéristiques propres, comme le fait que les données manipulées soient des modèles, qui permettent de développer des techniques de test spécifiques. La seconde contribution de cette thèse est une technique de test de transformations de modèles. Cette technique [FSB04, BFS$^+$06] s'appuie sur les méta-modèles qui fournissent une description formelle des domaines d'entrée et de sortie des transformations de modèles afin de définir des critères de test. Ces critères de test sont ensuite utilisés pour générer des modèles de test pour la validation de transformations de modèles.

La suite de ce document est organisée en cinq chapitres. Tout d'abord, le chapitre 2 rappelle le contexte dans lequel se placent nos travaux. L'objectif de ce chapitre est à la fois de fournir une introduction aux domaines de l'ingénierie des modèles et du test de logiciel mais aussi de positionner nos travaux par rapport à l'existant. L'ensemble des concepts importants utilisés dans les chapitres suivants est défini dans ce chapitre. Le chapitre 3 détaille la conception et la réalisation du langage Kermeta au coeur d'une

plateforme d'ingénierie des modèles. Le chapitre 4 présente quatre études de cas menées pour valider le langage Kermeta. Le chapitre 5 présente les travaux que nous avons mené sur le test de programmes de transformations de modèles. Enfin, le chapitre 6 conclut ce document et propose un ensemble de perspectives. Ces travaux ont donné lieu à des publications dans des journaux et conférences internationales (cf page 215).

Chapitre 2

Contexte et état de l'art

Ce chapitre détaille le contexte des travaux présentés dans ce document. L'objectif de cette thèse est de proposer des techniques pour améliorer la qualité des processus de développement basés sur les modèles. Cet état de l'art est donc naturellement divisé en deux axes : l'ingénierie des modèles et la validation de logiciels. La section 2.1 présente l'ingénierie des modèles d'un point de vue général et détaille les techniques employées pour la définition de modèles et de transformations de modèles. La section 2.2 présente les principes du test de logiciel et détaille un ensemble de travaux pour l'automatisation de la génération de données de test. Enfin, la section 2.3 présente des travaux existants pour la fiabilisation des processus de développement basés sur les modèles.

2.1 Ingénierie des modèles

L'Ingénierie dirigée par les modèles (IDM) [EFB+05] propose des pistes pour permettre aux organisations de surmonter la mutation des exigences du développement de logiciel. L'IDM est une forme d'ingénierie générative, par laquelle tout ou partie d'une application informatique est générée à partir de modèles. Les idées de base de cette approche sont voisines de celles de nombreuses autres approches du génie logiciel, comme la programmation générative, les langages spécifiques de domaines (DSL pour *domain-specific language*) [vDKV00, CM98], le MIC (Model Integrated Computing) [SKB+95, SK97], les usines à logiciels (Software Factories) [GSC+04], etc.

Dans cette nouvelle perspective, les modèles occupent une place de premier plan parmi les artefacts de développement des systèmes, et doivent en contrepartie être suffisamment précis afin de pouvoir être interprétés ou transformés par des machines. Le processus de développement des systèmes peut alors être vu comme un ensemble de transformations de modèles partiellement ordonné, chaque transformation prenant des modèles en entrée et produisant des modèles en sortie, jusqu'à obtention d'artefacts exécutables.

Pour donner aux modèles cette dimension opérationnelle, il est essentiel de spécifier leurs points de variations sémantiques, et donc aussi de décrire de manière précise les langages utilisés pour les représenter. On parle alors de meta-modélisation.

L'intérêt pour l'Ingénierie Dirigée par les Modèles a été fortement amplifié, en novembre 2000, lorsque l'OMG (*Object Management Group*) a rendu publique son initiative MDA [StOs00a] qui vise à la définition d'un cadre normatif pour l'IDM. Il existe cependant bien d'autres alternatives technologiques aux normes de l'OMG, par exemple Ecore dans la sphère Eclipse, mais aussi les grammaires, les schémas de bases de données, les schémas XML. Finalement, l'IDM dépasse largement le MDA, pour se positionner plutôt à la confluence de différentes disciplines.

Cette section d'état de l'art sur l'ingénierie des modèles a deux objectifs. Le premier est de préciser le contexte et les travaux existant dans ce domaine. Le second est de définir un certain nombre de concepts qui sont utilisés dans les chapitres suivants de ce document. La section 2.1.1 présente un tour d'horizon des différentes approches de développement basées sur l'utilisation de modèles. La section 2.1.2 définit les notions de modèle, méta-modèle et méta-méta-modèle qui sont largement utilisées dans la suite du document. La section 2.1.3 présente rapidement les principaux langages de méta-modélisation existants et s'attarde sur les concepts propre à ces langages. Enfin, la section 2.1.5 définit la notion de transformation de modèle et présente les différentes techniques de transformations existantes.

2.1.1 Les approches de développement basées sur les modèles

Dans le courant des langages spécifiques et de la programmation générative, un certain nombre d'approches basées sur les modèles se développent depuis une décennie. Parmi ces approches, et par ordre chronologique d'apparition, on peut citer le Model-Integrated Computing (MIC), le Model-Driven Architecture (MDA) et les Software Factories. Les sous-sections qui suivent présentent les grandes lignes de chacune de ces approches.

Model-Integrated Computing (MIC)

Les travaux autour du Model-Integated Computing (MIC) [SKB+95, SK97] proposent, dès le milieu des années 90, une vision du génie logiciel dans laquelle les artefacts de base sont des modèles spécifiques au domaine d'application considéré. Initialement, le MIC est conçu pour le développement de systèmes embarqués complexes [KSLB03]. Dans [LBM+01] les auteurs discutent de l'implantation et de l'outillage des idées du MIC.

La méthodologie proposée décompose le développement logiciel en deux grandes phases. La première phase est réalisée par des ingénieurs logiciels et systèmes. Elle consiste en une analyse du domaine d'application ayant pour but de produire un environnement de modélisation spécifique à ce domaine. Au cours de cette phase, il faut choisir les paradigmes de modélisation et définir formellement les langages de modélisation qui seront utilisés. A partir de ces éléments, un ensemble de méta-outils sont utilisés pour générer un environnement spécifique au domaine. Cet environnement a l'avantage d'être utilisable directement par des ingénieurs du domaine. La seconde phase consiste à modéliser l'application à réaliser en utilisant l'environnement généré. Cette phase est

réalisée uniquement par des ingénieurs du domaine et permet de synthétiser automatiquement le logiciel.

En pratique, les idées du MIC sont implantées dans la suite d'outils (GME) [Dav03] (The Generic Modeling Environment). Dans sa dernière version (5.0), GME s'intègre à l'environnement de développement Visual Studio .NET 2003 de Microsoft et propose des outils pour la création de langages de modélisation et de modèles. La définition de la sémantique des langages de modélisation, et donc la semantique des modèles manipulés, n'est pas supportée directement : l'interprétation sémantique des modèles doit être réalisée dans une phase aval.

Model-Driven Architecture (MDA)

Le Model-Driven Architecture (MDA) [StOS00b] est une démarche de développement à l'initiative de l'OMG (Object Management Group) rendue publique fin 2000. L'idée de base du MDA est de séparer les spécifications fonctionnelles d'un système des spécifications de son implémentation sur une plate-forme donnée. Pour cela, le MDA définit une architecture de spécifications structurée en modèles indépendants des plates-formes (Platform Indepandant Models, PIM) et en modèles spécifiques (Platform Specific Models, PSM).

L'approche MDA permet de réaliser le même modèle sur plusieurs plates-formes grâce à des projections normalisées. Elle permet aux applications d'interopérer en reliant leurs modèles et supporte l'évolution des plates-formes et des techniques. La mise en oeuvre du MDA est entièrement basée sur les modèles et leurs transformations.

L'initiative MDA de l'OMG a donné lieu à une importante normalisation des technologies pour la modélisation. La proposition initiale était d'utiliser le langage UML (Unified Modeling Langage) et ses différentes vues comme unique langage de modélisation. Cependant, il a fallu rapidement ajouter la possibilité d'étendre le langage UML, par exemple en créant des profils, afin de permettre d'exprimer de nouvelles notions. Ces extensions devenant de plus en plus importantes, la communauté MDA a progressivement délaissé UML au profit d'approches utilisant des langages spécifiques. Le langage de méta-modélisation MOF, et plus récemment MOF 2.0, a été créé pour permettre la spécification de méta-modèles, c'est-à-dire de langages de modélisation. Il est intéressant de noter que, par exemple, UML est construit à partir de MOF.

Les usines logicielles (Software Factories)

Les usines logicielles (Software Factories) [GSCK04] sont la vision de Microsoft de l'ingénierie des modèles. Le terme d'usine logicielle fait référence à une industrialisation du développement logiciel et s'explique par l'analogie entre le processus de développement proposé et une chaine de montage industrielle classique. En effet, dans l'industrie classique :

– Une usine fabrique une seule famille de produits. Si l'on considère par exemple l'industrie automobile, une chaîne de montage ne fabrique généralement qu'un seul type de voiture avec différentes combinaisons d'options.

- Les ouvriers sont relativement spécialisés. Il n'est pas rare de trouver des ouvriers ayant des compétences sur plusieurs postes de la chaîne de montage mais il est très rare qu'un ouvrier ait toutes les compétences depuis l'assemblage à la peinture des véhicules.
- Les outils utilisés sont très spécialisés et fortement automatisés. Les outils utilisés sur une chaîne de montage sont conçus uniquement pour cette chaîne de montage ce qui permet d'atteindre des degrés d'automatisation importants.
- Toutes les pièces assemblées ne sont pas fabriquées sur place. Une chaîne de montage automobile ne fait généralement qu'un assemblage de pièces normes ou préfabriquées à un autre endroit.

L'idée des usines logicielles est d'adapter ces caractéristiques aux développements logiciels, qui ont fait leur preuve pour la production de masse de familles de produits matériels. Les deux premiers points correspondent à la spécialisation des fournisseurs de logiciels et des développeurs à l'intérieur des équipes de développement. Le troisième point correspond à l'utilisation d'outils de génie logiciel spécifiques au domaine d'application, c'est-à-dire de langages, d'assistants et transformations spécifiques. Le dernier point correspond à la réutilisation de composants logiciels sur étagères. L'environnement de développement Visual Studio .NET 2005 de Microsoft a été conçu autour de ces idées et propose un environnement générique extensible pouvant initialement être configuré pour un ensemble de domaine d'application prédéfinis.

Conclusion

L'ensemble des approches décrites dans la section précédente partagent l'utilisation des modèles comme base pour le processus de développement. Pour chacune de ces approches de développement, l'utilisation de modèles nécessite la définition de langages de modélisation et de transformations de modèles. L'*ingénierie des modèle* est la disipline qui s'intéresse à ces problèmes. La section suivante définit avec précision ce que l'on entend par modèle dans le contexte de l'ingénierie des modèles.

2.1.2 Modèles, méta-modèles et méta-méta-modèle

Cette section définit les concepts de base de l'ingénierie des modèles et les relations qui existent entre ces concepts. Les définitions données dans cette section s'appuient sur les résultats de l'action spécifique MDA du CNRS [EFB+05]. La suite de ce document s'appuie fortement sur ces définitions.

Modèles et Systèmes

La première notion importante est la notion de *modèle*. Quelle que soit la discipline scientifique considérée, un modèle est une abstraction d'un système construite dans un but précis. On dit alors que le modèle *représente* le système. Un modèle est une abstraction dans la mesure où il contient un ensemble restreint d'informations sur un système. Il est construit dans un but précis dans la mesure où les informations qu'il

contient sont choisies pour être pertinentes vis-à-vis de l'utilisation qui sera faite du modèle.

A titre d'exemple, une carte routière est un modèle d'une zone géographique conçu pour circuler en voiture dans cette zone. Pour être utilisable, une carte routière ne doit inclure qu'une information très synthétique sur la zone cartographiée : une carte à l'échelle 1, bien que très précise, ne serait d'aucune utilité.

Méta-modèle : langage de modélisation

Afin de rendre un modèle utilisable il est nécessaire de préciser le langage de modélisation dans lequel il est exprimé. On utilise pour cela un *méta-modèle*. Ce méta-modèle représente les concepts du langage de méta-modélisation utilisé et la sémantique qui leur est associée. En d'autres termes, le méta-modèle décrit le lien existant entre un modèle et le système qu'il représente. Dans le cas d'une carte routière, le méta-modèle utilisé est donné par la légende qui précise, entre autres, l'échelle de la carte et la signification des différents symboles utilisés.

On dit qu'un modèle est *conforme* à un méta-modèle si l'ensemble des éléments du modèle sont définis par le méta-modèle. Cette notion de conformité est essentielle à l'ingénierie des modèles mais n'est pas nouvelle : un texte est conforme à une orthographe et une grammaire, un programme JAVA est conforme au langage JAVA et un document XML est conforme à sa DTD.

Méta-méta-modèle : langage de méta-modélisation

De la même manière qu'il est nécessaire d'avoir un méta-modèle pour interpréter un modèle, pour pouvoir interpréter un méta-modèle il faut disposer d'une description du langage dans lequel il est écrit : un méta-modèle pour les méta-modèles. C'est naturellement que l'on désigne ce méta-modèle particulier par le terme de *méta-méta-modèle*. En pratique, un méta-méta-modèle détermine le paradigme utilisé dans les méta-modèles et que les modèles qui sont construits à partir de lui. De ce fait, le choix d'un méta-méta-modèle est un choix important et l'utilisation de différents méta-méta-modèles conduit à des approches très différentes du point de vue théorique et du point de vue technique.

A titre d'exemple, on peut citer deux approches différentes. Tout d'abord, EBNF qui permet de définir des grammaires qui jouent le role de méta-modèles et dont les mots jouent le rôle de modèles. Ensuite, XML pour lequel la DTD joue le rôle de méta-modèle pour des documents XML pouvant alors être considérés comme des modèles. Le formalisme retenu dans les approches à base de modèles, tels que MIC, MDA ou les usines logicielles, est basé sur le paradigme orienté-objets. Dans la suite de ce document nous ne considérons donc que les langages de méta-modélisation orientés-objets.

Afin de se soustraire au problème de la définition des méta-méta-modèles (et ainsi éviter d'avoir à définir un méta-méta-méta-modèle), l'idée généralement retenue est de concevoir les méta-méta-modèles de sorte qu'ils soient auto-descriptifs, c'est-à-dire capables de se définir eux-mêmes. Dans le cas des grammaires, on spécifie ainsi EBNF

par une grammaire et dans le cas d'XML, c'est une DTD XML pour les DTD qui joue le rôle de méta-méta-modèle.

Résumé

Pour résumer, la figure 2.1 représente les différentes relations qui existent entre les modèles, méta-modèles et méta-méta-modèles. On distingue sur cette figure les trois niveaux de modélisation M1, M2 et M3 correspondant respectivement au modèle, méta-modèle et méta-méta-modèle. Dans la littérature, et en particulier dans certain documents de l'OMG, il est fait état d'un niveau M0 correspondant au système que représente un modèle du niveau M1. La nature de la relation qui existe entre le niveau M0 et M1 est donc différente de la nature des relations existant entre les niveaux M1 et M2 et M2 et M3. Pour éviter ce problème d'homogénéité, le niveau M0 n'est généralement pas considéré.

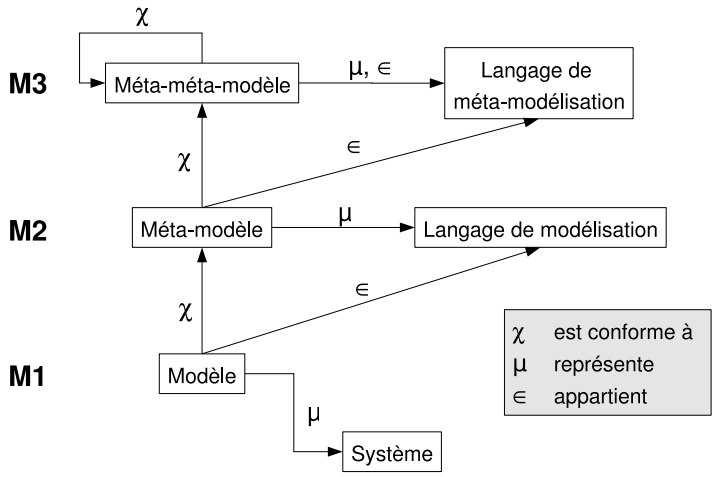


FIG. 2.1 – Niveaux de modélisation.

On retrouve également, sur la figure, les trois relations de base de l'ingénierie des modèle : la conformité (χ), la représentation (μ) et l'appartenance ($\in$). Cela permet de mettre en évidence les relations qui existent entre modèles et langages : un modèle est exprimé dans un langage de modélisation (c'est-à-dire qu'il appartient à ce langage) et est conforme au modèle qui représente ce langage. Cela permet également de préciser ce que l'on entend par auto-définition (ou *boot-strap*) pour le méta-méta-modèle : le méta-méta-modèle représente le langage dans lequel il est exprimé, il est donc conforme à lui même.

2.1.3 Méta-modélisation orientée-objets

Comme évoqué précédemment, les principes de base de l'ingénierie des modèles ne sont pas récents et l'architecture en trois couches décrite dans la section précédente à été

mise en oeuvre pour différents paradigmes tel que les grammaires, XML ou les langages dédiés. Dans ce document, nous nous restreignons aux approches orientés-objets dont l'émergence est plus récente et qui constitue le cadre de travail de ce document.

Les principaux langages de méta-modélisation existants, dans ce domaine sont MOF 1.4 [OMGa], CMOF , EMOF et ECore [ECo]. MOF (Meta-Objet Facilities) 1.4 a été normalisé par l'OMG lors de la construction d'UML 1.4. EMOF (pour Essential MOF) et CMOF (pour Complete MOF) [OMGc] sont deux évolutions de MOF 1.4 définies lors de la normalisation de UML 2.0. Comme leurs noms l'indique, la différence entre ces deux langages réside dans le nombre de concepts qu'ils contiennent. CMOF contient un ensemble de concepts plus important que EMOF qui lui ne contient que les concepts essentiels à la méta-modélisation. Enfin, le langage ECore n'est pas une norme mais le langage de méta-modélisation défini par IBM et utilisé dans le framework de modélisation d'Eclipse EMF (Eclipse Modeling Framework) [EMF]. En pratique, ECore a été défini avant la fin du processus de normalisation de EMOF mais a depuis été aligné sur le norme EMOF.

Parmi les langages de méta-modélisation existants, nous avons retenu EMOF pour les travaux présentés dans ce document. Les deux raisons du choix de EMOF sont, d'une part, le fait que ce langage soit normalisé et d'autre part le fait qu'il soit bien supporté par des outils comme Eclipse/EMF [EMF]. Cette section utilise un exemple simple de langage de méta-modélisation orientée-objets et discute des principales constructions présentes dans les langages de méta-modélisation orientés-objets. Toutes les constructions présentes dans EMOF sont définies en détail.

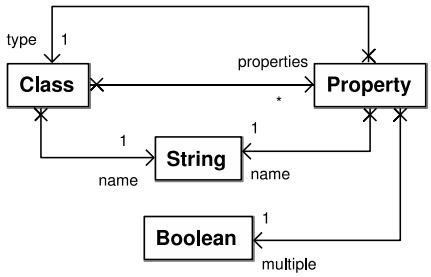


FIG. 2.2 – Diagramme de classes d'un méta-méta-modèle orienté-objets minimal.

La figure 2.2 présente un méta-méta-modèle orienté-objets minimaliste en utilisant la notation UML pour les diagrammes de classes. Ce méta-méta-modèle correspond à un langage de méta-modélisation très simple qui permet de définir des classes (class *Class*) et des propriétés (class *Property*). Chaque classe a un nom et un ensemble de propriétés. Chaque propriété a un nom (propriété *name*), un type (propriété *type*) qui fait référence à une classe et un booléen *multiple* qui spécifie si l'attribut fait référence à un objet unique ou à un ensemble d'objets. Dans ce langage, il est possible de créer un ensemble de classes et de relations entre ces classes à travers la notion de propriété. Un des avantages de ce langage est qu'il permet de se décrire lui même. Afin de s'en convaincre, la figure 2.3 présente un diagramme d'objets qui montre comment les classes

du méta-méta-modèle de la figure 2.2 sont instanciées pour se reproduire lui-même.

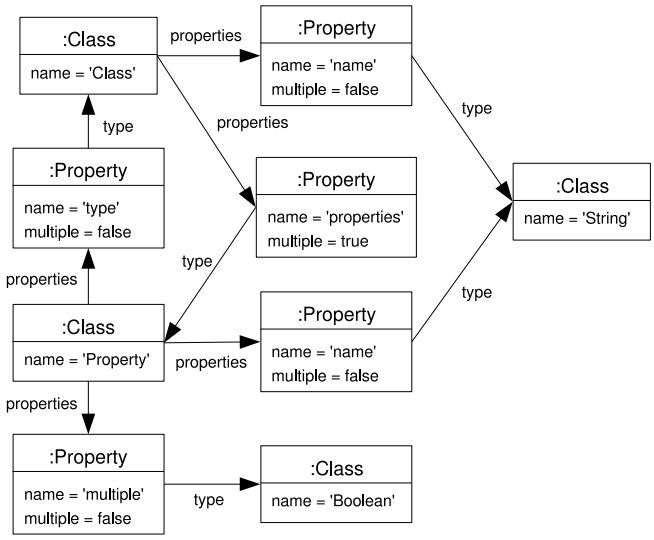


FIG. 2.3 – Diagramme d'objets correspondant au méta-méta-modèle orienté-objets de la figure 2.2.

Mis à part le fait qu'il puisse se définir lui-même, ce méta-méta-modèle permet de définir des méta-modèles. Par exemple, la figure 2.4 présente un méta-modèle simplifié pour des catalogues de produits. Dans la suite de cette section, nous utiliserons cet exemple pour illustrer les discussions. Un catalogue regroupe un ensemble de produits classés dans des catégories. Le méta-modèle fait donc apparaître trois classes correspondants aux catalogues (classe *Catalog*), aux catégories (classe *Category*) et aux produits (classe *Product*). Chaque concept est concrètement traduit en une classe dans le méta-modèle. Les relations entre ces concepts sont réalisés grâce aux propriétés. Dans la classe *Catalog* on a par exemple ajouté une propriété *category* qui permet à un catalogue de faire référence à un ensemble de catégories.

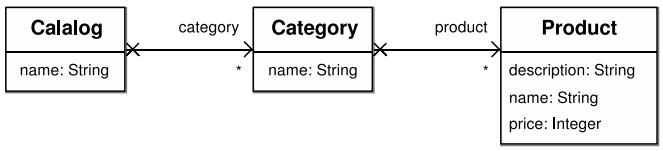


FIG. 2.4 – Méta-modèle de catalogues conforme au méta-méta-modèle de la figure 2.2

Une fois le méta-modèle défini, il est possible de créer des modèles en instanciant les classes du méta-modèle. La figure 2.5 montre, par exemple, un extrait de catalogue comportant deux catégories et trois produits. On a maintenant un exemple des trois niveaux de méta-modelisation évoqué précédemment :

– le méta-méta-modèle sur la figure 2.2,

– un méta-modèle sur le figure 2.4,
– un modèle sur la figure 2.5.

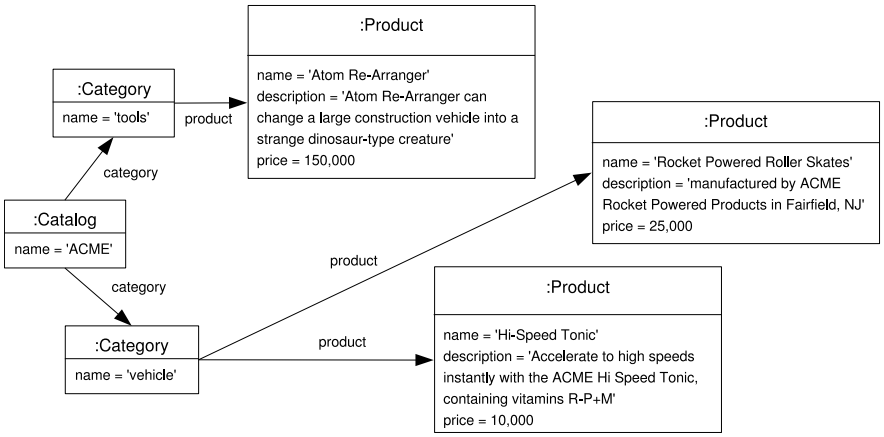


FIG. 2.5 – Modèle de catalogue conforme au méta-modèle de la figure 2.4.

Le langage de méta-modélisation utilisé dans l'exemple précédent (c'est-à-dire le langage correspondant au méta-méta-modèle de la figure 2.2) est limité dans la mesure où il ne propose que les concepts de classes et de propriétés. L'expressivité de ce langage est approximativement équivalente à celle d'un schéma de bases de données : les classes correspondent à des tables et les propriétés à des colonnes (qui peuvent contenir des clefs étrangères). En pratique un certain nombre concepts supplémentaires, comme l'héritage, les associations ou les compositions, a été proposé pour améliorer l'expressivité des langages de méta-modelisation orientée-objets. Les paragraphes suivants détaillent ces concepts en insistant sur les concepts présents dans le langage EMOF utilisé dans la suite de ce document.

Héritage et classes abstraites

La première amélioration, intégrée dans tous les langages orientés-objets, est l'héritage. L'héritage a fait ses preuves dans le domaine de la programmation orienté-objet et constitue un des mécanismes de base de la modelisation orientée-objets. L'héritage est intéressant pour deux raisons :

– Il permet de factoriser les caractéristiques communes à plusieurs classes. Dans l'exemple du méta-modèle de catalogues, cela permet par exemple, comme le montre la figure 2.6 de factoriser les attributs *name* présents dans chacune des classes en créant une super-classe abstraite *NamedElement* contenant une propriété *name*.
– Il induit une relation de sous-typage qui permet le polymorphisme. Sur l'exemple présenté sur la figure 2.6 cela permet de définir plusieurs types de produits (des imprimantes et des disques durs) dont les caractéristiques sont différentes. La

relation de sous-typage induite par l'héritage permet à une catégorie d'être en relation avec les différents types de produits.

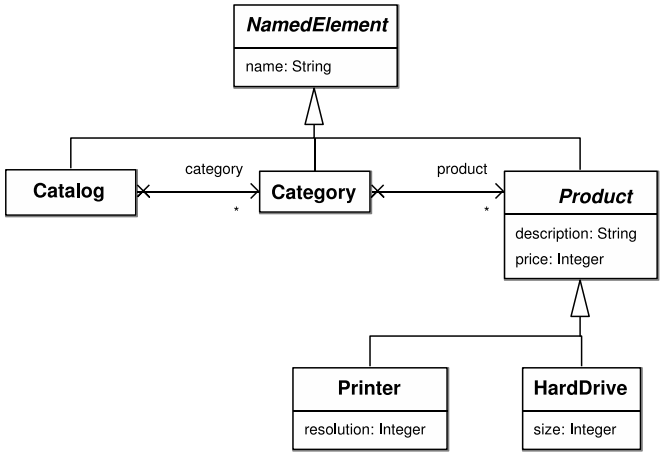


FIG. 2.6 – Méta-modèle de catalogues utilisant l'héritage.

Association

Le second concept présent dans la plupart des langages de méta-modélisation est le concept d'association. Contrairement à l'héritage qui est issu des langages de programmation orienté-objets, les associations sont typiques des langages de modélisation. Dans les langages orientés-objets classiques (et dans le langage de méta-modélisation présenté précédemment), les relations entre classes sont spécifiées grâce à des propriétés (ou références). Une propriété possède simplement un nom et un type et peut éventuellement faire référence à plusieurs objets. Pour permettre d'exprimer des relations entre classes plus complexes, les associations réifient ces relations pour les enrichir d'informations supplémentaires tels que des noms de rôle, des multiplicités ou des informations de navigabilité.

Pour l'exemple du méta-modèle de catalogue, la figure 2.7 montre l'utilisation d'associations entre les classes *Catalog* et *Category* et entre les classes *Category* et *Product*. Ces associations permettent de spécifier des noms de rôles et des multiplicités pour chacune des extrémités des relations. Il apparait clairement sur ce nouveau méta-modèle que chaque catégorie est propre à un catalogue mais que les produits peuvent apparaître dans plusieurs catégories.

Dans les langages de méta-modélisation orientée-objets, les associations sont modélisées de façons différentes et présentent une expressivité plus ou moins grande. Les principaux éléments de spécification d'une relation communs aux langages de méta-modélisation sont :

– Les noms de rôles qui nomment les extrémités d'une relation. Ces noms de rôles sont utilisés pour naviguer les relations.

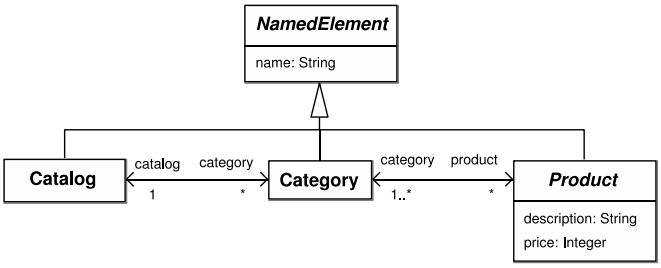


FIG. 2.7 – Méta-modèle de catalogues utilisant des associations.

- Les multiplicités qui spécifient les nombres d'objets pouvant être associés pour chaque extrémité d'une relation. Une multiplicité est généralement composée d'une borne inférieure positive ou nulle, et d'une borne supérieure positive ou infinie.
- Dans le cas où la multiplicité de l'extrémité d'une relation permet de mettre en relation plusieurs objets, il est possible de spécifier si ces objets sont ordonnés ou pas, et si un objet peut être présent plusieurs fois.
- Une association peut être dérivée, dans ce cas la relation n'existe pas physiquement mais est calculée au besoin à partir d'autres éléments.

En plus de ces quatre éléments, certains langages comme par exemple CMOF [OMGc] proposent des associations bien plus complexes qui permettent de représenter des relations ternaires ou n-aires, de spécifier des relations telles que l'inclusion entre plusieurs relations, etc.

Aggrégation et composition

Les agrégations et les compositions sont des associations particulières. Elles ont initialement été proposées dans UML. Les symboles graphiques qui leur sont associés sont un diamant blanc pour les agrégations et un diamant noir pour les compositions. Dans UML, la sémantique de ces relations était initialement informelle et souvent ignorée par les utilisateurs. Depuis, des travaux ont permis de préciser et formaliser ces relations [BHSPLB03]. Dans les langages de méta-modélisation comme EMOF, seule la relation de composition est conservée.

La composition est une association particulière qui permet de spécifier des relations de "contenance" entre objets. Lorsqu'un objet o_1 est en relation de composition avec un objet o_2, on dit que l'objet o_1 contient l'objet o_2. La principale contrainte liée à cette relation est que chaque objet ne peut être contenu que par un seul autre. De plus, les cycles d'objets par la relation de composition sont interdits (la relation de contenance est transitive). C'est là tout l'intérêt de cette relation par rapport aux associations qui engendrent un graphe quelconque entre les objets d'un modèle. La structure engendrée par la relation de composition est nécessairement un arbre. En pratique, cet arbre permet de faciliter le parcours, la sérialisation, la visualisation ou l'édition des modèles. Le norme de sérialisation XMI 2.0 [OMGb] s'appuie par exemple sur la relation de

composition pour construire la représentation XML d'un modèle.

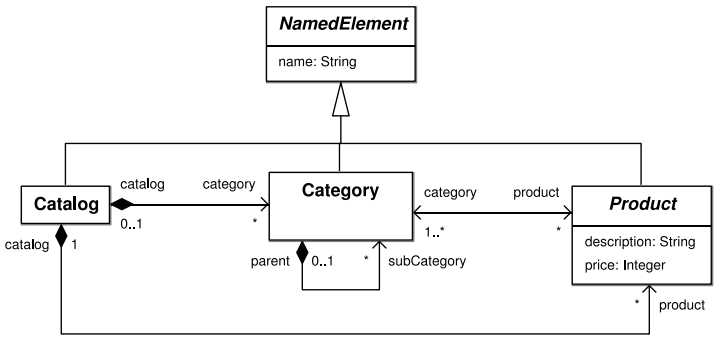


FIG. 2.8 – Méta-modèle de catalogues utilisant des compositions.

La figure 2.8 présente un méta-modèle de catalogue utilisant des compositions. Un catalogue contient des catégories (composition de la classe *Catalog* vers la class *Category*) et des produits (composition de la classe *Category* vers la class *Product*). Dans ce méta-modèle nous avons ajouté la possibilité de définir des sous-catégories. Chaque catégorie peut donc contenir des sous-catégories grâce à la relation de composition définie sur la classe *Category*. Comme un objet ne peut être contenu que par un seul autre, en utilisant notre méta-modèle, chaque catégorie sera contenue soit par un catalogue soit par une catégorie parente.

En pratique, la composition se révèle être une construction très utile des langages de méta-modélisation car un grand nombre d'outils l'utilisent pour la structure arborescente des modèles qu'elle engendre. Si la relation de composition est correctement utilisée, c'est-à-dire dans le sens de la contenance, l'arbre obtenu est sémantiquement intéressant : chaque objet encapsule les objets qu'il contient. Les bonnes pratiques de méta-modélisation consistent à concevoir les méta-modèles afin que tous les objets qui sont sémantiquement contenus par un autre le soient explicitement par une relation de composition. Dans le cas du méta-modèle de catalogue, nous avons par exemple ajouté une relation de composition entre un catalogue et les produits qu'il contient.

Remarque 2.1 *Pour utiliser certains outils founis avec le framework Eclipse/EMF [EMF], il est nécéssaire que chaque modèle ait un seul objet racine par la relation de composition. Dans ce cas, le méta-modèle doit être conçu pour assurer cette contrainte. Le méta-modèle de la figure 2.8 est satisfaisant si chaque modèle ne contient qu'un seul catalogue. Si l'on souhaitait construire des modèles contenant plusieurs catalogues, on pourrait ajouter une classe CatalogSet pour contenir l'ensemble des catalogues de chaque modèle.*

Opération et comportement

Dans les langages de programmation orienté-objets, le comportement des objets est spécifié par des opérations dans les classes. La dynamique de chaque opération est

généralement décrite en utilisant un langage impératif comme c'est le cas dans les langages Java, Eiffel ou C♯. Dans les langages de méta-modélisation comme EMOF, Ecore ou CMOF, il est possible de définir des opérations sur les classes mais aucun support n'est prévu pour la spécification du comportement de ces opérations. Initialement, ces langages ont été conçus pour la description structurelle de méta-données et non pour la description du comportement ou de la sémantique de ces données. Dans la pratique il faut avoir recours à d'autres langages pour spécifier le comportement des opérations. Les langages utilisés sont soit une langue naturelle, soit un langage de programmation généraliste.

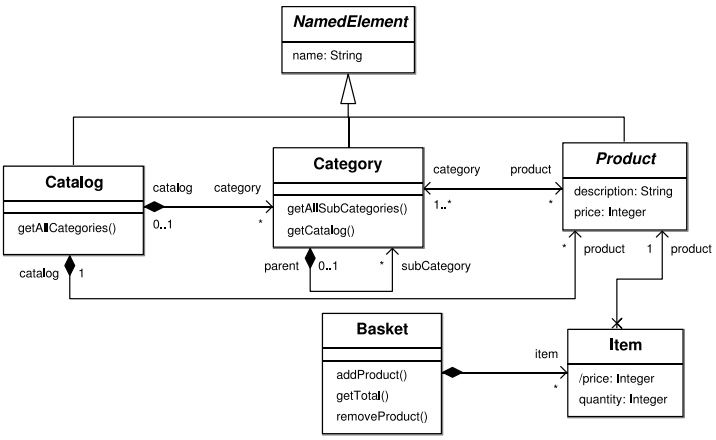


FIG. 2.9 – Méta-modèle de catalogues avec des opérations et propriétés dérivées.

Les deux points d'ancrage pour la sémantique et le comportement que proposent les langages de méta-modélisation comme EMOF sont les opérations et les propriétés dérivées. En ce qui concerne les opérations, seule leur signature peut être définie, c'est-à-dire un nom, un ensemble de paramètres typés et un type de retour. Les propriétés dérivées sont un équivalent des *getter* et *setter* utilisés en Java ou des *Properties* définies en C♯. Elles permettent de définir des propriétés qui au lieu de faire référence directement à une donnée permettent de calculer cette donnée. Lors de la définition d'une propriété dérivée, il est nécessaire de spécifier comment la valeur de cette propriété est calculée. A l'utilisation, les propriétés dérivées sont identiques aux autres propriétés. Dans les langages de méta-modélisation comme EMOF, il est possible de spécifier qu'une propriété est dérivée mais, comme pour les opérations, rien ne permet de spécifier le calcul à effectuer.

La figure 2.9 représente le méta-modèle de catalogue enrichi d'opérations et de propriétés dérivées. Afin d'illustrer le propos, nous avons ajouté au méta-modèle les classes *Basket* et *Item* qui représentent le panier d'un éventuel acheteur. Un panier (*Basket*) contient un ensemble d'articles (*Item*) qui correspond à un produit et une quantité. Dans la classe *Basket* nous avons défini des opérations permettant d'ajouter et d'enlever des produits du panier et de calculer le total du panier. Sur la classe *Item*,

la propriété dérivée *price* permet de calculer le prix de l'article en fonction du prix et de la quantité demandée. Du point de vue graphique, le / devant le nom de la propriété *price* spécifie que la propriété est dérivée.

Le langage EMOF

EMOF inclut tous les concepts de modélisation présentés dans les sections précédentes : héritage multiple, associations, compositions, opérations et propriétés dérivées. Cette section présente rapidement le langage EMOF et ses particularités.

La figure 2.10 présente le méta-méta-modèle EMOF dans son ensemble. EMOF permet de définir des packages qui contiennent des types. Les types EMOF sont soit des *DataType* (types primitifs ou types énumérés), soit des classes. Les classes EMOF peuvent avoir un nombre quelconque de super-classes (propriété *superClass* de la classe *Class*) et sont composées de propriétés et d'opérations. En tant que langage de description de structures de données, EMOF ne permet pas de spécifier de corps pour les opérations. Le système de type de EMOF est très simple : une classe définit un type et le sous-typage n'est possible que grâce au sous-classage.

Une des raisons pour lesquelles EMOF présente peu de concepts par rapport à d'autres langages de méta-données est la représentation "compacte" qui est utilisée pour les attributs, les références, les associations et les compositions. En effet, la notion de propriété (classe *Property* sur la figure 2.10) permet de représenter tous ces éléments. Une propriété est définie par :

- un identifiant (*name*) qui doit être unique parmi les propriétés de la classe à laquelle elle appartient.
- un type (*type*) qui spécifie le type des éléments que peut contenir la propriété. Le type d'une propriété peut être une classe, un type primitif ou une énumération.
- une multiplicité (*lower*, *upper*, *isOrdered* et *isUnique*) qui précise combien d'éléments peut contenir la propriété et si ces éléments doivent être uniques et/ou ordonnés.
- un booléen *isComposite* qui spécifie si la propriété correspond à une composition.
- un booléen *isDerived* qui spécifie que la propriété est dérivée. Dans ce cas la propriété ne contient pas directement de valeur mais permet de calculer une valeur.
- une éventuelle propriété opposée (*opposite*). Un couple de propriétés opposées permet de représenter une association bi-directionnelle entre deux classes.

Remarque 2.2 *Il est important de noter que les* ***propriétés*** *définies dans EMOF sont des structures complexes qui permettent de représenter des références mais aussi des associations bi-directionnelle et de compositions. Dans la suite de ce document, cette notion de propriété est largement utilisée.*

Bien que EMOF ne permette pas de décrire le comportement des opérations, le modèle de la figure 2.10 comporte des opérations dont la sémantique est décrite en anglais dans la norme définissant le langage. Ces opérations définissent comment les classes définies en EMOF sont instanciées pour former des modèles.

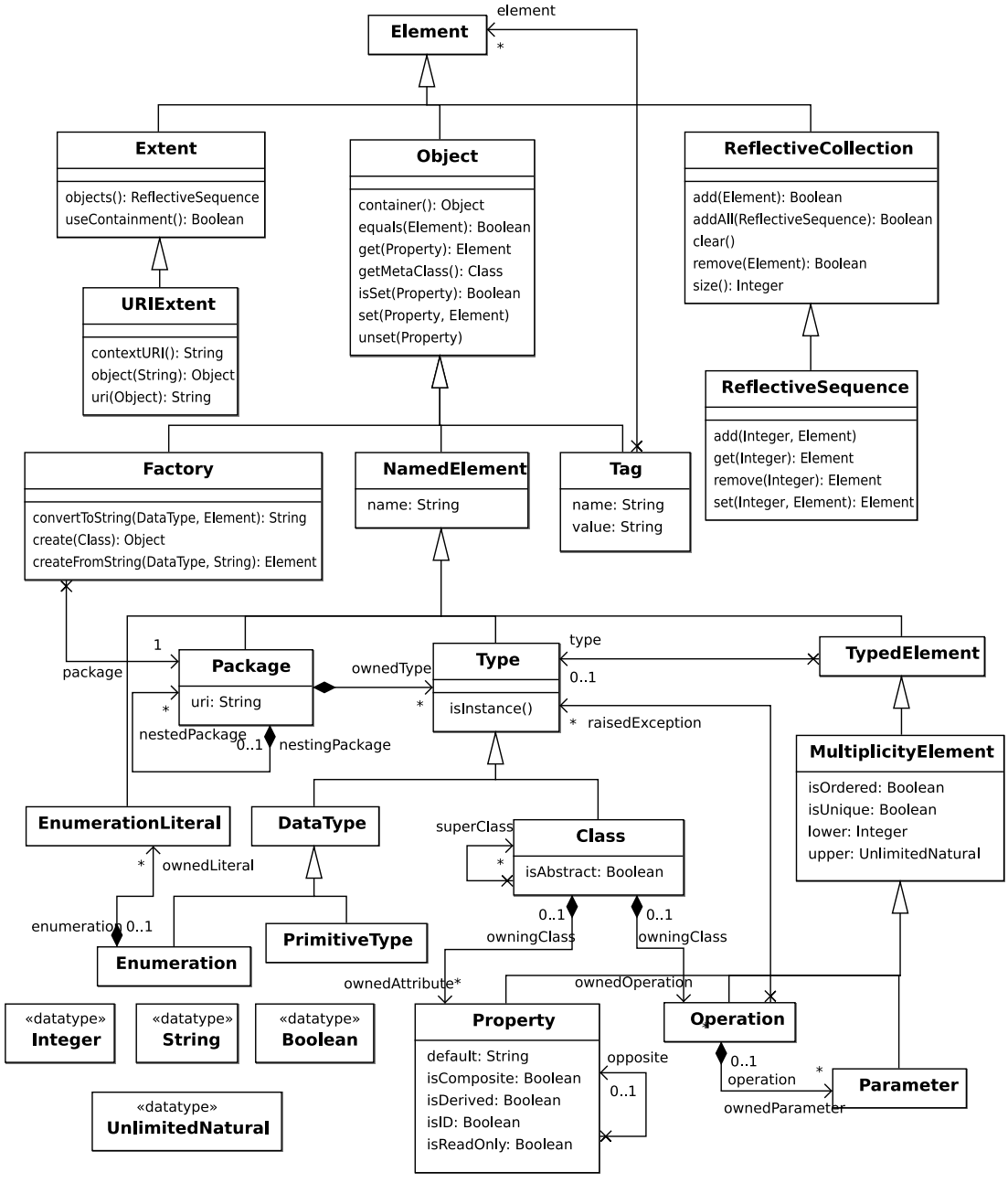


FIG. 2.10 – Essential Meta-Object Facilities (EMOF).

L'opération *create* la classe *Factory* permet de créer des instances des classes EMOF. La classe *Object* de EMOF définit la structure et les opération disponible pour tout objets EMOF. L'opération *getMetaClass* de la classe *Object* permet par exemple à partir de tout objet de retrouver sa méta-classe. Les opérations *get* et *set* permettent de consulter ou modifier les valeurs des propriétés d'une classe. Dans le cas ou une propriété a une multiplicité supérieure à un, les classes *ReflectiveCollection* et *ReflectiveSequence* sont utilisées pour représenter la collection de valeurs correspondant à cette propriété.

Les notions d'*Extent* et d'*URIExtent* définies dans EMOF sont les notions les plus proches de la notion de *Modèle*. Ces classes permettent simplement d'encapsuler un ensemble d'objets dans une collection sans offrir de réels services.

2.1.4 Expression de contraintes et utilisation d'OCL

Un méta-modèle permet de définir et donc de contraindre la struture des modèles qu'il décrit. Par exemple, les multiplicités des propriétés spécifient les nombres minimum et maximum d'objets qui peuvent participer à chaque association. Sur le méta-modèle de la figure 2.9, les multiplicités de l'association entre les classes *Category* et *Product* imposent que chaque produit soit référencé par au moins une catégorie. Cependant, les langages de méta-modélisation ne permettent pas d'exprimer toutes les contraintes relatives à un méta-modèle. Sur l'exemple de la figure 2.9, il est par exemple impossible de spécifier que les produits ne peuvent être en relation qu'avec des catégories d'un même catalogue. Cette contrainte étant relative à plusieurs associations du méta-modèle, elle n'est pas possible à assurer par construction.

En pratique il est fréquent que des contraintes doivent être ajoutées à un méta-modèle pour assurer la cohérence des modèles qu'il permet de définir. Les langages de méta-modélisation ne fournissent pas directement de solution à ce problème. Dans certains cas, les contraintes peuvent être simplement exprimées en langage naturel. Lorqu'il est nécessaire de vérifier automatiquement des contraintes, le langage de contraintes OCL (Object Constraint Language) [(OM03] est utilisé. Le langage OCL a été initialement conçu pour ajouter des contraintes sur les modèles UML sous la forme d'invariants de classes et de pré-conditions et post-conditions pour les opérations. Les diagrammes de classes d'UML et les méta-modèles MOF étant des formalismes très proches, OCL peut être directement réutilisé pour définir des invariants sur des classes MOF et des pré-conditions et post-conditions pour les opérations d'un méta-modèle.

Le langage OCL inclut un ensemble de constructions permettant de naviguer parmi les objets d'un modèle afin de vérifier des contraintes. Les constructions d'OCL ne permettent pas de créer, détruire ou modifier les objets d'un modèle : la vérification des contraintes se fait sans effet de bord. En plus de la définition de contraintes, OCL peut être utilisé pour spécifier le comportement de propriétés dérivées et d'opérations à condition qu'aucune modification du modèle n'intervienne.

Le listing de la figure 2.11 présente des exemples d'utilisation d'OCL pour le méta-modèle de la figure 2.9. Les lignes 1 à 5 définissent le corps de l'opération *getCatalog* définie sur la classe *Category*. Cette opération retourne le catalogue auquel appartient une catégorie. Les lignes 7 à 10 du listing spécifient un invariant pour la classe *Product*.

```
1  -- Définition du corps de l'opération getCatalog
2  context Category::getCatalog() : Catalog
3    body : if self.calalog->isEmpty() then parent.getCatalog()
4           else self.catalog
5           endif
6
7  -- Définition d'un invariant de la classe Product
8  context Product
9    inv : self.category->forall { c | c.getCatalog() = self.catalog }
10
11 -- Définition de la propriété dérivée price
12 context Item::price : Integer
13   derive : quantity * product.price
```

FIG. 2.11 – Exemple d'utilisation d'OCL

Cet invariant assure que tout produit est en relation avec un ensemble de catégories qui appartient au catalogue dans lequel le produit est défini. Enfin, les lignes 11 à 13 du listing spécifient la propriété dérivée *price* de la classe *Item*.

La limitation à l'utilisation d'OCL pour spécifier le corps des opérations des méta-modèles est qu'il est impossible de spécifier des modifications modèles. Il est par exemple impossible de spécifier le corps des opérations *addProduct* et *removeProduct* de la class *Basket* du méta-modèle de la figure 2.9. OCL fournit donc une bonne solution pour exprimer des contraintes sur les modèles mais pas pour décrire complètement le comportement des modèles. Un des objectif du langage Kermeta, proposé dans le chapitre 3, est de permettre à la fois la définition de contraintes et la spécification du corps des opérations et propriétés dérivées.

2.1.5 Transformation de modèles

Les deux principaux artefacts de l'ingénierie des modèles sont les modèles et les transformations de modèles. D'un point de vue général, on appelle *transformation de modèles* tout programme dont les entrées et les sorties sont des modèles. Cette section propose un rapide tour d'horizon des techniques de transformation de modèles existantes. De nombreux outils, tant commerciaux que dans le monde de l'open source sont aujourd'hui disponibles pour faire la transformation de modèles. On peut grossièrement distinguer quatre catégories d'outils :

1. les outils de transformation génériques qui peuvent être utilisés pour faire de la transformation de modèles ;
2. les facilités de type langages de scripts intégrés à la plupart des ateliers de génie logiciel ;
3. les outils conçus spécifiquement pour faire de la transformation de modèles et prévus pour être plus ou moins intégrables dans les environnements de développement normalisés ;

4. les outils de méta-modélisation "pur jus" dans lesquels la transformation de modèles revient à l'exécution d'un méta-programme.

Outils génériques

Dans la première catégorie on trouve notamment d'une part les outils de la famille XML, comme XSLT [W3C05b] ou Xquery [W3C05a], et d'autre part les outils de transformation de graphes (la plupart du temps issus du monde académique) [EGdL+05]. Les premiers ont l'avantage d'être déjà largement utilisés dans le monde XML, ce qui leur a permis d'atteindre un certain niveau de maturité. En revanche, l'expérience montre que ce type de langage est assez mal adapté pour des transformations de modèles complexes (c'est-à-dire allant au-delà des problématiques de transcodage syntaxique), car ils ne permettent pas de travailler au niveau de la sémantique des modèles manipulés mais simplement à celui d'un arbre couvrant le graphe de la syntaxe abstraite du modèle ce qui impose de nombreuses contorsions qui rendent rapidement ce type de transformation de modèles complexes à élaborer, à valider et surtout à maintenir sur de longues périodes.

Outils intégrés aux AGLs

Dans la seconde catégorie, on va trouver une famille d'outils de transformation de modèles proposés par des vendeurs d'ateliers de génie logiciel. Par exemple, l'outil Arcstyler [Hub] de Interactive Objects propose la notion de MDA-Cartridge qui encapsule une transformation de modèles écrite en JPython (langage de script construit à l'aide de Python et de Java). Ces MDA-Cartridges peuvent être configurées et combinées avant d'être exécutées par un interpréteur dédié. L'outil propose aussi une interface graphique pour définir de nouvelles MDA-Cartridges. Dans cette catégorie on trouve aussi l'outil Objecteering [Sof] de Objecteering Software (filiale de Softeam), qui propose un langage de script pour la transformation de modèles appelé J, ou encore OptimalJ [Com] de Compuware qui utilise le langage TPL, et bien d'autres encore, y compris dans le monde de l'open source avec des outils comme Fujaba [BGN+04] (From UML to Java and Back Again).

L'intérêt de cette catégorie d'outils de transformation de modèles et d'une part leur relative maturité (car certains d'entre eux comme le J d'Objecteering sont développés depuis plus d'une décennie) et d'autres pas leur excellente intégration dans l'atelier de génie logiciel qui les héberge. Leur principal inconvénient est le revers de la médaille de cette intégration poussée : il s'agit la plupart du temps de langages de transformation de modèles propriétaires sur lesquels il peut être risqué de miser sur le long terme. De plus, historiquement ces langages de transformation de modèles ont été conçus comme des ajouts au départ marginaux à l'atelier de génie logiciel qui les héberge. Même s'ils ont aujourd'hui pris de l'importance, ils ne sont toujours pas vus comme les outils centraux qu'ils devraient être dans une véritable ingénierie dirigée par les modèles. De nouveau ces langages montrent leur limitation lorsque les transformations de modèles deviennent complexes et qu'il faut les gérer, les faire évoluer et les maintenir sur de

longues périodes.

Langages spécifiques

Dans la troisième catégorie, c'est-à-dire les outils conçus spécifiquement pour faire de la transformation de modèles et prévus pour être plus ou moins intégrables dans les environnements de développement standards, on va trouver par exemple Mia Transformation [MS] de Mia-Software qui est un outil qui exécute des transformations de modèles prenant en charge différents formats d'entrée et de sortie (XMI, tout autre format de fichiers, API, dépôt). Les transformations sont exprimées comme des règles d'inférence semi-déclaratives (dans le sens où il n'y a pas de mécanisme de type de programmation logique), qui peuvent être enrichies en utilisant des scripts Java pour des services additionnels tels que la manipulation de chaînes. PathMATE [Sol] de Pathfinder Solutions est un autre environnement configurable de transformation de modèles qui s'intègre particulièrement bien avec les ateliers de modélisation UML dominant le marché. Dans le monde académique on va trouver de nombreux projets open source s'inscrivant dans cette approche : les outils ATL [BRST05] et MTL [Pol05, VJ04] de l'INRIA, AndroMDA [And], BOTL [Bra03] (Bidirectional Object oriented Transformation Language), Coral [AP04] (Toolkit to create/edit/transform new models/modeling languages at run-time), ModTransf [Dum] (XML and ruled based transformation language), QVTEclipse [Tra] (une implantation préliminaire de quelques unes des idées de la norme QVT dans Eclipse) ou encore UMT-QVT [UQ] (UML Model Transformation Tool).

Outils de méta-modélisation

La dernière catégorie d'outils de transformation de modèles est celle des outils de méta-modélisation dans lesquels la transformation de modèles revient à l'exécution d'un méta-programme. Le plus ancien et le plus connu de ces outils est certainement MetaEdit+ [SLTM91, TR03] de MetaCase. Celui-ci permet de définir explicitement un méta-modèle, et au même niveau de programmer tous les outils nécessaires, allant des éditeurs graphiques aux générateurs de code en passant par des transformations de modèles. Dans une veine similaire, l'outil MetaGen [Rev96, RSBP95] propose un environnement pour l'édition de méta-modèles et un language a base de règles pour exprimer des transformations de modèles. Les méta-modèles sont compilé vers des classes Smalltalk qui peuvent être instanciées par un éditeur de modèles générique.

Plus récement, l'environnement XMF-Mosaic [Xac] de Xactium a été conçu comme un environnement complet pour la définition de langage. Au coeur de XMF-Mosaic on trouve un noyau exécutable de définition de langage (qui est d'ailleurs auto-définissant). Tout est modélisé sur cette base, que ce soit les langages (e.g. UML), les outils, les GUI, parsers et autres analyseurs XML, mais à la différence de l'environnement MetaEdit+ obtenu par génération plutôt que par instantiation. Le langage de méta-modélisation de base est un langage orienté objet qui a toute la puissance d'un langage de programmation complet ce qui en fait un outil particulièrement puissant pour la transformation

de modèles.

Le langage Kermeta, présenté dans le chapitre 3 fait partie de cette catégorie. Par rapports aux approches existantes et notament MetaEdit+, MetaGen ou XMF-Mosaic, la contribution de Kermeta est d'être statiquement typé, libre et conforme aux normes de méta-modélisation actuelles. Comme le détaillent les chapitres suivants, Le language Kermeta a été défini dans le but de prendre en compte les besoins liés à la validation et la vérification de méta-modèles et de transformations de modèles. La section 4.2 du chapitre 4 est consacrée à l'utilisation de Kermeta pour l'implantation de transformations de modèles.

2.2 Le test de logiciel

La seconde partie de ce chapitre d'état de l'art présente le test de logiciel et un ensemble de travaux de ce domaine. Le chapitre 5 de ce document présente des travaux sur le test de transformations de modèles, en particulier sur la génération de données de test pour les transformations de modèles. L'objectif de cette section est de définir la problématique et les concepts du test de logiciel et de placer nos travaux par rapport à l'existant.

Le test apparaît aujourd'hui comme une technique incontournable pour la validation des logiciels [Bin99, Bei90]. En effet quel que soit le domaine d'application ou la complexité des logiciels développés, le test est un complément indispensable aux techniques de vérification mises en oeuvre au cours du développement. Cette section présente brièvement les problèmes liés au test de logiciel et un tour d'horizon des approches existantes. Dans la pratique, le processus de test pour un logiciel est intimement lié aux techniques de développement utilisées. Ainsi, il est nécessaire de faire évoluer conjointement les techniques de développement et les techniques de test qui leur sont associées.

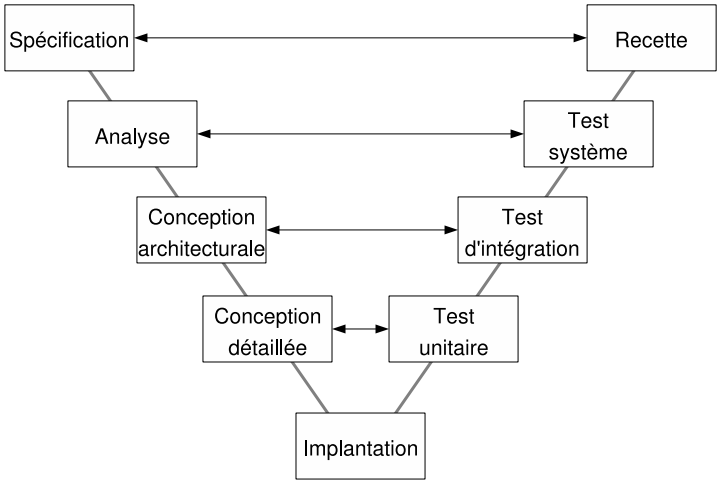


FIG. 2.12 – Cycle de développement en V.

A titre d'exemple, la Figure 2.12 rappelle le cycle de développement en V qui présente un parallèle entre les phases de développement et les phases de test qui leur sont associées. On distingue sur cette figure quatre phases de test successives :

- Tout d'abord le *test unitaire* consiste à tester l'implantation de chaque module logiciel séparément. S'il s'agit d'un logiciel orienté-objets, cela consistera, par exemple, à tester chaque classe séparément pour s'assurer de leur bon fonctionnement. Le test unitaire est généralement réalisé conjointement avec le développement du module sous test.
- Vient ensuite le *test d'intégration* qui a pour objectif de tester le bon fonctionnement des différents modules logiciels entre eux. Chaque sous-système est testé séparément afin de vérifier le bon fonctionnement des interactions entre les modules unitaires.
- Une fois le système intégré, l'étape de *test système* permet de vérifier le bon fonctionnement des services rendus par le système.
- La dernière phase de test est le *test de recette*. Son objectif est de tester la validité du système implanté par rapport à sa spécification initiale.

Quelle que soit la technique de développement ou la phase de test considérée, l'objectif du test est toujours de détecter des fautes. Pour cela il existe deux méthodes complémentaires : le test statique et le test dynamique. Lorsque l'on fait du test statique, on cherche à détecter des erreurs sans exécuter le programme. Le test statique passe par exemple par l'analyse statique de programmes ou la relecture de code. Le test dynamique consiste à exécuter le programme afin mettre en évidence des defaillances. La suite de cette section ne s'intéresse qu'au test dynamique.

2.2.1 Principes et problématiques du test

Comme le montre la figure 2.13, l'activité de test commence par l'élaboration de cas de test. Ensuite, ces cas de test sont confrontés à l'implantation, les résultats obtenus sont alors comparés avec ceux attendus. Il faut donc avoir déterminé au préalable un *oracle*, c'est-à-dire les sorties attendues. Si cet oracle met en évidence un défaut de l'implantation (i.e. au moins un des cas de test *échoue*), la faute à l'origine de cette défaillance doit être localisée (c'est le *diagnostic*) et corrigée. Si l'oracle ne détecte aucune défaillance du programme, c'est-à-dire si l'ensemble des cas de test passe, un *critère d'arrêt* permet de déterminer si l'ensemble de cas de test utilisés est suffisant pour garantir une certaine confiance dans la correction du programme sous test. En cas de non satisfaction du *critère d'arrêt*, d'autres cas de test sont générés pour compléter l'ensemble de cas de test initial. L'activité de test ne prend fin que lorsque tous les cas de test passent et que le critère d'arrêt est satisfait.

On distingue deux grandes familles de méthodes pour le test dynamique : le *test structurel* (ou *boîte blanche*) et le *test fonctionnel* (ou *boîte noire*). Dans le cadre du test structurel, l'implantation du programme sous test est connue et utilisée pour la génération des cas de test et l'obtention du critère d'arrêt. Le test fonctionnel, au contraire, est indépendant de l'implantation du programme sous test et seules ses entrées et sorties sont accessibles. La spécification est alors utilisée à la fois pour la génération des

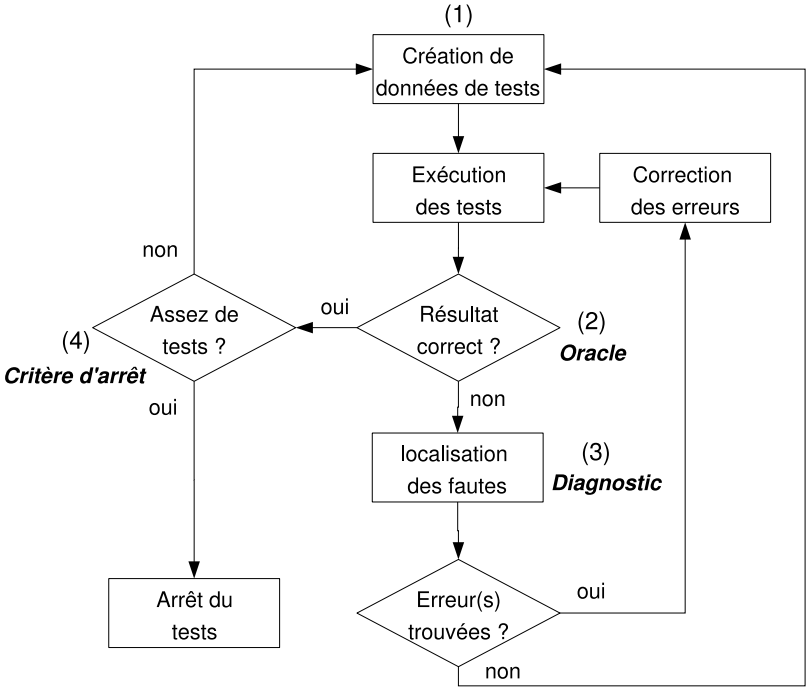


FIG. 2.13 – Processus de test.

données de test et l'obtention du critère d'arrêt. Dans les deux cas, l'oracle est obtenu en prédisant selon la spécification, les sorties attendues pour chaque cas de test.

Les section suivantes détaillent les principaux problèmes liés au test de logiciel : le critère d'arrêt, l'oracle, la minimisation de suite de test, la génération de données de test et la localisation d'erreurs.

2.2.2 Le critère d'arrêt

Le test a pour but de trouver des erreurs dans une implantation mais ne permet pas, dans le cas général, de s'assurer de la correction d'un programme. En effet, le seul moyen de prouver par le test la correction d'un programme est de le tester sur tout son domaine d'entrée *(test exhaustif)*. C'est généralement impossible car les domaines d'entrée des programmes usuels son soit bien trop grands, soit infinis. Il est nécessaire de choisir judicieusement un sous-ensemble du domaine d'entrée du programme pour le test : les *données de test*. Le rôle du critère de test (ou critère d'arrêt) est de définir comment choisir l'ensemble des données de test.

Définition 2.1 *Un* ***critère de test*** *est un critère qui doit être vérifié par l'ensemble des cas de test d'un programme à tester. L'utilisation d'un critère de test permet d'assurer le pouvoir de détection d'erreur des test utilisés. En pratique, les critère de tests peuvent être structurels ou fonctionnels.*

Dans le cadre du test structurel (ou boite blanche), c'est-à-dire lorsque que l'on s'appuie sur le code du programme à tester, des critères de test classiques sont la couverture du code ou la couverture du graphe de flot de contrôle. L'objectif est de trouver un ensemble de données en entrée du programme qui permette soit d'exécuter au moins une fois chaque instruction du programme soit de passer au moins une fois par un ensemble de chemins extrait du graphe de flot de contrôle. Dans le cadre du test fonctionnel (ou boite noire) l'idée est de s'appuyer sur la spécification du programme à tester pour la sélection des données de tests. Par exemple, pour les programmes dont le comportement est spécifié par des machines à états, il existe des critères de test spécifiques qui consistent à sélectionner des données qui permettent de passer au moins une fois par chaque état ou par chaque transition de la machine à état [ABDL02, OA99].

D'autres approches sont également possibles comme l'*analyse de mutation* [DLS78] par exemple. Le test par mutation consiste à créer une grande quantité de versions (ou *mutants*) du programme sous test en introduisant dans chacune une erreur élémentaire. L'hypothèse faite est que si une suite de test est capable de rejeter (ou *tuer*) tous les mutants alors cette suite de test est capable de détecter des erreurs plus complexes dans le programme original. Le critère d'arrêt s'exprime ici comme le pourcentage de mutants rejetés par la suite de test, appelé *score de mutation*.

2.2.3 Le problème de l'oracle

L'oracle est est une fonction qui permet de décider si un cas de test est passé ou a échoué (i.e. a détecté une erreur) en faisant le lien entre l'implantation et la spécification du programme sous test.

Définition 2.2 *Soit P une implantation sous test d'une spécification F. Soit $In(P)$ et $Out(P)$ les domaines d'entrée et de sortie de P. L'**oracle** est une fonction ok_F de $In(P) \times Out(P)$ à valeur booléenne tel que :*

$$\forall x \in In(P), ok_F(x, P(x)) = vrai \Longleftrightarrow P(x) = F(x)$$

Cet oracle peut être, dans la pratique, manuel ou automatique. La décision peut être prise par le testeur en examinant les entrées et sorties du programme mais ce travail peut devenir fastidieux si le testeur utilise de grandes suites de test ou si la spécification est complexe. Dans beaucoup de cas, il est donc préférable d'automatiser l'oracle.

Si l'on dispose d'une spécification exécutable du programme (ou d'une autre version du programme) on pourra par exemple comparer ses sorties avec celles de l'implantation sous test pour décider du succès d'un cas de test. L'oracle peut également dans certains cas être embarqué dans le code sous forme d'assertions. Cette idée est implantée dans certains langages orienté-objets sous forme de pré/post-conditions pour les méthodes, d'invariants de classe et de variants et invariants de boucle [Mey92, CL02].

Dans certain cas particuliers des oracles automatiques propre au domaine peuvent être utilisés. Si l'on teste, par exemple, un algorithme de compression sans perte, on pourra utiliser l'algorithme de décompression correspondant pour vérifier si l'intégrité est conservée par l'algorithme de compression.

Disposer d'un oracle efficace et automatique est primordial si l'on utilise des méthodes de génération de test nécessitant l'exécution d'un grand nombre de cas de test. Malheureusement, à notre connaissance peu de travaux existent dans ce domaine.

2.2.4 La réduction/minimisation de suite de tests

Dans les deux sections précédentes, nous avons détaillé la nécessité d'utiliser un critère de test et de disposer d'une fonction d'oracle. Pour des raisons de coût liées à la production de l'oracle, à l'exécution des test et à la réexécution des tests lors de la maintenance des logiciels, il est important de minimiser le nombre de cas de test utilisés pour satisfaire le critère de test. Harrold et al. [HGS93] formalise le problème de la réduction de suite de test ainsi :

Définition 2.3 *Problème de la **réduction/minimisation de suite de test** :*

Étant donné une suite de test TS et un ensemble d'objectif de test $r_0, r_1, \ldots, r_n$ couverts par TS, trouver un ensemble minimal de cas de test, inclus dans TS et couvrant tous les objectifs de test. Les objectifs de test r_i peuvent représenter n'importe quel critère de test : couverture d'instruction, de fonction, des arcs du graphe de contrôle, etc.

L'objectif de la réduction/minimisation de suite de test est de trouver un ensemble de cas de test, de cardinalité minimale, satisfaisant le critère de test (c'est-à-dire un ensemble d'objectifs de test), ce qui dans le cas général est un problème N-P complet. Les méthodes proposées sont donc des heuristiques permettant de réduire une suite de test sans toutefois garantir la minimalité de la suite de tests obtenue, mais en conservant

le critère de couverture retenu. Dans [HGS93], les auteurs proposent un algorithme permettant de réduire des suites de tests en éliminant les cas de test redondants.

La réduction/minimisation de suite de tests peut être utilisée pour sélectionner des cas de test après en avoir générer un grand nombre, mais aussi pour limiter le nombre de cas de test à exécuter lors de la maintenance du logiciel sous test. Le limites de la minimisation sont que, bien que l'on assure que le critère de test est toujours vérifié par la suite et test minimisée, il est possible que le pouvoir de détection d'erreurs de la suite de test minimisé soit moins bon que celui de la suite de test initiale [RUCH99]. Un autre inconvenient de la minimisation est que lors de la maintenance et de l'évolution d'un programme, la suite de test minimisée risque de ne plus vérifier le critère de test. En effet, des travaux ont montré [EGR01] que des modifications mineures à un programme induisent des modifications significatives et imprévisibles à la couverture de son code par une suite de tests.

Pour éviter la perte d'information liée à la minimisation de suites de test, lorsque l'on dispose d'une fonction d'oracle peu coûteuse à produire, le simple fait d'ordonner l'exécution des cas de test permet de diminuer le coût du test de non régression. Cette technique s'appelle la prioritisation de suite de tests [RUCH99]. L'idée est d'ordonner les cas de tests afin d'augmenter le *taux de détection de faute* de cette suite de tests. Le taux de détection de faute d'une suite de tests étant définie comme sa capacité à détecter le plus tôt possible lors de son exécution des erreurs dans le composant sous test. Cette technique est particulièrement utile dans le cadre du *test de non-regréssion*. Les travaux [EMR00, EMR02, RUCH01, RUCH99] définissent un grand nombre d'heuristiques utilisables pour prioritiser des suites de test.

2.2.5 La localisation d'erreurs

La localisation d'erreurs (ou *diagnostic*) consiste à déterminer, à partir de l'observation d'anomalies de fonctionnement d'une implantation, la partie défectueuse de cette implantation. C'est une des phases les plus coûteuses du processus de test/mise-au-point de programme lorsque celui-ci est réalisé entièrement à la main.

Dans la plupart des cas le diagnostic est facilité par l'utilisation de "debuggers" permettant au programmeur de tracer l'exécution du programme qu'il teste. Il peut ainsi visualiser l'évolution de l'état du programme, instruction par instruction, et localiser finement les causes des anomalies observées. Cela étant cette méthode a ses limites. En effet, un debugger n'offre qu'une vision très locale du programme en cours d'exécution et nécessite que l'utilisateur ait fait un effort de compréhension globale du programme qu'il teste. Pour répondre à ces problèmes des méthodes de diagnostic complémentaires existent, leur but est de réduire l'ensemble de instructions suspectes qu'il faudra inspecter.

Les techniques d'assistance au diagnostic ont pour objectif de déterminer un ensemble d'instructions suspectes dans un programme sous test. Pour cela l'idée est d'utiliser plusieurs exécutions du programme sous test parmi lesquelles certaines conduisent à un resultat correct et d'autres a un résultat erroné. Lorsqu'une exécution produit un résultat erroné, au moins une instruction erronée a été exécutée. Lorsque le résultat

obtenu est correct, cela n'implique pas que toute les instructions exécutées soit correcte mais cela contribue tout de même a renforcer la confiance que l'on a dans ces instructions.

La première technique basée sur ce principe de l'utilisation des traces d'exécutions à été proposée par Agrawal et al dans [AHLW95]. Cette technique utilise seulement deux exécutions à la fois : une correcte et une erronée. Les instructions identifiées comme suspectes sont celles qui sont exécutées uniquement par l'exécution menant à un résultat erroné. Cette technique a depuis été améliorée [JHS01, EHJS01, JHS02] pour utiliser toutes les exécutions disponibles et établir un classement des instructions de la plus suspecte à la moins suspecte.

Une des limitations des ces techniques de localisation d'erreurs est que pour permettre un diagnostic précis il faut disposer d'un grand nombre d'exécutions du programme sous test, c'est-à-dire utiliser un grand nombre de cas de test. Ce besoin est contradictoire avec la pratique du test que nous avons discuté précédemment et qui consiste à minimiser les suites de test utilisées afin de réduire le coût globale du test. Dans [BFLT06] nous avons proposé un critère pour la sélection des données de test pour le diagnostic. Ce critère permet d'optimiser une suite de test existante en ajoutant un minimum de cas de test destinés à améliorer les performances des algorithmes de diagnostic qui utilisent le recoupement de traces d'exécutions.

2.2.6 Génération de données de test

La génération des données de test est la première activité du test de logiciel. L'objectif de cette phase est de construire un ensemble de cas de test de taille minimale permettant de satisfaire un critère de test. L'utilisation d'un critère de test permet d'assurer le qualité des cas de test utilisés et la minimisation du nombre de tests permet de limiter le coût du test. Les sections suivantes détaillent différentes approches de génération automatique de test.

Techniques structurelles

Les techniques structurelles se basent sur le composant lui même pour la génération de cas de tests. On distingue, parmi les méthodes proposées, deux types de technique : statique ou dynamique. La génération statique consiste à générer des cas de test sans exécuter le composant sous test alors que la génération dynamique tire partie de l'exécution du composant sous test afin de générer et sélectionner des cas de test pertinents.

Les techniques de génération statiques de tests sont, dans la plupart des cas, fondées sur la résolution de contraintes sur les données en entrée. Dans [Cla76], par exemple, les auteurs utilisent l'exécution symbolique de programme qui consiste à donner à chaque variable du programme une valeur symbolique. L'exécution statique du programme permet ensuite d'inférer les contraintes sur les valeurs de variable permettant d'atteindre tel ou tel point du programme. Un solveur de contraintes permet ensuite de générer des données de test safisfaisant ces contraintes.

La génération dynamique de cas de test initialement proposée dans [MS76] consiste

à instrumenter le code du programme sous test pour collecter des informations sur le programme au cours de l'exécution d'un cas de test. Ces informations sont utilisées pour évaluer la qualité du cas de test par rapport à un critère donné. Les données de test sont ensuite modifiées pour tenter d'améliorer les cas de test afin qu'il satisfassent entièrement le critère choisi. De nombreux travaux sur la génération dynamique de cas test sont inspirés des travaux de Korel qui ramène ce problème à un problème de minimisation de fonction [Kor92]. L'idée est la suivante : un objectif de test peut s'exprimer sous la forme d'une fonction sur les données en entrée, trouver une donnée de test pertinente consiste alors à trouver une donnée qui minimise la fonction. Korel utilise la descente de gradient pour résoudre la minimisation de fonction. Un grand nombre de chercheurs ont utilisé les algorithmes génétiques pour résoudre ce problème [JHD96, Jon98, PHP99, MMS01, WBS01]. Ces travaux, que nous détaillons dans la suite, diffèrent par la modélisation des prédicats comme fonction des variables d'entrée.

Dans [MMS01], Michael et al. présentent leur outil GADGET (Genetic Algorithm Data GEneration Tool) qui génère des données de test scalaire. Les individus sont modélisés comme une chaîne de bits correspondant à la représentation binaire de la donnée de test et la fonction d'utilité correspond à la fonction à minimiser. L'article détaille la modélisation du problème et donne des résultats expérimentaux avec plusieurs petits programmes (30 lignes de code) et un programme plus important (2000 lignes de code). Au cours de ces expériences, les auteurs ont utilisé deux types d'algorithmes génétiques pour la génération de données de test. Les résultats sont comparés à la génération aléatoire, et à l'utilisation de la descente de gradient pour la minimisation de fonction. Parmi les trois algorithmes de génération, l'algorithme génétique classique est le plus efficace dans une large majorité des cas.

Pargas et al. [PHP99] utilisent aussi un algorithme génétique pour la génération dynamique de données de test. Ici, la fonction d'utilité est fondée sur la couverture du graphe de dépendance de contrôle (GDC). Les auteurs ont développé un prototype appelé TGen. L'outil génère une donnée de test pour un objectif de test. Cet objectif correspond à un noeud du graphe de flot de contrôle (GFC) que l'on veut couvrir. En exécutant le cas de test sur une version instrumentée du programme sous test, TGen repère les noeuds couverts par le cas de test. La valeur d'utilité du cas de test est évaluée par le nombre de noeuds en commun entre le chemin couvert et le chemin correspondant à l'objectif de test. L'article décrit précisément le processus de génération fondé sur l'algorithme génétique, puis plusieurs expériences sur 6 petits programmes (entre 32 et 82 lignes de code). La pertinence de la technique présentée est validée par comparaison des résultats de l'algorithme génétique et d'un générateur aléatoire.

Jones et al. [JHD96, Jon98] donnent différents calculs possibles pour la fonction d'utilité. Pour tous ces modèles, les prédicats sont vus comme une fonction des données en entrée. Il faut ensuite trouver des données de test qui minimisent ces fonctions. Dans [Jon98], les mêmes auteurs proposent d'utiliser le score de mutation comme fonction d'utilité.

L'ensemble de travaux présentés ici montre l'efficacité des algorithmes évolutionnistes pour la génération de cas de test. Cependant, ces approches restent difficiles à appliquer pour le test de composant de taille importante manipulant des types de données

complexes. En effet, la plupart de ces méthodes ne sont adaptées que pour des données de test scalaires. Dans [BFJT05, BFJLT05] nous avons proposé un algorithme évolutionniste original baptisé algorithme bactériologique qui permet de palier aux limitations des algorithmes génétiques. Cet algorithme à été mis au point par analogie au processus naturel d'adaptation des bactéries. L'algorithme a été validé pour plusieurs sur différentes études de cas avec plusieurs critères de test [BTLJ00, BFJT02b, BFJT02a]. Nous revenons en détail sur cet algorithme dans la section 5.4.3 qui propose de l'adapter pour la génération de modèles.

Techniques fonctionnelles

Les techniques de génération fonctionnelle se basent sur une spécification du programme sous test. L'idée est d'utiliser ce modèle du programme pour la recherche et la sélection de cas de test pertinents. Un grand nombre de techniques existent dans ce domaine, elles diffèrent principalement par le formalisme de la spécification exploitée.

Nombre de travaux s'intéressent à la génération de cas de test à partir d'une spécification formelle. Ainsi dans [LPU02], Legeard et al. utilisent une spécification B ou Z du composant à tester afin de générer des cas de test aux limites. Dans [MMSC98], les auteurs utilisent une spécification Object-Z d'une classe pour la génération de cas de tests unitaires.

Dans [PJT+02], les auteurs décrivent une technique fondée sur une spécification fonctionnelle du composant à tester. Cette technique tire parti des diagrammes de classes et machines d'états associées au composant sous test afin de construire un système de transitions reflétant le comportement global du système. Un simulateur permet alors de générer des cas de tests à partir d'objectifs de test décrits en utilisant des diagrammes de séquence. Cette technique est implantée dans l'atelier de génie logiciel UMLAUT [HJGP99] qui utilise TGV [JM99] pour la génération de cas de test exécutables.

Dans [BL02], Briand et al. proposent une technique utilisant les diagrammes d'activités UML pour exprimer les dépendances séquentielles entre cas d'utilisation. Ce modèle est utilisé pour construire des objectifs de test sous forme de séquences valides de cas d'utilisation. Des cas de test sont ensuite construits à partir de ces objectifs de test en substituant chaque cas d'utilisation par un des scénarii qui lui est attaché.

Dans [NFLTJ06], nous avons proposé une technique de génération automatique de test qui exploite une spécification sous forme de cas d'utilisation. Chaque cas d'utilisation comporte des paramètres, une pré-condition et une post-condition. Les pré-conditions et les post-conditions permettent de déterminer les enchaînements de cas d'utilisation valides. Plusieurs critères permettent d'extraire des objectifs de tests de ces enchaînements valides. L'utilisation de diagrammes de séquence associés à chaque cas d'utilisation permet de transformer les objectifs de tests en cas de test exécutables.

Partionnement du domaine d'entrée

Qu'elles soient structurelles ou fonctionnelles, les techniques présentées dans les deux sections précédentes utilisent des informations sur le contrôle du programme à tester. A l'inverse, les techniques de test par partitionnement du domaine d'entrée sont des techniques orientées données. L'idée du test par partitions est d'utiliser les informations disponibles sur la topologie du domaine d'entrée du programme à tester afin de sélectionner des données de test pertinentes. Parmi les approches existantes, la plus aboutie est le test par partition. Initialement proposé par Ostrand et al. dans [OB88], le test par partition a fait l'objet de nombreux travaux [BGM91, DF93, vABM97, LPU02, KLPU04]. Ces techniques diffèrent par le formalisme de spécification et les stratégies qu'elles utilisent pour partitionner le domaine d'entrée du programme à tester. La technique de test de transformation de modèles que nous proposons dans le chapitre 5 s'appuie sur les idées du test par partition. Les paragraphes suivants détaillent ces idées.

Le test par partitions permet de sélectionner dans le domaine d'entrée d'un programme un ensemble restreint de données de test qui représente au mieux ce domaine d'entrée. Le principe est de partitionner le domaine à couvrir en définissant un ensemble de classes d'équivalence puis de choisir des représentants de chaque classe d'équivalence.

Si on considère, par exemple, un programme qui permet de calculer la valeur absolue d'un entier, le domaine d'entrée de ce programme est l'ensemble de nombres entiers. Dans la pratique, on ne peut pas le tester avec tous les entiers et il faut choisir un petit ensemble d'entiers qui servira au test. Intuitivement, il n'est pas nécessaire de tester un tel programme avec tous les entiers pour se convaincre de son bon fonctionnement : si le programme fonctionne correctement pour l'entier 5 il parait peu probable qu'il ne fonctionne pas pour l'entier 6. De façon plus générale, si le programme fonctionne avec un entier strictement positif, il est probable qu'il fonctionne avec tous les entiers strictement positifs.

De même, si le programme fonctionne avec un entier strictement négatif il est raisonnable de penser qu'il doit fonctionner pour tous les entiers strictement négatifs. Il reste ensuite à s'assurer que le programme fonctionne correctement pour l'entier 0 pour avoir couvert complètement le domaine d'entrée. On a alors partitionné le domaine d'entrée du programme en trois classes d'équivalences : $[-\infty, -1]$, $\{0\}$ et $[1, +\infty]$. A partir de cette partition, on choisit ensuite une donnée de test dans chaque classe d'équivalence et éventuellement des données de test aux limites de chaque classe d'équivalence. Pour tester le programme de calcul de valeur absolue, on choisira par exemple l'ensemble de données de test $\{-8, -1, 0, 1, 6\}$.

Dans la pratique il existe deux techniques complémentaires pour le calcul des classes d'équivalence : le partitionnement par défaut (*default partitionning*) et partitionnement sémantique (*knowledge-based partitionning*). Le partitionnement par défaut consiste à créer les classes d'équivalence en se basant uniquement sur la topologie du domaine à partitionner. Par exemple, si le domaine est l'ensemble des entiers, les trois classes d'équivalences négatifs, nul et positifs sont naturelles quel que soit le programme à tester. Le partitionnement sémantique permet de raffiner les classes d'équivalence par

défaut en tenant compte des spécificités du programme à tester.

Considérons, par exemple, un programme permettant de calculer le reste de la division euclidienne d'un entier x par 5 (c'est-à-dire x modulo 5). Pour le test de ce programme, les classes d'équivalence par défaut pour l'entier x $[-\infty, -1]$, $\{0\}$ et $[1, +\infty]$ sont certes intéressantes mais intuitivement insuffisantes pour construire un ensemble de test pertinent : à la lumière de la spécification du programme il apparaît, par exemple, que la valeur 5 joue un rôle particulier.

2.3 Validation et ingénierie des modèles

L'ingénierie des modèles étant une approche de développement logiciel relativement récente, il existe peu de techniques de validation spécifiques à ce domaine. Pourtant, le besoin de telles techniques est clairement identifié dans la littérature : une condition pour que l'ingénierie des modèles tienne ses promesses en terme de qualité est d'assurer la validation des modèles et des transformations utilisées. C'est le thème principal de cette thèse. Les deux sous-sections suivantes détaillent respectivement un ensemble de travaux pour la validation des modèles et un ensemble de travaux pour la validation des transformations.

2.3.1 Validation des modèles

Dans le contexte du développement basé sur les modèles, il existe deux approches pour la validation des modèles. La première est la validation directe des modèles qui consiste à vérifier ou à tester les modèles. L'inconvénient de cette technique est qu'elle impose de disposer d'un environnement permettant de simuler ou d'exécuter des modèles. La seconde approche tire avantage du fait que les modèles sont utilisés pour produire une partie de l'application développée. L'idée est de tester le code produit à partir des modèles plutôt que de tester directement les modèles. L'avantage de cette technique est qu'il n'est pas nécessaire de disposer d'un environnement modélisation exécutable. Son inconvénient est que la distance entre les modèles et le code produit est potentiellement importante et que le passage de l'un à l'autre peut introduire des erreurs.

Validation directe des modèles

Dans [RG00] les auteurs s'intéressent à la validation de diagrammes de classes UML comportant des contraintes OCL. Les auteurs proposent l'outil USE [RG02, USE] qui permet d'interpréter des contraintes OCL et d'assurer la conformité d'un diagramme d'objets à un diagramme de classes comportant des contraintes OCL. Pour valider un modèle statique UML, le testeur doit construire et animer des diagrammes d'objets et l'outil permet de vérifier qu'aucune contrainte n'est violée. Afin de valider leur approche, les auteurs l'ont appliquée à une partie du méta-modèle UML et aux contraintes de bonne formation qui lui sont associées. Dans la technique initialement proposée, les diagrammes d'objets sont construits manuellement par le testeur. Dans [GBR03] les

auteurs s'intéressent à automatiser la construction de diagrammes d'objets à partir d'une description déclarative des structures attendues.

Dans [GFB+03, AFGC03], Andrews et al. s'intéressent au test de modèles de conception UML. Les modèles UML sont rendus exécutables grâce à un langage d'action proche des diagrammes d'activités. Les auteurs définissent des critères de test qui s'appuient sur les diagrammes de classes et les diagrammes de collaborations du modèles UML. L'idée des critères proposés est d'assurer la couverture des structures des diagrammes de classes et des collaborations. Les scénarii de test sont générés sous la forme de diagrammes de séquences qui peuvent être complétés par des assertions. Une technique de génération de tests permettant de satisfaire ces critères a été implantée dans un outil présenté dans [DTKG+05]. Lors de la conception de la technique de test de transformations de modèles présentée dans le chapitre 5 nous nous sommes inspiré de ces travaux.

Validation du code produit à partir des modèles

Dans [RW03], Rutherford et al. présente un rapport d'expérience de génération conjointe de code et de tests pour un système développé à partir de modèles. Pour leurs expériences, les auteurs utilisent un outil de programmation générative appelé MODEST. L'article discute des avantages et de la rentabilité de développer des templates spécifiques à la génération de code de test pour l'outil MODEST. Ces templates permettent de générer des drivers de test et des cas de test pour le code généré. Un des bénéfices rapporté de cette approche est une amélioration du processus de développement obtenue par la familiarisation des développeurs et testeurs avec le code généré par l'outil MODEST. Le coût et la rentabilité de l'approche sont évalués par une étude comparative de la complexité des templates de génération de code et des templates de génération de tests.

Dans [HL03], les auteurs s'interessent également à la génération de test dans le contexte du Model-Driven Architecture. Le Model-Driven Architecture distingue deux types de modèles : les modèles métiers (indépandant de toute plateforme) et les modèles d'implantation (spécifiques à une plateforme). Dans l'approche proposée par les auteurs, la génération des tests et la production de l'oracle sont réalisés au niveau indépendant de la plateforme et l'exécution des tests est réalisée sur la plateforme cible, après que les modèles et les tests aient été transformés. L'étude de cas du développement d'une application web est utilisée afin d'illustrer l'approche proposée.

2.3.2 Validation des transformations de modèles

Dans [SL04] les auteurs présentent un retour d'expérience issue de la validation d'un moteur de transformation de modèles déclaratif. Pour le test de ce moteur de transformation, chaque donnée de test est une transformation et un modèle en entrée de cette transformation. La transformation est exécutée par le moteur avec le modèle en entrée donné et le modèle de sortie obtenu est comparé avec un modèle en sortie attendu. Comme le remarquent les auteurs, le processus de test mis en place pour la

validation du moteur de transformation est très similaire à un processus de validation de transforamtion de modèles. L'article discute des problèmes rencontrés et détaille les solutions adoptées par les auteurs. La première difficulté identifiée est la complexité de manipuler des modèles comme donnée de test : il est nécessaire de disposer d'outils pour éditer et sérialiser les modèles. Le problème du critère de test est également évoqué mais la solution adoptée par les auteurs n'est ni systématique ni automatisable.

Dans [KÖ4], l'auteur identifie le besoin de techniques pour la vérification et le test de transformations de modèles. La technique proposée permet la vérification de propriétés syntaxiques pour des transformations de modèles décrites par un ensemble de règles de transformations. Nous ne détaillons pas ici ce travail car nous nous interessons avant tout aux techniques de test.

Dans [LZG05], Lin et al. s'attardent sur trois problèmes liés à la mise au point de programmes de transformation de modèles. Le premier est le problème de la comparaison entre modèles qui est nécessaire pour la production de l'oracle lors de l'exécution des tests. Le second est la visualisation des erreurs, indispensable pour permettre un diagnostic lorsqu'un cas de test échoue. Enfin, le troisième est le développement de techniques de debuggage adaptées aux langages de transformation de modèles pour permettre de corriger les erreurs détectées. Dans l'article, les auteurs se concentrent sur les deux premiers problèmes et proposent un premier algorithme de comparaison de modèles basé sur des techniques de comparaison de graphes existantes. Les auteurs détaillent également un framework permettant d'organiser le test de transformations de modèles. Un exemple est utilisé pour illustrer les différentes étapes du test d'une transformation de modèles.

Dans le chapitre 5 nous nous interessons à ce problème et nous proposons des critères de test et des techniques de génération automatique de modèles.

2.4 Conclusion

Dans ce chapitre nous avons défini le contexte dans lequel se place cette thèse et détaillé les principaux travaux du domaine de l'ingénierie des modèles et du test de logiciel qui sont en relation directe avec les techniques que nous proposons. L'étude des travaux existants montre que bien que le besoin soit clairement établie dans la littératirature, il existe relativement peu de travaux dans le domaine de la validation des techniques d'ingenierie des modèles. Les travaux de cette thèse s'inscrivent dans cette voie et constituent un pas vers la fiabilisation des techniques d'ingénieries des modèles.

Les chapitre suivants de ce document contribue à la fois dans le sens de la qualité des modèles et dans le sens de la correction des transforamtions de modèles. Le chapitre 3 présente le langages Kermeta au coeur d'une plateforme d'ingénierie des modèles. Ce langage a pour but de permettre de définir de manière unifiée des structures, des contraintes du comportement et des transformation. La construction du langages Kermeta est réalisé par extention du langages de méta-modélisation EMOF présenté dans ce chapitre. Le chapitre 4 présente un ensemble d'études dont l'objectif est la validation

du langages Kermeta par la pratique. On retrouve parmis ces études de cas l'utilisation de Kermeta pour l'implantation de transformations de modèles. Enfin, le chapitre 5 s'interesse au problème du test des programmes de transforamations de modèles. Dans ce domaine nous proposons un ensemble de critères de test basées sur une technique de partitionnement de méta-modèles et nous discutons de la génération automatique de modèles de test.

Chapitre 3

Noyau pour la méta-modélisation

Dans la communauté de l'ingénierie des modèles, les méta-langages tel que MOF, EMOF ou ECore sont de plus en plus utilisés pour spécifier des méta-modèles. Ces langages se concentrent sur la spécification structurelle de méta-modèle mais ne supportent pas nativement la définition de comportements. De ce fait, ils ne peuvent pas être utilisés pour exprimer des contraintes, spécifier des transformations ou définir la sémantique opérationnelle des méta-modèles. Lorsque cela est nécessaire, il faut alors avoir recours à l'utilisation d'autres langages. En fonction des besoins les langages choisis peuvent être très différents : parfois impératif (Java par exemple), parfois déclaratif (OCL par exemple), et même hybrides dans certains cas (QVT par exemple).

Dans ce contexte, au sein d'un projet manipulant des modèles, les *artefacts relatifs à un méta-modèle* coexistent dans des langages aussi hétérogènes que Java, OCL ou QVT. On désigne par *artefacts relatifs à un méta-modèle* l'ensemble des éléments qui font référence à un méta-modèle. Ce peut être des contraintes associées au méta-modèle, des transformations de modèles ou l'expression d'éléments sémantiques liés au méta-modèle. L'hétérogénéité des langages utilisés engendre à la fois des problèmes techniques liés à l'interopérabilité entre les langages utilisés et des problèmes pour la validation des différents artefacts les uns par rapport aux autres.

Afin de remédier à ces problèmes, ce chapitre propose un noyau de méta-modélisation permettant de définir à la fois de la structure et la sémantique des méta-modèles. L'intérêt principal de l'approche est de permettre de définir les différents artefacts relatifs à un méta-modèle de façon homogène. Un méta-modèle peut par exemple encapsuler directement les contraintes qui lui sont attachées et la spécification de sa sémantique opérationnelle. Le noyau de méta-modélisation proposé se présente sous la forme d'un méta-langage orienté-objets construit comme une extension de EMOF. L'implantation de ce langage a été baptisée *Kermeta* (par contraction de *Kernel Meta*).

Ce chapitre est organisé comme suit. La section 3.1 présente le contexte et les motivations de ce travail à travers l'exemple de la définition et de l'utilisation d'un méta-modèle pour des automates. La section 3.2 détaille les différentes approches possibles pour la construction d'un méta-langage exécutable et détaille les choix que nous avons faits pour la réalisation de Kermeta. La section 3.3 présente le langage Kermeta et

détaille ses originalités et spécificités pour la manipulation de modèles. La section 3.4 détaille le système de type statique de Kermeta. Enfin la section 3.5 conclut ce chapitre et propose un ensemble de perspectives à ce travail. Des exemples d'utilisation de Kermeta et les domaines d'applications de Kermeta sont détaillés dans le chapitre suivant.

3.1 Contexte et motivations

Afin de présenter le contexte et les motivations de ce travail et de pointer avec précision les défauts des pratiques actuelles de modélisation, ce chapitre commence par détailler sur un exemple la définition et l'utilisation d'un méta-modèle simple. L'exemple utilisé met en jeu des automates simples, l'expression de leur structure, de leur sémantique, de contraintes et de transformations en utilisant les langages et techniques de modélisation actuelles.

3.1.1 Exemple de la pratique actuelle

Cette section détaille les moyens qui sont utilisés pour la spécification des différents artefacts relatifs à langage d'automates. Les artefacts que nous détaillons sont : la structure (méta-modèle), les contraintes, la sémantique et les eventuelles transformations.

Structure

Conformément à ce qui est présenté dans la section 2.1.3 de l'état de l'art, les langages de modélisation tel que MOF [(OM97] permettent de définir des méta-modèles grâce à des packages, classes, attributs, etc...La figure 3.1 présente un exemple de méta-modèle pour des machines à états finis très simples. Une machine à états (classe *FiniteAutomaton*) est composée d'un ensemble d'états (classe *State*) et d'un ensemble de transitions (classe Transition). Parmi ses états, une machine à états doit avoir un état initial et un ou plusieurs états finaux. Chaque état possède un nom, un ensemble de transitions sortantes et un ensemble de transitions entrantes. Une transition a un état source, un état cible et porte un symbole.

Cette description d'automates est suffisante pour permettre de créer des automates en instanciant les classes du méta-modèle. La figure 3.2 présente le diagramme d'objets correspondant à un automate simple. Cet automate comporte trois états nommés $S1$, $S2$ et $S3$, parmi lesquels $S1$ et $S3$ sont respectivement l'état initial et l'état final, et six transitions entre ces états. La notation utilisée figure 3.2, sous forme de diagramme d'objets a l'avantage de mettre en évidence la relation entre l'automate et son méta-modèle (figure 3.1) mais a l'inconvénient d'être peu lisible. La figure 3.3 présente le même automate en utilisant une notation graphique plus classique pour la représentation des automates.

A partir de la définition d'un méta-modèle utilisant un langage de méta-modélisation tel que MOF il est donc possible, en instanciant les classes du méta-modèle, de créer des modèles. Dans la pratique, un certain nombre d'outils tel que MDR ou EMF implantent

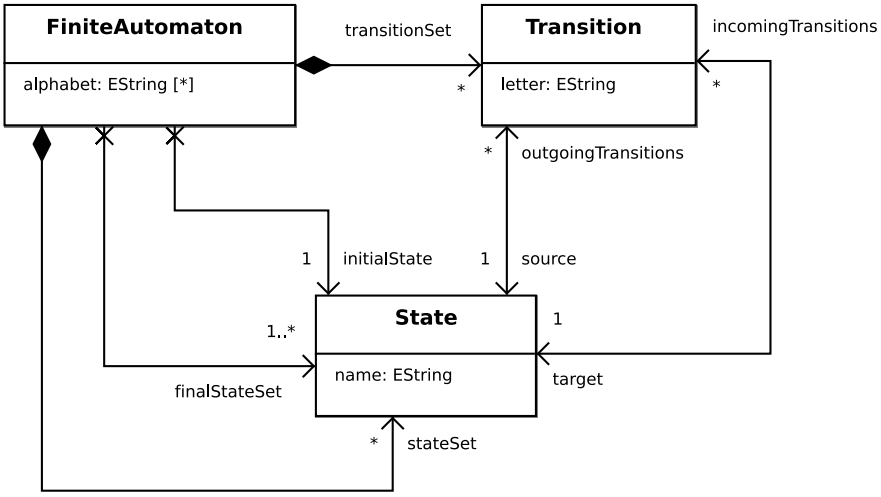


FIG. 3.1 – Méta-modèle d'automate.

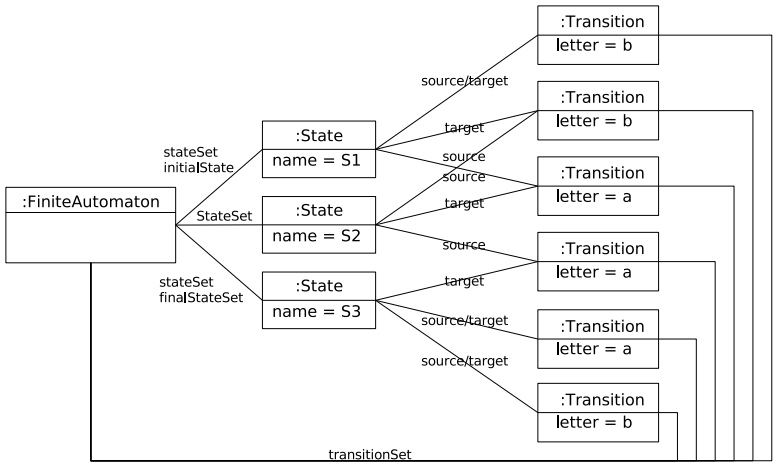


FIG. 3.2 – Diagramme d'objets correspondant à un automate.

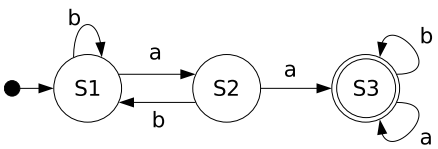


FIG. 3.3 – Notation classique pour l'automate de la figure 3.2.

cette fonctionnalité et proposent, par exemple, de générer des éditeurs à partir d'un méta-modèle afin de permettre à l'utilisateur de créer et éditer des modèles. Cependant, afin de rendre ces modèles exploitables d'un point de vue pratique, il est nécessaire de pouvoir exprimer des contraintes sur les modèles, spécifier la sémantique associée à un méta-modèle, ou implanter des transformations de modèles. Pour cela les langages de méta-modélisation, et MOF en particulier, ne fournissent aucun support, et, dans la pratique, différents langages sont utilisés pour la spécification de chacun de ces aspects.

Contraintes

Pour exploiter le méta-modèle d'automate présenté figure 3.1, il est nécessaire, dans un premier temps, d'exprimer les conditions sous lesquelles un automate est valide et de spécifier sa sémantique. Dans un second temps, il est alors possible d'écrire des programmes ou des transformations qui manipulent ces automates.

Concernant l'expression de contraintes, le langage OCL[(OM03] (Object-Constraint Language) initialement défini pour exprimer des contraintes dans des modèles UML, a été adapté afin de permettre d'exprimer des contraintes sur des méta-modèles. Cette adaptation consiste à réutiliser la partie d'OCL relative au diagrammes de classes UML pour exprimer des contraintes sur des modèles MOF. La réutilisation est possible du fait de la grande similarité structurelle des deux formalismes. Sur le méta-modèle d'automate, un exemple de contrainte est que toute les transitions entrantes et sortantes des états d'un automate doivent être contenues par cet automate. En d'autres termes, on interdit les transitions inter-automates. Cette contrainte peut facilement être écrite en OCL :

```
context Transition
  inv : self.container() = self.source.container() and
        self.container() = self.target.container()
```

Sémantique

La sémantique des méta-modèles est généralement exprimée en langage naturel. A titre d'exemple, la norme définissant le langage MOF[(OM97] est principalement écrite en anglais même si certains éléments ont été réécrits de manière déclarative en OCL. L'inconvénient du langage naturel est qu'il contient des ambiguïtés, ce qui réduit considérablement le niveau d'automatisation pouvant être atteint. Dans le cas de l'exemple des automates, on se contentera pour le moment d'une description en langage naturel de la sémantique associée au méta-modèle :

> Un automate définit un langage et permet de vérifier si un mot appartient à ce langage. L'état courant de l'automate est initialisé par l'état initial de l'automate. L'automate lit un à un les symboles sur la chaîne d'entrée. Si le symbole lu est l'étiquette de l'une des transitions sortantes de l'état courant alors cette transition est tirée. L'état courant devient alors l'état cible de cette transition et on lit le

symbole suivant. Sinon, le mot en entrée n'appartient pas au langage reconnu par l'automate. Lorsque la fin de la chaîne d'entrée est atteinte, si l'état courant est un état final, alors le mot en entrée fait partie du langage reconnu par l'automate, sinon le mot en entrée n'appartient pas au langage reconnu par l'automate.

Transformations

Concernant les transformations de modèles, un grand nombre d'approches et de langages existent. Ces approches vont de l'utilisation d'un langage généraliste à la définition de langages déclaratifs dédiés. Parmi les approches proposées, la norme QVT[(OM02] (Query/View/Transformation) propose un langage dédié à la transformation de modèle qui inclut à la fois des constructions déclaratives et impératives. A partir des automates définis précédemment, un grand nombre de transformations de modèles, allant de la génération de code à la détermination ou minimisation d'automate peuvent être envisagées. En pratique en fonction de la transformation à réaliser, tel ou tel langage sera utilisé. Par exemple, pour générer du code on utilisera un langage de *templates*, pour réaliser une transformation structurelle simple on utilisera un langage de déclaratif et pour réaliser une transformation plus algorithmique on préférera un langage impératif.

3.1.2 Analyse et proposition

Le fait d'utiliser des langages différents pour définir les différents artefacts relatifs à un méta-modèle (c'est-à-dire la structure, les contraintes, la sémantique et les transformations dans l'exemple précédent) présente l'avantage de pouvoir choisir pour chaque aspect un langage parfaitement adapté. Cependant, le fait que ces langages soient aujourd'hui complètement indépendants présente deux inconvénients majeurs. Premièrement, cela empêche les interactions entre les différents artefacts. En effet, si les contraintes associées au méta-modèle sont en OCL et que la sémantique est exprimée en langage naturel, il est difficile de faire appel aux contraintes ou aux éléments sémantiques dans une transformation de modèles écrite en Java ou QVT. Deuxièmement, cela rend très difficile toute vérification relative à la cohérence entre les différents artefacts.

Dans ce contexte, nous proposons d'identifier et de concevoir un noyau de modélisation afin de palier les limitations de cohérence et l'interopérabilité liées à l'hétérogénéité des langages. L'idée générale, illustrée figure 3.4, est de factoriser les concepts communs aux différents langages utilisés dans un coeur contenant les éléments permettant de spécifier des méta-modèles, leur sémantique opérationnelle mais aussi d'instancier ces méta-modèles ou de réaliser des transformations de modèles. L'objectif de ce coeur n'est pas de remplacer l'ensemble des langages dédiés utilisés mais de fournir une base sémantique commune à ces langages afin de leur permettre d'interopérer naturellement. En pratique, ce coeur se présente sous forme d'un langage de méta-modélisation appelé Kermeta et d'une machine virtuelle répondant aux besoins spécifiques de la manipulation de méta-modèles et de modèles.

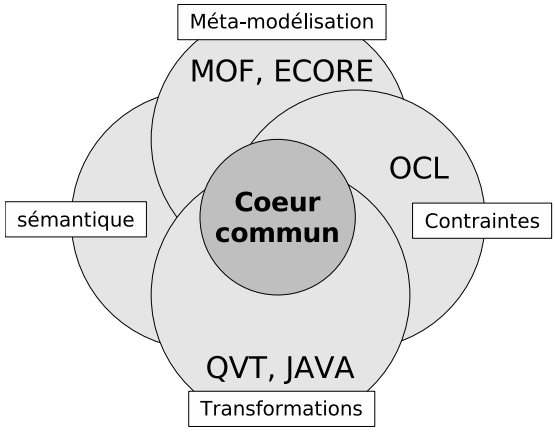


FIG. 3.4 – Un coeur commun.

3.2 Un méta-méta-modèle exécutable

L'inconvénient des méta-langages existants tel que MOF, est que ce ne sont en réalité que des langages de méta-données : ils ne permettent de définir que des structures sans fournir de support pour la définition de sémantiques, de comportements ou d'algorithmes. Cette section propose un méta-langage plus expressif permettant de définir de façon homogène des méta-modèles et leur sémantique. La conception d'un tel langage est un compromis à faire entre des exigences d'expressivité, d'utilisabilité et de minimalité :

Expressivité : Comme l'illustre la figure 3.4 l'objectif de ce méta-langage est de factoriser le dénominateur commun à l'ensemble des techniques relatives à l'utilisation de modèles. Il doit donc être possible de définir des méta-modèles, d'instancier ces méta-modèles, de spécifier leur sémantique opérationnelle, d'exprimer des contraintes, etc.

Utilisabilité : Le méta-langage a vocation à être manipulé par l'utilisateur pour la création de méta-modèles, ou de programmes manipulant des modèles. Les concepts proposés doivent donc être spécifiques au domaine de la modélisation et suffisement haut niveau pour répondre au mieux aux attentes de l'utilisateur. Pour permettre la réalisation et la maintenance de systèmes complexes, le langage doit supporter la pratique de techniques de génie logiciel conformes à l'état de l'art. L'expérience dans le domaine des langages a montré, par exemple, que le typage statique, l'utilisation de contrats (*design by contract*) et les mécanismes orientés-objets constituent de bons moyens de gérer la complexité des systèmes et de s'assurer de leur qualité.

Minimalité : L'objectif du noyau n'est pas de fournir un langage universel et généraliste pour la méta-modélisation, mais de permettre une définition sémantique homogène des méta-modèles. Le nombre de concepts du langage doit donc res-

ter aussi petit que possible tout en satisfaisant les contraintes d'expressivité et d'utilisabilité.

La plus grosse difficulté de conception est d'ajuster le compromis entre minimalité et utilisabilité. D'un côté, il est nécessaire d'introduire des concepts et des constructions dédiées qui rendent facile la manipulation de modèles et de l'autre, il faut prêter attention à introduire un minimum de concepts. Afin de répondre à ces exigences, trois options peuvent être envisagées vis à vis des langages existants :

- L'utilisation d'un langage généraliste orienté-objets existant tel que Java. Cela présente l'avantage de ne pas avoir à développer un nouveau langage et donc de permettre la réutilisation de l'ensemble des outils et frameworks disponibles pour le langage choisi. D'un autre côté, cette solution présente l'inconvénient que les langages existants, en tant que langages généralistes, ne sont pas spécialement adaptés à la manipulation de modèles. A titre d'exemple, des concepts de premier ordre de la modélisation tels que les associations ne font pas partie des langages orienté-objets existants. Cette solution ne permet donc pas de répondre de façon satisfaisante aux critères d'utilisabilité fixés.
- La définition d'un nouveau langage dédié. Cette solution est celle qui offre le plus de liberté pour la définition du langage, mais c'est également la solution qui requiert l'effort le plus important. Cette solution a l'inconvénient de conduire à des efforts inutiles car bien qu'aucun ne puisse être directement utilisé, un certain nombre de langages tel que MOF, OCL ou QVT contiennent des constructions qui ont été spécialement conçues pour la modélisation et qui devraient être réutilisées dans la mesure du possible.
- L'extension d'un méta-langage existant. La dernière solution est de choisir comme base un des méta-langages déjà utilisés pour la méta-modélisation puis de l'enrichir avec des primitives spécifiques et adaptées. Cette solution présente deux avantages majeurs. Le premier est qu'elle permet de ne pas risquer de régresser par rapport à l'existant tout en laissant une grande liberté en ce qui concerne les extensions. Le second est que le langage obtenu est compatible par construction avec le langage étendu et donc avec les outils existants. C'est la solution que nous avons choisie et qui est développée dans les sections suivantes.

L'approche que nous proposons consiste à partir d'un langage de méta-données existant et éprouvé puis à le compléter par un langage d'actions offrant les constructions manquantes afin d'obtenir un méta-langage complet. Nous avons appelé le langage obtenu Kermeta (Kernel Meta). Les sections suivantes présentent les choix que nous avons fait pour le langage de méta-données (section 3.2.1) et le langage d'actions (section 3.2.2) et détaillent notre approche pour l'intégration de ces deux langages (section 3.2.3).

3.2.1 Choix du langage de méta-données

Les langages de méta-modélisation existants, tel que MOF 1.4, EMOF, CMOF ou ECore se basent tous sur les concepts de packages, de classes et d'associations, et diffèrent par des variantes pour la définition de chacun de ces concepts. Malgré ces

différences, l'ensemble de ces langages présente un pouvoir d'expression sensiblement équivalent : il est toujours possible de traduire les concepts présents dans un langage en utilisant les concepts du langage cible. A titre d'exemple, les langages CMOF et MOF 1.4 permettent la définition de classe-associations, ce qui n'est pas le cas de EMOF ou ECore. Cependant, il est toujours possible de traduire une classe-association dans un langage qui ne les supporte pas nativement, en utilisant une classe et deux associations.

Afin de construire un méta-langage aussi "petit" que possible, le choix le plus logique est de retenir le langage de méta-données qui définit l'ensemble de concepts le plus compact. Dans cette optique nous avons choisi EMOF qui a été défini par l'OMG comme le langage ne contenant que les concepts "essentiels" à la méta-modélisation. La section 2.1.3 de l'état de l'art détaille les particularités de EMOF. Le fait de réutiliser EMOF comme base présente également l'intérêt qu'un certain nombre d'outils le supportent du fait de sa normalisation. C'est le cas par exemple des outils Eclipse qui, bien que basés sur le langage ECore, supportent l'import-export des modèles EMOF.

3.2.2 Choix du langage d'actions

Pour compléter un langage de méta-données, des possibilités très différentes peuvent être envisagées en fonction du type de langage d'action que l'on souhaite utiliser et de la façon dont il peut être intégré au langage de méta-données. Par exemple, si l'on souhaite utiliser des machines à états comme langage d'action, on décidera par exemple que le comportement de chaque classe EMOF doit être spécifié par une machine à état.

Afin de répondre au mieux aux objectifs et aux contraintes fixés nous avons choisi de construire un langage impératif, orienté-objets et statiquement typé. Le choix de réaliser un langage orienté-objets est motivé d'une part par le fait que les langages de méta-données que nous utilisons sont déjà fondés sur des principes orientés-objets, et d'autre part par la capacité des langages orientés-objets à permettre la réalisation et la maintenance de systèmes complexes [Jéz96, RBP$^+$91, GHJV95]. Le choix d'un langage statiquement typé est motivé par une volonté de permettre l'implantation de techniques de vérification et d'analyse statique à l'état de l'art.

Un certain nombre de langages existants, tel que l'AS (Action Semantics), OCL ou QVT, ont été spécialement conçus pour satisfaire les besoins liés à la manipulation de modèles. Il est donc naturel de se poser la question de la réutilisation de tout ou partie de ces langages comme langage d'actions. Chacun de ces langages présente des caractéristiques intéressantes mais aucun d'entre eux ne peut être réutilisé tel qu'il est.

OCL Le langage OCL a été initialement conçu pour exprimer des contraintes sur des modèles UML mais il existe aujourd'hui une version d'OCL liée à MOF qui permet d'exprimer des contraintes sur des méta-modèles. Le langage OCL inclut des expressions et constructions particulièrement adaptées à la navigation dans les modèles. Cependant, OCL étant un langage de contraintes, aucune construction OCL ne permet de créer ou de modifier des modèles. OCL ne peut être utilisé directement comme langage d'action mais une grande partie des expressions OCL peut être réutilisée tel quel dans le langage d'action. Dans cette optique, le langage

d'action peut être vu comme une extension à OCL qui permette les effet de bord nécessaires à la modification de modèles.

Action Semantics L'Action Semantics (AS) est défini comme le langage d'action d'UML. De la même manière que OCL a migré de UML vers MOF pour être utilisé sur des méta-modèles, on pourrait isoler la partie de l'AS applicable à MOF et l'utiliser comme langage d'action. Cette solution est cependant difficile à mettre en oeuvre car l'AS n'est pas un langage destiné à l'utilisateur mais une machine virtuelle permettant l'exécution d'actions au sein de modèles UML. De ce fait, les concepts présents dans l'AS sont trop bas-niveau par rapport à ce que l'utilisateur est prêt à manipuler pour la définition de méta-modèles.

QVT Le langage QVT est le langage de transformation de modèles normalisé par l'OMG. QVT est composé d'un coeur déclaratif basé sur des règles dédiées à la transformation de modèles et propose une extension impérative permettant d'implanter des règles de transformations. Les concepts présents dans QVT sont trop spécifiques à la transformation de modèles pour pouvoir être réutilisés dans le langage d'action.

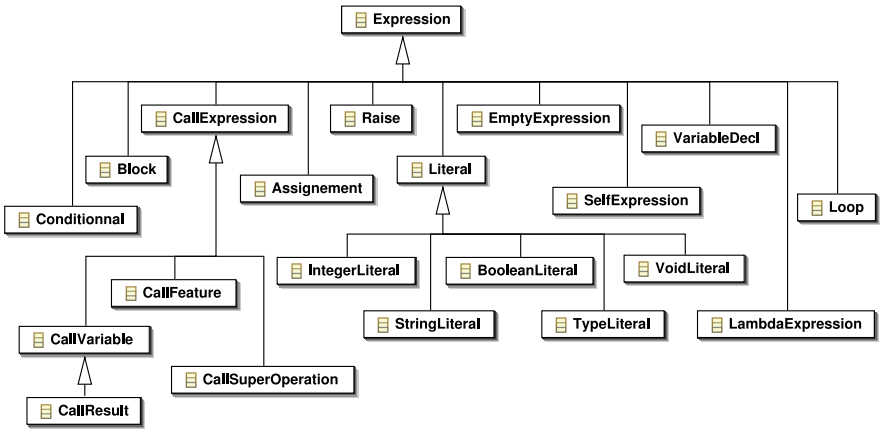


FIG. 3.5 – Les différents types d'expressions de Kermeta.

Le langage que nous avons choisi pour Kermeta réutilise les constructions d'OCL pour la navigation et les expressions et ajoute des constructions permettant la modification de modèles. La figure 3.5 présente la hiérarchie d'héritage entre les différent types d'expressions Kermeta. Les principales constructions de ce langage d'action sont :

- Des structures de contrôle classiques comme les boucles (*Loop*), les conditionnelles (*Conditional*) et les blocs (*Block*). Nous avons exclu du langage toute construction telle que *break* ou *goto* car elles brisent le flot de contrôle ce qui tend à rendre les programmes difficiles à comprendre et à maintenir.
- La déclaration de variables locales (*VariableDecl*). Étant donné que nous souhaitons réaliser un langage statiquement typé, les variables locales doivent être déclarées avec leur type.

- Des expressions permettant de lire les variables locales et les propriétés ainsi que d'appeler des opérations (*CallExpression* et ses sous-classes).
- Des affectations permettant d'affecter les variables locales et les propriétés des objets (*Assignement*).
- Des expressions littérales pour les types primitifs et les types (*Literal* et ses sous-classes).
- Un mécanisme d'exceptions permettant le traitement des erreurs. L'expression *Raise* permet de lever une exceptions.
- Des clôtures lexicales. Elles permettent l'implantation facile d'itérateurs analogues à ceux d'OCL tel que *select*, *reject* ou *collect*.

La section suivante présente la démarche que nous avons utilisée pour l'intégration de ce langage d'action dans EMOF. La section 3.3 revient ensuite en détail sur chacune des constructions du langage d'actions et discute des raisons d'être de chacune d'entre elles.

3.2.3 Processus de construction de Kermeta

La figure 3.6 schématise le processus de construction du langage Kermeta et sa promotion au rang de méta-langage (ie. au niveau M3). On retrouve sur cette figure les deux niveaux de méta-modélisation M2 et M3 de la traditionelle pile méta. Pour construire l'environnement de modélisation Kermeta, c'est-à-dire l'environnement dans lequel le méta-méta-modèle est Kermeta (à droite sur la figure 3.6), nous utilisons un environnement de modélisation EMOF (à gauche sur la figure). Dans cet environnement, les méta-modèles sont exprimés en EMOF. Étant donné que nous souhaitons construire Kermeta comme une extension de EMOF, les modèle initiaux du processus de construction de Kermeta sont d'une part EMOF (exprimé en EMOF) et d'autre part le méta-modèle d'actions présenté dans la section précédente (lui aussi exprimé en EMOF).

A partir de ces deux méta-modèles, une première transformation (appelée *Composition* sur la figure 3.6) a pour but de composer EMOF et le langage d'action en un méta-modèle cohérent. Le résultat de cette transformation est le méta-modèle de Kermeta en EMOF. La seconde transformation représentée sur la figure 3.6 est la promotion. Cette transformation vise à "remonter" le langage Kermeta au niveau M3 afin de le poser comme base d'un nouvel environnement de modélisation. Ce processus est proche de la démarche qui est utilisée pour faire évoluer un langage et son compilateur d'une version n à une version $n+1$. On utilise la version n d'un langage pour implanter le compilateur de la version $n+1$ de ce même langage. Le compilateur de la version n du langage permet ainsi de compiler le compilateur de la version $n+1$. Une fois compilé, le compilateur de la version $n+1$ peut se recompiler seul et la version n du compilateur peut être abandonnée. Dans le cas présent EMOF joue le rôle de la version n et Kermeta celui de la version $n+1$.

La composition de EMOF avec le langage d'action parait simple au premier abord mais elle nécessite en réalité un soin particulier pour assurer les qualités du langage obtenu par composition. Comme le montre la figure 3.7, la jonction entre EMOF (pré-

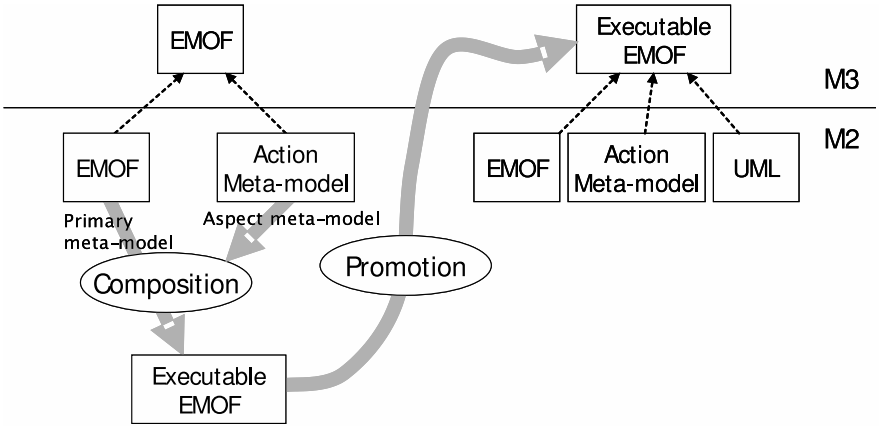


FIG. 3.6 – Construction d'un EMOF exécutable. Les flèches pointillées symbolisent la relation de conformité d'un modèle à son méta-modèle et les flèches grises et les ellipses représentent des transformations de modèles.

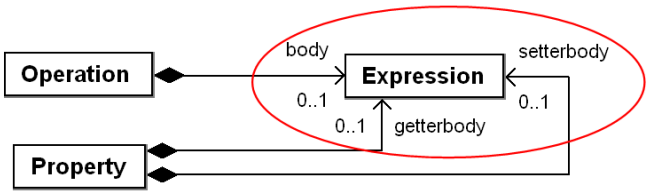


FIG. 3.7 – Spécifier le corps des opérations grâce à des expressions.

senté figure 2.10) et le langage d'actions est faite par trois associations entre les classes *Properties* et *Operation* de EMOF et la classe *Expression* du langage d'action. Ainsi, on offre la possibilité de spécifier, dans un méta-modèle, le comportement des opérations et la façon dont les propriétés dérivées sont calculées par des expressions. Bien que structurellement ces trois associations suffisent à établir le lien entre EMOF et le langage d'action, un certain nombre de problèmes sémantiques restent à régler quant au système de types et au mécanisme de redéfinition du langage résultant. Les sections suivantes détaillent les solutions que nous avons choisies dans le langage Kermeta et les ajustements nécessaires à l'obtention d'un langage cohérent.

3.3 Le langage Kermeta

La figure 3.8 présente le méta-modèle du langage Kermeta construit par extension de EMOF. Ce méta-modèle est divisé en deux parties : une partie structurelle et une partie comportementale. La partie structurelle (23 classes), en haut sur la figure, est constituée des concepts issus de EMOF. La partie comportementale (26 classes) quant à elle correspond au langage d'action, c'est-à-dire aux sous-classes de la classe *Expression* sur la figure. Les sous-sections suivantes détaillent les différentes constructions de Kermeta et les raisons pour lesquelles chacune d'entre elle a été retenue.

La composition du langage d'action statiquement typé de Kermeta a nécessité un certain nombre d'ajouts et d'ajustement dans EMOF. La section 3.3.1 détaille les élément de Kermeta qui sont directement issus de EMOF. Cependant, EMOF n'étant pas conçu pour supporter du comportement un certain nombre d'ajustements liés à la sémantique de l'héritage (notament vis-à-vis de la redéfinition d'opérations) ont dû être réalisés. Ces ajustement sont détaillés section 3.3.2. Pour permettre le typage statique d'expressions de navigation OCL, Kermeta intègre un mécanisme de généricité des classes et des opérations. La section 3.3.3 montre en quoi les génériques fournissent une solution élégante au problème de typage et détaille leurs implantations dans Kermeta. La section 3.3.4 présente l'ensemble des expressions du langage d'action de Kermeta. Enfin, la section 3.3.5 détaille comment les clôtures lexicales intégrées à Kermeta permettent d'implanter des itérateurs comparables aux itérateurs d'OCL.

3.3.1 Structures issues de EMOF

Du fait de sa construction par extension de EMOF, la structure d'un programme Kermeta est très proche de celle d'un méta-modèle EMOF. Dans les deux cas un méta-modèle est un ensemble de packages qui contiennent des définitions de type. Ces définitions de type peuvent être des types primitifs, des énumérations ou des classes. Cette section détaille, à travers des exemples simples, la définition de packages Kermeta, la définition de types primitifs et de types énumérés et enfin la définition de classes et associations.

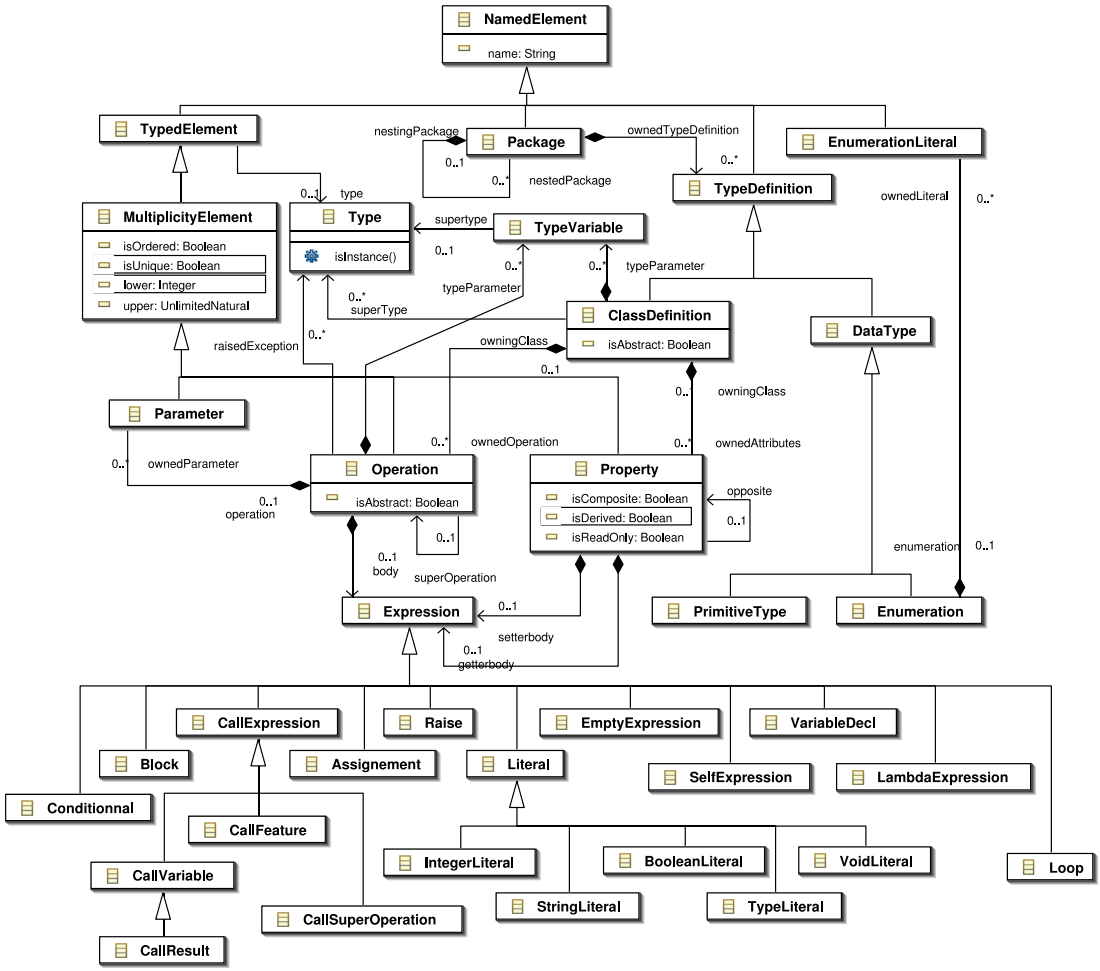


FIG. 3.8 – Principales classes du méta-modèle de Kermeta.

Packages Kermeta

Lors de la définition d'un méta-modèle, les packages servent à la fois de conteneurs pour les méta-classes et d'espaces de nommage. Ainsi, en Kermeta, un des premiers éléments de tout programme est la déclaration du package dans lequel sont contenus tous les éléments que l'on définit par la suite. Sur le listing de la figure 3.9, cette déclaration correspond à la ligne 2 : tous les éléments définis ensuite sont dans le package *A*. A l'intérieur d'un package, on peut bien sûr définir des classes, des énumérations ou des types primitifs mais également des sous-packages. La ligne 8 de la figure 3.9 montre la définition d'un sous-package *B*. Il est intéressant de noter, comme le montre la ligne 14 du listing, qu'un package existant peut être réouvert afin de lui ajouter de nouveaux éléments.

```
1  /** déclaration du nom du package */
2  package A;
3
4  /** définition de la classe classe A::C */
5  class C { ... }
6
7  /** définition du sous package A::B */
8  package B {
9    /** définition de la classe A::B::D */
10   class D { ... }
11 }
12
13 /** réouverture du package A::B */
14 package B {
15   /** définition de la classe A::B::E */
16   class E { ... }
17 }
```

FIG. 3.9 – Définition de packages

En plus de servir de conteneurs, les packages servent d'espaces de nommage. Le nom qualifié d'un élément est le nom du package qui contient cet élément, suivi du nom de l'élément. Dans un programme Kermeta, il est possible d'utiliser systématiquement les noms qualifiés des éléments auxquels on souhaite faire référence, mais il est également possible d'utiliser des noms courts. Les noms courts utilisés sont recherchés dans le package courant, puis dans un ensemble de packages que le programmeur peut spécifier au moyen de la directive **using**. Cette directive est proche de la directive "import" de Java.

Types primitifs et énumérés

Les types primitifs (*PrimitiveType*) sont un outil de modélisation qui permettent, lorsque l'on crée un méta-modèle, de ne pas décrire les types de base (tels que les nombres entiers, les chaînes de caractères, les date ou les booléens) utilisés et supposés

connus. Une fois le méta-modèle défini, pour pouvoir instancier des modèles utilisant ces types primitifs, il est nécessaire de donner la correspondance entre ces types primitifs et un type concret existant dans l'environnement.

En Kermeta, les types primitifs sont donc une simple indirection vers des types existants. La figure 3.10 montre la définition d'un type primitif *Entier* faisant référence au type *Integer* défini par une classe dans le package *standard*. Le mot-clé **alias** a été choisi pour expliciter le fait que le type primitif *Entier* n'est en fait qu'une référence au type *Integer* existant par ailleurs.

Du point de vu de la vérification de type et du sous-typage, les types primitifs sont "transparents", c'est-à-dire que seul le type auquel ils font référence est considéré. Sur l'exemple cela signifie que pour chaque utilisation du type *Entier* tout se passe comme si le type *Integer* avait été utilisé.

```
1  package standard {
2    /** Definition of class Integer */
3    class Integer {
4      [...]
5    }
6    [...]
7  }
8
9  package monMétaModèle {
10   /** Définition du type primitif Entier et de sa représentation */
11   alias Entier : standard::Integer
12   [...]
13 }
```

FIG. 3.10 – Définition d'un type primitif.

Les types énumérés (*Enumeration*) permettent de définir des types de données constitués par un petit ensemble de valeurs particulières données en extension. La figure 3.11 donne un exemple de définition d'une énumération de couleurs en Kermeta. Les types énumérés de EMOF ou Kermeta ne sont pas différents des types énumérés existants dans la plupart des langages de programmation.

```
1  /** Enumération des couleurs primaires */
2  enumeration CouleurPrimaire {
3    rouge;
4    vert;
5    bleu;
6  }
```

FIG. 3.11 – Définition d'un type énuméré

Classes et associations

Les classes et les associations sont les concepts centraux de la méta-modélisation. Bien que le concept de classe soit également central à la programmation orienté-objets, il existe deux grandes différences entre le concept de classe des langages de programmation et le concept de classe utilisé en modélisation. La première est la présence, en modélisation, d'associations là où les langages orientés-objets se contentent de simples références. La seconde est la notion de composition (*containment*), absente des langages de programmation, qui offre un moyen de structuration des objets.

Les associations et les relations de composition existent dans UML et sont naturellement utilisées par les concepteurs de systèmes orientés-objets. Cependant, la plupart des générateurs de code à partir de modèles UML ne tiennent pas compte de la composition et se contentent de transformer les associations en attributs. La principale raison à cela est que ces notions n'existent pas dans les langages de programmation, mais c'est aussi probablement parce que la sémantique de ces constructions est longtemps restée floue pour les utilisateurs d'UML. Dans Kermeta, les concepts d'association et de composition ont été conservés tels qu'ils sont définis dans EMOF et la sémantique des actions du langage a été élaborée en accord avec leurs sémantiques.

La figure 3.12 présente, à travers des exemples d'associations et de compositions, la correspondance entre la syntaxe de Kermeta et la syntaxe graphique classique de UML/MOF. Le premier exemple montre une référence simple de la classe *A1* vers la classe *B1*. Ce genre de référence correspond à la notion de référence existant dans les langages orientés-objets comme Java. Dans le second exemple, on a ajouté une multiplicité à la référence afin de lier plusieurs instances de *B2*. La construction obtenue est proche de ce que l'on peut exprimer avec des tableaux dans les langages existants. A l'opposé de ces deux premiers exemples, les quatre exemples suivants ne correspondent à aucune notion disponible dans les langages orientés-objets classiques.

Les exemples 3 et 4 montrent des associations avec des multiplicités variables. Dans la syntaxe textuelle, on peut constater que ces associations sont construites par jumelage de deux références que l'on qualifie alors d'opposées. Le fait de représenter les associations de cette manière est issu de EMOF. D'un point de vue sémantique deux références opposées engendrent une association. Le troisième exemple présente une association d'une instance de *A3* avec une instance de *B3*. La signification de cette association est :

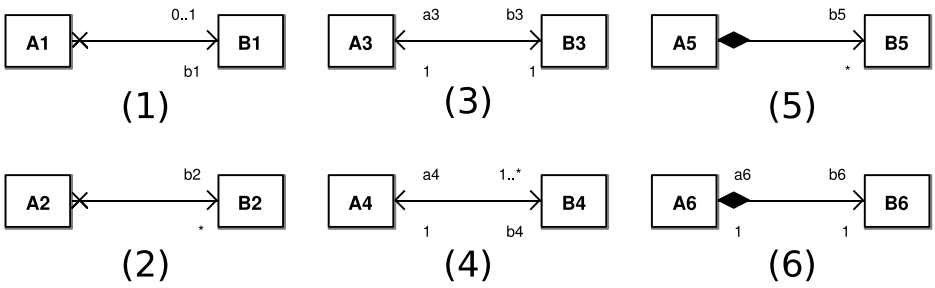
$$\text{Si } Ia : A3 \text{ et } Ib : B3 \text{ alors } Ia.b3 = Ib \Leftrightarrow Ib.a3 = Ia$$

C'est-à-dire que, si une instance Ib de *B3* associée à une instance Ia de *A3*, alors la propriété *b3* de Ia référence Ib et la propriété *a3* de Ib référence Ia. De la même manière, il est possible de décrire la sémantique de l'association entre les classes *A4* et *B4* du quatrième exemple de la figure 3.12 :

$$\text{Si } Ia : A4 \text{ et } Ib_{1..n} : B4 \text{ alors } Ib_i \in Ia.b4 \Leftrightarrow Ib.a4 = Ia$$

Étant donné cette sémantique des associations, il est nécessaire que l'ensemble des constructions du langage Kermeta préserve ces propriétés pour assurer l'intégrité des

associations. Ainsi il ne doit, par exemple, pas être possible de réaliser une simple affectation à l'une des extrémités d'une association sans que l'autre extrémité de l'association soit convenablement mise à jour. Les solutions que nous avons choisies pour le langage Kermeta, notament en ce qui concerne la sémantique de l'affectation, sont détaillées dans la section 3.3.4.




```
1  // Référence simple vers une instance de B1
2  class A1 { reference b1 : B1 }
3  class B1 {}
4
5  // Référence à un ensemble de B2
6  class A2 { reference b2 : B2[0..*] }
7  class B2 {}
8
9  // Association entre 1 A3 et 1 B3 (deux références jumelées)
10 class A3 { reference b3 : B3[1..1]#a3 }
11 class B3 { reference a3 : A3[1..1]#b3 }
12
13 // Association entre 1 A4 et 1 ou plusieurs B4
14 class A4 { reference b4 : B4[1..*]#a4 }
15 class B4 { reference a4 : A4[1..1]#b4 }
16
17 // Composition d'un ensemble de B5 dans un A5
18 class A5 { attribute b5 : B5[0..*] }
19 class B5 {}
20
21 // Composition d'un B6 dans un A6
22 class A6 { attribute b6 : B6[1..1]#a6 }
23 class B6 { reference a6 : A6[1..1]#b6 }
```

FIG. 3.12 – Associations et compositions.

Les deux derniers exemples de la figure 3.12 concernent l'utilisation de la composition. La sémantique des relations de composition et d'agrégation d'UML a mis un certain temps avant d'être clairement établie [BHSPLB03] et ces relations ont souvent été utilisées comme des décorations étant donné que, dans la majorité des cas, le code généré à partir des diagrammes UML n'en tient pas compte. Afin de palier ces problèmes, lors de la définition de MOF, seule la relation de composition a été conservée et une sémantique claire lui a été attribuée. Lorsqu'un objet C réfère un objet O par

une composition (que l'on notera $C \overset{\diamond}{\rightarrow} O$), c'est-à-dire que O est conceptuellement "contenu" par C, on a :

$$C' \overset{\diamond}{\rightarrow} O \implies C' = C$$
$$O.container() = C$$

Tout graphe d'objets est a-cyclique par la relation $\overset{\diamond}{\rightarrow}$

La première propriété exprime le fait qu'un objet ne peut être contenu, à un instant donné, que par un seul autre. La seconde propriété établit la sémantique de l'opération *container* disponible pour tout objet. Elle permet, à partir d'un objet, de remonter à l'objet qui le contient, c'est-à-dire à celui qui est en relation de composition avec lui. La dernière propriété concerne plus généralement la composition et impose qu'il n'y ait pas de cycle par la relation de composition dans un graphe d'objets. Cette dernière propriété est importante car il en résulte que la structure associée à la compostion est un arbre. Tout comme pour les associations précédemment discutées, il est important que toutes les constructions du langage Kermeta permettent de préserver les contraintes liées à la composition.

3.3.2 Héritage et redéfinition

L'expérience du paradigme orienté-objets montre qu'un mécanisme de redéfinition d'opérations couplé à l'héritage est un bon moyen de représenter la variablilité dans le comportement des objets. EMOF, en tant que langage de méta-données orienté-objets, présente bien un mécanisme d'héritage entre les classes mais ne donne de sémantique ni a l'appel d'opération, ni à l'héritage ou la redéfinition d'opération. Cette lacune doit être comblée afin de donner une sémantique précise au langage Kermeta.

Le problème de choisir une sémantique pour la redéfinition d'opération à été largement étudié dans la littérature [AC96] sans qu'une solution unique soit toutefois adoptée. Ces solutions se distinguent par différents choix en fonction que l'on souhaite un héritage simple ou multiple, que l'on souhaite autoriser ou non la surcharge d'opérations et, que l'on autorise ou non la variation de la signature d'une opération lors de sa redéfinition.

Pour réaliser un langage par extension de EMOF, le choix de l'héritage multiple est imposé car EMOF le supporte naturellement. Pour ce qui est de la surcharge et des redéfinitions les choix sont libres à condition de leur donner une sémantique claire. Afin de réaliser un langage le plus simple possible, nous avons choisi dans la première version de Kermeta d'interdire la surcharge et de n'autoriser que les redéfinitions invariantes (ie. sans aucune modifications de la signature de l'opération redéfinie).

Pour permettre la redéfinition d'opérations, comme le montre la figure 3.13, nous avons ajouté deux propriétés à la classe *Operation* de EMOF. La première, *isAbstract*, permet de spécifier qu'une opération est abstraite, c'est à dire qu'elle n'a pas de corps et qu'elle doit être définie dans toutes les sous-classes de la classe dans laquelle elle se trouve. La seconde propriété, nommée *superOperation*, permet de spécifier quelle est l'opération redéfinie lorsqu'une opération est une redéfinition. Un certain nombre de contraintes sont liées à cette dernière association :

- La super-opération d'une opération dans une classe C doit être définie dans une classe dont C hérite directement ou indirectement.
- Si une opération possède une super-opération, son nom doit être identique à celui de sa super-opération. Ceci est nécessaire car les opérations sont identifiées par leur nom lors de l'appel.
- La signature d'une opération et celle de sa super-opération doivent être identiques. Nous avons, en effet, choisi une politique de redéfinition invariante.
- Toutes les opérations d'une classe, incluant les opérations héritées, doivent avoir des noms uniques. Ceci est lié au choix que nous avons fait d'interdire la surcharge. Ce choix est motivé par le fait que les interactions entre surcharge et redéfinition, en particuler dans un contexte d'héritage multiple, sont difficiles à maîtriser.

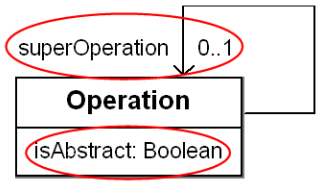


FIG. 3.13 – Redéfinition d'opérations.

Le mécanisme de super-opération mis en place permet non seulement de représenter les redéfinitions d'opérations dans une hiérarchie d'héritage, mais aussi de résoudre de façon simple les conflits qui émergent de l'utilisation de l'héritage multiple. En effet, le fait qu'une opération ne puisse avoir qu'une seule super-opération implique qu'un choix doit être fait explicitement, si plusieurs super-classes présentent des opérations qui entrent en conflit. Dans la première version de Kermeta, nous avons décidé de nous contenter de ce simple mécanisme. Cela induit certaines limitations car des programmes dans lesquels une classe hérite des deux autres présentant des méthodes de même nom, mais de signatures différentes seront considérés comme invalides. Pour résoudre ce problème, des mécanismes moins naïfs tel que le renommage d'opérations disponibles dans le langage Eiffel pourraient être mis en oeuvre.

3.3.3 Classes et opérations génériques

Comme il a été discuté précédemment, nous avons choisi pour des raisons de testabilité et de fiabilité de vérifier statiquement le typage des expressions Kermeta. Pour permettre le typage statique d'expression, un certain nombre de déclarations telles que les déclarations de type pour les variables, doit être rendu obligatoire. Cependant il faut porter attention à ne pas surcharger le langage avec trop d'informations redondantes sous peine de nuire gravement à son utilisabilité. Cette section discute du typage statique d'expression de navigation dans les modèles et présente le système de types que nous avons choisi dans le langage Kermeta.

Exemple et motivations

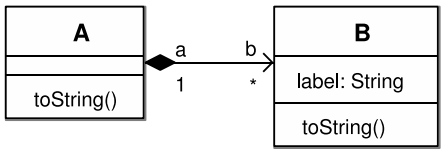


FIG. 3.14 – Un méta-modèle très simple.

Considérons le méta-modèle très simple présenté sur la figure 3.14. Ce méta-modèle contient deux classes et une association entre ces deux classes. Voici un exemple d'expression que l'on souhaite pouvoir écrire :

```
myA.b.one.label
```

Cette expression permet de naviger l'association entre les classes A et B et retourne le *label* associé à une instance de B. L'opération *one* permet d'extraire un élément d'une collection. Dans le système de types proposé par EMOF, chaque classe définit un type. Essayons donc de typer les différentes parties de cette expression en utilisant ce système de type. Le type de `myA` est A. L'expression `myA.b` retourne un ensemble d'instances de B, son type est donc la classe *Set*. On suppose que la classe *Set* possède une méthode *one* qui retourne un élément de l'ensemble. L'expression `myA.b.one` retourne un objet contenu dans l'ensemble, son type, donné par le type de retour de la méthode `one` de la classe *Set*, est *Object*. Étant donné ce résultat, l'appel à l'attribut label sera refusé dans l'expression `myA.b.one.label` car le type *Object* ne contient pas d'attribut label.

Dans l'exemple précédent, l'expression est refusée car le système de types ne permet pas de représenter que le type de `myA.b` n'est pas un ensemble d'objets arbitraires mais un ensemble de B. Une façon de résoudre ce problème, sans modifier le système de type, est d'imposer à l'utilisateur de spécifier le type des objets extraits de la collection. On pourrait écrire par exemple, si l'on suppose que l'on dispose d'une pseudo-opération *o.asType(T)* qui permet d'indiquer que l'objet o est de type T :

```
myA.b.one.asType(B).label
```

Dans le cas de cette expression, l'utilisateur indique explicitement que `myA.b.one` est de type B et l'expression dans son ensemble sera acceptée car une propriété *label* existe bien sur la classe B. Cependant, cette approche n'est pas satisfaisante pour deux raisons. Premièrement l'information est inutilement dupliquée puisque l'association sur le méta-modèle de la figure 3.14 exprime déjà le fait que la propriété b de la classe A dénote un ensemble d'instances de B. Deuxièmement, d'un point de vue purement syntaxique, cela surcharge grandement les expressions de navigation ce qui tend à les rendre moins naturelles et moins lisibles. La seule solution reste alors de modifier le système de types pour permettre de typer statiquement les expressions de navigation. Dans cette optique deux solutions peuvent être envisagées.

La première solution est d'introduire dans le système de types des types spéciaux pour représenter les collections. Cela peut être vu comme une extension de la notion de tableau dans les langages orientés-objets. C'est le choix qui a été fait dans les langages tel que Xion [MSFB05] pour permettre de typer statiquement les expressions de navigation dans les modèles UML. Cependant cela impose d'introduire des constructions syntaxiques et sémantiques dédiées aux collections et de définir des politiques appropriées pour la vérification des types. Pour ces raisons, nous avons rejeté cette option pour le langage Kermeta au profit des classes paramétriques qui offrent une solution plus générale.

La seconde solution est de permettre la définition de classes paramétrées. De cette façon, les collections typées peuvent être définies comme n'importe quelle autre classe. La généricité offre une solution élégante au typage statique et est aujourd'hui adoptée par la plupart des langages orientés-objets statiquement typés tel que Java, Eiffel, C++ ou C♯. Cependant, introduire la généricité dans le système de types de EMOF n'est pas trivial car cela impose de modifier en profondeur la relation entre classes et types.

Dans le cas de l'expression `myA.b.one.label`, en utilisant des collections génériques le type de `myA.b` est une collection de B ($Set<B>$), `myA.b.one` est donc bien de type B.et la propriété *label* peut être lue sans problèmes.

Ajout de la généricité à EMOF

Comme indiqué sur la figure 2.10, dans le système de types de EMOF un type peut être un type primitif (*PrimitiveType*), une énumération (*Énumération*) ou une classe (*Class*). L'introduction des classes paramétriques rend plus complexe la relation entre types et classes. Si par exemple on définit une classe $Set<G>$, qui représente un ensemble d'éléments de type G, cette classe ne définit pas un type *Set* mais engendre un ensemble de types en fonction de la valeur que prend le paramètre de type G.

La figure 3.15 présente l'approche adoptée pour créer un système de types compatible avec celui d'EMOF et incluant la généricité. Le cadre A représente le modèle de classes de EMOF. La classe *Class* hérite de *Type* et contient un ensemble de *Features*, à savoir des propriétés et des opérations. Le cadre B représente un système de types supportant la généricité. Ce modèle fait explicitement la différence entre une définition de classe (*ClassDefinition*) qui contient des propriétés, des opérations et des variables de type (*TypeVariable*) et une classe paramétrée (*ParametrizedClass*) qui définit un type. Une classe paramétrée fait référence à une définition de classe et associe un type effectif à chacune des variables de type de cette définition de classe par l'intermédiaire de la classe *TypeVariableBinding*. Dans ce modèle, la notion de définition de classe et la notion de type, c'est-à-dire de classe paramétrée, sont distinctes.

Sur la figure 3.15 les liens pointillés entre le modèle de classe de EMOF dans le cadre A et le modèle permettant la généricité dans le cadre B, figurent les éléments communs à ces deux modèles. Ainsi on retrouve sur les deux modèles, les notions de *Type* et de *Feature* mais la classe Classe de EMOF semble contenir à la fois des caractéristiques d'une définition de classe car elle contient des *Features* mais aussi des caractéristiques d'une classe paramétrée car elle hérite de *Type*. Afin d'ajouter la généricité à EMOF

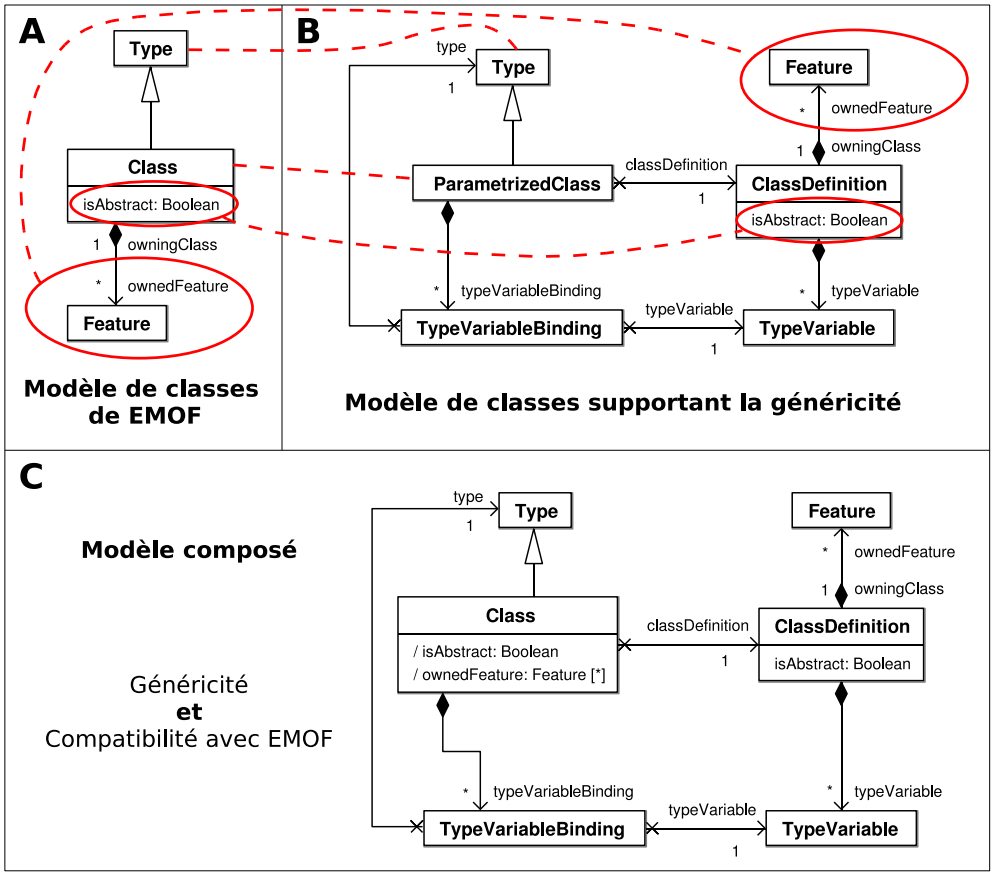


FIG. 3.15 – Ajout de la généricité au système de types de EMOF.

l'idée est donc de séparer la classe *Class* en deux pour séparer la notion de type de la notion de classe.

Le cadre C sur la figure 3.15 présente le résultat de la composition du modèle de généricité dans EMOF. On a gardé le nom *Class* pour désigner une classe paramétrée afin d'assurer la compatibilité avec MOF et on a ajouté les classes *ClassDefinition*, *TypeVariable* et *TypeVariableBinding* afin de représenter les définitions de classes. Afin d'assurer la compatibilité avec EMOF nous avons ajouté à la classe *Class* des propriétés dérivées (dont le nom est préfixé par un / sur la figure) permettant d'accéder aux propriétés telles que *isAbstract* ou *ownedFeature* qui étaient disponibles sur les classes EMOF et qui ont été déplacées dans la classe *ClassDefinition*.

Nous avons choisi d'intégrer la généricité à Kermeta non seulement parce que cela fournit une solution au problème du typage des expressions de navigation mais aussi, parce que cela fournit un mécanisme plus général qui offre une solution élégante au traitement d'expressions intégrant des itérateurs analogues à ceux d'OCL. Les principes de la généricité implantés dans Kermeta sont peu différents des concepts présents dans des langages tel que Eiffel, .NET 2.0 ou Java 5.

Classes génériques en Kermeta

La figure 3.16 illustre l'utilisation des classes paramétriques à travers la définition de files d'attentes. La ligne 2 montre la définition de la classe *Queue* comportant un paramètre de type *G*. La variable de type *G* peu être utilisée comme n'importe quel type à l'intérieur de la classe. A la ligne 4, par exemple, on définit une association vers un ensemble ordonné d'objets de type *G*. Le type *G* est également utilisé comme type de paramètre pour l'opération *enqueue* et comme type de retour de l'opération *dequeue*. La sous-classe *SortedQueue*, définie à la ligne 17, est également une file d'attente mais avec une politique différente. Pour pouvoir utiliser cette file d'attente, il faut que les éléments que l'on met en attente soient munis d'un opérateur de comparaison qui détermine leur priorité. De la même manière que pour la classe *Queue*, on définit une classe générique, mais, pour imposer l'existence d'une opération permettant de comparer les éléments deux-à-deux, on contraint la variable de types aux sous-types du type *Comparable*. C'est cette contrainte qui permet de comparer deux éléments à la ligne 22.

Opérations génériques en Kermeta

L'utilisation de variables de types ne se limite pas aux seules classes mais permet également de définir des opérations paramétriques. L'intérêt de définir de telles opérations est de permettre de donner au vérificateur de type le lien existant entre le type des paramètres et le type de retour d'une opération. La figure 3.17 présente la définition d'une opération *max* permettant de comparer deux éléments entre eux et de rendre le plus grand des deux. De la même manière que pour les classes dans l'exemple précédent, pour pouvoir comparer les éléments entre eux, on a contraint le type *T* à être un sous-type de *Comparable*.

Il existe une différence importante entre l'utilisation des variables de type dans le

```
1  /** Generic FIFO queue */
2  class Queue<G>
3  {
4    reference elements : oset G[*]
5
6      opération enqueue(e : G) : Void is do
7          elements.add(e)
8      end
9
10     opération dequeue() : G is do
11       result := elements.first
12         elements.removeAt(0)
13     end
14 }
15
16 /** Generic Sorted queue */
17 class SortedQueue<C : Comparable> inherits Queue<C>
18 {
19   method enqueue(e : C) : Void is do
20     var i : Integer
21     from i := 0
22     until i == elements.size or e > elements.elementAt(i)
23     loop
24       i := i + 1
25     end
26     elements.addAt(i, e)
27   end
28 }
```

FIG. 3.16 – Définition de classes paramétriques

```
1  class Utils {
2
3    /** Returns max(a,b) */
4    opération max<T : Comparable>(a : T, b : T) : T is do
5      result := if a > b then a else b end
6    end
7  }
```

FIG. 3.17 – Définition d'une opération générique

contexte d'une classe et dans le contexte d'une opération. Dans le cas des variables de classes, le type effectif doit être précisé explicitement par le programmeur. Ainsi pour instancier une file d'entiers on écrira :

```
var fifo : Queue<Integer> init Queue<Integer>.new
```

Dans le cas des opérations génériques le type effectif est inféré lors de la vérification de type à partir du type des paramètres effectifs. Pour appeler l'opération *max* définie précédemment, on écrira donc simplement :

```
var m : Integer init max(2, 3)
```

Le type de `max(2, 3)` est *Integer*. Ce type est inféré à partir des types des littéraux `2` et `3`. En effet, l'opération *max* définie sur le listing de la figure 3.17 définie une variable de type T qui est utilisé pour le type des deux paramètres et le type de retour de l'opération. Lors de l'appel `max(2, 3)`, le type T est associé au type *Integer* car les expression `2` et `3` sont de type *Integer*. Le type de retour de l'opération *max* étant T, le type calculé pour l'expression `max(2, 3)` est bien *Integer*.

Pour permettre l'inférence du type associé a chaque variable de type pour chaque appel, il est nécessaire d'utiliser toutes les variables de type d'une opération dans les types des paramètres de cette opération. Cette limitation est peu handicapante en pratique car il n'existe à notre connaissance que de très rares cas d'utilisation pour lequel il est interessant de ne pas utiliser les variables de types dans le type d'au moins un paramètre.

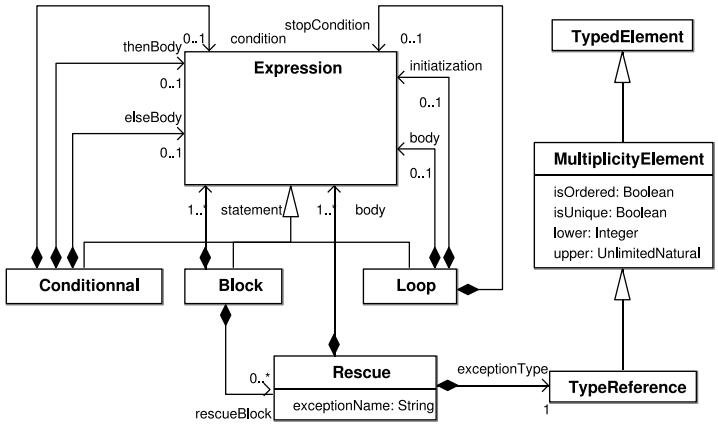
3.3.4 Expressions Kermeta

Cette section détaille les concepts retenus dans le langage d'action de Kermeta, leur sémantique et leur syntaxe. Un diagramme de classe faisant apparaître l'ensemble des classes du langage d'expression a été présenté précédemment figure 3.5. L'objectif de cette section est de donner une vision exhaustive et aussi précise que possible des éléments du langage d'action de Kermeta tout en insistant plus particulièrement sur les originalités sémantiques induites par le paradigme de la modélisation telle que la gestion des associations ou des compositions. Seules les clôtures lexicales ne sont pas détaillés dans cette section car elles font l'objet d'une discussion plus complète dans la section suivante. La sémantique formelle des principaux éléments du langage Kermeta est détaillée dans l'annexe A de ce document.

Structures de contrôles

Les premières constructions du langage d'action sont des structures de contrôle classiques tel que le bloc, la boucle ou la conditionelle. La figure 3.18 représente les classes correspondant à ces constructions dans le méta-modèle de Kermeta et précise la syntaxe associée à chacune de ces constructions.

Étant donné que toutes les constructions de notre langage d'action sont des expressions, lors de leur évaluation, elles doivent retourner une valeur. Dans le cas de la



$$
\begin{aligned}
\mathit{Block} &\rightarrow \textbf{do}\ (\mathit{Exp})^*\ (\ \textbf{rescue}\ [\ \epsilon \mid (\ \mathit{Ident}\ \textbf{:}\ \mathit{TRef}\ )\ ]\ (\mathit{Exp})^+\ )^*\ \textbf{end}\\
\mathit{Conditionnal} &\rightarrow \textbf{if}\ \mathit{Exp}\ \textbf{then}\ \mathit{Exp}\ [\ \epsilon \mid \textbf{else}\ \mathit{Exp}\ ]\ \textbf{end}\\
\mathit{Loop} &\rightarrow \textbf{from}\ \mathit{Exp}\ \textbf{until}\ \mathit{Exp}\ \textbf{loop}\ (\mathit{Exp})^*\ \textbf{end}
\end{aligned}
$$

FIG. 3.18 – Structures de contrôles.

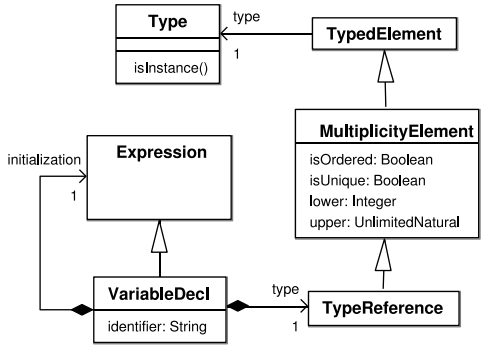
conditionelle, la valeur retournée est celle retournée par la branche ayant été exécutée ou *void*, si aucune branche n'est exécutée. Cette sémantique a été choisie en accord avec celle d'OCL. Dans le cas de la boucle, la valeur retournée est *void*.Enfin dans le cas du bloc, la valeur retournée correspond à celle retournée par la dernière expression du bloc ou *void* si le bloc est vide ou si une exception est levée lors de l'exécution du bloc. Cette solution est implantée dans un certain nombre de langages dynamiques tel que Ruby et dans OCL.

Un bloc permet d'exprimer une séquence d'instructions et constitue une unité pour le traitement des exceptions. Comme le montre le modèle de la figure 3.18 chaque bloc peut être muni d'un ensemble de clauses *rescue* qui sont évaluées si une exception est levée lors de l'exécution des instructions du corps du bloc. Chaque clause *rescue* peut avoir un paramètre dont le type correspond au type de l'exception qui peut être traité par cette clause. Une clause *rescue* sans paramètre sera exécutée quelque soit le type de l'exception ayant été levée. Lorsqu'une exception est levée dans le bloc, les clauses *rescue* sont examinées l'une après l'autre, et la première pouvant traiter l'exception est exécutée.

Déclaration de variables locales

La figure 3.19 présente l'expression correspondant à la déclaration d'une variable. Les variables en Kermeta ont un nom, un type et une multiplicité. Lors de la déclaration d'une variable il est possible de spécifier une expression d'initialisation. La seule originalité des variables Kermeta est d'avoir une multiplicité mais, cela peut être assimilé à une petite extension de la notion de tableau existant dans les langages orientés-objets.

La valeur retournée par une déclaration est la valeur à laquelle la variable est initialisée, c'est-à-dire la valeur retournée par l'expression d'initialisation ou *void* si aucune expression d'initialisation n'est donnée.



$$VariableDecl \quad \rightarrow \quad \textbf{var}\ Ident\ \textbf{:}\ TRef\ [\ \epsilon\ |\ \textbf{init}\ Exp\ ]$$

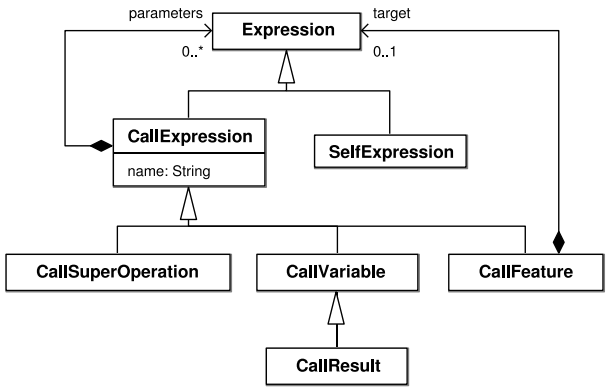
FIG. 3.19 – Déclaration et initialisation de variables.

Utilisation de variables, appel d'opérations et navigation

La figure 3.20 présente la classe correspondant aux appels d'opération et à la navigation d'associations (*CallFeature*) et la classe correspondant à l'utilisation de variables (*CallVariable*). En plus de ces deux principales classes, on distingue sur le diagramme les types d'expressions permettant la lecture des variables particulières *self* (classe *SelfExpression*) et *result* (classe *CallResult*) qui correspondent respectivement à l'objet dans le contexte duquel une opération est exécutée et au résultat de l'opération qui est exécutée. Ces deux variables sont définies et initialisées par la machine virtuelle et la variable *self* ne peut être modifiée. L'expression *CallSuperOperation* est un appel d'opération spécial, utilisable dans les redéfinitions d'opérations, et permettant d'appeler l'opération redéfinie.

Les expressions de type *CallFeature* correspondent indifféremment aux appels d'opérations et à la navigation de propriétés ou d'associations. La seule information qui soit présente pour toute instance de *CallFeature*, et permettant d'identifier la propriété à naviguer ou l'opération à appeler, est son nom. La liaison avec une propriété ou une opération est faite dynamiquement par l'interpréteur en fonction du type de l'objet cible. L'algorithme utilisé pour la liaison dynamique est tout à fait classique au paradigme orienté-objet et consiste à chercher les propriétés et opérations en remontant la hiérarchie d'héritage de la classe de l'objet cible.

Il est interessant de remarquer que le fait de ne pas différencier les propriétés et opérations à l'appel (comme cela est fait dans le langage Eiffel) permet de transformer facilement une propriété en procédure ou inversement sans que les clients aient besoin



$$
\begin{aligned}
SelfExpression &\rightarrow \textbf{self} \\
CallSuperOperation &\rightarrow \textbf{super}\ [\ \epsilon \mid \textbf{()} \mid \textbf{(}\ Exp\ (\textbf{,}\ Exp)^{*}\ \textbf{)}\ ] \\
CallVariable &\rightarrow Ident\ [\ \epsilon \mid \textbf{(}\ Exp\ (\textbf{,}\ Exp)^{*}\ \textbf{)}\ ] \\
CallResult &\rightarrow \textbf{result} \\
CallFeature &\rightarrow [\ \epsilon \mid Exp\ \textbf{.}\ ]\ Ident\ [\ \epsilon \mid \textbf{(}\ Exp\ (\textbf{,}\ Exp)^{*}\ \textbf{)} \mid \\
&\qquad \textbf{\{}\ Ident\ (\textbf{,}\ Ident)^{*} \mid (Exp)^{+}\ \textbf{\}}\ ]
\end{aligned}
$$

FIG. 3.20 – Utilisation de variables, appel d'opérations et navigation.

d'être modifiés. L'inconvénient de ce choix est que les noms des propriétés et opérations, qu'elles soit propres ou héritées, doivent être uniques pour chaque classe.

La valeur retournée par une expression de type *CallFeature* est soit la valeur de la propriété naviguée soit le résultat de l'exécution de l'opération invoquée. Dans le cas où l'expression correspond à une propriété, si la borne supérieure de la multiplicité de cette propriété est 1 alors la valeur de la propriété est retournée. Dans le cas contraire, c'est-à-dire si la multiplicité de la propriété est supérieure à 1, l'expression retourne une collection contenant les valeurs de la propriété.

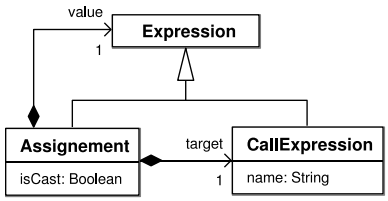
La classe *CallVariable* permet la lecture et eventuellement l'écriture des variables, c'est à dire des variables globales, des paramètres d'opérations et des variables locales. Les variables globales sont définies par l'environnement et permettent d'accéder à des ressources tel que l'entrée ou la sortie standard (variable *stdio*). Ces variables peuvent être utilisées mais sont non modifiables pour éviter les effets de bord qui pourraient en résulter.

En Kermeta, les variables sont organisées dans une pile de contextes. A la base de la pile se trouve le contexte contenant les variables globales. Lors d'un appel d'opération, un nouveau contexte, contenant la correspondance entre les noms des paramètres formels de l'opération et les paramètres effectifs, est empilé. Ensuite, lors de l'exécution du corps d'une opération, un nouveau contexte est créé et empilé à chaque début de bloc pour n'être dépilé qu'à la fin de l'exécution du bloc. Les variables locales définies dans un bloc sont ajoutées au contexte correspondant à ce bloc et ont donc une portée locale à ce bloc. Lors de l'utilisation d'une variable, le nom de la variable est recherché du haut de la pile vers le bas de la pile. En cas de conflit de nom, c'est donc la variable qui a la portée la plus restreinte qui masque les variables de même nom, qui ont une plus grande portée. La valeur retournée par une expression de type *CallVariable* est la valeur de la variable utilisée. La section 3.4.3 revient en détail sur ces mechanismes.

Affectation et modification de la valeur des propriétés

L'affectation permet de modifier la valeur des variables et des propriétés dont la borne supérieure de la multiplicité est 1. Les valeurs des propriétés dont la multiplicité est supérieure à 1 ne peuvent pas être modifiées en utilisant l'affectation mais simplement en utilisant les opérations sur les collections. Comme il a été discuté dans les sections précédentes, que l'on utilise l'affectation ou les opérations sur les collections, un point important est de préserver l'intégrité des associations et des compositions entre objets.

La valeur retournée par une affectation est la valeur qui a été affectée. En ce qui concerne les variables, seules les variables locales peuvent être affectées, les variables globales et paramètres d'opérations étant fixés par l'environnement afin d'éviter des effets de bord pouvant induire en erreur. La sémantique de l'affectation des variables locales est classique : une référence à l'objet retourné par l'expression *value* (partie droite de l'affectation) est associée à la variable cible. Le booléen *isCast* associé à chaque instance de la classe *Assignment* permet de préciser qu'il s'agit d'une affectation conditionnée par le type dynamique de la valeur à affecter. Si le type de cette valeur est



$$Assignment \quad \rightarrow \quad CallExp\ [\ :=\ |\ ?=\ ]\ Exp$$

FIG. 3.21 – Affectation.

compatible avec le type statique de la partie gauche de l'affectation, alors l'affectation a lieu normalement. Dans le cas contraire, la valeur *void* est affectée à la propriété ou à la variable.

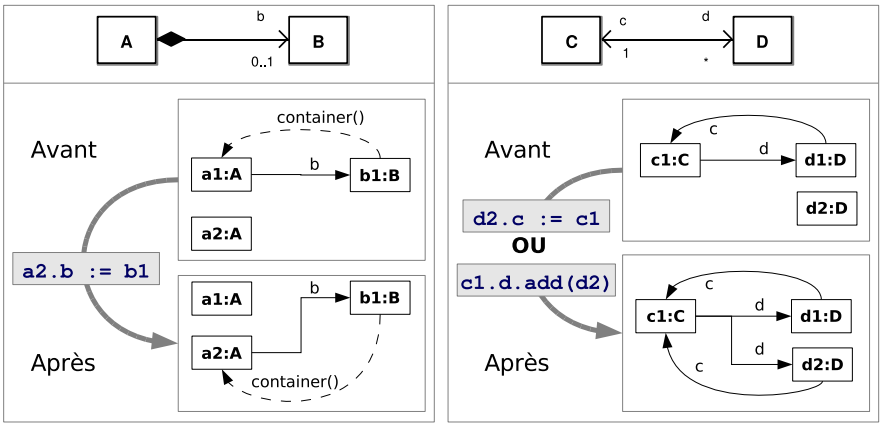


FIG. 3.22 – Sémantique de l'affectation vis-à-vis des associations et des compositions.

L'affectation d'une valeur à une propriété d'un objet entraîne, de façon atomique, les modifications nécessaires pour assurer la tenue des contraintes liées aux associations et compositions du méta-modèle. La figure 3.22 donne deux exemples d'affectation de propriété mettant en jeu ce mécanisme. Dans le premier exemple, à gauche de la figure, un objet $a1$ contient un objet b. Lorsque l'on affecte b à une autre instance $a2$ de la classe A, le conteneur de b devient $a2$ et étant donné qu'un objet ne peut avoir qu'un seul conteneur, la relation qui existait entre $a1$ et b est détruite.

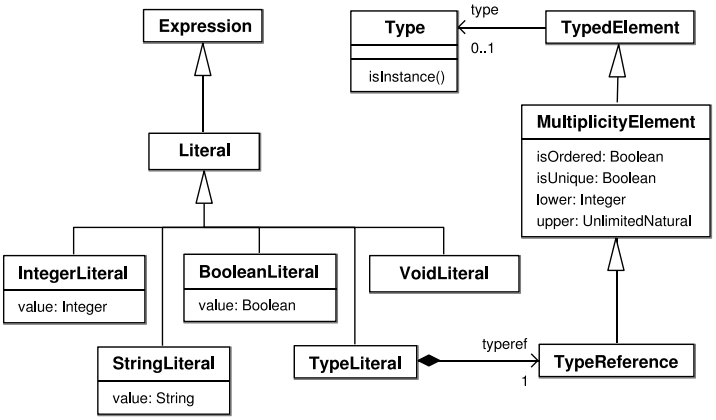
Le second exemple, à la droite de la figure 3.22 montre la manipulation d'une association entre deux classes C et D. Initialement une instance $c1$ de C et deux instances $d1$ et $d2$ de D existent et $c1$ est associé avec $d1$. La figure présente deux expressions, totalement équivalentes qui permettent d'associer $c1$ à $d2$. En effet, bien

que deux propriétés soient définies pour chaque extrémité de l'association, ces propriétés forment une unique association dont chaque extrémité est automatiquement mise à jour lorsque l'autre est modifiée.

L'ensemble des situations possibles pour l'affectation de références, associations et composition est détaillé plus formellement par les règles de sémantique naturelle de l'annexe A.

Littéraux

Les expressions littérales, présentées figure 3.23, permettent de créer ou de faire référence à des objets sans avoir à les créer explicitement. L'expression *VoidLiteral* permet de faire référence au singleton *void* qui désigne l'objet nul en Kermeta. Les expressions de type *BooleanLiteral*, *IntegerLiteral* et *StringLiteral* permettent respectivement de créer des booléens, des entiers et des chaînes de caractères. Enfin l'expression *TypeLiteral* permet de faire référence au type du langage pour instancier, par exemple, un objet à partir de sa classe.



$$
\begin{aligned}
VoidLiteral &\rightarrow \textbf{void} \\
BooleanLiteral &\rightarrow [\ \textbf{true} \mid \textbf{false}\ ] \\
InteterLiteral &\rightarrow Integer \\
StringLiteral &\rightarrow \text{"}\ String\ \text{"} \\
TypeLiteral &\rightarrow TRef \\
TRef &\rightarrow [\ \epsilon \mid \textbf{set} \mid \textbf{oset} \mid \textbf{seq} \mid \textbf{bag}\ ]\ Ident\ (\textbf{.}\ Ident)^{*} \\
&\quad\ [\ \epsilon \mid [\ Integer\ \textbf{..}\ Integer\ ]]
\end{aligned}
$$

FIG. 3.23 – Expression littérales.

3.3.5 Clôtures lexicales

Le dernier type d'expression de Kermeta est une forme de fonctions : Les clôtures lexicales. La spécificité des clôtures lexicales est qu'elles comportent non seulement un corps mais aussi une environnement dans lequel ce corps doit être exécuté. Les clôtures lexicales ont été initialement introduites dans les langages fonctionels [SGLS75] mais ont été adapté dans des langages orienté-objets tels que CLOS [BR88] ou smalltalk-80 [GR83] et ses descendants. Plus récement les clôtures lexicales on été intégrées a des langages orienté-objets statiquement typés tels que Eiffel [Mey] ou C# 2.0.

Cette construction fonctionnelle a été ajouté à Kermeta pour permettre la définition et l'utilisation dans un contexte de typage statique d'itérateurs analogues à ceux d'OCL. Dans cette section, nous détaillons tout d'abord en quoi les clôtures lexicales offrent une solution a ce problème. Nous discutons ensuite de l'impact de l'introduction des fonctions dans le système de types de Kermeta. Enfin nous présentons l'utilisation des clôtures lexicales sur des exemples simples.

Exemple et motivations

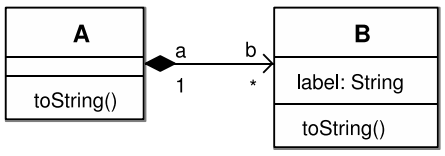


FIG. 3.24 – Un méta-modèle très simple.

Les itérateurs OCL tels que *select*, *collect*, *reject* ou *forall*, aussi appelés fermetures, sont des constructions particulièrement adaptées à la navigation dans les modèles parce qu'elles simplifient grandement la manipulation des collections. L'expression suivante, dans le contexte du méta-modèle de la figure 3.24, est un exemple d'expression que l'on souhaite pouvoir utiliser ; on suppose qu'il existe une opération *toUpper* sur la classe String :

```
myA.b.collect{ o | o.label }.one.toUpper
```

On rappelle que l'itérateur *collect* d'OCL permet d'évaluer une expression (ici `o.label`) sur l'ensemble des éléments d'une collection et de rendre une collection contenant la séquence de résultats obtenus. L'expression proposée permet donc de collecter l'ensemble des *label* des instances de B associées à myA, de selectioner un de ces *label* (opération *one*) et de le mettre en majuscule (opération *toUpper*).

La vérification de type sur une expression telle que celle-ci pose deux problèmes. Tout d'abord, pour pouvoir vérifier la sous expression `o.label` il est nécessaire de déterminer le type de `o` qui dépend du type des éléments contenus dans la collection `myA.b`. Ensuite, le type des éléments de la collection `myA.b.collect{ o | o.label }`

dépend du type statique de l'expression `o.label` contenue dans le corps de l'itérateur *collect*. Pour résoudre ces problèmes, encore une fois, deux solutions sont possibles.

La première solution est d'ajouter dans le langage des constructions spécifiques correspondants aux itérateurs et de mettre à jour le vérificateur de types afin de traiter correctement ces itérateurs. Cette solution est utilisée par la plupart des langage orienté-objets statiquement typés tels que Java ou C♯ qui proposent, par exemple, une construction *foreach* permettant d'itérer sur une collection. Par le passé, c'est la solution qui a été adoptée pour implanter les itérateurs analogues à OCL dans le langage Xion. L'inconvénient de cette technique est qu'elle oblige à câbler dans le langage les itérateurs disponibles et à concevoir les règles de typage correspondantes de façon ad-hoc. Dans le langage Kermeta nous avons choisi de ne pas retenir cette solution et d'adopter une approche plus générale permettant au programmeur lui même de concevoir et d'ajouter facilement ses propres itérateurs. Cela permet de consever l'aspect minimale du langage en repoussant dans des bibliothèques les constructions semblables aux collections d'OCL.

La seconde solution est donc d'étendre le langage pour permettre de considérer les itérateurs comme des opérations ordinaires de la classe *Collection* de la bibliothèque standard du langage. Il est alors possible, simplement en définissant des opérations, de construire autant d'itérateurs que l'on souhaite. En pratique un tel mécanisme est implanté dans les langages orientés-objets dynamiques tel que Ruby [TH00] par exemple. Cependant il est possible d'implanter de telles constructions dans un langage statiquement typé à condition d'étendre le système de types afin de gérer les opérations génériques et les types fonctionnels.

Considérons, par exemple, l'itérateur *collect* utilisé dans l'exemple précédent. Cet itérateurs permet d'évaluer une expression sur chacun des éléments d'une collection et de retourner la collection des résultats obtenus. L'opération *collect* correspondante (définie dans la classe *Collection*) a donc un paramètre : l'expression à évaluer sur chaque élément de la collection. L'opération *collect* retourne une collection d'éléments dont le type est donné par le type statique de l'expression passée en paramètre. Cet exemple met en lumière deux constructions qui doivent être supportées par le système de type :

- le paramètre de l'opération étant une expression à évaluer sur les éléments de la collection, c'est en fait une fonction à appliquer sur les éléments de la collection. Pour permettre de typer ce genre de paramètres, le système de types doit supporter les types fonctionnels.
- le type du résultat de l'opération *collect* n'est pas connu à priori car il dépend du type résultat de la fonction passée en paramètre. En d'autres termes, le type de retour de l'opération est déterminé, pour chaque appel à l'opération, en fonction du type du paramètre. Le système de types doit donc permettre de définir des opérations paramétriques.

La figure 3.25 présente l'implantation de l'opération *collect* sur la classe *Collection*. La classe *Collection* est paramétrée par la variable de type G. De la même manière qu'un paramètre de type G est défini dans le contexte de la classe collection, on définit le paramètre de type T dans le contexte de l'opération *collect*. Il est alors possible

```
1  abstract class Collection<G> {
2  [...]
3      operation collect<T> (collector : <G->T>) : Sequence<T> is do
4          result := Sequence<T>.new
5          from var it : Iterator<G> init self.iterator
6          until it.isOff
7          loop
8              result.add(collector(it.next))
9          end
10     end
11 [...]
12 }
```

FIG. 3.25 – Implantation de l'itérateur *collect*.

d'exprimer, en fonction de ces deux paramètres de type, le type du paramètre formel *collector* et le type de retour de l'opération. Le type du paramètre est *G->T*, c'est-à-dire une fonction de *G* dans *T*. Le type de retour est *Sequence<T>*, c'est-à-dire une collection ordonnée d'éléments de type *T*. L'implantation de l'opération est ensuite assez simple : on initialise un ensemble résultat (ligne 4) puis on parcourt la collection à l'aide d'un itérateur. Enfin, on applique la fonction *collector* grâce à l'appel `collector (it.next)` et on ajoute les résultats obtenus à la collection résultat (ligne 8).

Pour conclure cette section, revenons à l'exemple de l'expression discuté au début de cette section :

```
myA.b.collect{ o | o.label }.one.toUpper
```

Cette expression est un racourci syntaxique pour un simple appel d'opération et peut être réécrite pour mettre en evidence l'utilisation d'une fonction comme paramètre de l'opération *collect* :

```
myA.b.collect( function { o : B | o.label } ).one.toUpper
```

Dans cette expression le type du paramètre `function { o : B | o.label }` est *B->String*. D'après la signature de l'opération *collect* (listing de la figure 3.25) le type de retouné par l'appel à *collect* est *Sequence<String>*. Ce type est conforme à ce que nous attendions et permet d'assurer la vérification statique des types dans l'expression de départ.

Impact des fonctions sur le système de type

La figure 3.26 présente les modifications apportées au système de types pour supporter d'une part les opérations paramétriques et d'autre part les types fonctionnels. Afin de permettre la définition d'opérations génériques, les opérations contiennent maintenant un ensemble de variables de type analogue aux variables de type précédemment associées aux classes (classe *TypeVariable*). Ces variables de type doivent être utilisées

au moins une fois dans le type des paramètres de l'opération et peuvent être utilisées dans le type de retour de l'opération. Il est nécessaire d'utiliser les variables dans le type des paramètres car dans le cas d'opérations génériques le type concret associé aux variables de type est inféré, pour chaque appel, à partir du type des paramètres effectifs. Cette restriction, peu contraignante en pratique, permet d'alléger l'appel aux opérations génériques car les types concrets associés aux variables de type n'ont pas besoin d'être déclarés.

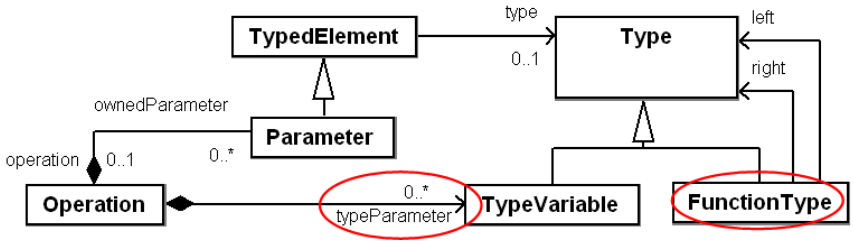


FIG. 3.26 – Ajout des opérations paramétriques et des types fonctionnels.

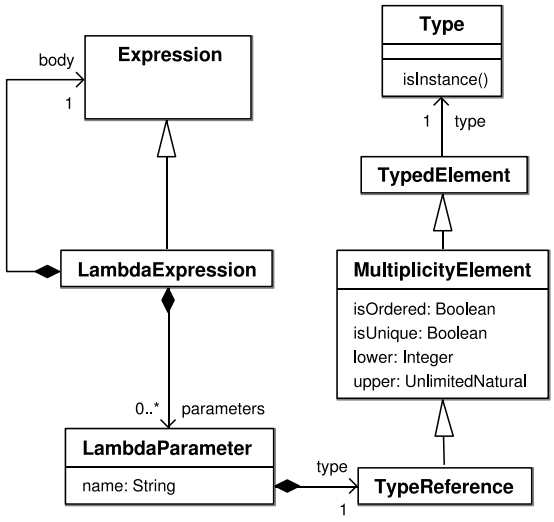
L'intégration des types fonctionnels dans le système de types est réalisé par l'ajout d'une sous-classe de la classe abstraite *Type* nommée *FunctionType* et comportant simplement deux associations vers un type source et un type destination. Cette nouvelle classe de types peut a priori être utilisée en lieu et place de tout autre type dans le langage. Cependant, dans la première version de Kermeta nous avons limité son utilisation aux variables et paramètres d'opérations, d'une part parce que cela est suffisant pour l'implantation des itérateurs et, d'autre part dans un souci de simplifier l'implantation du langage.

Utilisation des fonctions en Kermeta

Bien que Kermeta ne soit pas à proprement parler un langage fonctionnel, les clôtures lexicales ont été incluses au langage d'action afin de supporter l'implantation d'itérateurs analogues à ceux d'OCL. Il est donc possible en Kermeta de définir des fonctions grâce à la classe *LambdaExpression*. Comme le montre la figure 3.27, une clôture lexicale a un ou plusieurs paramètres et un corps constitué d'une expression. La valeur retournée par la définition d'une fonction est la fonction elle-même.

Le listing de la figure 3.28 présente la définition et l'utilisation d'une fonction permettant d'élever un entier au carré. La ligne 2 définit une variable *carre* dont le type est un type fonctionnel d'entier vers entier (<Integer -> Integer>). La ligne 4 définit et affecte à la variable *carre* une fonction ayant un paramètre entier et retournant le carré de cet entier. La fonction peut alors être invoquée comme le montre la ligne 6 avec comme paramètre l'entier 2.

Les fonction Kermeta sont des clôtures lexicales : quelque soit le contexte dans lequel elles sont invoquées, sont évaluées dans le contexte de leur définition. Dans les langages orientés-objets, les opérations sont exécutées dans le contexte de la classe



$$LambdaExpression \rightarrow \textbf{function (}\ Ident\ \textbf{:}\ TRef\ (\textbf{,}\ Ident\ \textbf{:}\ TRef)^{*}\ \textbf{)\{}\ (Exp)^{+}\ \textbf{\}}$$

FIG. 3.27 – Lambda expressions.

dans laquelle elles sont définies. Cela permet, dans le corps des opérations de faire référence aux propriétés de la classe courante. De manière analogue, le fait d'évaluer les fonction dans le contexte du bloc de leur définition permet non seulement l'utilisation des éléments de la classe courante mais aussi l'utilisation des variables locales.

```
1 // définition d'une variable de type fonctionnel
2 var carre : <Integer->Integer>
3 // définition de la fonction
4 carre := function { i : Integer | i*i }
5 // appel de la fonction et affichage du résultat
6 stdio.writeln( carre(2) )
```

FIG. 3.28 – Définition et appel d'une fonction.

La raison première pour laquelle les fonctions ont été introduites dans Kermeta est pour permettre aux opérations de prendre des fonctions comme paramètres. Le premier listing de la figure 3.29 présente la définition de la fonction *times* de la classe *Integer* de Kermeta. Cette opération permet d'exécuter une fonction *body* passée en paramètre un nombre de fois correspondant à l'entier sur lequel l'opération est appelé.

Le second listing de la figure 3.29 présente deux appels à cette fonction pour afficher cinq fois "hello world" sur la sortie standard. Le premier appel, des lignes 1 à 5, consiste à définir une fonction *hello* permettant d'afficher "hello world" sur la sortie standard puis

de passer cette fonction en paramètre de l'opération *times*. Le second appel, ligne 8, est équivalent mais écrit dans une syntaxe alternative, il permet simultanément de définir la fonction et de la passer en paramètre à l'opération *times*. Cette syntaxe permet de rendre l'utilisation des itérateurs tels que *select*, *collect*, *reject* ou *detect* analogue à ce qui existe dans OCL.

```
1  class Integer inherits Numeric
2  {
3  [...]
4     opération times(body : <Integer->Object>) : Void is do
5      from var i : Integer init 0
6      until i == self
7      loop
8        body(i)
9        i := i + 1
10     end
11    end
12 [...]
13 }
```

```
1  // Définition de la fonction
2  var hello : <Integer->Object> init
3  function { i:Integer | stdio.writeln("hello␣world␣" + i.toString) }
4  //Appel à l'opération times
5  5.times(hello)
6
7  // Définition de la fonction dans le contexte de l'appel d'opération
8  5.times { i | stdio.writeln("hello␣world␣" + i.toString) }
```

FIG. 3.29 – Définition et utilisation de l'opération *times* de la classe *Integer*.

Dans l'implantation actuelle de l'intérpréteur Kermeta, les clôture lexicales sont implantées comme des appels de fonctions dans le contexte de leurs définition. Cela nécessite le stockage d'un contexte associé a chaque fonction et l'installation de ce contexte pour chaque appel de fonction. Dans la pratique, ce mode de fonctionnement peu être couteux en mémoire et en temps d'execution. Il existe dans la littérature [SA00] des techniques permettant d'optimiser l'interprétation et la compilation des clôture lexicale. Les problèmes de performance ne faisant pas partie des exigences premières pour l'environnement Kermeta nous n'avons pas mis en place ces techniques dans les premières versions de Kermeta.

3.3.6 Contraintes

En Kermeta, les contraintes sont utilisées à la fois pour assurer la consistance des objets (et donc des modèles) manipulés et pour détecter des comportements erronés lors de l'exécutions d'opérations. La première utilisation correspond à l'utilisation (courante en ingénierie dirigée par les modèles) de contraintes OCL sur un méta-modèle

qui permet de préciser les règles de bonne formation des modèles conformes à ce méta-modèle. La seconde utilisation est analogues au contrats qui existent dans les langages orientés-objets comme Eiffel [Mey92].

Afin de permettre l'écriture de contraintes sur les méta-modèles Kermeta, nous avons ajouté la possibilité de définir des contraintes directement dans le langage sous la forme d'invariants pour les classes et de pré-conditions et de post-conditions pour les opérations. L'implantation des contraintes dans Kermeta est similaire au contrats du langage Eiffel et aux possibilités offertes par OCL sur des modèles UML. Cette section détaille comment les contraintes sont intégrées au méta-modèle de Kermeta, présente un exemple simple d'utilisation de contraintes et discute de la sémantique des contraintes vis-à-vis de l'héritage.

Intégration des contraintes

La figure 3.30 présente un diagramme de la partie du méta-modèle de Kermeta qui permet de définir les contraintes. Les contraintes sont représentées par la classe *Constraint* et il existe trois types de contraintes : les invariants, les pré-conditions et les post-conditions. Les invariants doivent impérativement être contenus par une définition de classe. Les pré-conditions et post-conditions doivent être contenues par des opérations à travers les propriété correspondantes[1].

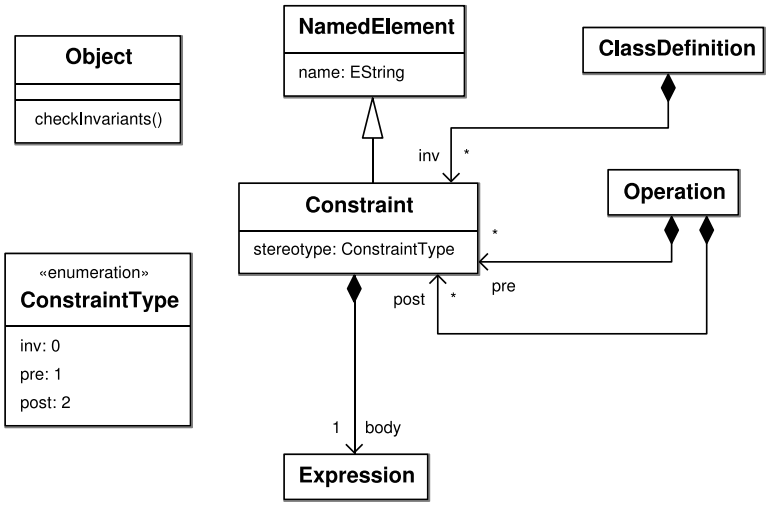


FIG. 3.30 – Contraintes Kermeta.

L'opération *checkInvariant* définie dans la classe *Object* permet d'appeler l'ensemble des vérifications d'invariant sur un objet. Cette opération renvoie *true* si l'ensemble des invariants sont vérifiés et lève une exception dans le cas contraire. L'exception contient la liste des invariants violés.

[1]Les contraintes sont intégrées au langage Kermeta depuis la version 0.2.2 et ont été implantées par Jean-Marie Mottu.

L'ensemble des expressions Kermeta peut être librement utilisé dans les contraintes Kermeta. Il appartient à l'utilisateur de s'assurer que les expressions contenues dans les contraintes n'ont pas d'effet de bord. Dans le futur, nous envisageons d'implanter des vérifications statiques pour interdire les constructions, tel que l'affectation d'un attribut de classe, qui ont toujours des effets de bord. Cependant, dans le cas général le problème de déterminer si une expression Kermeta a des effets de bord n'est pas décidable car comme en OCL, on permet l'appel de fonctions arbitraires.

En plus des expressions Kermeta, l'opérateur spécifique *@pre* peut être utilisé dans les post-conditions afin de faire référence à un état avant l'exécution de l'opération. Cet opérateur est directement issu d'OCL et permet d'exprimer les changements d'états attendus à l'issue de l'exécution d'une opération. Cet opérateur correspond à l'opérateur *old* du langage Eiffel.

Exemple d'utilisation

Le listing de la figure 3.31 présente un exemple simple d'utilisation des contraintes dans une classe Kermeta. La classe *Account* possède une propriété *balance* qui est un entier qui doit toujours être positif ou nul. L'invariant de classe *positiveBalance* (ligne 14) permet de s'en assurer. L'opération *deposit* permet d'incrémenter la propriété balance d'un montant *amount* strictement positif. La pré-condition de l'opération *deposit* (ligne 6) permet de s'assurer que le montant est bien positif. Enfin, une fois l'opération exécutée, sa post-condition (ligne 10) permet de s'assurer, grâce à l'opérateur *@pre*, que la valeur de la propriété *balance* a bien été incrémentée.

```
1  class Account {
2
3    attribute balance : Integer
4
5    operation deposit(amount : Integer) : Void is
6    pre positiveAmount is amount > 0
7    do
8      balance := balance + amount
9    end
10   post balanceUpdated is balance == balance@pre + amount
11
12   [...]
13
14   invariant positiveBalance is balance >= 0
15 }
```

FIG. 3.31 – Exemple de contrats dans une classe Kermeta.

Dans l'environnement Kermeta, les invariants sont vérifiés par l'appel à l'opération *checkInvariants* définie sur la classe *Object*. Les pré-conditions et les post-conditions peuvent être automatiquement vérifiées par l'interpréteur Kermeta lors de chaque appel d'opération. Ces vérifications restent toutefois optionnelles car leur impact sur les

performances des programmes n'est pas négligeable.

Héritage des contraintes

Cette section précise la sémantique des contraintes vis-à-vis de l'héritage, c'est-à-dire la sémantique des invariants définis dans des sous-classes et la sémantique des pré-conditions et des post-conditions définies sur des opérations redéfinies.

Tout d'abord, les invariants d'une classe sont hérités au même titre que les propriétés ou les opérations par ses sous-classes. Dans une sous-classe, il est uniquement possible de définir de nouveaux invariants. Cela permet de s'assurer que tous les objets conformes à un type vérifient les invariants associés à ce type.

En ce qui concerne les pré-conditions et les post-conditions associées à une redéfinition le problème est un peu plus complexe. En effet, l'opération redéfinie doit pouvoir être utilisée "à la place" de l'opération qu'elle redéfinit :

- L'ensemble de domaine d'entrée de l'opération originale doit être accepté : la précondition ne peut pas être plus contrainte.
- Le résultat doit respecter le contrat de l'opération originale : la post-condition ne peut pas être relâchée.

Ainsi, les pré-conditions d'une redéfinition sont évaluées en disjonction avec la précondition de l'opération originale et les post-conditions sont évaluées en conjonction avec la post-condition originale. Cette sémantique est conforme à ce qui implanté dans le langage Eiffel.

3.4 Système de types de Kermeta

Pour la construction de Kermeta, nous avons choisi un système de types statique afin de permettre un maximum de vérifications statiques. Comme il a été évoqué dans la section précédente, afin d'assurer les vérifications de types tout en conservant une bonne utilisabilité du langage nous avons été amenés à modifier et à compléter le système de types de EMOF. Nous avons en particulier ajouté la généricité et les types fonctions. L'ensemble de ces modifications implique que le système de types de Kermeta présente non seulement l'ensemble des caractéristiques d'un système de types orienté-objets mais aussi des constructions empruntées au paradigme fonctionnel. Les sous-sections suivantes détaillent les différentes facettes du système de types de Kermeta.

3.4.1 Les types Kermeta

La figure 3.32 présente la hiérarchie complète des types Kermeta. On retrouve dans cette hiérarchie les types primitifs (*PrimitiveType*), les énumérations (*Enumeration*) et les classes (*Class*) de EMOF. Les variables de type (TypeVariable) ont été ajoutées pour supporter la généricité. Les types produits (*ProductType*) et les types fonctions (*FunctionType*) permettent de créer de nouveaux types par composition de types existants. Enfin la classe *VoidType* correspond au type particulier *Void* qui est utilisé comme sous-type universel.

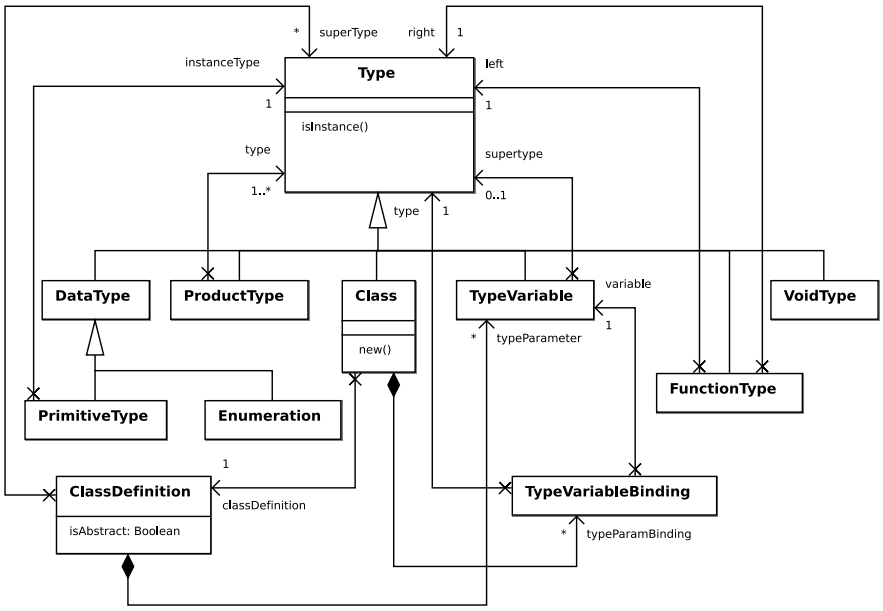


FIG. 3.32 – Hiérarchie des types en Kermeta.

3.4.2 Relations et règles de sous-typage

Cette section détaille les règles de sous-typage qui s'appliquent sur les différents types Kermeta présentés précédemment et qui définissent la relation de sous typage. Si τ_1 et τ_2 sont deux types, nous utilisons la notation $\tau_1 \preceq \tau_2$ pour préciser de τ_1 est un sous-type de τ_2.

Règles générales

Pour commencer, les règles suivantes sont valables (par convention) pour l'ensemble des types Kermeta :

- Un type τ est un sous-type de lui même : $\forall \tau, \tau \preceq \tau$. Cette règle purement technique permet de simplifier l'écriture du système de type.
- La relation de sous-typage est transitive : si τ, τ' et τ'' sont des types alors si $\tau \preceq \tau'$ et $\tau' \preceq \tau''$ alors $\tau \preceq \tau''$.
- Le type $Object$ est un super-type universel : $\forall \tau, \tau \preceq Object$.
- Le type $Void$ est un sous-type universel : $\forall \tau, Void \preceq \tau$. En pratique, l'unique valeur du type *Void* est le singleton *void*.

Relation de sous-typage entre classes

L'ajout de la généricité dans Kermeta impose de corriger la définition de la relation de sous-typage de EMOF qui se base uniquement sur le sous-classage. En effet, les éventuels paramètres de types associés à une classe doivent être pris en compte pour

le sous-typage. Si l'on considère par exemple les quatre classes définies sur le listing suivant :

```
1 class Collection<G> { ... }
2 class Stack<F> inherits Collection<F> { ... }
3 class Map<K, V> inherits Collection<V> { ... }
4 class Properties inherits Map<String, String> { ... }
```

A partir de la classe $Collection < G >$ il est possible de construire un ensemble de types en substituant la variable de type par un type concret. Vis-à-vis du sous-typage on a $Collection < \tau > \preceq Collection < \tau' >$ si et seulement si $\tau = \tau'$. Si l'on prend un exemple cela signifie par exemple qu'*une collection de chaînes n'est pas une sorte de collection d'objet*. Cela peut paraître contre-intuitif au premier abord mais cela se justifie par le fait que dans une collection d'objets il est possible d'ajouter n'importe quel type d'objets, alors que dans une collection de chaînes on ne peut ajouter que des chaînes. De ce point de vue il est clair qu'une collection de chaîne ne peut se substituer à une collection d'objets.

Considérons maintenant la classe $Stack < H >$. Cette classe définit une variable de type H et hérite du type $Collection < H >$. Il est important de noter que l'héritage est une relation entre une classe et un ou plusieurs super-types : ce n'est pas une relation entre classes. Ceci constitue une différence importante par rapport à ce qui existe dans EMOF et résulte de l'introduction des génériques. Cette relation d'héritage signifie que pour tout type τ on a : $Stack < \tau > \preceq Collection < \tau >$.

La classe $Map < K, V >$ définit deux variables de type K et V et hérite de $Collection < V >$. Du point de vue du sous-typage cela signifie que pour tout type τ et τ' on a, de la même manière que précédemment : $Map < \tau, \tau' > \preceq Collection < \tau' >$.

Enfin, la classe *Properties* ne définit aucun paramètre générique mais hérite du type $Map < String, String >$. Cela signifie que le type *Properties* (engendré par la classe *Properties*) est un sous-type de $Map < String, String >$: $Properties \preceq Map < String, String >$. En utilisant la transitivité de la relation de sous-typage on peut alors déduire que $Properties \preceq Collection < String >$.

En procédant comme pour la classe *Properties*, c'est-à-dire en navigant les relations d'héritages et en substituant les variables de types, il est possible de calculer et d'énumérer l'ensemble des super-types d'une classe en fonction des variables de types qu'elle définit. Pour une classe $C < \tau_1, \ldots, \tau_n >$, on notera $\Delta C(\tau_1, \ldots, \tau_n)$ cet ensemble de super-types. Grâce à cet ensemble, la relation de sous-typage entre types engendrés par des classes s'exprime simplement :

$$X < \tau_1^X, \ldots, \tau_n^X > \preceq Y < \tau_1^Y, \ldots, \tau_n^Y > \iff X < \tau_1^X, \ldots, \tau_n^X > \in \Delta Y(\tau_1^Y, \ldots, \tau_n^Y)$$

Types primitifs et types énumérés

Les *DataType* Kermeta, c'est-à-dire les types primitifs et les types énumérés, sont proche de ce qui existe dans EMOF. L'unique différence, comme nous l'avons discuté dans le paragraphe consacré aux types primitifs de la section 3.2.3, est que les types primitifs Kermeta doivent être mis en relation avec un type concret pour pouvoir être

utilisés. Du point de vue de la vérification de types, les types primitifs peuvent donc être traités de façon transparente : ils sont simplement substitués par le type auquel ils correspondent.

En ce qui concerne les types énumérés, aucune relation de sous-typage entre énumérations n'est implantée ni dans EMOF, ni dans Kermeta. Si τ_E et τ'_E sont des types énumérés on a donc $\tau_E \preceq \tau'_E \Rightarrow \tau_E = \tau'_E$.

Types produits et types fonctions

Un type produit est un type construit comme une séquence de types existants. Les types produits ont été introduit dans Kermeta pour permettre de représenter et de créer les types fonctions correspondant à des fonctions ou des opérations ayant plusieurs paramètres. En Kermeta, si τ_1, ..., τ_n sont des types, le type produit correspondant à $\tau_1 \times \cdots \times \tau_n$ est noté $[\tau_1, \ldots, \tau_n]$. Vis-à-vis du sous-typage, les types produits ont un comportement simple :

$$\tau_1 \times \cdots \times \tau_n \preceq \tau'_1 \times \cdots \times \tau'_n \Longleftrightarrow \forall i \in [1..n], \tau_i \preceq \tau'_i$$

Les types fonctions sont utilisés non-seulement pour représenter le type des clôtures lexicales de Kermeta mais aussi pour représenter le type des opérations. Un type fonction est construit à partir de deux types existant : un type "paramètre" et un type "résultat". Si τ_p et τ_r sont deux types, on notera $\tau_p \rightarrow \tau_r$ le type fonction ayant τ_p comme type paramètre et τ_r comme type résultat. La règle de sous-typage pour les types fonctions est covariante pour le type résultat, et contravariante pour le type paramètre :

$$\tau_p \rightarrow \tau_r \preceq \tau'_p \rightarrow \tau'_r \Longleftrightarrow \tau'_p \preceq \tau_p \text{ et } \tau_r \preceq \tau'_r$$

Variables de types

Les variables de types (classe *TypeVariable* sur la figure 3.32) sont utilisées pour la définition de classes et d'opérations génériques. A l'intérieur des classes ou opérations auxquelles elles appartiennent, les variables de type sont utilisées comme des types. Un super-type peut être associé à chaque variable de type pour permettre de contraindre les types pouvant être associés à la variable. Dans le cas où aucun super-type n'est fourni, tout se passe comme si le super-type donné était *Object*. Le fait de fournir un super-type τ à une variable de type V permet de s'assurer que seuls les sous-types de τ sont acceptables comme types effectifs pour la variable V. On a donc : $V \preceq \tau$.

3.4.3 Environnement de typage

L'environnement de typage de Kermeta, que l'on notera E dans la suite, contient toutes les informations de contexte utilisées lors des vérifications de types statiques d'un programme Kermeta. Le contenu de cet environnement de typage varie en fonction des éléments sur lesquels portent les vérifications.

Table des définitions de types

L'environnement de typage contient dans tous les cas l'ensemble des packages et l'ensemble des types définis dans le contexte de l'unité de compilation à vérifier. Dans les programmes Kermeta les définitions types sont identifiées par leurs noms qualifiés. Le nom qualifié d'une définition de type est calculé à partir du nom qualifié du package qui la contient. L'ensemble de définitions de types de l'environnement de typage se présente sous la forme une table contenant l'ensemble des définitions de types (c'est-à-dire les classes, les types primitifs et les énumérations) indexés par leur noms qualifiés.

Par défaut seul l'ensemble des types de l'unité de compilation à vérifier est présent mais la directive *require* permet d'importer les définitions de types d'autres unités de compilations. L'argument de la directive *require* est l'URI de l'unité à importer. L'unité importée peut être un fichier source Kermeta, un modèle Kermeta ou un modèle ECore. Pour chacun de ces formats d'unité, une stratégie adaptée permet de renseigner l'environnement de typage.

Les listings de la figure 3.33 présentent un exemple d'utilisation de la directive *require*. Le premier listing (fichier *unit1.kmt*) défini une classe A dans un package $p1$. Dans le second listing (fichier *unit2.kmt*) la directive *require* de la ligne 4 permet d'importer la première unité de compilation, c'est-à-dire les types qu'elle contient. La ligne 11 de ce même listing montre une utilisation du type *p1 : :A*. Il est interessant de remarquer l'autre directive *require* utilisée à la ligne 3 du second listing. Cette directive permet d'importer la bibliothèque standard de Kermeta.

La directive *using* permet d'ajouter à l'environnement de typage d'une unité de compilation un ensemble ordonné de packages pour la recherche de types à partir de leur nom simple. L'argument de la directive *using* est le nom qualifié d'un package qui doit appartenir à l'environnement de typage. Ceci permet dans un grand nombre de cas d'éviter d'avoir à utiliser le nom qualifié des types utilisés et permet donc de simplifier l'écriture des programmes Kermeta. La résolution d'une référence à un type Kermeta se fait donc par une recherche dans les packages désignés par une directive *using* si un nom simple est fournit et dans la table des types si un nom qualifié est fourni.

Dans le second listing de la figure 3.33, la directive *using* utilisée à la ligne 6 permet d'eviter d'écrire le nom qualifié du type *p1 : :A* lors de son utilisation. La définition de la propriété *a2* de la classe B illustre cette possibilité.

Table des symboles

En plus des informations de contexte globale à l'unité de compilation à vérifier, l'environnement de typage contient les informations propres à la classe, la propriété, l'opération ou la contrainte en cours de vérification. Ces informations sont regroupées dans une pile de table de symboles. Chaque symbole est une chaîne de caractère à laquelle on associe un type. Ces symboles peuvent correspondre à différents éléments : des propriétés, des opérations, des paramètres ou des variables.

Les tables de symboles sont organisées en une pile qui correspond aux différents contextes dans lesquels les symboles sont définis. la figure 3.34 présente un exemple de pile de symboles pour un programme simple. La recherche d'un symbole commence

```
1  package p1;
2
3  class A {
4    [...]
5  }
6  [...]
```

fichier unit1.kmt

```
1  package p2;
2
3  require kermeta // Permet d'importer la librairie standard
4  require "unit1.kmt" // Permet d'importer l'unité unit1
5
6  using p1 // permet d'éviter de préciser le nom qualifié des types
7           // contenus dans p1 lors de leur utilisation dans ce fichier
8
9  class B {
10   // une propriété de type p1::A
11   attribute a1 : p1::A
12   // une autre propriété de type p1::A (grâce à la directive using)
13   attribute a2 : A
14   [...]
15 }
16 [...]
```

fichier unit2.kmt

FIG. 3.33 – Exemple d'utilisation des directives *require* et *using*.

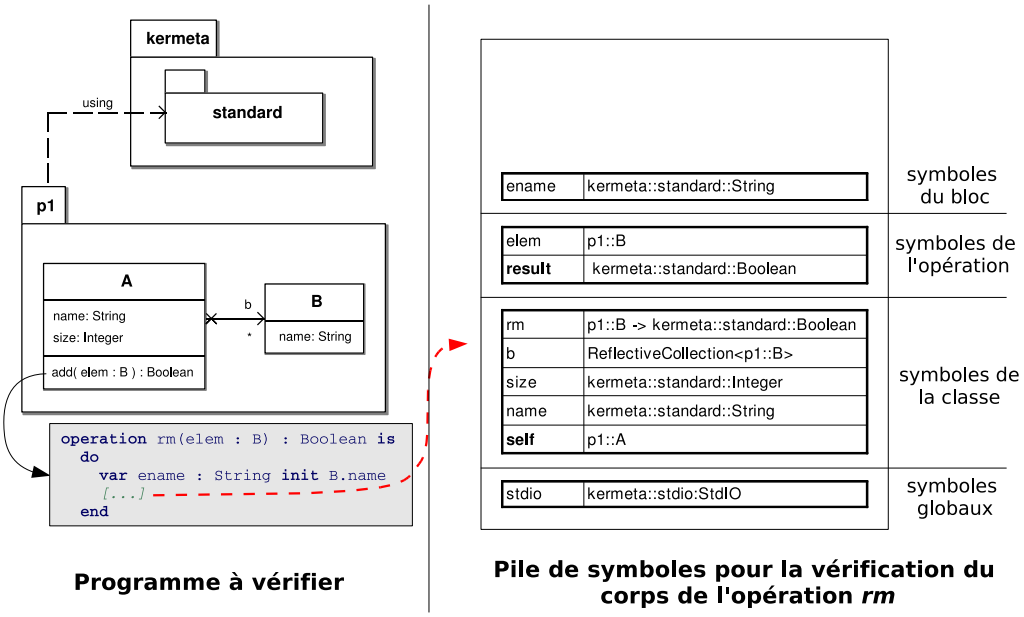


FIG. 3.34 – Exemple de pile de symboles.

par le sommet de la pile puis descend dans la pile jusqu'à trouver le premier symbole correspondant. La conséquence directe de cette politique de recherche est que, en cas de conflit de noms, c'est le symbole dont la portée est la plus faible qui masque les symboles de portée plus grande.

A la base de la pile se trouvent des variables globales. La seule variable globale définie à ce jour est une variable *stdio* dont le type est définit par la classe *kermeta : :stdio : :StdIO* et qui permet l'accès a l'entrée et la sortie standard du processus dans lequel s'exécute un programme Kermeta. Cette variable est automatiquement positionnée par l'environnement Kermeta lors son initialisation.

Les symboles propres à la classe à vérifier sont empilés juste au dessus des variables globales. Parmi ces symboles, la variable particulière *self* représente l'objet courant. Le type qui lui est associé est le type correspondant à la classe à vérifier. Les autres symboles de ce niveau sont les propriétés et opérations disponibles dans la classe à vérifier, c'est-à-dire les propriétés et les opérations qu'elle contient et celles dont elle hérite.

Le niveau suivant dans la pile des symboles est utilisé pour les paramètres et valeurs de retours des opérations et des propriétés dérivées. Au dessus de ce niveau on trouve les contextes correspondant aux blocs et aux fonctions. Les contextes correspondant aux blocs contiennent les symboles correspondants aux variables locales. Les contextes correspondant aux clôtures lexicales contiennent les paramètres libres.

3.4.4 Typage de la structure

Cette section présente les contraintes de types portant sur la structure d'un programme Kermeta, c'est-à-dire les contraintes associées au classes, opérations et propriétés.

Typage des packages

Une seule règle portant sur les noms des types contenus dans un package permet de s'assurer de la bonne formation d'un package : à l'intérieur d'un package les classes, énumérations et types primitifs doivent avoir des nom uniques. Cela permet d'assurer que les noms qualifiés de deux types soient toujours différents dans la table des types de l'environnement de typage.

Typage des classes

Une classe Kermeta $C < v_1, \ldots, v_n >$ est correctement typée si l'ensemble de règles suivantes est vérifié :

- Les noms des opérations qui ne sont pas des redéfinitions et des propriétés définies et héritées par C sont tous distincts. Kermeta n'admet pas la surcharge.
- Toutes les propriétés et opérations de C sont correctement typées (cf. paragraphes suivants).
- Il n'y a pas de cycles dans l'héritage :

$$\nexists \tau \in E \text{ tel que } \tau \in \Delta C(v_1, \ldots, v_n >) \text{ et } \tau \preceq C < v_1, \ldots, v_n >$$

Typage des propriétés

En Kermeta, les propriétés correspondent à la fois aux attributs de classe des langages orientés-objets classiques mais aussi aux associations et aux compositions. Dans le cas où un propriété n'a pas de propriété opposé, aucune contrainte de type particulière ne doit être satisfaite si ce n'est l'existence du type de la propriété.

Dans le cas ou une propriété est dérivée, elle ne peut pas avoir d'opposée et elle est valide si, et seulement si, le type de l'expression de son *getter* est un sous-type du type de la propriété et si l'expression correspondant au *setter* est correctement typée. Cela permet de s'assurer que le type de la valeur retournée par le *getter* est bien conforme au type de le propriété.

Enfin, lorsque qu'une propriété p a une propriété opposée p', le couple de propriétés (p, p') représente une association ou une composition et les contraintes suivantes doivent être vérifiées :

- Le type de la propriété p doit être engendré par la classe qui contient p'
- La propriété opposée de p' doit être p.
- Le type de la propriété p' doit être engendré par la classe qui contient p
- Au plus une des propriétés p et p' peut être une composition.

Typage des opérations

Il existe deux types d'opérations dans une classe Kermeta : les définitions d'opérations (qui correspondent à l'utilisation du mot clé *operation* dans la syntaxe concrète) et les redéfinitions (qui correspondent à l'utilisation du mot clé *method*). Dans les deux cas il est nécessaire que l'expression correspondant au corps de l'opération soit correctement typée.

Dans le cas où il ne s'agit pas d'une redéfinition, les contraintes suivantes doivent être vérifiées :

- Les noms des paramètres de l'opération sont uniques.
- Les types de paramètres et de retours de l'opération existent.
- les variables de types définies dans l'opération sont utilisées chacune au moins une fois dans le type d'un des paramètre de l'opération. Cela est nécessaire pour permettre d'inférer statiquement pour chaque appel à une opération générique le type associé a chaque variable à partir de type. Ce problème est discuté en détail dans le paragraphe consacré aux opérations génériques de la section 3.3.3.

Dans le cas où une opération est une redéfinition, les noms et types de cette opération doivent être strictement identiques aux noms et types de l'opération qu'elle redéfinie : en Kermeta toutes les redéfinitions sont invariantes.

3.4.5 Typage des expressions

Cette section détaille les règles de typage applicables aux principales constructions du langage d'action de Kermeta.

Structures de contrôles

La première structure de contrôle à laquelle nous nous intéressons est le bloc. Un bloc est une séquence d'expressions. Un bloc est correctement typé si l'ensemble des instructions qu'il contient sont correctement typées. Lors de la vérification de type d'un bloc, un nouveau contexte est empilé dans la table des symboles de l'environnement de typage. Ce nouveau contexte contient les éventuelles variables locales définies dans le bloc. Il est dépilé a la fin de l'analyse du bloc. Lors de son exécution, un bloc retourne la valeur retournée par la dernière expression qu'il contient. Comme le montre la règle de déduction 3.1, le type d'un bloc est donc le type de la dernière expression qu'il contient.

$$\frac{E \vdash exp_1 : \tau_1, \ldots, E \vdash exp_n : \tau_n}{\text{do } exp_1, \ldots, exp_n \text{ end} : \tau_n} \tag{3.1}$$

Les règles 3.2 et 3.3 détaille le typage des conditionnelles. Il est nécessaire que le type de la condition soit *Boolean*. Dans le cas où il n'y a qu'une seule branche, le type de la conditionnelle correspond au type de l'expression de cette branche. Dans le cas où il y a deux branches, le type de la valeur retournée par la conditionnelle peut correspondre soit au type de l'expression de la branche *then* soit au type de l'expression de le branche *else*. Nous avons choisi une solution simple à ce problème : le type de

retour de la conditionnelle correspond au type le plus général des deux. C'est le rôle de la fonction $ancestor(E, \tau_1, \tau_2)$ utilisée dans la règle 3.3. Cette fonction vaut :

- τ_1 si $\tau_2 \preceq \tau_1$,
- τ_2 si $\tau_1 \preceq \tau_2$,
- *Object* dans les autre cas.

Dans les versions futures de Kermeta, il serait intéressant d'introduire les *types unions* afin de typer au mieux les expressions conditionelles.

$$\frac{E \vdash cond : Boolean,\ E \vdash exp : \tau_1}{\text{if } cond \text{ then } exp \text{ end} : \tau_1} \tag{3.2}$$

$$\frac{E \vdash cond : Boolean,\ E \vdash exp_1 : \tau_1,\ E \vdash exp_2 : \tau_2}{\text{if } cond \text{ then } exp_1 \text{ else } exp_2 \text{ end} : ancestor(E, \tau_1, \tau_2)} \tag{3.3}$$

Pour le typage des boucles, comme le montre la règle 3.4 nous avons choisi une solution très simple. Il est simplement nécessaire que la condition soit une expression booléenne et le type de retour est toujours *Void*.

$$\frac{E \vdash init : \tau_i,\ E \vdash cond : Boolean,\ E \vdash exp : \tau}{\text{from } init \text{ until } cond \text{ loop } exp \text{ end} : Void} \tag{3.4}$$

Variables

Les règles 3.5 et 3.6 correspondent à la déclaration de variables locales avec et sans expression d'initialisation. Le type de la déclaration de variable est le type de l'expression d'initialisation. Si aucune expression d'initialisation n'est donnée alors le type est *Void*. Lorsqu'une déclaration de variable est vérifiée, un symbole correspondant à la variable est ajouté en haut de la table des symboles de l'environnement de typage. Le type associé à ce symbole est le type τ_v déclaré pour la variable.

$$\frac{E \vdash \tau_v}{E \vdash \text{var } v : \tau_v : Void} \tag{3.5}$$

$$\frac{E \vdash \tau_v,\ E \vdash exp : \tau_e,\ \tau_e \preceq \tau_v}{E \vdash \text{var } v : \tau_v \text{ init } exp : \tau_e} \tag{3.6}$$

Lors de l'appel à une variable, le symbole correspondant est recherché dans l'environnement de typage et, comme le montre la règle 3.7, le type rendu est le type associé au symbole trouvé. La fonction *symbol* utilisé dans la règle correspond à la recherche du haut en bas de la pile de symbole dans l'environnement de typage.

$$\frac{E \vdash symbol(v, E) = \tau}{E \vdash v : \tau} \tag{3.7}$$

Affectation et cast

La règle 3.8 détaille le typage des affectations Kermeta. La valeur retournée par une affectation est la valeur qui a été affectée. Le type de l'affectation correspond donc au type de la partie droite de l'affectation. Pour que l'affectation soit correcte, il faut que le type de la partie droite soit un sous-type de la partie gauche.

$$\frac{E \vdash lhs : \tau,\ E \vdash rhs : \tau',\ \tau' \preceq \tau}{E \vdash lhs := rhs : \tau'} \tag{3.8}$$

Dans le cas du cast dynamique c'est aussi la valeur affectée qui est retournée. La valeur rendue par l'expression en partie droite est affectée si son type dynamique est un sous-type du type statique de la partie gauche. Sinon, la valeur *void* est affectée. Comme le montre la règle 3.9 le type d'un cast dynamique correspond au type statique de la partie gauche.

$$\frac{E \vdash lhs : \tau,\ E \vdash rhs : \tau',\ \tau \preceq \tau'}{E \vdash lhs\ ?= rhs : \tau} \tag{3.9}$$

Appel de propriétés et d'opérations

La règle 3.10 correspond au typage des appels de propriété et d'opérations. La fonction $feature(f, \tau, E)$ utilisée dans cette règle correspond à la recherche du type d'une propriété ou d'une opération nommée f disponible sur le type τ (ou un de ses super-types) dans l'environnement E. Pour une propriété ou une opération sans paramètres le type obtenu correspond directement au résultat du typage. En revanche pour une opération qui a des paramètres le type obtenu est un type fonction. Il faut alors vérifier que le nombre adéquat de paramètres a été fourni à l'appel, et que le type des paramètres effectifs sont des sous-types des types spécifiés pour les paramètres formels. Dans ces conditions, le résultat du typage correspond à la partie droite du type fonction.

$$\frac{\begin{array}{c} E \vdash exp : \tau,\ \tau : Class \\ feature(f, \tau, E) = \{\tau_1 \times \cdots \times \tau_n \rightarrow \tau_f\} \\ E \vdash exp_1 : \tau'_1,\ \ldots,\ E \vdash exp_n : \tau'_n \\ \tau'_1 \preceq \tau_1,\ \ldots,\ \tau'_n \preceq \tau_n \end{array}}{exp.f(exp_1, \ldots, exp_n) : \tau_f} \tag{3.10}$$

Fonctions

Les dernières constructions de Kermeta dont nous détaillons le typage ici sont la définition et l'utilisation de fonctions ou clôtures lexicales. La règle 3.11 correspond à la définition d'une fonction. Les fonctions doivent avoir au moins un paramètre. Le type d'une fonction est un type fonction pour lequel la partie gauche est le type produit contenant les types déclarés pour les paramètres, et la partie droite est le type de l'expression correspondant au corps de la fonction.

$$\frac{E \vdash \tau_1,\ \ldots,\ E \vdash \tau_n,\ E \vdash exp : \tau}{\text{function } \{v_1 : \tau_1, \ldots, v_n : \tau_n \mid exp\} : \{\tau_1 \times \cdots \times \tau_n \rightarrow \tau\}} \tag{3.11}$$

La règle 3.12 détaille le typage de l'application d'une fonction. Cette règle est très similaire à l'appel d'opérations avec paramètres. Afin de ne pas compliquer le langage, nous avons dans un premier temps interdit les mécanismes fonctionnels plus évolués comme la currification par exemple. Cependant dans une version future du langage, il serait intéressant d'enrichir les capacités fonctionnelles de Kermeta.

$$\frac{\begin{array}{c} exp : \{\tau_1 \times \cdots \times \tau_n \rightarrow \tau_f\},\ E \vdash \tau_1,\ \ldots,\ E \vdash \tau_n,\ E \vdash \tau_f \\ E \vdash exp_1 : \tau'_1,\ \ldots,\ E \vdash exp_n : \tau'_n \\ \tau'_1 \preceq \tau_1,\ \ldots,\ \tau'_n \preceq \tau_n \end{array}}{exp(exp_1, \ldots, exp_n) : \tau_f} \tag{3.12}$$

3.5 Conclusion

Au sein d'un projet de modélisation, il est avantageux d'utiliser un ensemble de langages dédiés pour la définition de méta-modèles, de modèles, de contraintes de transformations, etc. Cependant, dans la plupart des environnements existants, les différents langages utilisés sont indépendants les uns des autres et donc difficilement interopérables. L'avantage lié à l'utilisation de langages dédiés est alors nuancé par le fait que les différents artefacts d'un même projet restent déconnectés les uns des autres du fait de leur hétérogénéité. Dans ces conditions, il difficile d'assurer et de maintenir la cohérence globale d'un projet.

La réponse que nous proposons à ce problème est une plateforme de modélisation dont le coeur est le méta-langage Kermeta. L'expressivité de Kermeta est suffisante pour exprimer sans difficultés l'ensemble des artefacts d'un projet de modélisation. L'objectif de ce langage n'est pas de remplacer les langages dédiés existants mais de fournir une base sémantique commune à ces langages. L'utilisation de Kermeta comme dénominateur commun permet l'intégration des différents artefacts d'un projet de modélisation dans une représentation qui permet l'interopérabilité.

En plus de la contrainte liée à l'expressivité de Kermeta, nous avons défini un langage le plus simple possible, en terme de nombre de concepts, tout en restant à un niveau d'abstraction satisfaisant pour l'utilisateur. Nous avons porté une attention toute particulière au système de types de Kermeta afin de fournir un maximum de vérifications statiques. Ces vérifications statiques permettent d'assurer un minimum de cohérence entre les différents artefacts d'un projet et constitue un premier pas indispensable pour la construction de processus de validation spécifiques à l'ingénierie dirigée par les modèles.

La première version comportait un parseur, un vérificateur de types (*type-checker*), un interpréteur et un éditeur intègré à Eclipse avec coloration et complétion. Le projet open-source Kermeta compte, en juin 2006, plus de douze développeurs actifs[2] et un

[2]Parmis lesquels Zoé Drey, Jean-Marie Mottu et Didier Vojtisek qui ont joué un rôle de premier

nombre d'utilisateurs croissant. En plus des éléments implantés dans la première version de l'environnement Kermeta, les principales nouveautés sont l'interopérabilité avec Java et EMF, un metteur au point (*debugger*), un éditeur graphique et des frameworks associés à des méta-modèles standards tel que UML2 par exemple.

Dans ce chapitre nous avons détaillé la construction du langage Kermeta. C'est un langage dédié à l'ingénierie des modèles. C'est un langage généraliste par rapport aux langages MOF, OCL ou QVT qui sont plus spécialisés, mais c'est un langage dédié par rapport à des langages comme Java ou Eiffel. Les cinq principales forces du langages Kermeta sont :

- Son **expressivité** qui permet de spécifier dans un formalisme unique les quatres principaux types d'artefacts de l'ingénierie des modèles : les méta-modèles, les contraintes, la sémantique et les transformations.
- Ses **mécanismes orienté-objets** qui permettent de réutiliser des bonnes pratiques issues de la programmation orienté-objets : utilisation de *design patterns*, traitement d'exceptions, utilisation d'assertions (ou contrats), etc.
- Ses **mécanismes orienté-modèles** (comme les associations et compositions) qui permettent de garantir des propriétés de conformité entre les modèles et leurs méta-modèles.
- Son **système de types statique** qui permet de garantir un premier niveau de cohérence entre les différents artefacts d'un projet.
- Son **mechanisme d'introspection** qui permet d'implanter facilement des transformations de programmes et des techniques de validation basées sur l'analyse statique.
- Sa **conformité au standard EMOF** qui permet une bonne intégration avec d'autres outils d'ingénierie des modèles (comme Eclipse/EMF par exemple).

Une faiblesse potentielle du langage Kermeta est d'être un langage généraliste. En effet, là où des langages comme MOF, OCL ou QVT proposent des constructions spécifiques à chacune des activités de l'ingénierie des modèles, Kermeta offre un ensemble restreint de constructions plus générales. Afin de valider le langage Kermeta, il est nécessaire de montrer que son niveau abstraction de ces constructions reste suffisant pour concevoir, implanter, valider et maintenir les différents artefacts d'un projet d'ingénierie des modèles. Cette étude fait l'objet du chapitre suivant.

---

plan dans l'implantation du langage Kermeta et de son environnement.

Chapitre 4

Utilisation de Kermeta : études de cas

Le chapitre précédent détaille la conception et la réalisation du langage Kermeta comme coeur d'une plateforme de modélisation. Ce chapitre propose un ensemble d'études de cas pour la validation de Kermeta. Au début du chapitre précédent (section 3.2), nous avons défini trois exigences pour le langage Kermeta : une expressivité suffisante, une taille minimale et une bonne utilisabilité. Une partie de ces exigences a été satisfaite lors de la construction de Kermeta mais il reste nécessaire de valider par la pratique que le langage Kermeta répond de façon satisfaisante aux besoins de l'ingénierie des modèles, malgré les contraintes imposées par les préocupations d'expressivité et de fiabilité.

En ce qui concerne l'expressivité, Kermeta est un langage Turing-complet. D'un point de vue théorique cette première exigence est donc satisfaite. La taille minimale est une exigence qui a guidé les choix de conception de Kermeta : choix de EMOF comme langage de méta-données, conception d'un langage d'action compact, etc. Cependant l'utilisabilité n'est pas une exigence qui peut être assurée complètement par construction. L'utilisabilité est une exigence générale qui regroupe un ensemble de propriétés dont la satisfaction est parfois difficile à assurer formellement. Pour pouvoir qualifier un langage d'utilisable, nous avons retenu les six critères suivants :

1. Être au bon niveau d'abstraction. Si les constructions offertes sont trop "bas-niveau" l'utilisation du langage est inutilement coûteuse, ce qui constitue une source d'erreur potentiellement importante. A l'inverse, des primitives trop spécifiques ou trop nombreuses nuisent à l'expressivité du langage. Un compromis doit donc être trouvé en fonction du domaine d'application du langage considéré.
2. Permettre de gérer la complexité des applications. Le langage doit offrir des constructions pour assurer son passage à l'échelle vis-à-vis de la complexité du domaine considéré.
3. Permettre de la réutilisation de code. Les applications d'un domaine ont généralement des points communs qu'il est intéressant de factoriser. Afin d'éviter toute

duplication de code, le langage doit offrir la possibilité de réutiliser, spécialiser ou étendre du code existant.

4. Faciliter la validation des programmes. La validation des programmes est toujours une phase très coûteuse du développement. Il est donc important que le langage utilisé mette en oeuvre toutes les techniques existantes pour assurer la qualité et la robustesse des programmes.
5. Faciliter la maintenance et l'évolution des applications. Les applications de ce domaine doivent pouvoir s'adapter aux évolutions des besoins de ce domaine. Pour que les évolutions soient le moins coûteuse possible, il est nécessaire que le langage offre des mécanismes permettant de faire évoluer les programmes.
6. Être supporté par des outils et interopérer avec les outils du domaine considéré. Ce dernier point est primordial pour permettre l'utilisation du langage. Les outils doivent assister l'écriture des programmes, permettre leur mise-au-point et leur exécution. Les outils et le langage doivent également interopérer avec les autres outils du domaine considéré pour être utilisables en pratique.

Pour assurer que Kermeta vérifie l'ensemble de ces critères dans le contexte de l'ingénierie des modèles, la démarche que nous avons adopté comporte deux étapes. La première est une étape amont qui consiste à intégrer dès la conception de Kermeta des concepts et des constructions qui ont fait leurs preuves dans le domaine de la programation : concepts orienté-objets, langage d'action impératif, traitement d'exceptions, typage statique, contrats, etc. La seconde est une étape avale qui consiste à vérifier, par des études de cas, que le langage Kermeta permet de concevoir et d'implanter les un ensemble de cas d'applications caractéristiques de l'ingénierie des modèles : la définition de langages de modélisation, l'implantation de transformations de modèles et de composition de modèles. Cette dernière étape est le seul moyen de valider les qualités pratiques de Kermeta. Dans cette optique, ce chapitre détaille quatre études de cas qui couvrent les principaux cas d'utilisations d'une plateforme d'ingénierie des modèles.

La première étude, présentée section 4.1, traite l'exemple du langage d'automate utilisé au début du chapitre précédent. L'objectif de cette étude est d'illustrer sur un exemple simple et complet les possibilités et les avantages de Kermeta pour la spécification des différents artéfacts relatifs à un méta-modèle : sa structure, sa sémantique, des contraintes et des transformations. Cette étude est également l'occasion de présenter brièvement les outils que nous avons développé autour du langage Kermeta.

La seconde étude, présentée section 4.2, discute de l'implantation de transformations de modèles. Les transformations de modèles sont une des applications caractéristiques de l'ingénierie des modèles. L'objectif de l'étude est de valider les capacités de Kermeta pour leur implantation. L'étude que nous avons menée repose sur un benchmark de transformations de modèles défini par les organisateurs du workshop MTIP'05 [BRST05] pour évaluer différentes techniques de transformation de modèles. Cette étude montre, par exemple, comment tirer avantage des possibilités offertes par Kermeta pour l'écriture de code de transformation de modèles réutilisable.

La troisième étude, présentée section 4.3, détaille la mise en oeuvre en Kermeta de travaux académiques sur la modélisation orientée aspects. Pour cette étude, Ker-

meta à été utilisé pour l'implantation de deux techniques de tissage de modèles. La première permet de composer des diagrammes de classes et la seconde de composer des diagrammes de séquences. Le principal intérêt de Kermeta pour cette application est de permettre d'intégrer la sémantique (et les variations sémantiques) du tissage directement dans le méta-modèle considéré.

La quatrième étude, présentée section 4.4, revient sur des travaux utilisant l'ingénierie des exigences menée en lien avec l'industrie. Ces travaux proposent une chaîne de traitement des exigences basée sur l'ingénierie des modèles. Cette chaîne comporte, entre autres, deux langages de modélidation dédiés et un certain nombre de transformations de modèles. L'objectif de cette étude est de montrer que Kermeta fournit un bon moyen pour la mise en oeuvre de processus complets d'ingénierie des modèles.

4.1 Exemple des automates

Dans la section 3.1 nous avons pris l'exemple d'un méta-modèle d'automate afin de montrer les limites des environnements de modélisation existants. Pour spécifier le langage d'automates, nous avions utilisé EMOF pour définir la structure du méta-modèle, OCL pour spécifier des contraintes, le langage naturel pour définir la sémantique du méta-modèle et un langage dédié pour implanter des transformations de modèles. Les limitations de cette approche sont la conséquence de l'hétérogénéité et de la non-intéroperabilité des langages utilisés. Par exemple, cela rend difficile toute vérification automatique relative à la cohérence entre les différents artéfacts considérés. Cette section montre comment l'utilisation de Kermeta permet une bonne intégration entre la structure, des contraintes, la sémantique et des transformations de modèles.

La figure 4.1 rappelle le méta-modèle d'automate que nous utilisons. En pratique ce méta-modèle a été défini graphiquement en utilisant Eclipse/EMF et Omondo afin d'obtenir un diagramme. Les outils Eclipse/EMF étant conformes au standard EMOF, ce méta-modèle est directement utilisable dans l'environnement Kermeta. Cela est rendu possible par le fait que Kermeta à été construit par extension du standard EMOF. Dans l'environnement Kermeta, des transformations permettent d'importer et d'exporter des méta-modèles depuis Eclipse/EMF. La figure 4.2 présente le listing Kermeta correspondant exactement au méta-modèle d'automate de la figure 4.1. On retrouve sur ce listing les classes, les propriétés et les opérations du méta-modèle. Pour la propriété dérivée *alphabet* un corps vide et un commentaire ont été ajoutés.

4.1.1 Définition de contraintes

Plusieurs possibilités peuvent être utilisées pour l'écriture de contraintes en Kermeta. La première est d'utiliser le support des contrats qui permet de spécifier des pré-conditions et post-conditions pour les opérations et des invariants pour les classes (cf. section 3.3.6). L'avantage de ces contrats est qu'ils sont automatiquement vérifiés à l'exécution par l'interpréteur Kermeta. Un autre solution qui permet de n'évaluer une contrainte qu'à la demande est d'implanter cette contrainte dans une opération ou dans une propriété dérivée. L'intérêt de cette solution est que les contraintes ne sont

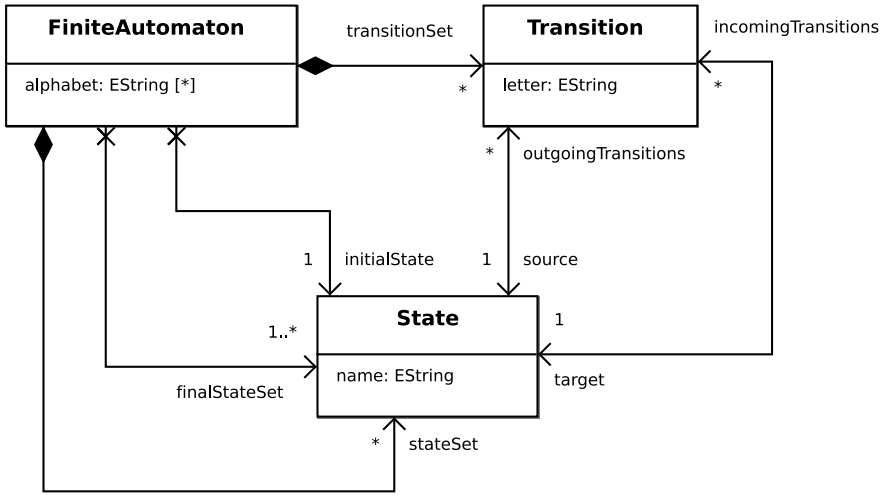


FIG. 4.1 – Méta-modèle d'automate.

alors vérifiées qu'explicitement ce qui permet de manipuler des modèles qui violent ces contraintes.

Pour illustrer les possibilités offertes par Kermeta, nous avons ajouté une contrainte de déterminisme sur le méta-modèle d'automates de deux manières différentes. Le listing 4.3 montre les ajouts faits au méta-modèle de la figure 4.2. Nous avons tout d'abord définit une opération *hasConflictingTransitions* dans la class *State* (lignes 21 à 28 sur le listing). Cette opération permet de déterminer si un état possède plusieurs transitions sortantes portant le même symbole. L'utilisation des fermetures tel que *collect*, *exists* ou *select* permet une écriture facile et lisible de la contrainte.

A partir de l'opération *hasConflictingTransitions*, il est possible de définir le déterminisme comme une simple propriété dérivée ou comme un invariant. Dans le cas où l'on définit une propriété, il reste possible de manipuler des automates non-déterministes alors que si l'on impose cette contrainte comme un invariant, les machines à états non-déterministes seront rejetées par l'environnement. Sur le listing de la figure 4.3 nous avons ajouté une propriété dérivée dans la classe *FiniteAutomaton* (lignes 6 à 11). Cette propriété vérifie que l'ensemble des états de la machine à états considérée a des transitions sortantes déterministes.

Enfin, pour illustrer la définition d'invariants, nous avons ajouté un invariant dans la classe *State* (ligne 34 sur le listing). Cet invariant assure qu'aucun état ne porte de transitions conflictuelles et assure donc que toute machine à états soit déterministe.

En utilisant des contrats, des opérations ou des propriétés dérivées il est ainsi possible, en Kermeta, d'intégrer des contraintes directement dans les classes du méta-modèle concerné. Ces contraintes peuvent être, soit automatiquement prises en charge par l'interpréteur Kermeta dans le cas des contrats, soit simplement encapsulées et disponibles à l'évaluation dans le cas de l'utilisation d'opérations ou de propriétés dérivées.

```
1  /*
2   * Definition of a simple metamodel for automata
3   */
4  package automaton;
5
6  alias String : kermeta::standard::String;
7
8  class FiniteAutomaton
9  {
10   attribute stateSet : State[0..*]
11   attribute transitionSet : Transition[0..*]
12   property readonly alphabet : String[0..*]
13     getter is do
14       //TODO: implement getter for derived property alphabet
15     end
16   reference finalStateSet : State[1..*]
17   reference initialState : State[1..1]
18 }
19
20 class State
21 {
22   attribute name : String
23   reference incomingTransitions : Transition[0..*]#target
24   reference outgoingTransitions : Transition[0..*]#source
25 }
26
27 class Transition
28 {
29   reference source : State[1..1]#outgoingTransitions
30   reference target : State[1..1]#incomingTransitions
31   attribute letter : String
32 }
```

FIG. 4.2 – Source Kermeta pour le méta-modèle d'automates de le figure 3.1

```
1  class FiniteAutomaton {
2    [...]
3
4    // Définition d'une propriété dérivée qui vérifie
5    // si l'automate est déterministe
6    property readonly isDeterministic : Boolean
7      getter is do
8        result := not stateSet.exists{ state |
9          state.hasConflictingTransitions
10       }
11     end
12
13   [...]
14 }
15
16 class State {
17   [...]
18
19   // Définition d'une opération qui vérifie si l'état possède
20   // des transitions sortantes portant des symboles identiques
21   operation hasConflictingTransitions() : Boolean is do
22     result :=
23     outgoingTransitions.collect{ t | t.letter }.exists{ symb |
24       outgoingTransitions.select{ trans |
25         trans.letter == symb
26       }.size > 1
27     }
28   end
29
30   [...]
31
32   // Définition d'un invariant qui assure que l'état ne possède
33   // pas de transitions sortantes portant le même symbole
34   inv no_conflicting_transitions is not hasConflictingTransitions
35
36   [...]
37 }
```

FIG. 4.3 – Contrainte de déterminisme pour les automates

4.1.2 Définition d'une sémantique opérationnelle et simulation

Au delà de permettre l'expression de contraintes, un des avantages de Kermeta est de permettre de définir formellement la sémantique opérationnelle des méta-modèles considérés. En Kermeta, la sémantique opérationelle d'un méta-modèle est directement intégrée aux classes de ce méta-modèle sous la forme d'opérations. Grâce à Kermeta il est également possible de spécifier les éventuelles propriétés dérivées des méta-modèles manipulés. L'ensemble des opérations et propriétés spécifiées en Kermeta peut alors être exécuté dans l'environnement Kermeta et permet donc de rendre les modèles "exécutables" ou "simulables". Cette section détaille ce processus sur l'exemple des automates.

Propriété dérivées

Le premier point intéressant est la définition de propriétés dérivées. Le méta-modèle d'automate présenté figure 3.1, et dont la représentation textuelle est donnée sur le listing de la figure 4.2, contient une propriété dérivée *alphabet* dans la class *FiniteAutomaton*. Cette propriété renvoie l'ensemble des symboles qui sont utilisés sur les transitions d'un automate. Le calcul de cette propriété est donc relativement simple : il suffit de parcourir l'ensemble des transitions de l'automate et de collecter les symboles qu'elles portent. Le listing 4.4 présente l'implantation en Kermeta de cette propriété. Une fois implantée de la sorte, cette propriété dérivée peut être utilisée dans toute opération Kermeta manipulant des automates.

```
1 property readonly alphabet : set String[0..*]
2   getter is do
3     // collect all symbols on transitions
4     result := Set<String>.new
5     transitionSet.each{ t |
6       result.add(t.letter)
7     }
8   end
```

FIG. 4.4 – Définition de la propriété dérivée *alphabet* dans la classe *FiniteAutomaton*.

Sémantique opérationelle

Les propriétés dérivées étant implantées, intéressons nous maintenant à la spécification de la sémantique opérationnelle du méta-modèle. Dans le cas des automates, la sémantique retenue consiste à lire une chaîne d'entrée et à répondre vrai si cette chaîne d'entrée appartient au langage décrit par l'automate (cf. section 3.1). Cette sémantique simple peut être implantée facilement en utilisant les techniques classiques de la programmation impérative orienté-objets. Du fait que Kermeta soit un langage orienté-objets, il est important de réutiliser l'ensemble du savoir-faire de conception issu de la programmation orienté-objets afin de concevoir des méta-modèles lisibles, fiables et extensibles.

```
1  /** The current state */
2  reference currentState : State
3
4  /** Determines if a word w is recognized by the automaton */
5  operation process(w : String[*]) : Boolean is do
6    // pre-condition of the operation
7    pre deterministic is isDeterministic
8    // initialize current state with initial state
9    currentState := initialState
10   // read symbols
11   w.each{ symbol | currentState := currentState.step(symbol) }
12   // check if current state is final
13   result := finalStateSet.contains(currentState)
14 rescue (e : InvalidSymbolException)
15   // One symbol was not recognized
16   result := false
17 end
```

FIG. 4.5 – Définition de l'état courant et de l'opération *process* dans la classe *FiniteAutomaton*.

Le listing de la figure 4.5 présente les éléments ajoutés à la classe *FiniteAutomaton*. En premier lieu, afin de représenter l'état courant de l'automate, nous avons ajouté une référence nommée *currentState* (ligne 2). En utilisant cette référence, l'opération *process* permet de lire une chaîne d'entrée (exprimée comme un tableau de chaînes de caractères) et rend vrai si cette chaîne est reconnue par l'automate. Une des pré-conditions pour cette opération est que l'automate considéré doit être déterministe. Cette pré-condition est exprimée (ligne 7) en utilisant l'opération *isDeterministic* définie dans la section précédente. L'implantation de cette opération comporte tout d'abord l'initialisation de l'état courant par l'état initial de l'automate (ligne 9) puis pour chaque symbole de la chaîne d'entrée, la lecture du symbole est déléguée à l'état courant (ligne 11).

```
1  /** Read the symbol and return the new state */
2  operation step(symbol : String) : State is do
3    // select valid transitions
4    var transitions : seq Transition[0..*] init
5      out{ t | t.letter == symbol }
6    // error : no valid transitions
7    if transitions.size == 0 then raise InvalidSymbolException.new end
8    // return the new state
9    result := transitions.one.target
10 end
```

FIG. 4.6 – Définition de l'opération *step* dans la classe *State*.

Le listing de la figure 4.6 présente l'implantation Kermeta de l'opération *step* dans

```
1  class InvalidSymbolException
2    inherits kermeta::exceptions::Exception {}
```

FIG. 4.7 – Définition d'une nouvelle classe d'exceptions.

la classe *State*. Cette opération recherche dans les transitions sortantes de l'état une transition qui porte le symbole correspondant au symbole lu, puis tire cette transition en mettant à jour l'état courant de l'automate. Dans le contexte de cette méthode, il n'est pas possible que plusieurs transitions sortantes correspondent au symbole lu car on s'est assuré en pré-condition de l'opération *process* que l'automate était déterministe. Par contre, il est possible qu'aucune transition sortante ne corresponde au symbole lu. Dans ce cas, nous avons choisi d'utiliser le mécanisme d'exception de Kermeta afin de faire remonter l'information à l'opération *process*.

Comme le montre le listing de la figure 4.7, nous avons défini une nouvelle classe d'exception que nous utilisons à la ligne 7 de l'opération *step* si aucune transition sortante n'a été trouvée. Dans l'opération *process* cette exception est traitée par le bloc *rescue* (ligne 14) qui met alors le résultat de l'opération à *faux* afin de refléter le fait que la chaîne n'a pu être reconnue par l'automate. Une fois la sémantique opérationelle des automates définie de la sorte, l'environnement Kermeta, qui inclut un interpréteur, permet d'exécuter les opérations définies et donc de simuler des automates.

4.1.3 Validation de la sémantique

Comme pour n'importe quel programme, une fois la sémantique des automates décrite, il est nécessaire de s'assurer la correction de ce qui a été implanté. Afin de faciliter cette étape de test l'environnement Kermeta intègre un framework de test unitaire similaire au framework JUnit [GB] de JAVA. Ce framework a naturellement été baptisé *KUnit*. Le listing de la figure 4.8 présente la définition d'une classe de test permettant de vérifier les fonctions de base du méta-modèle d'automates.

Une classe de test KUnit est une classe qui hérite de la classe *TestCase* définie dans le framework. Cette classe *TestCase* comporte deux opérations *setUp* et *tearDown* qui sont exécutées respectivement avant et après chaque cas de test. Ces opérations peuvent être redéfinies dans les classes de test afin de factoriser du code d'initialisation ou de finalisation. Dans la classe *TestAutomata* de la figure 4.8, la méthode *setUp* est utilisée pour la création de l'automate très simple représenté figure 4.9. Cet automate reconnaît les chaînes d'entrée correspondant à l'expression régulière $(ab)*$.

Dans une classe de test KUnit, chaque cas de test est implanté par une opération dont le nom commence par "test" et qui n'a ni paramètre ni type de retour. Afin de tester le méta-modèle d'automates, la classe *TestAutomata* comporte quatre cas de test. Le premier test permet de s'assurer que la chaîne "ab" est bien reconnue par l'automate. Le second permet de s'assurer que la chaîne vide est reconnue (l'état initial de l'automate étant également final). Le troisième test s'assure que la chaîne "a" n'est pas reconnue par l'automate car l'état 2 n'est pas final. Enfin le quatrième test permet de s'assurer

```
1  class TestAutomata inherits TestCase {
2    reference automaton : FiniteAutomaton
3
4    method setUp() : Void is do
5      automaton := FiniteAutomaton.new
6      var s1 : State init State.new s1.name := "1"
7      var s2 : State init State.new s2.name := "2"
8      var ta : Transition init Transition.new
9      var tb : Transition init Transition.new
10     ta.letter := "a"  tb.letter := "b"
11     ta.source := s1   tb.source := s2
12     ta.target := s2   tb.target := s1
13     automaton.stateSet.add(s1)  automaton.initialState := s1
14     automaton.stateSet.add(s2)  automaton.finalStateSet.add(s1)
15   end
16
17   method tearDown() : Void is do
18     automaton := void
19   end
20
21   operation testAB() : Void is do
22     var input : OrderedSet<String> init OrderedSet<String>.new
23     input.add("a")  input.add("b")
24     assert(automaton.process(input))
25   end
26
27   operation testEmpty() : Void is do
28     var input : OrderedSet<String> init OrderedSet<String>.new
29     assert(automaton.process(input))
30   end
31
32   operation testA() : Void is do
33     var input : OrderedSet<String> init OrderedSet<String>.new
34     input.add("a")
35     assert(not automaton.process(input))
36   end
37
38   operation testB() : Void is do
39     var input : OrderedSet<String> init OrderedSet<String>.new
40     input.add("b")
41     assert(not automaton.process(input))
42   end
43
44   operation main() : Void is do
45     name := "testAutomata"
46     var tr : TestRunner init TestRunner.new
47     tr.run(TestAutomata)
48     tr.printTestResult
49   end
50 }
```

FIG. 4.8 – Définition d'une classe de test "KUnit".

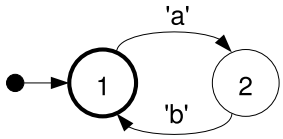


FIG. 4.9 – Automate de test créé par l'opération *setUp* de la classe *TestAutomata*.

que la chaîne "b" n'est pas reconnue faute de transition étiquetée "b" dans l'état 1.

La méthode *main* de la classe *TestAutomata* permet de lancer les tests et d'afficher le résultat sur la sortie standard. La figure 4.10 présente une capture d'écran saisie lors de l'exécution des test unitaires. Le rapport de passage est affiché dans la console : les quatre tests sont passés.

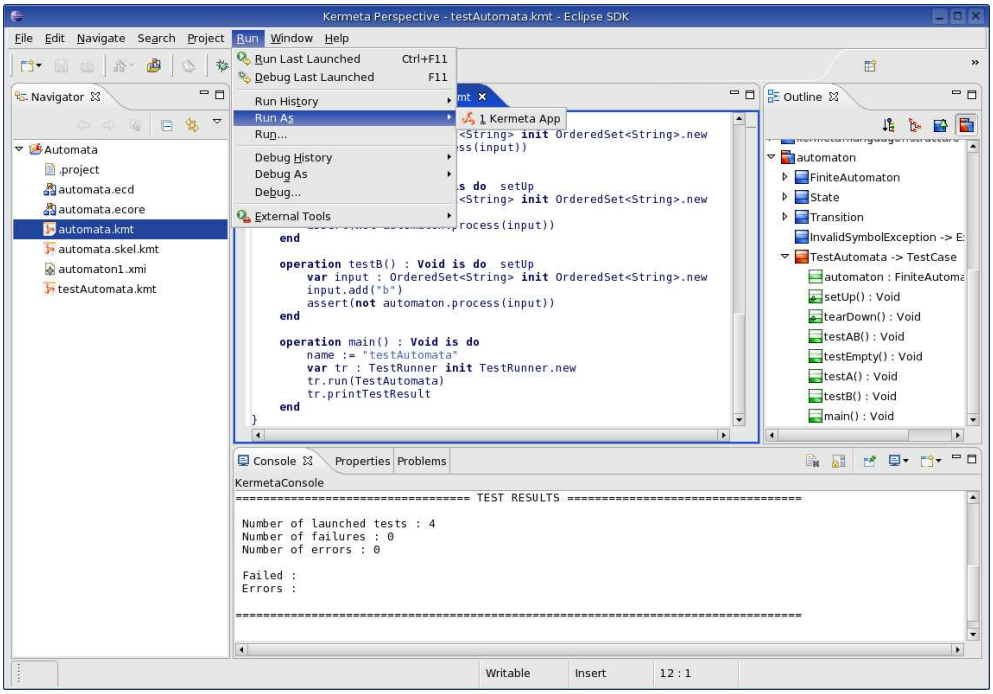


FIG. 4.10 – Capture d'écran de l'exécution des test unitaires dans l'environnement Eclipse/Kermeta.

4.1.4 Interopérabilité et sérialisation

Afin de tester la sémantique implantée dans le méta-modèle d'automates, nous avons créé un automate en instanciant en Kermeta les classes du méta-modèle (ligne 6 à 15 du listing de la figure 4.8). En pratique la création des modèles en utilisant cette méthode "programmatique" est fastidieuse car le code produit est peu lisible, peu maintenable

et inaccessible au non-programmeur.

Afin d'éviter d'avoir recours à cette méthode pour la création de modèles et du fait de la compatibilité de Kermeta avec le standard EMOF, il est possible de s'appuyer sur des outils externes pour l'édition et la sérialisation de modèles. A ce titre, un framework complet d'import/export de modèle EMF (Eclipse Modeling Framework) [EMF] est implanté dans la librairie standard de Kermeta. Grâce à ce framework, il est possible d'utiliser les outils EMF pour la création, l'édition et la visualisation des modèles d'automates.

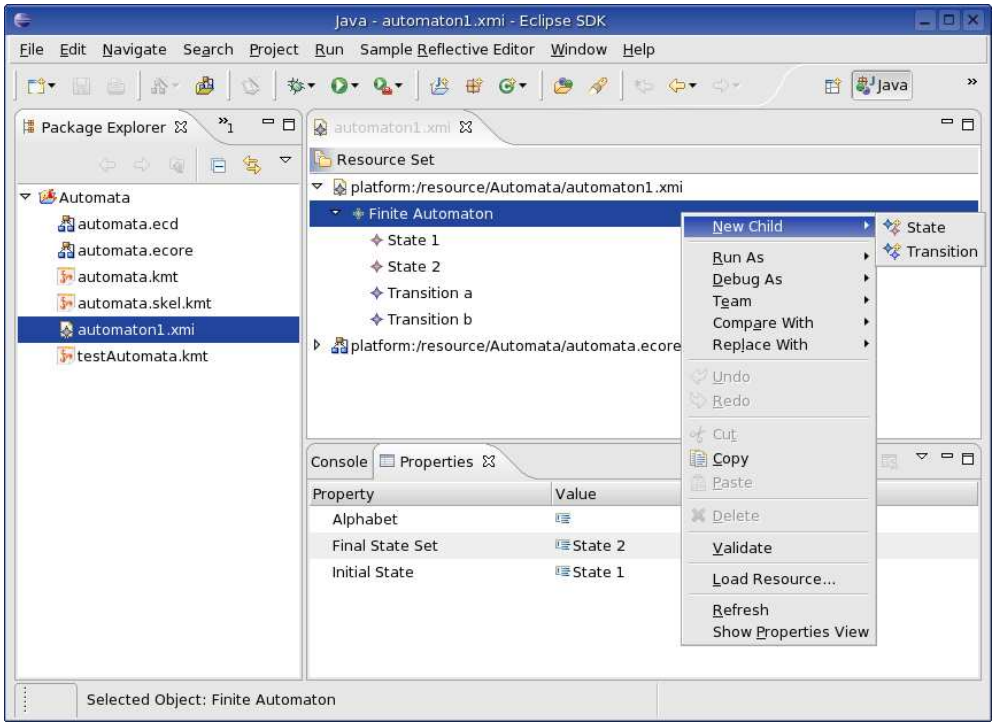


FIG. 4.11 – Édition d'une machine à états dans l'éditeur générique de EMF.

La figure 4.11 présente une capture d'écran de l'éditeur générique d'Eclipse/EMF. Cet éditeur présente les modèles sous la forme d'une arborescence d'objets et permet d'éditer les propriétés de chaque objet. Nous avons reproduit l'automate présenté figure 4.9 dans cet éditeur. L'éditeur produit un modèle au format XMI 2.0 qui peut être ensuite chargé en Kermeta. Le listing de la figure 4.12 présente un listing permettant de charger ce modèle en Kermeta. L'opération *setUp* de ce listing peut être utilisée pour remplacer l'opération *setUp* du listing de la figure 4.8.

L'intéropérabilité de Kermeta avec les outils Eclipse/EMF est une qualité qui a été assurée lors de la construction de Kermeta par le respect du standard EMOF. Cette interoperabilité permet non seulement l'import/export de méta-modèles mais aussi la réutilisation des outils existants pour l'édition, la visualisation et la sérialisation de modèles.

```
1  /**
2  * Load a simple automaton
3  */
4  method setUp() : Void is do
5    var repository : EMFRepository init EMFRepository.new
6    var resource : EMFResource
7    resource := repository.createResource("automaton1.xmi",
8                                          "automata.ecore")
9    resource.load()
10   automaton ?= resource.instances.one
11 end
```

FIG. 4.12 – Chargement d'un modèle en Kermeta.

4.1.5 Manipulation de modèles et transformations

Les sections précédentes ont montré comment ajouter des contraintes et de la sémantique directement dans les classes d'un méta-modèle. En pratique, une fois que l'on dispose de modèles validés par des contraintes, dont la sémantique est établie et qui ont pu être testés par simulation, l'étape suivante est l'exploitation de ces modèles. Pour cela, on utilise des transformations de modèle ou plus généralement des programmes qui manipulent les modèles. Ces programmes, qui utilisent des modèles, sont intimement liés aux méta-modèles correspondants aux modèles qu'ils exploitent mais, contrairement aux contraintes ou à la sémantique, ils ne font pas partie de ces méta-modèles.

Il faut donc à la fois assurer le découplage et la cohérence entre les méta-modèles et les programmes qui utilisent ces méta-modèles. Pour cela, Kermeta permet de créer, à côté des méta-modèles, des packages, classes et opérations afin d'implanter des programmes de manipulation de modèles. Le découplage est assuré comme dans n'importe quel programme orienté-objets. De façon analogue à ce qui existe dans les langages de programmation orienté-objets, ces nouvelles classes peuvent importer les classes des méta-modèles pour les utiliser. Les vérifications des types statiques implantés dans Kermeta permettent alors d'assurer que les programmes sont consistants avec le méta-modèle.

Le listing de la figure 4.13 définit une classe *PrintAutomata* qui permet d'afficher un automate sur la sortie standard. La directive *require* à la ligne 4 du listing permet d'importer le méta-modèle d'automate. La classe *PrintAutomata* comporte une opération *main* qui est le point d'entrée du programme et deux opérations *printAutomaton* et *printState* qui permettent d'afficher un automate sur la sortie standard. La figure 4.14 présente une capture d'écran de l'environnement Kermeta après l'exécution du programme.

Dans le cadre de l'exemple des automates, un certain nombre de transformations peut être envisagé : la détermination, la minimisation, l'application du patron de conception *Etat* (*State*) ou encore la génération de code. Dans le cadre de travaux

```
1  package automaton;
2
3  require kermeta
4  require "automata.kmt"
5
6  using kermeta::standard
7  using kermeta::persistence
8
9  class PrintAutomata
10 {
11   operation main() : Void is do
12     var automaton : FiniteAutomaton
13     // Load an automaton
14     var repository : EMFRepository init EMFRepository.new
15     var resource : EMFResource
16     resource := repository.createResource("automaton1.xmi",
17                                           "automata.ecore")
18     resource.load()
19     automaton ?= resource.instances.one
20     // Print the automaton to the standard output
21     printAutomaton(automaton)
22   end
23
24   operation printAutomaton(automaton : FiniteAutomaton) is do
25     // Print states and transitions
26     automaton.stateSet.each{ s | printState(s) }
27     // Initial state
28     stdio.writeln("Inital␣state␣:␣" + automaton.initialState.name)
29     // Final states
30     stdio.write("Final␣states␣:␣")
31     automaton.finalStateSet.each{ s | stdio.write(s.name + "␣")}
32   end
33
34   operation printState(state : State) is do
35     state.outgoingTransitions.each{ t |
36       stdio.writeln(state.name + "␣--(" + t.letter +
37                     ")->␣" + t.target.name)
38     }
39   end
40 }
```

FIG. 4.13 – Affichage d'une machine à états.

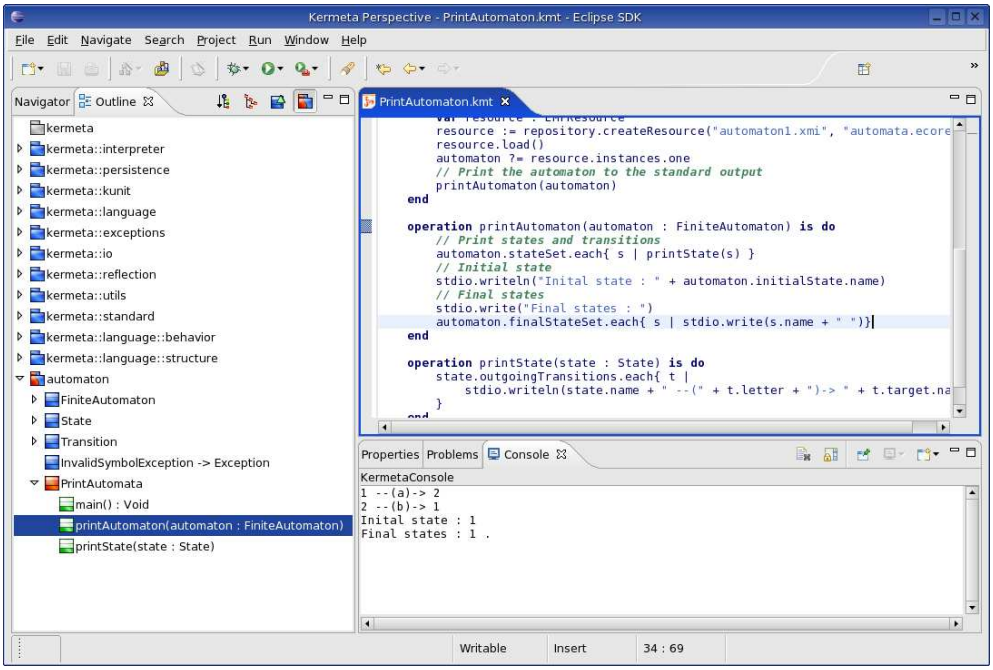


FIG. 4.14 – Capture d'écran de l'environnement Kermeta lors de l'affichage d'une machine à états.

sur la transformation de modèles, les transformations de détermination et minimisation d'automates en Kermeta ont été implantées. Ces transformations sont détaillées dans [MFV+05] et disponibles en téléchargement sur le site de Kermeta [Ker]. Nous ne détaillons pas ces exemples dans cette section car l'utilisation de Kermeta pour l'écriture de transformations de modèles fait l'objet d'une discussion plus complète dans la section 4.2.

4.1.6 Conclusion

Cette première étude a permis de s'assurer de la possibilité de définir un ensemble d'artéfacts relatifs au méta-modèle d'automate en Kermeta. Nous avons tout d'abord défini le méta-modèle dans Eclipse/EMF. Ce méta-modèle a pu être importé directement dans Kermeta grâce à la compatibilité des deux outils avec le standard EMOF. Ensuite, Kermeta a permis d'ajouter des contraintes et la sémantique opérationnelle directement dans le méta-modèle. Nous avons utilisé le framework KUnit afin de tester le code écrit. Pour créer et éditer facilement des modèles (dans notre cas des machines a états) nous avons utilisé l'éditeur de modèle générique de Eclipse/EMF. Les modèles créés dans eclipse/EMF sont directement utilisables en Kermeta. Enfin nous avons écrit un programme Kermeta permettant d'afficher une machine à états sur la sortie standard. Ce programme met en évidence la possibilité de manipuler des automates sans ajouter de code directement dans le méta-modèle. La section suivante présente une étude complète de l'utilisation de Kermeta pour la transformation de modèles.

4.2 Transformation de modèles

La seconde étude que nous présentons ici est l'utilisation de Kermeta pour la transformation de modèles. Le langage Kermeta n'est pas dédié à la transformation de modèle car il n'intègre aucune construction spécifique à la transformation. Cependant, un des objectifs de Kermeta est de permettre de manipuler des modèles, c'est-à-dire de parcourir, créer et modifier des modèles. A ce titre, il doit donc permettre, et même faciliter, l'écriture de transformations de modèles. Afin de vérifier les aptitudes de Kermeta pour la transformation de modèles, nous nous sommes appuyés sur un ensemble de transformations de modèles proposées dans le cadre du workshop MTIP (Model Transformation In Practice) [BRST05]. L'objectif de ce workshop était d'évaluer et de comparer les différentes approches de transformation de modèles. Pour cela l'ensemble des participants devait implanter une étude de cas obligatoire : la génération de schémas de base de données à partir de diagrammes de classes. En plus de cette étude, quatre transformations optionnelles étaient proposées :

- la conversion de chiffres arabes en chiffres romains (et vice-versa),
- la déterminisation et minimisation d'automates,
- des refactorings sur des diagrammes de classes,
- la génération de code "non-conventionel".

En utilisant Kermeta, nous avons implanté l'étude de cas obligatoire ainsi que les trois premières études facultatives[1]. Il est intéressant de noter que parmi les approche proposées au workshop, Kermeta est la seule qui ait présenté autant de transformations. Le résultat de ce travail a été présenté au workshop MTIP [BRST05]. La suite de cette section détaille les différentes étapes de l'écriture de transformations de modèles en Kermeta à travers l'étude de cas obligatoire : la génération de schéma de base de données à partir d'un diagramme de classes, nommée *CLS2RDB*. Cette étude a été spécifiée par les organisateurs du workshop pour être la plus représentative possible des transformations de modèles classiques.

4.2.1 Méta-modèles d'entrée et de sortie

La première étape pour l'écriture de transformations de modèles est la spécification des méta-modèles d'entrée et de sortie. Dans le cas de la transformation *CLS2RDB*, les modèles d'entrée sont des diagrammes de classes et les modèles en sortie des schémas de base de données. La figure 4.15 présente les deux méta-modèles correspondants. En pratique, le fait que Kermeta ait été développé conformément au standard EMOF permet d'utiliser les outils supportant ce standard pour la définition de méta-modèles. Ainsi, les méta-modèles de la figure 4.15 ont été définis en utilisant l'éditeur Omondo [Omo] qui permet de définir des méta-modèles au format ECore compatible avec EMOF.

Une fois les méta-modèles définis de la sorte, il peuvent être utilisés directement dans un programme Kermeta mais il est également possible de les traduire automatiquement en Kermeta si l'on souhaite leur ajouter du comportement. Pour l'écriture de transformations de modèles, l'ajout de code directement dans les méta-modèles permet

[1] Zoé Drey, Jean-Marc Jézéquel et Damien Pollet ont participé a l'implantation de ces études de cas.

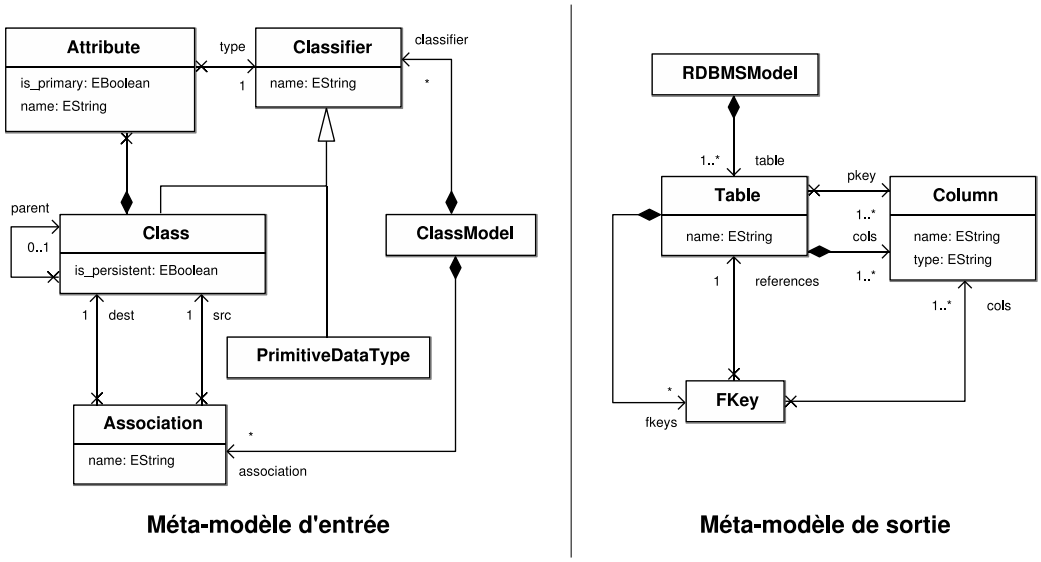


FIG. 4.15 – Méta-modèles d'entrée et de sortie pour la transformation *CLS2RDB*.

d'encapsuler des traitements directement dans les classes des méta-modèles. Dans la majorité des cas, ces traitements ne sont pas spécifiques à la transformation implantée mais simplement au méta-modèle considéré et peuvent donc être réutilisés dans plusieurs transformations.

A titre d'exemple, dans un modèle de classes, il peut être utile de disposer de l'ensemble des super-classes d'une classe. Plutôt que d'intégrer du code permettant de calculer l'ensemble des super-classes d'une classe dans chaque transformation qui en a besoin, il est avantageux de définir et d'implanter une opération ou une propriété dérivée directement dans la classe *Class* du méta-modèle.

Afin d'illustrer les deux possibilités offertes par l'environnement Kermeta nous avons choisi pour la transformation *CLS2RDB* d'utiliser directement le méta-modèle d'entrée dans le format ECore et de traduire le méta-modèle de sortie en Kermeta afin de pouvoir lui ajouter des traitements. Le listing de la figure 4.16 détaille la représentation Kermeta du méta-modèle de sortie de la transformation *CLS2RDB*. Lors de la conception ou de l'implantation de la transformation, il est possible d'y ajouter toute propriété ou opération nécessaire à la transformation. En pratique, il est préférable de n'intégrer que des propriétés ou des opérations non spécifiques à la transformation et ayant vocation à être réutilisées dans d'autres contextes.

4.2.2 Spécification de la transformation

La spécification de la transformation *CLS2RDB* a été donnée de façon précise dans l'appel à participation au workshop MTIP. La spécification a été donnée en anglais sous la forme des sept règles suivantes.

```
1  package RDBMSMM;
2  require kermeta
3  using kermeta::standard
4
5  class Table
6  {
7    attribute name : String
8    attribute cols : Column[1..*]
9    reference pkey : Column[1..*]
10   attribute fkeys : FKey[0..*]
11 }
12 class FKey
13 {
14   reference references : Table
15   reference cols : Column[1..*]
16 }
17 class Column
18 {
19   attribute name : String
20   attribute type : String
21 }
22 class RDBMSModel
23 {
24   attribute table : Table[1..*]
25 }
```

FIG. 4.16 – Méta-modèle de schéma de base de donnée en Kermeta.

This version of the transformation contains several subtleties that authors will need to be aware of. In order to facilitate comparisons between approaches, authors should ensure that they accurately implement the transformation.

1. Classes that are marked as persistent in the source model should be transformed into a single table of the same name in the target model. The resultant table should contain one or more columns for every attribute in the class, and one or more columns for every association for which the class is marked as being the source. Attributes should be transformed as per rules 3-5.
2. Classes that are marked as non-persistent should not be transformed at the top level. For each attribute whose type is a non-persistent class, or for each association whose dst is such a class, each of the classes' attributes should be transformed as per rule 3. The columns should be named name transformed attr where name is the name of the attribute or association in question, and transformed attr is a transformed attribute, the two being separated by an underscore character. The columns will be placed in tables created from persistent classes.
3. Attributes whose type is a primitive data type (e.g. String, Int) should be transformed to a single column whose type is the same as the primitive data type.
4. Attributes whose type is a persistent class should be transformed to one or more columns, which should be created from the persistent classes' primary key attributes. The columns should be named name transformed attr where name is the attributes' name. The resultant columns should be marked as constituting a foreign key ; the FKey element created should refer to the table created from the persistent class.
5. Attributes whose type is a non-persistent class should be transformed to one or more columns, as per rule 2. Note that the primary keys and foreign keys of the translated non-persistent class need to be merged in appropriately, taking into consideration that the translated non-persistent class may contain primary and foreign keys from an arbitrary number of other translated classes.
6. When transforming a class, all attributes of its parent classes (which must be recursively calculated), and all associations which have such classes as a src, should be considered. Attributes in subclasses with the same name as an attribute in a parent class are considered to override the parent attribute.
7. In inheritance hierarchies, only the top-most parent class should be converted into a table ; the resultant table should however contain the merged columns from all of its subclasses.

4.2.3 Conception et implantation en Kermeta

Comme pour le développement de tout composant logiciel, la première étape de développement est la conception. Le fait d'utiliser un langage généraliste comme Kermeta plutôt qu'un langage dédié à la transformation de modèle impose de porter un soin particulier à la conception. En effet, les langages dédiés à la transformation fournissent des constructions et des structures adaptées à la définition de transformations de modèles alors qu'une transformation écrite en Kermeta n'est qu'un programme orienté-objets qui manipule des éléments de modèles.

Conception

La transformation *CLS2RDB* consiste à créer des tables pour chaque classe marquée comme persistante dans le modèle source. Pour chaque attribut ou association

des classes persistantes, des colonnes ou clés étrangères sont ajoutées dans les tables correspondantes. En pratique, cette transformation peut s'implanter par trois étapes successives :

1. Création des tables. Création d'une table dans le modèle de sortie pour chaque classe marquée persistante dans le modèle d'entrée.
2. Création des colonnes. Création d'un attribut pour chaque attribut ou référence des classes persistantes dans le modèle d'entrée. Les colonnes correspondants aux clés étrangères ne peuvent pas être créées lors de cette étape car toutes les clés primaires des tables ne sont pas encore créées.
3. Mise-à-jour des clés étrangères. Dans cette troisième phase, les clés primaires de toutes les tables sont à jour et il est donc possible de créer l'ensemble des colonnes correspondant aux clés étrangères.

Entre les deux premières phases de la transformation il est nécessaire de garder une information de traçabilité entre les éléments du modèle source et les éléments du modèle résultat. En effet, afin d'ajouter les colonnes dans les tables appropriées lors de la seconde phase, il est nécessaire de conserver un lien entre les classes persistantes et les tables qui leur correspondent lors de la première phase. Ce besoin de stocker des informations de traçabilité n'est pas propre à la transformation *CLS2RDB* mais est observé de façon récurrente pour l'écriture de transformations de modèles.

En Kermeta plusieurs possibilités existent pour stocker les informations de traçabilité. La solution la plus simple est d'utiliser une structure de donnée disponible dans le framework de Kermeta tel qu'une table de hachage par exemple. Cependant, étant donné qu'il s'agit d'un problème récurrent il peut être avantageux de concevoir un solution réutilisable sous la forme d'un framework de traçabilité. Le fait que Kermeta soit un langage orienté-objets généraliste fait que son application à un domaine particulier tel que l'écriture de transformations de modèles passe par la création de frameworks spécialisés qui encapsulent les constructions du domaine.

La figure 4.17 présente l'implantation Kermeta d'une classe générique permettant de stocker la correspondance entre deux types d'objets. L'utilisation des variables de types génériques SRC et TGT permet de rendre la classe Trace réutilisable dans un contexte de typage statique fort. L'implantation proposée est très simple et permet de stocker des correspondances "un-un" unidirectionnelles entre deux ensembles d'objets. Ce système de trace très simple est suffisant pour la transformation *CLS2RDB* mais mériterait d'être enrichi en offrant par exemple la possibilité de stocker des traces bidirectionnelles ou des correspondances mettant en jeu plus de deux objets.

La conception d'une transformation en Kermeta se fait comme la conception de n'importe quel programme orienté-objets : conception de classes, création de frameworks, utilisation de patrons de conceptions, etc. Une des particularités de Kermeta pour la transformation de modèles est la possibilité d'encapsuler une partie du code de la transformation dans le méta-modèle d'entrée et de sortie de la transformation. Afin d'illustrer cette possibilité nous avons choisi d'intégrer au méta-modèle de base de données l'étape 3 de la transformation qui consiste a créer les colonnes correspondant aux clés étrangères. En effet, ce traitement est utile à la transformation *CLS2RDB*

```
1  package trace;
2  require kermeta
3  using kermeta::utils
4
5  /**
6   * This class represents a simple one to one
7   * unidirectional mapping
8   */
9  class Trace<SRC, TGT>
10 {
11   /** Mapping between source and target objects */
12   reference src2tgt : Hashtable<SRC, TGT>
13
14   operation create() is do
15     src2tgt := Hashtable<SRC, TGT>.new
16   end
17
18   /** get a target element */
19   operation getTargetElem(src : SRC) : TGT is do
20     result := src2tgt.getValue(src)
21   end
22
23   /** Store a trace */
24    operation storeTrace(src : SRC, tgt : TGT) is do
25     src2tgt.put(src, tgt)
26    end
27 }
```

FIG. 4.17 – Framework de trace (très simple) en Kermeta.

mais peut être également être utile à tout programme ou transformation qui utilise le méta-modèle de base de données et qui crée des clés étrangères.

Pour ce qui est de la transformation elle-même et des traitements correspondant aux étapes un et deux, nous avons choisi d'utiliser simplement une classe nommée *Class2RDBMS*.

Implantation

La figure 4.18 présente le listing Kermeta contenant la classe *Class2RDBMS*, ses propriétés et la méthode *transform* qui constitue le point d'entrée de la transformation. La première ligne du listing précise dans quel package est définie la classe *Class2RDBMS*. Les lignes de 3 à 6 permettent d'importer les éléments qui sont utilisés dans la transformation : le framework de base de Kermeta, le framework de trace que l'on a définit précédemment et enfin les deux méta-modèles. On remarque que le méta-modèle de classes est directement importé dans le format ECore (ligne 5) alors que pour le méta-modèle de base de données, c'est la version Kermeta qui est utilisée (ligne 6).

La classe *Class2RDBMS* comporte deux références définies respectivement aux lignes 13 et 15. La première, appelée *class2table* qui est utilisée pour stocker les correspondances entre classes et tables entre les deux premières phases de la transformation en utilisant la classe générique *Trace* définie précédemment. La seconde, nommée *fkeys* permet de stocker l'ensemble des clés étrangères crées au cours de la transformation afin de créer les colonnes correspondantes lors de la dernière phase. Pour représenter cette collection, on a utilisé la classe générique *Collection* du framework de Kermeta.

L'opération *transform* est le point d'entrée de la transformation. On retrouve dans le corps de cette méthode, un bloc d'initialisation suivi des trois phases de la transformation. Le bloc d'initialisation (lignes 18 à 22) permet de créer les structures de données utilisées et d'initialiser le modèle résultat. La première phase de transformation assure la création des tables correspondant aux classes marquées persistantes (ligne 23 à 29). Pour chacune des classes marquées comme persistantes, une nouvelle table est créée, ajoutée au modèle de sortie, et l'information de trace entre la classe et la nouvelle table est stockée.

La seconde phase consiste à créer les colonnes correspondant à chaque table. Pour cela on a défini une opération *createColumns* dont l'implantation est détaillée sur le listing de la figure 4.19. Le code de cette opération est simple : il consiste tout d'abord à ajouter l'ensemble des colonnes correspondant aux attributs (appel à l'opération *createColumnsForAttribute*) de la classe puis à ajouter les colonnes correspondant aux associations (appel à l'opération *createColumnsForAssociation*).

L'opération *createcolumnsForAttribute* permet la création des colonnes correspondant à un attribut. Trois cas doivent être pris en compte :

- Si le type de l'attribut est simple alors une seule colonne doit être créée.
- Si le type de l'attribut est une classe persistante alors une clé étrangère doit être créée. Les colonnes correspondantes ne seront ajoutées dans la table que lors de la troisième étape de la transformation.
- Si le type de l'attribut est une classe non persistante alors l'ensemble des colonnes

```
1  package Class2RDBMS;
2
3  require kermeta // The kermerta standard library
4  require "trace.kmt" // The trace framework
5  require "../metamodels/ClassMM.ecore"  // Input metamodel in ecore
6  require "../metamodels/RDBMSMM.kmt"    // Output metamodel in kermeta
7
8  [...]
9
10 class Class2RDBMS
11 {
12    /** The trace of the transformation */
13    reference class2table : Trace<Class, Table>
14    /** Set of keys of the output model */
15    reference fkeys : Collection<FKey>
16
17    operation transform(inModel : ClassModel) : RDBMSModel is do
18      // Initialize the trace
19      class2table := Trace<Class, Table>.new
20      class2table.create
21      fkeys := Set<FKey>.new
22      result := RDBMSModel.new
23      // Create tables
24      getAllClasses(inModel).select{ c | c.is_persistent }.each{ c |
25        var table : Table init Table.new
26        table.name := c.name
27        class2table.storeTrace(c, table)
28        result.table.add(table)
29      }
30      // Create columns
31      getAllClasses(inModel).select{ c | c.is_persistent }.each{ c |
32        createColumns(class2table.getTargetElem(c), c, "")
33      }
34      // Create foreign keys
35      fkeys.each{ k | k.createFKeyColumns }
36    end
37
38   [...]
39 }
```

FIG. 4.18 – Extrait de la classe Kermeta *Class2RDBMS*.

```
1  operation createColumns(table:Table, cls:Class, prefix:String) is
2  do
3    // add all attributes
4    getAllAttributes(cls).each{ att |
5      createColumnsForAttribute(table, att, prefix)
6    }
7    // add all associations
8    getAllAssociation(cls).each{ asso |
9      createColumnsForAssociation(table, asso, prefix)
10   }
11 end
12
13 operation createColumnsForAttribute(table : Table,
14                                      att : Attribute,
15                                      prefix : String) is
16 do
17   // The type is primitive : create a simple column
18   if PrimitiveDataType.isInstance(att.type) then
19     var c : Column init Column.new
20     c.name := prefix + att.name
21     c.type := att.type.name
22     table.cols.add(c)
23     if att.is_primary then table.pkey.add(c) end
24   else
25     var type : Class  type ?= att.type
26     // The type is persitant
27     if isPersistentClass(type) then
28       // Create a FKey
29       var fk : FKey init FKey.new
30       fk.prefix := prefix + att.name
31       table.fkeys.add(fk)
32       fk.references :=
33         class2table.getTargetElem(getPersistentClass(type))
34       fkeys.add(fk)
35     else
36      // Recusively add all attrs and asso of the non persistent table
37      createColumns(table, type, prefix + att.name)
38     end
39   end
40 end
```

FIG. 4.19 – Implantation des opérations *createColumns* et *createColumnsForAttribute*.

correspondant aux attributs de cette classe doit être créé. Cela est fait par un appel récursif à l'opération *createColumns*.

De la même manière, l'opération *createcolumnsForAssociation* permet la création des colonnes correspondant à une association. Nous ne détaillons pas ici son implantation car elle est très similaire à l'opération *createcolumnsForAttribute* à l'exception près que le type cible d'une association ne peut pas être un type simple.

```
1  class FKey
2  {
3    reference references : Table
4    reference cols : Column[1..*]
5
6    /**
7     * prefix for the name of the columns
8     * used by the createFKeyColumns method
9     */
10   attribute prefix : String
11
12   /**
13    * Create the FKey columns in the table
14    */
15   operation createFKeyColumns() is do
16     var src_table : Table
17     src_table ?= container
18     // add columns
19     references.pkey.each{ k |
20       var c : Column init Column.new
21       c.name := prefix + k.name
22       c.type := k.type
23       self.cols.add(c)
24       src_table.cols.add(c)
25     }
26   end
27 }
```

FIG. 4.20 – Implantation de l'opération *createFKeyColumns* dans la classe *FKey* du méta-modèle de base de données.

Enfin, la figure 4.20 présente l'implantation de la troisième étape de la transformation, c'est-à-dire la création des colonnes correspondant à une clé étrangère. Comme nous l'avons évoqué précédemment, l'opération permettant la création de ces colonnes a été directement intégrée dans la classe *FKey* du méta-modèle de base de donnée. En plus de l'opération *createFKeyColumns* nous avons ajouté un attribut *prefix* dans la classe *FKey* afin de stocker le préfixe à utiliser pour les noms des colonnes correspondant à la clé étrangère.

4.2.4 Conclusion

Cette étude sur l'utilisation de Kermeta pour l'implantation de transformations de modèles montre que, bien que le langage Kermeta ne soit pas dédié à la transformation il présente des qualités intéressantes pour la réalisation de transfomations.

La première de ces qualités est de permettre la conception de code réutilisable dans plusieurs transformations. Dans l'exemple de la transformation *Class2RDBMS* nous avons, par exemple, intégré des opérations dans le méta-modèle de base de données. Ce code, indispensable à la transformation *Class2RDBMS*, peut être réutilisé dans d'autre transformations ou programmes qui manipulent des modèles de base de données. Nous avons également créé un framework de traçabilité générique qui peut être réutilisé dans toute application ayant besoin de garder une correspondance entre deux types d'objets.

Un autre avantage de Kermeta est la grande liberté de conception offerte par le paradigme orienté-objets. Dans le contexte du workshop MTIP, Kermeta a permis d'implanter aussi bien les transformations très structurelles comme la transformation *Class2RDBMS* que les transformations plus algorithmiques comme la déterminisation ou la minimisation d'automates. Les approches dédiées à la transformation ciblent généralement des type de transformation particuliers. Par exemple, les approches purement déclaratives [LS05] ont de grandes qualités pour exprimer des transformations structurelles mais sont peu adaptées, bien que théoriquement utilisables, pour l'implantation de transformation plus algorithmiques.

4.3 Modélisation orientée-aspect

La seconde étude de cas que nous avons menée avec Kermeta est l'implantation de techniques académiques de *tissage* de modèles. La modélisation orientée-aspect est un domaine de recherche actif de l'ingénierie des modèles. L'idée est de permettre d'exprimer dans des modèles séparés, appelés *modèles d'aspects*, les différentes aspects d'un modèle complexe. Cette séparation permet une bonne gestion de la complexité, de découpler des préoccupations transverses et de réutiliser certains modèles d'aspects. La principale difficulté des techniques de modélisation orienté-aspect est le *tissage* des modèles d'aspect qui permet de construire le modèle complet.

Pour cette étude, nous avons travaillé sur deux techniques de tissage de modèles. La première permet de composer des diagrammes de classes[2] et la seconde de tisser des diagrammes de séquences [3]. L'objectif de cette étude est de vérifier que Kermeta est adapté à l'implantation de ces techniques. La section 4.3.1 présente brièvement le contexte de la modélisation orienté-aspect. Ensuite, les section 4.3.2 et 4.3.3 développent les deux techniques de tissage que nous avons implantées. Enfin, la section 4.3.4 conclut cette étude.

[2]Cette étude a été réalisé en collaboration avec Robert France, Sudipto Ghosh et Raghu Reddy de Colorado State University

[3]Cette étude a été réalisé en collaboration avec Jacques Klein

4.3.1 Contexte et motivations

La séparation des préoccupations transverses permet un meilleur contrôle sur les variations et les évolutions du logiciel. Dans le domaine de la programmation, cette idée a été popularisée par le langage AspectJ [Mil04], mais aujourd'hui, la communauté aspect s'intéresse aussi à opérer cette séparation plus tôt dans le cycle de développement : dès les phases d'analyse et conception et même d'expression des besoins. Dans cette optique, un certain nombre de travaux sont menés afin de définir des techniques de modélisation orientée-aspects qui permettent l'utilisation d'aspects non seulement dans le code mais aussi dans les modèles. Quel que soit le contexte, le prosessus de tissage d'aspects se divise en deux phases. Tout d'abord, une phase de détection permettant d'identifier des parties particulières d'un modèle de base, puis une phase de composition permettant de construire le modèle prenant en compte l'aspect.

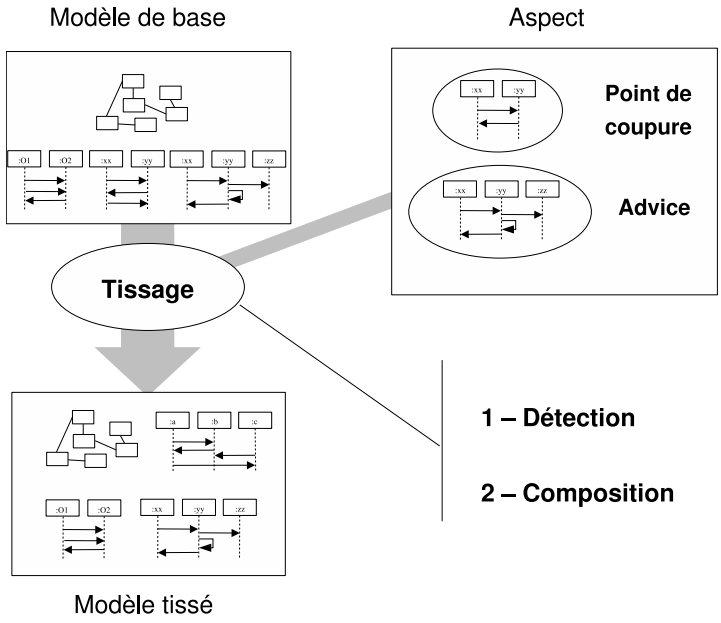


FIG. 4.21 – Processus général de tissage d'aspects.

La figure 4.21 présente les différents artefacts qui participent au processus de tissage. En haut à gauche, le modèle de base correspond au modèle dans lequel l'aspect doit être tissé. Un aspect, à droite sur la figure, est composé de deux parties : un *point de coupure* et un *advice*. Le point de coupure décrit les structures ou les comportements du modèle de base qui doivent être modifiés et l'advice spécifie la structure ou le comportement qui doit être composé dans le modèle de base. Lors de la phase de détection, le point de coupure sert donc à trouver l'ensemble des points du modèle de base auxquels l'advice doit être composé lors de la seconde phase.

Dans le cas ou l'on se place à un niveau de modélisation, les phases de détection et

composition sont des opérations de manipulation de modèles relativement complexes et intimement liées au langage de modélisation considéré. En effet, le tissage d'aspect pour des modèles statiques comme les diagrammes de classes met en jeu des opérateurs tout à fait différents de ce qui peut exister pour le tissage de modèles dynamiques tel que des diagrammes de séquences. Dans les deux cas, l'utilisation de Kermeta pour l'implantation du tissage d'aspects présente deux avantages majeurs. Le premier est qu'il permet d'ajouter des opérateurs de détection et de composition directement dans le méta-modèle du langage de modélisation considéré. Le second est lié à la compatibilité de Kermeta avec des standards de modélisation. Grâce à cette compatibilité, les fonctionnalités implantées en Kermeta sont utilisables avec des modèles issus d'outils standards et en particulier Eclipse/EMF.

4.3.2 Composition de diagrammes de classes

Dans [RFG+05] nous définissons un algorithme générique pour la composition de modèles structurels tel que les diagrammes de classes. Lorsque l'on doit composer des diagrammes de classes, au niveau le plus haut il s'agit de composer des packages. Ensuite, à l'intérieur des packages il faut composer des classes, puis on s'intéresse aux attributs et aux opérations, et ainsi de suite. Il se trouve qu'à chacun de ces niveaux, les problèmes sont sensiblement les mêmes, il faut :

- déterminer si deux objets doivent être composés,
- détecter les éventuels conflits afin d'assurer la correction du modèle résultat,
- appeler récursivement l'algorithme de composition sur les éléments contenus dans deux objets composés.

La seule chose qui diffère d'un type d'objet à l'autre est la politique qui permet de comparer deux objets pour déterminer s'il doivent être composés. Par exemple, dans le cas des diagrammes de classes, on compose généralement les packages sur la base de leurs noms alors que, pour la composition des opérations, il faut tenir compte non seulement de leurs noms mais aussi du nombre et des types de leurs paramètres. En dehors de cette politique de composition, il est possible d'écrire un algorithme de composition valable pour n'importe quel type d'objet. Les paragraphes suivants présentent cet algorithme, et son utilisation sur un exemple simple.

La figure 4.22 présente un extrait du méta-modèle que nous utilisons pour les diagrammes de classes. Comme dans la plupart des méta-modèles, tout en haut de l'arbre d'héritage entre les classes du méta-modèle se trouve une classe abstraite *ModelElement*. Cette classe permet d'encapsuler les caractéristiques communes à tous les concepts du langage de modélisation considéré. Juste en dessous de cette classe se trouve la classe abstraite *NamedElement* qui factorise ce qui est commun aux éléments ayant un nom. Enfin, les sous-classes de *NamedElement* correspondent aux concepts classiques des modèles de classes : *Package*, *Classifier*, *Operation*, *Parameter*, ...

Ce méta-modèle permet de créer des diagrammes de classes tel que les modèles d'entrée (a et b) de la figure 4.23. Chacun de ces modèles est composé d'un package nommé *Model1*. Le premier modèle (a) possède une seule classe *Customer* qui comporte deux attributs et deux méthodes *update* qui permettent de mettre à jour ces attributs.

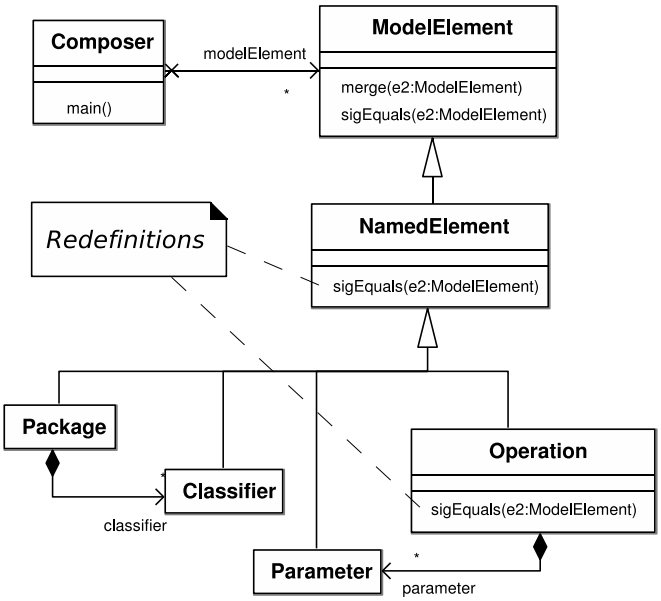
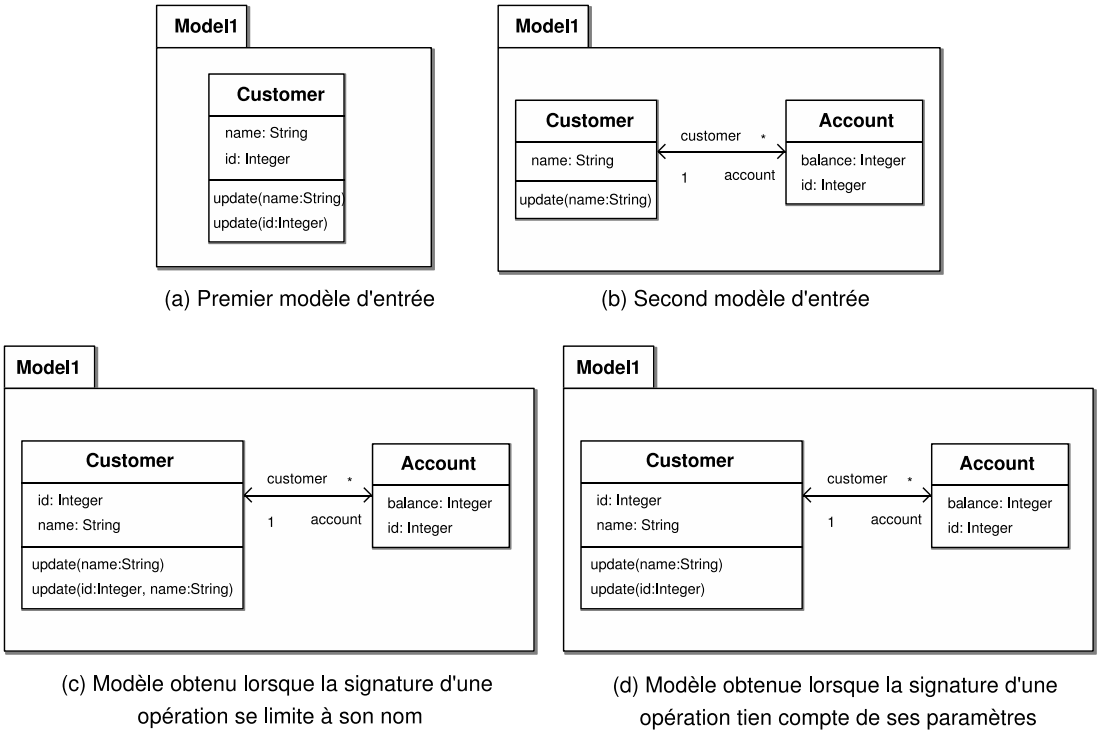


FIG. 4.22 – Méta-modèle de classes enrichie pour la composition.

Le second modèle (b) comporte deux classes : une classe *Customer* contenant un attribut et la méthode *update* correspondante et une classe *Account*. Au cours du processus de tissage d'aspects, on suppose qu'une phase de détection préalable a déterminée que les modèles (a) et (b) devaient être composés. Intuitivement, le résultat attendu est simple : on souhaite ajouter au modèle (a) la notion d'*Account* et une association entre la classe *Customer* et la classe *Account*. Sur la figure 4.23, ce modèle attendu correspond au modèle (d).

Pour permettre la composition de tels modèles de classes, nous avons ajouté des opérations dans le méta-modèle de la figure 4.22. Grâce aux mécanismes d'introspection implantés dans Kermeta, il est possible d'écrire l'algorithme de composition de façon générique, c'est-à-dire indépendante du type d'objet considéré. Pour cela, nous avons ajouté l'opération *merge* correspondant à la composition de deux objets dans la classe *ModelElement*. Un extrait du code de cette opération *merge* correspondant à l'algorithme de composition est donné figure 4.24.

Comme nous l'avons évoqué précédemment, il existe de légères différences quant aux politiques de sélection des objets devant être composés en fonction de leur type. Pour tenir compte de cela, l'algorithme de composition s'appuie sur un calcul de signature définie pour chaque type d'objet. L'idée de cette signature est d'intégrer l'ensemble des caractéristiques significatives d'un objet vis-à-vis de la composition. Par exemple, la signature d'une classe ne comporte que son nom alors que la signature d'une opération est constituée de son nom et du nombre et des types de ses paramètres. Pour permettre le calcul et la comparaison de signatures nous avons intégré l'opération abstraite *sigEquals* à la classe *ModelElement*. Cette opération peut être redéfinie librement dans



(a) Premier modèle d'entrée

(b) Second modèle d'entrée

(c) Modèle obtenu lorsque la signature d'une opération se limite à son nom

(d) Modèle obtenue lorsque la signature d'une opération tien compte de ses paramètres

FIG. 4.23 – Composition de deux packages en utilisant deux signatures différentes pour les opérations.

chacune des sous-classes du méta-modèle afin de spécialiser la signature de chaque type d'élément.

En pratique, les signatures des objets pour la composition de diagrammes de classes sont relativement simples. Dans la classe *NamedElement* nous avons implanté une signature très simple basée sur les noms. Cette signature est valable pour la plupart des classes qui héritent de *NamedElement*. Le modèle (c) de la figure 4.23 présente le résultat de la composition des modèles (a) et (b) en utilisant cette signature basée sur les noms pour tout les types d'objets (y compris pour les opérations). Le résultat obtenu est proche du résultat attendu excepté pour l'opération *update*. En effet, les opérations *update* comportant des paramètres ont été composées pour former une nouvelle opération *update* comportant deux paramètres. Afin d'éviter ce problème nous avons simplement redéfini à nouveau l'opération *sigEqual* dans la classe Operation afin de tenir compte des paramètres. Grâce à cette redéfinition le modèle composé obtenu est le modèle (d) c'est-à-dire le modèle qui était intuitivement attendu.

Le fonctionnement de l'algorithme de composition générique est simple. La précondition de l'algorithme est que les deux objets à composer aient le même type et la même signature. Les deux objets à composer ayant le même type, le résultat de la composition est un troisième objet du même type. Sur le listing de la figure 4.24 ce résultat est initialisé à la ligne 5. Ensuite, l'idée est de parcourir l'ensemble des propriétés de la classe des objets à composer et de calculer pour chacune des propriétés la valeur qui doit être affectée au résultat :

- Les ligne 8 à 16 de l'algorithme traitent des propriétés dont le type est simple (Boolean, Integer, String, ...). Pour une telle propriété si les deux objets à composer présentent des valeurs différentes un conflit est alors détecté. Si les deux objets présentent la même valeur, ou que seul l'un des deux objets a une valeur pour cette propriété, alors cette valeur est choisie.
- Les lignes 17 à 31 traitent des propriétés dont la multiplicité est 1 et dont le type est complexe. Dans ce cas, si les deux objets à composer font référence à des objets de signatures différentes alors un conflit est détecté. Si un seul des deux objets fait référence à un objet non nul alors cet objet est cloné et associé au résultat. Dans le cas ou l'on a affaire à deux objets de même signature et que la propriété est une composition alors l'opération *merge* est appelée récursivement pour ces objets. Le fait d'appeler l'algorithme de composition uniquement pour les propriétés qui sont des compositions permet d'assurer la terminaison de l'algorithme car la structure d'objets en relation par composition est un arbre.
- Enfin les lignes 32 à 51 traitent des propriétés dont la multiplicité est supérieure à un. Dans ce cas, la propriété contient une collection d'objets. Afin de construire la collection d'objets composée, l'idée est de garder les objets propres à chaque objet à composer et de composer ceux qui ont des signatures identiques. Là encore, ils ne sont composés récursivement que si la propriété est une composition.

Afin de valider cet algorithme et d'évaluer ses qualités nous l'avons implanté en Kermeta et testé pour un ensemble de diagrammes de classes simples. Ce travail fait actuellement l'objet de travaux plus avancés afin d'intégrer cet algorithme de composition dans une démarche complète de tissage de modèles (incluant notamment une phase

```
operation merge(e2: ModelElement) is
pre same_signature is e1.sigEquals(e2)
do
// Initialize the merged object
result := self.getmetaClass.new
// Iterate on the properties of the objects to merge
self.getMetaClass.getAllProperties.each { p |
  if isPrimitive(p.type) then // handle simple attributes
    if self.get(p) != void and e2.get(p) != void and
       self.get(p) != e2.get(p) then
      // A conflict has been detected
    else
      if self.get(p) != void then result.set(p, self.get(p))
                             else result.set(p, e2.get(p)) end
    end
  else // deal with attributes whose type is not primitive
    if p.upper == 1 then // it is a sigle object
      if self.get(p) != void and e2.get(p) != void and
         not self.get(p).sigEquals(e2.get(p)) then
        // A conflict has been detected
      else
        if self.get(p) == void then result.set(p, e2.get(p))
        else if e2.get(p) == void then result.set(p, e2.get(p))
        else
          if p.isComposite then
            result.set(p, e2.get(p).merge(self.get(p)))
          else
            result.set(p, self.get(p).clone)
          end
        end end
      end
    else // thr property contains several objects
      // Merge abjects that have to be merged
      self.get(p).each{ v1 |
        e2.get(p).select{ v2 | v1.sigEquals(v2)}.each{ v2 |
          if p.isComposite then result.get(p).add(v1.merge(v2))
          else result.get(p).add(v1.clone) end
        }
     }
     // add objects that are refered only by self
     self.get(p).select{ v1 |
     e2.get(p).select{ v2 | v1.sigEquals(v2)}.size == 0 }.each{ v |
       result.get(p).add(v.clone)
     }
     // add object that are only refered by e2
     e2.get(p).select{ v2 |
     self.get(p).select{ v1 | v1.sigEquals(v2)}.size == 0 }.each{ v |
       result.get(p).add(v.clone)
     }
end end } end // of operation merge
```

FIG. 4.24 – Algorithme de composition en pseudo-code Kermeta.

de détection et des directives de compositions permettant de lever un certain nombre de conflits). Une des qualités de cet algorithme est d'être très peu dépendant du type de modèles considérés et une perspective intéressante serait d'évaluer la pertinence de cet algorithme pour la composition de modèles autres que les diagrammes de classes.

4.3.3 Tissage de diagrammes de séquences

Le second type de modèles pour lequel nous avons implanté des opérateurs à la fois de détection et de composition sont les diagrammes de séquences. Contrairement aux diagrammes de classes, les diagrammes de séquences sont utilisés pour modéliser le comportement et non la structure d'un système. De ce fait, les opérateurs de détection ou de composition sont beaucoup plus liés à la sémantique des diagrammes de séquences qu'à leur structure. L'implantation Kermeta du tissage d'aspects comportementaux est basée sur les opérateurs sémantiques présentés dans [KF06].

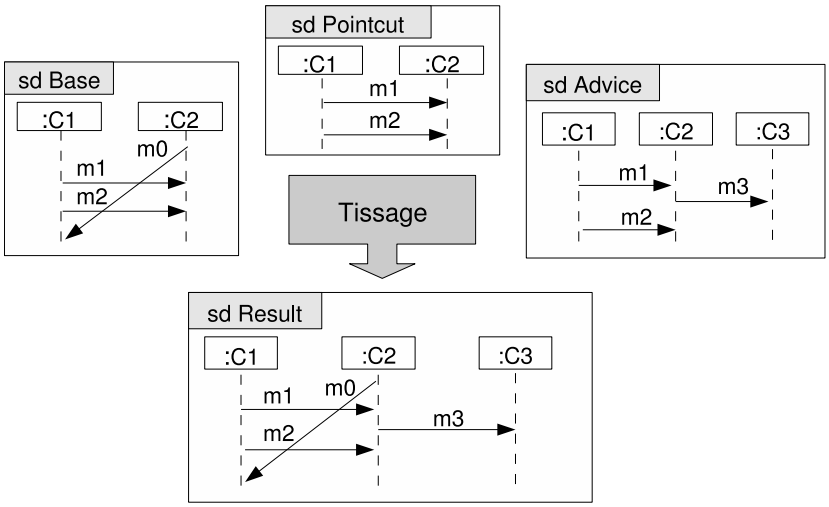


FIG. 4.25 – Exemple de tissage d'un aspect comportemental dans un scénario de base.

La figure 4.25 présente un exemple simple de tissage de diagrammes de séquences. Un des avantages de l'approche proposée est l'homogénéité des formalismes utilisés : le modèle de base, le point de coupure et l'advice sont tous trois des diagrammes de séquences. L'aspect à tisser est composé du point de coupure qui spécifie le comportement à détecter dans le modèle de base, et de l'advice qui correspond au comportement qui doit être tissé. Dans l'exemple de la figure 4.25 le comportement à détecter est la succession de message $m1$ suivi de $m2$. L'advice spécifie alors qu'un message $m3$ doit être émis vers une instance $C3$ après réception de $m1$ par $C2$. Le diagramme de séquences résultat présente bien une nouvelle instance $C3$ et le nouveau message $m3$.

Il est intéressant de noter que le tissage ne pose pas de problème si des messages, tel que $m0$ par exemple, sont émis avant la partie détectée et reçus après. Dans certain cas il est également possible de réaliser le tissage si des messages sont intercalés à l'intérieur de

la partie détectée. Dans [KF06] trois politiques de détection, et les algorithmes qui leur correspondent, sont définis. En pratique, chacune des stratégies de détection répond à des besoins de tissage différents et présente donc un intérêt suffisant pour être implanté.

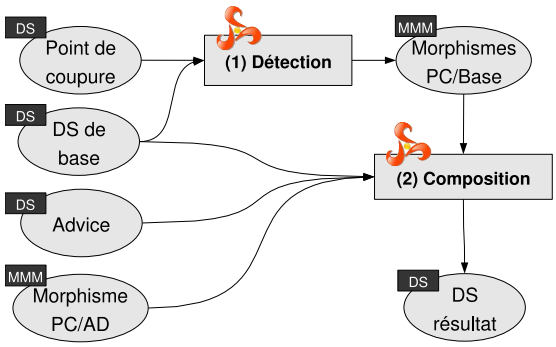


FIG. 4.26 – Processus de tissage d'un aspect comportemental.

La figure 4.26 détaille les entrées et sorties en termes de modèle des deux étapes de tissages. Chaque ellipse représente un modèle et le carré en haut à gauche spécifie le méta-modèle correspondant. Comme nous l'avons évoqué précédemment, la plupart des modèles manipulés sont des diagrammes de séquences. En plus des diagrammes de séquences, il existe deux modèles de morphismes qui sont utilisés lors de la phase de composition. Un morphisme est une simple correspondance entre les éléments de deux diagrammes de séquences. Dans l'exemple de tissage dont nous avons discuté précédemment, ces correspondances étaient implicites et basées sur les noms des instances et des messages. En pratique le fait d'utiliser une structure de données explicite permet plus de souplesse pour exprimer la relation entre un point de coupure et un advice, et elle est indispensable pour représenter le résultat de la phase de détection.

La première phase du processus de tissage utilise le point de coupure et le modèle de base. L'objectif de cette phase est de trouver toutes les occurrences du comportement décrit dans le point de coupure à l'intérieur du modèle de base. Pour chacune des occurrences trouvées un morphisme entre les éléments du point de coupure et les éléments correspondant dans le modèle de base sonr créés. Chacun de ces morphismes correspond en fait à un point de jonction dans la terminologie aspect. La seconde phase du processus de tissage utilise les morphismes produits afin de composer le comportement décrit dans l'advice à chaque point de jonction. L'opérateur de composition utilisé au cours de cette phase est une somme amalgamée entre diagramme de séquences. La définition formelle de cet opérateur est donnée dans [KF06].

L'ensemble du processus de tissage est implanté en Kermeta. L'utilisation de Kermeta a permis d'introduire les opérations de tissage directement dans le méta-modèle de diagramme de séquences utilisé. Un diagramme de classe de ce méta-modèle est donné sur la figure 4.27. L'implantation en Kermeta à permis de valider les différentes stratégies de détection proposées et l'algorithme de composition. Un certain nombre de travaux futurs est à envisager pour permettre par exemple l'utilisation de modèles

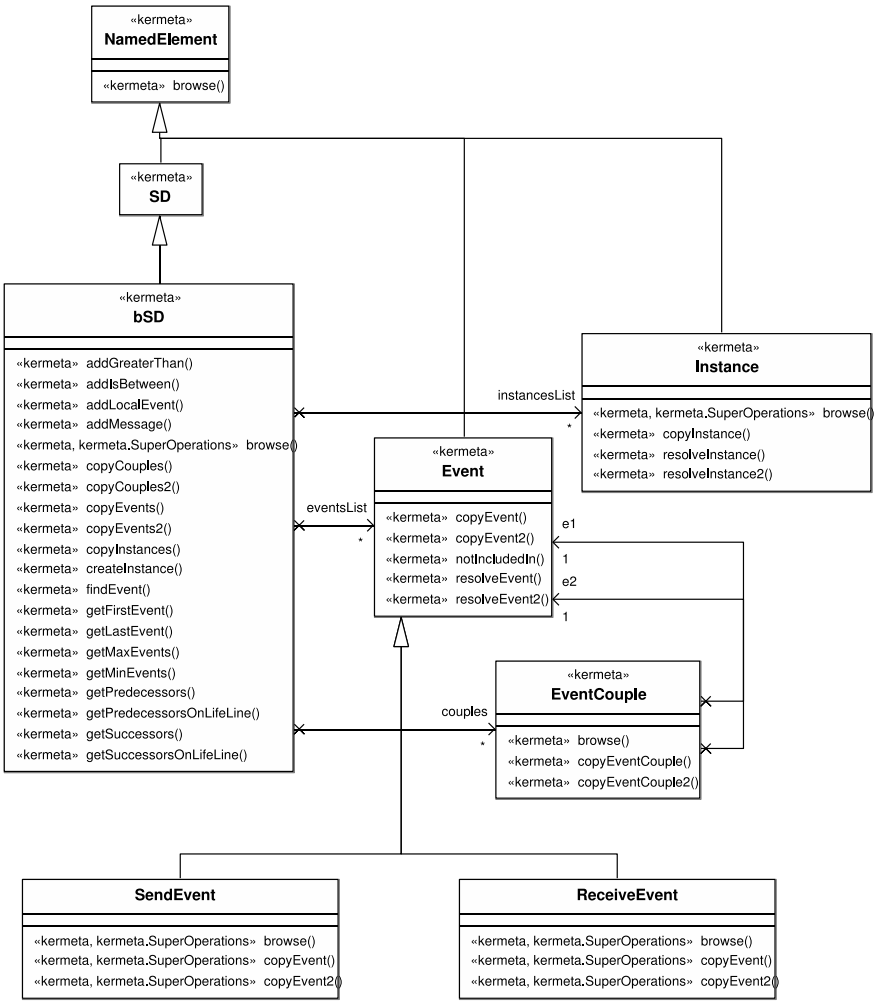


FIG. 4.27 – Méta-modèle de diagrammes de séquences augmenté d'opération Kermeta permettant le tissage d'aspects comportementaux.

de base faisant apparaître des comportements infinis. En effet les constructions telles que les boucles présentes dans les diagrammes de séquence d'UML 2.0 autorisent un pouvoir d'expression plus riche que de simples diagrammes de séquences de base. Une seconde perspective à ce travail est de combiner des techniques de tissage d'aspects dans des modèles statiques (comme les diagrammes de classes) et le tissage d'aspect comportementaux.

4.3.4 Conclusion

Au cours de cette étude nous avons implanté deux techniques différentes de tissage de modèles. Pour ces deux techniques, la compatibilité de Kermeta avec d'autres outils de modélisation (en particulier Eclipse/EMF) est un élément important qui permet d'assurer la bonne intégration des opérateurs de tissage dans un environnement de modélisation complet. Dans les deux cas nous avons également utilisé les possibilités offertes par Kermeta pour ajouter des opérations dans les classes des méta-modèles et ainsi encapsuler les opérateurs de tissage dans les méta-modèle considérés.

Dans le cas du tissage de diagrammes de classes, la composition est structurelle, c'est-à-dire que la composition des modèles revient à la composition des objets qu'ils contiennent. Nous avons profité de cette propriété pour implanter l'opérateur de composition de façon générique en utilisant les mechanismes d'introspection (ou reflexion) offerts par Kermeta. Dans le cas du tissage de diagrammes de séquence, la détection et la composition repose sur la sémantique des diagrammes de séquences. Les opérateurs correspondant sont donc moins génériques mais s'intègrent naturellement sous forme d'opérations à ce méta-modèle. Le fait que Kermeta soit un langage orienté-objets a permis de réutiliser le patron de conception *strategie* afin d'implanter élégamment les différentes stratégies de détection et composition pour les diagrammes de séquences.

Les résultats de cette étude montrent des qualités intéressantes de Kermeta pour l'implantation de techniques de modélisation orientée-aspects. Chacune des deux techniques a pu être conçue et implantée de façon satisfaisante, en particulier de façon réutilisable (opérateur générique dans le cas des modèles de classes) et extensible (signature personalisable dans le cas des modèles de classes et stratégies de compositions variables dans le cas des diagrammes de séquences).

4.4 Langages spécifiques à l'ingénierie des exigences

Pour la dernière étude de cas présentée dans ce chapitre, nous avons implanté en Kermeta une chaine de traitement des exigences logicielles basée sur l'utilisation de modèles[4]. Cette chaîne de traitement des exigences a été conçue dans le cadre du projet CAROLL/Mutation [CAR] (faisant intervenir l'INRIA, le CEA et Thalès) et utilisée plus récemment par France Télécom. Ces travaux ont fait l'objet d'un certain nombre de publications parmis lesquelles [NFLJ03, NF05, NFLTJ06]. Nous avons

[4]L'implantation a été réalisée fin 2005 par Waqas Ahmed Saeed au cours d'un stage de Master 2 Recherche dans l'équipe Triskell.

choisi cette application pour valider Kermeta vis-à-vis d'un processus d'ingénierie des modèles complet mettant en jeu la définition de plusieurs langages de modélisations, de leurs sémantiques et de transformations de modèles. Nous illustrons en particulier deux approches pour la définition de la sémantique d'un langages dédiés. La première solution est l'implantation d'une transformation de modèles permettant de cibler un langage dont la sémantique est connue (compilation). La seconde solution est l'implantation de la sémantique en Kermeta directement dans le méta-modèle du langages dédié (interprétation).

4.4.1 Processus de traitement des exigences

L'objectif de ces travaux est la formalisation progressive des exigences logicielles en des modèles sémantiquement précis et donc exploitables. Dans la pratique actuelle du génie logiciel, les exigences sont généralement écrites en langage naturel. Cela a l'avantage indispensable de permettre aux non-informaticiens, spécialistes d'un domaine, de rédiger des exigences. Cependant, du fait de sa complexité et des ambiguïtés qu'il peut contenir, l'utilisation du langage naturel a l'inconvénient de rendre les exigences difficiles à exploiter automatiquement.

Une solution à ce problème serait d'imposer l'utilisation de langages formels, à la sémantique bien définie, pour l'écriture d'exigences. Malheureusement, dans la majorité des cas cette solution n'est pas réaliste car, comme nous l'avons évoqué précédemment, les rédacteurs d'exigences sont rarement des informaticiens. L'approche que nous proposons est plus progressive et se base sur un "langage naturel contrôlé" pour la description d'exigences. Ce langage est conçu pour être utilisable directement par des experts, non-informaticiens, d'un domaine.

La figure 4.28 détaille le processus général de traitement des exigences que nous utilisons. Les entrées de ce processus sont des exigences textuelles dans un langage syntaxiquement contraint et un ensemble de patrons d'interprétation. Ces patrons d'interprétation permettent une analyse sémantique des exigences proposées et la construction d'un modèle de données et d'un modèle de cas d'utilisations. Le modèle de cas d'utilisations que nous utilisons est étendu par rapport à ce qui existe dans UML : les cas d'utilisations possèdent des paramètres, des pré-conditions et des post-conditions. Grâce à ces extensions, notre modèle de cas d'utilisations permet de simuler l'enchaînement des cas d'utilisations de l'application décrite. Le modèle sémantique et le modèle de simulation sont utilisés, en bout de chaîne, pour générer des objectifs de test pour l'application.

L'utilisation de cette chaîne de traitement des exigences a deux intérêts principaux. Le premier est bien sûr la production d'un modèle simulable et d'objectifs de test pour l'application décrite. Le second découle de l'ensemble des informations qui est remonté automatiquement au rédacteur des exigences par chacune des phases de transformation. Tout d'abord, l'analyse syntaxique permet de s'assurer que les exigences ne contiennent pas d'erreurs syntaxiques. Ensuite, la phase d'analyse sémantique permet de détecter les éventuelles contradictions ou les ambiguïtés existantes. Ces erreurs ou omissions peuvent alors être corrigées itérativement par le rédacteur des exigences. Enfin, grâce

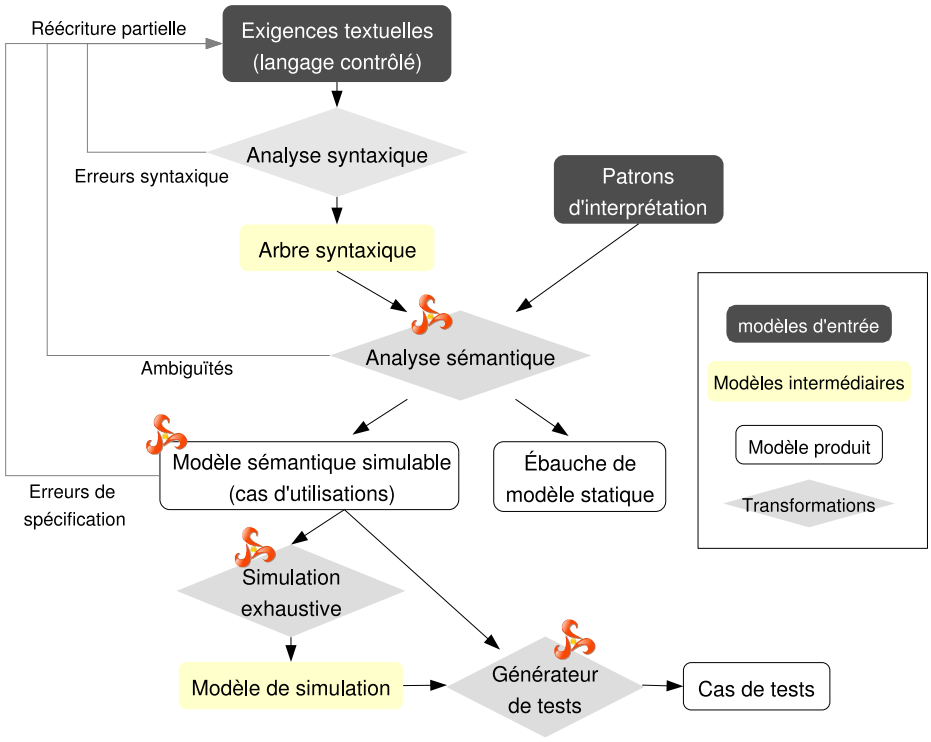


FIG. 4.28 – Processus de traitement des exigences.

au simulateur, le rédacteur des exigences peut facilement mettre à l'épreuve par le test les exigences qu'il a écrit et corriger les éventuelles incorrections qu'elles contiennent. Le processus proposé permet ainsi la construction progressive d'un modèle sémantiquement précis de l'application à réaliser.

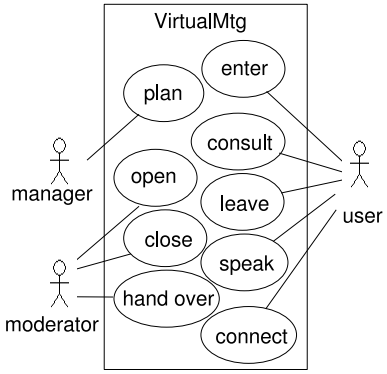


FIG. 4.29 – Cas d'utilisations du système de réunion virtuelles.

D'un point de vue pratique, le processus de la figure 4.28 est implanté en utilisant les outils de l'ingénierie des modèles. Chaque langage utilisé est spécifié par un méta-modèle et les différentes phases d'analyse et de génération sont implantées comme des transformations de modèles. Seule l'analyse syntaxique qui consiste à parser les exigences textuelles est réalisé en JAVA grâce à ANTLR [ANT]. Le parseur produit permet de générer un arbre syntaxique sous la forme d'un modèle EMF. La compatibilité de Kermeta avec EMF permet ensuite d'exploiter ce modèle en Kermeta dans la suite de la chaîne.

Le processus de traitement des exigences utilise deux langages dédiés pour représenter les exigences : le langage de description des exigences et le langage de cas d'utilisation. D'un point de vue pratique, il existe deux manières de fixer la sémantique d'un langage. La première est d'utiliser une transformation vers un langage dont la sémantique est connue. C'est la technique que nous avons utilisé pour fixer la sémantique du langage de description des exigences. La seconde est, comme nous l'avons fait pour les automates dans la section 4.1, d'utiliser Kermeta pour intégrer la sémantique dans le langage. C'est la technique que nous avons utilisé pour rendre simulable le modèle de cas d'utilisation.

Afin d'illustrer les discussions des sections suivantes, nous prenons l'exemple d'un système de réunions virtuelles. Ce système permet à des utilisateurs distants de simuler des réunions de travail en ligne. Chaque utilisateur se connecte au système de réunions virtuelles grâce à son email et à un mot de passe. Il a ensuite la possibilité d'organiser ou de rejoindre des réunions. Au sein de chaque réunion, un modérateur gère la parole des participants. La figure 4.29 présente un diagramme de cas d'utilisations UML de l'application de réunions virtuelles. Les sections suivantes détaillent les différentes étapes du processus de rédaction des exigences en utilisant cet exemple.

4.4.2 Modèle sémantique et simulation

Cette section détaille le modèle sémantique de cas d'utilisations que nous utilisons. La figure 4.30 présente le méta-modèle de ce langage. Ce méta-modèle précise qu'un modèle de cas d'utilisations est constitué d'une part d'un ensemble de types de données et d'autre part d'un ensemble de cas d'utilisations. Les types de données possèdent simplement un nom et un ensemble d'instances. Les cas d'utilisations ont un nom, des paramètres, une pré-condition et une post-condition. Les pré-conditions et les post-conditions sont des expressions booléennes. Pour permettre de représenter ces expressions, notre modèle sémantique inclus un langage qui comporte les constructions suivantes :

- des prédicats (classe *Predicate*) : un prédicat possède un nom et fait éventuellement référence à des paramètres.
- des opérateurs logiques : la négation (classe *Negation*), la conjonction (classe *Conjonction*) et la disjonction (classe *Disjonction*).
- des quantificateurs : *forall* (classe *Forall*) et *exists* (classe *Exists*) qui permettent de quantifier une expression pour un ensemble de paramètres.
- une forme d'implication (classe *PreImplies*) utilisable dans les post-conditions. Cette implication permet d'évaluer des gardes dans le contexte de la pré-condition de façon similaire à l'opérateur *@pre* d'OCL ou de Kermeta.
- un opérateur d'égalité (classe *Equals*).

A titre d'exemple, le listing de la figure 4.31 présente deux cas d'utilisations pour le système de réunions virtuelles. Chacun des cas d'utilisation possède deux paramètres : un utilisateur u et une réunion m. Les cas d'utilisations sont spécifiés de manière déclarative au moyen de leur pré-condition et de leur post-condition. On considère que l'état du système est représenté par un ensemble de prédicats.

La pré-condition d'un cas d'utilisation défini les conditions nécessaires sur l'état du système pour pouvoir exécuter l'action correspondant à ce cas d'utilisation. Par exemple, pour le cas d'utilisation *open*, la pré-condition impose que la réunion m soit préalablement créée (prédicat `created(m)`), que l'utilisateur u soit le modérateur de la réunion m (prédicat `moderator(u,m)`), etc.

La post-condition d'un cas d'utilisation spécifie, de manière déclarative, les modifications de l'état du système conséquentes à l'exécution de l'action correspondant à ce cas d'utilisation. Pour le cas d'utilisation *open* les modifications apportées à l'état du système sont simples : le prédicat `open(m)` doit être mis à *vrai*. Les prédicats auxquels la post-condition ne fait pas référence conservent simplement leurs valeurs. Pour le cas d'utilisation *close*, la post-condition est plus intéressante. En effet, lorsqu'une réunion est fermée, il est nécessaire de mettre à jour l'état de tous ses participants. L'utilisation du quantificateur *forall* permet de spécifier qu'à l'issue de la fermeture de la réunion, plus aucun participant ne peut ni être présent dans cette réunion (prédicat `entered(v,m)`), ni demander la parole dans cette réunion (prédicat `asked(v,m)`) ni avoir la parole dans cette réunion (prédicat `speaker(v,m)`).

Les pré-conditions et les post-conditions des cas d'utilisations permettent de spécifier les conditions d'application et le résultat de l'application des cas d'utilisations. En

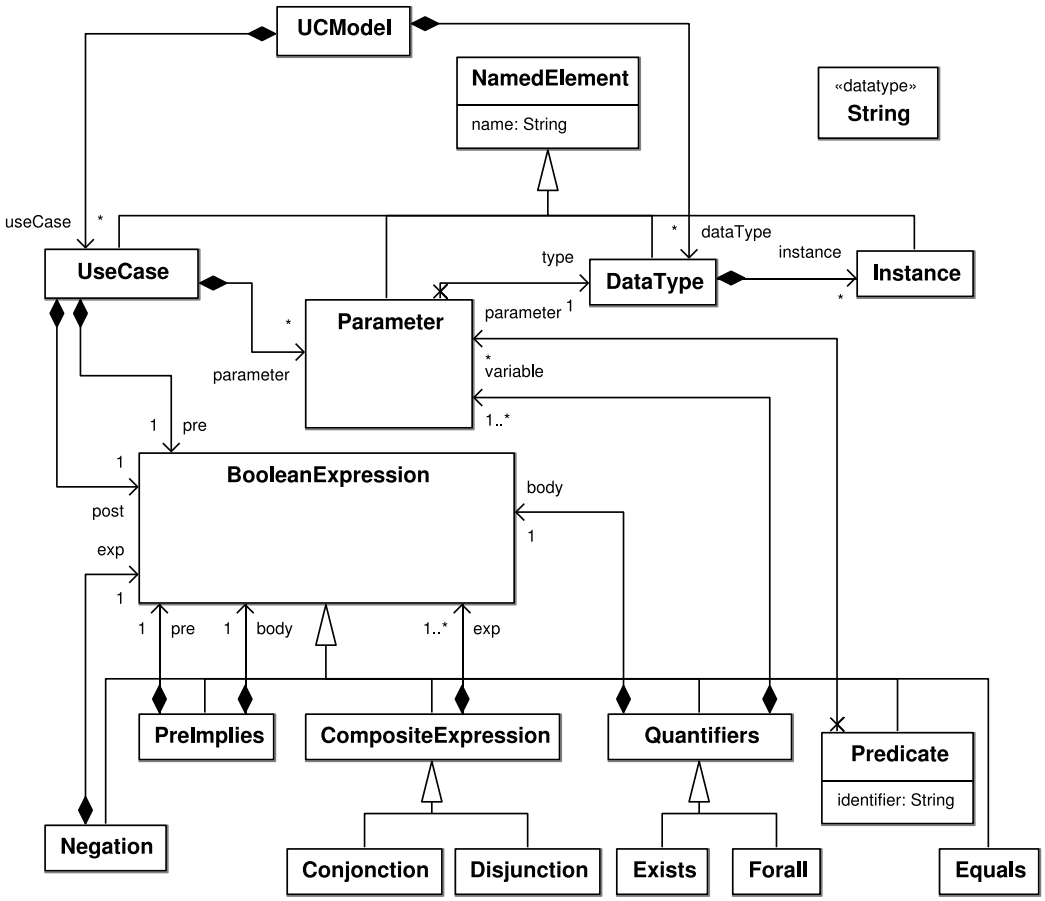


FIG. 4.30 – Méta-modèle de cas d'utilisations paramétrés et contractualisés.

```
1  # use case OPEN : user u opens the meeting m
2  UC open(u : participant; m : meeting)
3  pre  created(m) and moderator(u,m) and not closed(m) and
4       not opened(m) and connected(u)
5  post opened(m)
6
7  #use case CLOSE : user u closes the meeting m
8  UC close(u : participant; m : meeting)
9  pre  opened(m) and moderator(u,m)
10 post not opened(m) and closed(m) and forall(v:participant) {
11        not entered(v,m) and not asked(v,m) and not speaker(v,m)
12      }
```

FIG. 4.31 – Cas d'utilisation *open* et *close* du système de réunion virtuelle (la syntaxe utilisée est la syntaxe concrète associée au langage de cas d'utilisations de la figure 4.30).

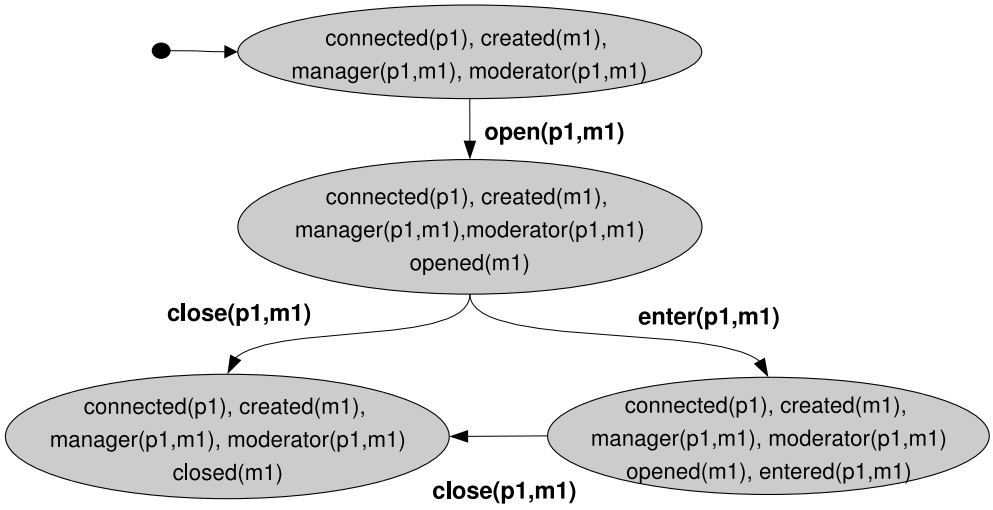


FIG. 4.32 – Exemple de graphe de simulation.

utilisant cette information, il est possible de simuler les enchaînements de cas d'utilisations possibles. La figure 4.32 montre un graphe de simulation obtenu pour l'exemple du système de réunions virtuelles. Pour que la simulation soit possible il est nécessaire de définir un état de simulation initial. Sur la figure, les états contiennent la liste des prédicats qui ont pour valeur *vrai*. Les prédicats non représentés sont donc *faux* par convention. A partir de l'état initial, on évalue les pré-conditions de l'ensemble des cas d'utilisations afin de déterminer ceux qui peuvent être appliqués à partir de l'état courant. Lorsqu'un cas d'utilisation est applicable, un nouvel état du système est calculé en appliquant sa post-condition sur l'état courant.

En pratique, nous avons utilisé Kermeta afin d'intégrer cette sémantique de simulation directement dans le méta-modèle de cas d'utilisations de la figure 4.30. Avant cette implantation en Kermeta, nous avions réalisé un prototype du simulateur en JAVA. Ce dernier se présentait alors comme un programme permettant de lire un modèle de cas d'utilisations dans un fichier texte puis de le simuler. L'utilisation de l'ingénierie des modèle et de Kermeta pour la définition du langage de cas d'utilisations lui même puis du simulateur à permis d'obtenir une application beaucoup plus facile à maintenir et à faire évoluer que le prototype initial.

La possibilité de simuler les cas d'utilisations présente deux intérêts : d'une part, la simulation permet au rédacteur des exigences de mettre à l'épreuve et de corriger ses exigences et, d'autre part, cela permet d'extraire des objectifs de tests pour l'application finale. Dans [NFLTJ06], nous détaillons un processus complet permettant d'obtenir des cas de tests pour les applications décrites par un modèle de cas d'utilisations contractualisés. L'implantation en Kermeta des algorithmes de génération de test que nous avons proposé est en cours.

La section suivante précise le processus d'analyse sémantique des exigences textuelles qui permet la création d'un ensemble de cas d'utilisations simulable à partir du Langage de Description d'Exigences (LDE).

4.4.3 Analyse sémantique d'exigences textuelles

Comme nous l'avons précisé précédemment, il est nécessaire de tenir compte du fait que les rédacteurs d'exigences peuvent ne pas être des informaticiens. L'écriture de cas d'utilisations nécessite, non seulement une connaissance de ce formalisme, mais aussi et surtout, une compréhension globale de l'ensemble des exigences relatives à chaque cas d'utilisation. En effet, afin de spécifier correctement la pré-condition et la post-condition d'un cas d'utilisation, il est nécessaire de tenir compte de toutes les exigences faisant référence à ce cas d'utilisation. Il est donc peu réaliste de penser que des exigences puissent être écrites directement dans le formalisme de cas d'utilisations présenté dans la section précédente.

Afin de résoudre ce problème, nous proposons un Langage de Description d'Exigences (LDE) syntaxiquement plus accessible et ne nécessitant pas de grouper les exigences relatives à chaque cas d'utilisation. Ce langage a été construit pour être le plus proche possible des structures utilisées dans des exigences écrites en langage naturel. En étudiant des exigences pour des systèmes avioniques, nous avons remarqué que pour la grande majorité des exigences, les tournures de phrases utilisées étaient relativement simples et peu nombreuses. L'approche que nous proposons n'est pas une analyse directe du langage naturel mais un langage contraint qui permet de reproduire des constructions du langage naturel.

```
if a "participant" does "close" a "meeting" then
  each "participant" does "leave" this "meeting".

before a "participant" does "enter" a "meeting",
  this "meeting" must be "opened".

after a "participant" did "open" a "meeting",
  each "participant" being "connected" can "enter" this "meeting".
```

FIG. 4.33 – Exemple d'exigences en LDE.

Le listing de la figure 4.33 présente trois exigences relatives au système de réunions virtuelles écrites en Langage de Description d'Exigences. La première exigence spécifie que lors de la fermeture d'une réunion tous ses participants doivent en sortir. La structure du LDE est très simple : les expressions entre guillemets sont les entités, les acteurs et les services du système et les autre termes sont considérés comme des mots clés. La seconde exigence exprime le fait que pour qu'un utilisateur puisse entrer dans une réunion, il faut que cette réunion soit ouverte. Enfin la troisième exigence spécifie que lorsqu'un utilisateur ouvre une réunion, les participants peuvent entrer dans cette réunion.

```
# requirement R1 :
if a "participant" is "connected" then
  this "participant" can "plan" a "meeting"

# requirement R2 :
a "participant" for a "meeting" is the "manager" after
  this "participant" did "plan" this "meeting"
```

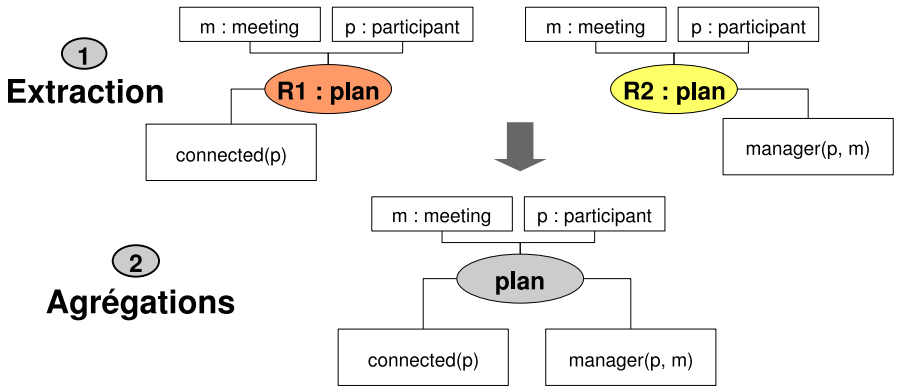


FIG. 4.34 – Exemple d'analyse sémantique : des phrases LDE au cas d'utilisations.

Il est intéressant de remarquer que chaque phrase de LDE ne fait que décrire partiellement chaque service. C'est lors de la phase d'analyse sémantique que les différentes exigences relatives à un service sont agrégées. Cette phase d'analyse sémantique est basée sur l'utilisation de patrons d'interprétation qui donnent une sémantique en termes de cas d'utilisations contractualisés à chaque construction du LDE.

La figure 4.34 présente un exemple d'analyse sémantique pour deux phrases LDE relatives à la planification de réunions dans le système de réunions virtuelles. La première phrase (R1) spécifie qu'un participant doit être connecté pour pouvoir planifier une réunion. La seconde phrase spécifie que le participant qui planifie une réunion devient le gérant de cette réunion. L'interprétation de ces phrases se déroule en deux étapes : l'extraction de cas d'utilisations élémentaires correspondant à chacune des phrases, puis l'agrégation des cas d'utilisations élémentaires portant sur les même services.

La première étape est l'extraction. Au cours de cette étape, un cas d'utilisation est créé pour chaque phrase LDE. Par exemple pour l'exigence (R1) de la figure 4.34, un cas d'utilisation contenant simplement la pré-condition `connected(p)` permet de rendre compte du fait que seul un utilisateur connecté peut planifier une réunion. Le cas d'utilisation correspondant à l'exigence (R2) comporte simplement une post-condition qui assure qu'au terme de la planification l'utilisateur qui à planifié la réunion est son gérant.

La phase d'extraction se base sur l'utilisation de patrons d'interprétations. La figure 4.35 présente un exemple de patron d'interprétation. Chaque patron d'interprétation

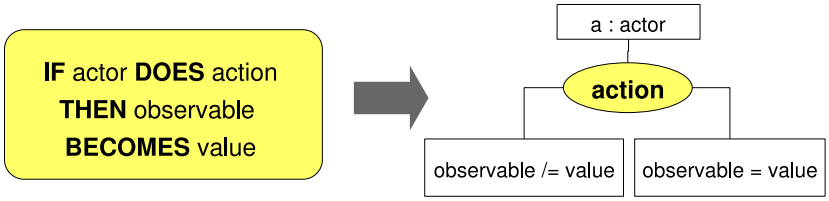


FIG. 4.35 – Extraction d'un cas d'utilisation grâce à un patron d'interprétation.

est constitué d'un motif (à gauche sur la figure) et d'un cas d'utilisation produit à partir de ce motif (à droite sur la figure). Lorsque le motif correspondant à un patron d'interprétation correspond à une phrase LDE, alors un cas d'utilisation peut être produit pour cette phrase. Pour réaliser l'extraction des cas d'utilisations à partir des phrases LDE, on essaie successivement les différents patrons d'interprétation disponibles afin de trouver celui qui permet de construire le cas d'utilisation correspondant à la phrase considérée. Deux cas d'erreur sont possibles :

- Aucun cas d'utilisation ne correspond à la phrase considérée. Dans ce cas, il faut soit modifier la phrase afin de la rendre interprétable, soit ajouter de nouveaux patrons d'interprétations faisant apparaître les constructions syntaxiques de la phrase.
- Plusieurs cas d'utilisations correspondent à la phrase considérée. Dans ce cas, une ambiguïté a été détectée dans les exigences. Ici encore, cette ambiguïté peut être levée, soit en modifiant la phrase, soit en modifiant les patrons d'interprétations.

La seconde phase est l'agrégation des cas d'utilisations élémentaires qui concerne un même service. Dans l'exemple présenté sur la figure 4.34, les deux cas d'utilisations correspondant au exigences (R1) et (R2) concernent la planification de réunion. Au cours de la phase d'agrégation, ces deux cas d'utilisations sont agrégés en un seul qui combine les pré-conditions et les post-conditions de deux. Un fois que l'on a agrégé les différents cas d'utilisations élémentaires correspondant à un service, on obtient un véritable cas d'utilisation pour ce service.

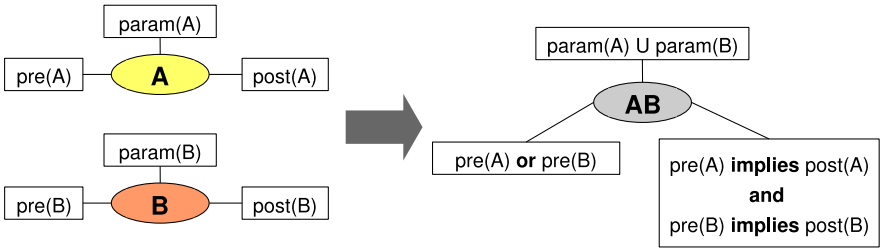


FIG. 4.36 – Agrégation de deux cas d'utilisations.

La figure 4.36 détaille le processus d'agrégation de deux cas d'utilisations. Les paramètres du cas d'utilisation résultat sont l'union des paramètres des cas d'utilisations

agrégés. La pré-condition agrégée est construite comme une disjonction entre les pré-conditions des cas d'utilisations initiaux. Enfin, la post-condition est construite grâce à l'opérateur d'implication afin de maintenir le lien entre les pré-conditions et post-conditions des cas d'utilisations élémentaires.

L'ensemble de ce processus d'analyse sémantique est implanté en Kermeta. Le programme Kermeta correspondant au modèle syntaxique du LDE et permettant, grâce aux patrons d'interprétation, de construire le modèle sémantique comporte environ 5000 lignes de code Kermeta.

4.4.4 Conclusion

Cette étude de cas a permis d'implanter en Kermeta un processus d'ingénierie des modèles complet issue de collaborations avec l'industrie. L'intérêt de cette étude est qu'elle concentre une grande variété d'éléments spécifiques à l'ingénierie des modèles : langages dédiés, transformations, simulations, etc. Pour implanter le processus complet deux langages dédiés ont été définis : le LDE et le langage de cas d'utilisation. La structure de ces langages a été décrite dans des méta-modèles. La sémantique du LDE a été implantée par une transformation de modèles ciblant le langage de cas d'utilisation. La sémantique du langage de cas d'utilisation a été spécifiée en Kermeta dans le méta-modèle de cas d'utilisation. Ceci a permis d'obtenir un simulateur de ce langage. Dans la suite de la chaîne, ce simulateur est utilisé pour produire des cas de tests.

Le résultat de cette étude est que les nouveaux développements, réalisés en collaboration avec France Télécom, autour de cette chaîne de traitement des exigences sont implantés en Kermeta[5]. Les prototypes initiaux implantés en JAVA ont été abondonnés au profit de Kermeta qui fournit un meilleur support pour la manipulation de modèles et donc la création d'applications plus facile à maintenir et à faire évoluer. Le seul maillon de la chaîne de traitement des exigence pour lequel l'environnement Kermeta ne founit pas de solution satisfaisante est la définition de syntaxes concrètes pour les différents langages dédiés. Pour ces éléments le générateur de parseur Java ANTLR est utilisé.

4.5 Conclusion

Comme le montrent les applications décrites dans cette section, Kermeta a été appliqué avec succès à divers travaux dans le domaine de l'ingénierie des modèles. En plus des cas d'applications décrites dans ce chapitre, l'ensemble des techniques de test de transformations de modèles proposées dans le chapitre suivant ont également été prototypées et implantées dans l'environnement Kermeta. En dehors des travaux de l'équipe Triskell, qui développe l'environnement Kermeta, un certain nombre de collaborateurs académiques et industriels évalue aujourd'hui la possibilité d'utiliser Kermeta pour leurs développements (Parmi lequels l'ENSTB, l'ENSIETA, le LIRMM et France Télécom R&D).

[5]Ces travaux sont menés, entre autres, par Erwan Brottier dans le cadre d'une thèse CIFRE commencée en octobre 2005 entre France Télécom R&D et l'équipe Triskell.

A la lumière des travaux présentés dans cette section, un des principaux avantages de Kermeta par rapport aux solutions existantes est de permettre l'intégration forte de sémantique et de contraintes dans les méta-modèles. Cela permet de bénéficier de l'ensemble des techniques de conception, de programmation, de test et de maintenance classiques du génie logiciel lors de la création de langages de modélisation. Les sections de ce chapitre on ainsi abordé la définition d'un langage d'automates, d'un langage de diagrammes de classes, d'un langage de diagrammes de séquences, d'un langage de cas d'utilisations et d'un langage d'exigences. Pour chacun de ces langages, des éléments sémantiques, des contraintes ou des transformations ont été intégrés directement dans le méta-modèle qui lui correspond.

Un des éléments qui fait défaut à l'environnement Kermeta, et sur lequel nous travaillons actellement [MFF+06], est la possibilité de définir des syntaxes textuelles pour les méta-modèles. Dans les cas ou nous avons eu ce besoin, jusqu'a présent la solution est d'utiliser un générateur de parseur tel que ANTLR [ANT]. Le parseur est alors utilisé pour construire un modèle exploitable en Kermeta grâce à Eclipse/EMF. Cette solution a plusieurs défauts et en particulier celui d'obliger à implanter en JAVA la construction des éléments de modèles dans les productions de la grammaire utilisée. Pour éviter cela, nous travaillons sur l'utilisation d'un modèle de syntaxe concrète directement lié au méta-modèle correspondant au langage considéré. Ce modèle de systaxe concrète est exploité par des transformations qui permettent de générer des parseurs et pretty-printers pour les modèles.

Chapitre 5

Test de transformations de modèles

Les transformations de modèles tiennent une place importante dans l'ingénierie dirigée par les modèles. Elles permettent non seulement l'automatisation de certaines activités de développement mais aussi la capitalisation et la réutilisation de savoir-faire. A ce titre, afin d'assurer la qualité du processus de développement, il est nécessaire d'assurer la qualité des transformations de modèles. Dans la communauté de l'ingénierie des modèles, un grand nombre de travaux s'intéressent aux langages et techniques pour le développement de transformations de modèles mais, parmi ces travaux, très peu s'intéressent aux problèmes de la validation.

D'un point de vue pratique, les transformations de modèles sont implantées comme des programmes et peuvent donc être validées comme n'importe quel autre programme. Cependant, les transformations de modèles ont la particularité de manipuler des modèles qui sont eux-mêmes décrits par des méta-modèles. Ce chapitre montre que cette particularité permet de développer des critères de test, des générateurs de données de test, des oracles et des techniques de diagnostic spécifiques aux programmes de transformation de modèles. Ces techniques spécifiques permettent de compléter les outils de validation disponibles pour les programmes. Les deux points que nous développons sont la définition de critères de test "boite noire" et la génération automatique de modèles d'entrée pour les transformations de modèles.

5.1 Validation de transformations de modèles

Cette section définit le contexte et les motivations de ce travail sur la validation de transformation de modèles. La section 5.1.1 revient en détail sur la définition de "transformation de modèles" que nous utilisons. La section 5.1.2 détaille les raisons pour lesquelles il est nécessaire de développer des techniques de test originales pour valider les programmes de transformation de modèles. Enfin, la section 5.1.3 rappelle les différentes étapes du test d'une transformation de modèles et détaille le plan de la suite de ce chapitre.

5.1.1 Définitions

Dans ce travail nous adoptons une définition assez large pour les transformations de modèles et nous ne faisons pas d'hypothèse sur les outils et les langages utilisés pour l'implantation des transformations de modèles. En d'autres termes, nous considérons les transformations de modèles comme des boites noires pour lesquelles les entrées et sorties sont des modèles. La figure 5.1 présente une transformation de modèles et les différents artefacts relatifs à sa définition et à son utilisation.

Un des éléments intéressants, propre aux transformations de modèles, est la présence des méta-modèles d'entrée et de sortie qui permettent de délimiter formellement les espaces d'entrée et de sortie de la transformation. Les pré-conditions et post-conditions, qui sont généralement associées à une transformation, permettent de contraindre les modèles en entrée et en sortie de la transformation par rapport aux méta-modèles utilisés. Comme pour la majorité des programmes, la spécification des transformations de modèles n'est souvent que partiellement formalisée et reste dans la majorité des cas exprimée en langage naturel.

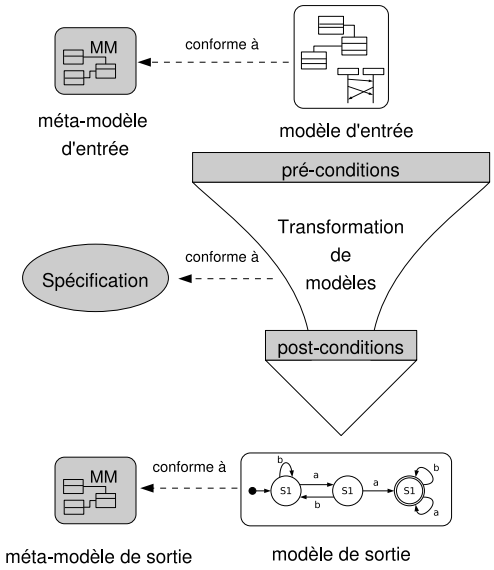


FIG. 5.1 – Une transformation de modèles et ses éléments de spécification.

5.1.2 Motivations

Les transformations de modèles sont des programmes qui ont la particularité de manipuler des modèles. Comme pour tous les programmes, la validation passe par une phase de test rigoureuse au cours de laquelle le programme est exécuté avec des données de test afin de mettre en évidence ses défauts. L'ensemble des techniques de tests existants pour le test de programme [Bin99, Bei90] doit donc povoir être réutilisé.

Il existe cependant au moins deux raisons de considérer le problème particuler du test de transformation de modèles :

- Les techniques de test existantes ne s'appliquent pas toujours de façon satisfaisante aux transformations de modèles car les structures de données manipulées par les transformations de modèles sont complexes. En effet, une grande partie des techniques de test classiques est adaptée à des programmes dont la complexité réside dans les traitements et non dans les structures de données. Dans le cas des transformations de modèles, c'est généralement le cas inverse : les structures de données sont souvent complexes alors que les traitements restent simples.
- Les particularités des transformations de modèles ne sont pas exploitées par les techniques de test classiques. En particulier, les méta-modèles qui offrent une description détaillée des données manipulées par les programmes de transformation ne sont pas exploités par les techniques de test généralistes. A ce titre, il est intéressant de faire évoluer les techniques existantes ou de développer de nouvelles techniques permettant d'exploiter au mieux les éléments propres aux transformations.

5.1.3 Test de transformations de modèles

De façon générale, le test d'une transformation de modèles se déroule de la même manière que le test de n'importe quel programme (cf chapitre 2). La première étape consiste à choisir un ensemble de données de test, c'est-à-dire de modèles d'entrée dans notre cas. Dans la suite nous appellerons *modèles de test* les modèles choisis pour tester une transformation. En pratique, il est impossible de tester exhaustivement une transformation, c'est-à-dire avec tous les modèles en entrée possibles. Il faut donc choisir judicieusement un ensemble de modèles de test, c'est la première difficulté de l'activité de test. Comme le précisent les définitions de la section 2.2.1, on appelle *critère de test* le critère utilisé pour déterminer si un ensemble de données de test est satisfaisant. Le second problème du test est la génération de données de test, c'est-à-dire la construction d'un ensemble de données de test qui vérifie le critère de test choisi.

Une fois que l'on dispose d'un ensemble de modèles de test satisfaisant, la transformation est exécutée avec chacune des données et on vérifie que les sorties produites sont conformes aux sorties attendues. En d'autres termes, il faut vérifier pour chaque modèle de test en entrée que le ou les modèles obtenus en sortie sont conformes à ce que prévoit la spécification de la transformation. La fonction qui détermine si les sorties obtenues sont conformes aux sorties attendues est appelée *oracle*. Dans la plupart des cas, cette fonction d'oracle est très difficile à automatiser lorsque la spécification est exprimée de façon informelle. La solution généralement retenue est coûteuse car elle consiste à produire l'oracle manuellement. Pour éviter un surcoût trop important il faut minimiser le nombre de modèles de test utilisés. Ainsi, lors de la génération des modèles de test, l'objectif est de générer un ensemble de données le plus petit possible permettant de vérifier le critère de test choisi.

Les techniques pour le test de transformation de modèles que nous présentons s'appuient sur l'existence des méta-modèles qui fournissent une description formelle des

données manipulées par les programmes de transformation. La section 5.2 propose un framework permettant d'exprimer des critères de tests et de vérifier qu'un ensemble de modèles en entrée vérifie ces critères de tests. La section 5.3 propose, en s'appuyant sur ce framework, un ensemble de critères de test simple pour le test boite noire de transformations de modèles. La section 5.4 détaille des propositions pour automatiser la génération de modèles satisfaisant un critère donné. Enfin, la section 5.5 conclut ce chapitre sur l'applicabilité des techniques proposées et sur les travaux nécessaires à l'obtention d'un processus de test complet pour les programmes de transformations de modèles.

5.2 Un framework pour la sélection de modèles de test

Dans le cadre de ce travail, nous ne faisons pas d'hypothèses sur les techniques d'implantation employées pour les transformations de modèles. Les seuls éléments sur lesquels nous pouvons nous appuyer pour le test sont les éléments de spécification des transformations de modèles. Ces éléments, représentés en gris sur la figure 5.1, incluent les méta-modèles d'entrée et de sortie, une pré-condition, une post-condition et une description de la transformation. La description de la transformation, et donc la spécification de son comportement, sont généralement informelles et donc difficiles à exploiter. Par contre, les méta-modèles sont des éléments de spécification formels, propres aux transformations de modèles, qui offrent une description détaillée des structures de données manipulées par la transformation de modèles. L'idée que nous proposons est d'exploiter ces méta-modèles pour la définition de critères de test. Sur la base de cette idée, cette section détaille un framework permettant de définir des critères de test structurels pour les transformations de modèles et de vérifier si un ensemble de modèles satisfait ces critères.

5.2.1 Exemple et intuition

Afin d'illustrer les discussions de ce chapitre, nous utilisons l'exemple d'une transformation de modèles permettant de "mettre à plat" des machines à états hiérarchiques. Par la suite, nous ferons référence à cette transformation sous le nom *SMFlatten*. La figure 5.2 présente l'application de cette transformation sur un exemple simple. Au cours de la transformation, les états composites sont supprimés et leurs transitions entrantes et sortantes sont distribuées sur leurs états internes.

Le méta-modèle d'entrée (et de sortie) de cette transformation est présenté figure 5.3. Selon ce méta-modèle, une machine à états est simplement composée d'un ensemble d'états, d'états composites et de transitions. Chaque état porte un numéro (propriété *label*) et chaque transition porte un évènement (propriété *event*). Grâce aux propriétés *isInitial* et *isFinal* chaque état peut être marqué comme respectivement initial et final.

Pour valider la transformation *SMFlatten*, il est nécessaire de choisir judicieusement un ensemble de machines à états hiérarchiques, d'exécuter la transformation et de vérifier pour chacune des machines à états que le résultat obtenu est conforme au résultat attendu. Il est intéressant de noter que, comme dans la plupart des cas, la

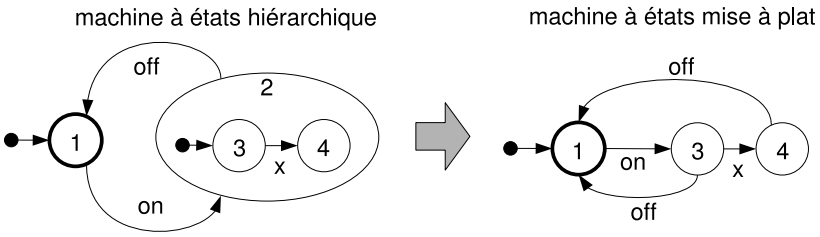


FIG. 5.2 – Mise à plat d'une machine à états hiérarchique.

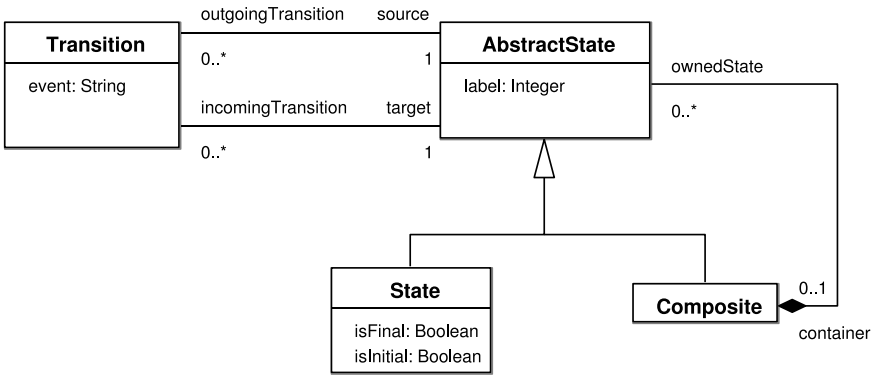


FIG. 5.3 – Méta-modèle de machines à état composites.

fonction d'oracle, qui permet de vérifier si les machines à état obtenues sont conformes à ce qui est attendu, n'est pas triviale et coûteuse à produire. En effet, le fait de décider que deux automates sont équivalents nécessite de les réduire à une forme canonique pour les comparer, ce qui est au moins aussi complexe que la transformation que l'on cherche à valider. Pour faciliter le calul de l'oracle, il est donc important, en amont, de minimiser le nombre et la complexité des machines à états hiérarchiques que l'on va utiliser pour le test.

Étant donné le méta-modèle proposé figure 5.3, on attend intuitivement quatre caractéristiques des machines à état utilisées pour le test :

- Toutes les classes concrètes du méta-modèle doivent être instanciées au moins une fois. Les machines à états de test doivent contenir des états simples, des états composites et des transitions.
- Les valeurs des attributs doivent être variées. Dans les différentes machines à états utilisées pour le test, il faut par exemple qu'il existe à la fois des états marqués comme initiaux et d'autres qui ne le soient pas. En ce qui concerne les transitions, la chaîne spécifiant l'évènement (propriété *event*) doit parfois être vide ou nulle afin de vérifier que la transformation fonctionne correctement dans ces cas limites.
- Toutes les associations doivent être instanciées dans différents cas de figures. S'il existe une transition dans une machine à états, elle doit être liée à un état source et un état cible car les multiplicités des propriétés *source* et *target* sont 1..1. En revanche, chaque état peut avoir un nombre arbitraire (éventuellement nul) de transitions entrantes et sortantes car les multiplicités pour les deux propriétés correspondantes sont $1..*$. Pour chacune de ces propriétés, il est souhaitable de tester la transformation avec différentes multiplicités : des états sans transition sortante, des états avec une seule transition sortante et des états avec plusieurs transitions sortantes.
- Les différentes valeurs d'attributs et les différentes multiplicités des associations doivent être combinées entre elles. Dans les deux points précédents, chaque propriété est considérée individuellement mais en pratique, les combinaisons de valeurs pour différentes propriétés sont intéressantes. Par exemple, il est souhaitable que dans les machines à états de test on trouve des états à la fois initiaux et finaux ou encore des états à la fois initiaux et comportant des transitions sortantes.

L'objectif des sous-sections suivantes est de formaliser ces intuitions et de proposer un framework qui permette de mettre en oeuvre les critères de test correspondants. Parmi les quatre points proposés précédemment, le premier est suffisement simple pour être directement utilisable : il suffit de vérifier que toutes les classes concrètes sont instanciées dans les modèles de test. Les second et troisième points doivent être clarifiés afin de formaliser ce que l'on entend par "valeurs variées" pour les attributs et "différents cas de figure" pour les associations. La section 5.2.2 propose d'adapter les techniques de test par partition pour permettre de sélectionner des valeurs et multiplicités pertinentes. Enfin, en ce qui concerne le dernier point qui consiste à combiner les valeurs et multiplicités de différentes propriétés, il est nécessaire de définir selon quels critères les combinaisons sont construites. La section 5.2.3 propose un framework permettant de représenter ces combinaisons.

5.2.2 Valeurs et multiplicité des propriétés

Comme le montre l'exemple de la section précédente, afin de choisir un bon ensemble de modèles de test, l'idée est de couvrir au mieux l'ensemble des structures décrites par le méta-modèle. A ce titre, une des difficultés est de s'assurer que chaque attribut dans le méta-modèle est instancié avec un ensemble pertinent de valeurs et que chaque association est instanciée avec un ensemble pertinent de multiplicités. Que ce soit pour les attributs ou les associations, il est souvent impossible de créer des modèles de test contenant toutes les valeurs ou toutes les multiplicités possibles, il faut donc choisir judicieusement un nombre restreint de valeurs ou de multiplicités le plus représentatif possible de l'ensemble des possibilités. Le fait de réduire ainsi un domaine de valeurs trop vaste pour être exploré complètement est une technique de test classique appelée test par partitions [OB88] (cf. section 2.2.6 de l'état de l'art). Par le passé, cette idée a été adaptée par France et al. [GFB+03, AFGC03] pour le test de modèles UML et nous l'adaptons ici pour le test de transformations de modèles.

Test par partitions appliqué aux transformations de modèles

Transition::**event**	**{''}, {'evt1'}, {'.+'}**
Transition::**#source**	**{1}**
Transition::**#target**	**{1}**
AbstractState::**label**	**{0}, {1}, {2..MaxInt}**
AbstractState::**#container**	**{0}, {1}**
AbstractState::**#incomingTransition**	**{0}, {1}, {2..MaxInt}**
AbstractState::**#outgoingTransition**	**{0}, {1}, {2..MaxInt}**
State:**isInitial**	**{true}, {false}**
State::**isFinal**	**{true}, {false}**
Composite::**#ownedState**	**{0}, {1}, {2..MaxInt}**

FIG. 5.4 – Partitions pour le méta-modèle de machines à états de la figures 5.3.

Les données d'entrée des transformations de modèles étant relativement complexes, il n'est pas possible de partitionner directement le domaine d'entrée d'une transformation de modèle (l'ensemble des modèles conformes à un méta-modèle). Par contre, la technique de partitionnement permet de définir des classes d'équivalences pour les valeurs et les multiplicités de chaque propriété du méta-modèle d'entrée. La figure 5.4 représente ces classes d'équivalence pour le méta-modèle de machines à états de la figure 5.3. Ces classe d'équivalences ont été obtenues en utilisant une stratégie de partitionnement pas défaut.

Un ensemble de classes d'équivalence est associé à chaque propriété du méta-modèle dont le type est simple. Lors de l'évaluation d'un ensemble de modèles de test l'idée est que les valeurs prises par un attribut couvrent les différentes classes d'équivalence. Sur l'exemple, les classes d'équivalence '', 'evt1' et '.+' ont été associées à la propriété *event* de la classe *Transition*. Cela permet d'assurer que dans les modèles de test il

faudra des transitions dont l'événement est vide, des transitions dont l'évènement est la valeur particulière 'evt1' (choisie arbitrairement) et des transitions dont la valeur est une chaîne non vide.

De la même manière, pour les propriétés dont la multiplicité est supérieure à 1, on définit des classes d'équivalence pour les multiplicités. Par exemple, pour la propriété *outgoingTransition* de la classe *AbstractState* on a défini les classes d'équivalence $\{0\}$, $\{1\}$ et $[2, +\infty]$ qui signifient que l'on souhaite des modèles de test qui contiennent des états sans transition sortante, des états avec une transition sortante et enfin des états avec plusieurs transitions sortantes.

Type	**Partition par défaut**
Boolean	{true}, {false}
Naturel	{0}, {1}, {2..MaxInt}
Entier	{MinInt..-2}, {-1}, {0}, {1}, {2..MaxInt}
Chaîne	{null}, {''}, {'.+'}
Enumération	chaque littéral

FIG. 5.5 – Partitionnement par défaut pour les principaux types de données.

La figure 5.5 présente les partitions pour les principaux types de données. Pour les types simples tel que les booléens, les entiers ou les chaînes, les partitions conçues pour que, lors du test, les attributs prennent des valeurs limites telle que la chaîne vide, une chaîne de caractères et des valeurs remarquables. En ce qui concerne les types énumérés, la politique par défaut est que chaque littéral constitue sa propre classe d'équivalence. Cette politique assure que tous les littéraux soient représentés dans les modèles de tests.

En pratique, le partitionnement par défaut permet de déterminer automatiquement un ensemble de classes d'équivalence pour les valeurs de chaque propriété du méta-modèle d'entrée d'une transformation. En ce qui concerne les associations, comme le montre l'exemple de la figure 5.4, l'idée est de partitionner leurs multiplicités afin d'assurer qu'elles soient instanciées avec un ensemble de multiplicité judicieux dans les modèles de test.

Partitionnement sémantique

Comme pour le test de programme, dans certains cas il est nécessaire de compléter le partitionnement par défaut en utilisant un partitionnement sémantique, c'est-à-dire non seulement basé sur le méta-modèle mais aussi sur la transformation à tester. C'est le cas, par exemple, pour la transformation présentée figure 5.6 qui permet de transformer un ensemble de classes marquées comme persistantes en un ensemble de tables correspondant. Le méta-modèle d'entrée de la transformation, présenté à gauche de la figure, est très simple : chaque classe a un nom, un tag et un ensemble d'attributs.

La transformation consiste a créer des tables à partir des classes pour lesquelles la valeur de l'attribut *tag* est la chaîne "persistant". Si l'on s'en tient au partitionnement

par défaut de la figure 5.5 pour les chaînes de caractères, le fait que la valeur "persistant" pour cet attribut joue un rôle de premier plan pour cette transformation n'apparaît pas. Pour assurer que cette valeur soit correctement couverte lors du test, il est nécessaire d'enrichir la partition par défaut en isolant la valeur "persistant" dans une nouvelle classe d'équivalence.

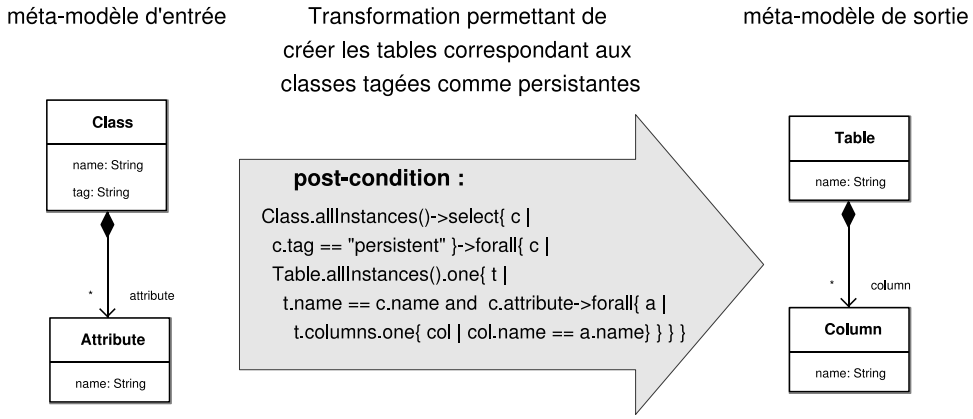


FIG. 5.6 – Transformation de classes en tables très simple.

Pour le test de transformations, les cas où il est nécessaire de prendre en compte des valeurs particulières, comme ici pour la valeur "persistant", sont relativement rares et ne concernent que des ensembles restreints de valeurs. En effet, les méta-modèles sont généralement conçus pour capturer au mieux la structure des modèles qu'ils permettent de créer. En d'autres termes, dans le cas de l'exemple précédent, un bon méta-modèle d'entrée devrait capturer clairement la notion de persistance afin d'éviter le flou induit par cet attribut *tag* de type chaîne de caractère. On pourrait par exemple avoir un attribut *isPersistant* de type Booléen ou encore une attribut *nature* permettant de choisir la nature d'une classe dans une énumération de possibilités. Cependant, l'exemple précédent est intéressant dans la mesure où l'on retrouve dans les méta-modèles, et en particuler les méta-modèles standardisés, des attributs génériques qui, comme ici l'attribut *tag* permettent de représenter des éléments non prévus lors de la conception du méta-modèle. Dans le contexte d'UML il s'agit par exemple des stéréotypes et des annotations.

Lorsqu'il est nécessaire de faire appel à un partitionnement sémantique, plusieurs stratégies peuvent être envisagées pour permettre de créer les classes d'équivalences. La stratégie la plus simple est de demander au testeur, à l'issue du partitionnement par défaut réalisé automatiquement, de raffiner les partitions obtenues grâce à sa connaissance de la transformation à tester. Cela ne représente pas un effort important dans la mesure où le nombre d'attributs du méta-modèle d'entrée d'une transformation nécessitant un partitionnement spécifique est généralement très faible. Cependant, afin de faciliter la tâche du testeur et d'améliorer la qualité des partitions obtenues, il est possible d'utiliser une stratégie semi-automatique basée sur l'analyse statique de la spé-

cification de la transformation à tester. Le principe, déjà mis en oeuvre dans le cadre du test de programme [XXXREF ?], consiste à chercher par une analyse statique les valeurs particulières avec lesquelles telle ou telle variable ou attribut est comparé.

Dans le cas de l'exemple permettant de transformer des classes en tables (figure 5.6) une analyse statique simple de la post-condition de la transformation rend compte du fait que l'attribut *tag* est comparé à la valeur "persistant" et permet donc d'enrichir de cette valeur la partition associée à cet attribut. Dans la plupart des cas, il sera difficile d'automatiser complètement la phase de partitionnement sémantique car la spécification des transformations est, la plupart du temps, partiellement en langage naturel.

5.2.3 Combinaisons de valeurs : fragments d'objets et de modèles

La section précédente s'attache à définir des domaines de valeurs et de multiplicités pertinents pour chacun des attributs du méta-modèle d'entrée d'une transformation. Dans ce contexte, chaque attribut du méta-modèle est traité indépendamment des autres car l'objectif de cette phase de partitionnement est d'assurer la couverture individuelle du domaine de valeur de chaque attribut. Cependant, un méta-modèle est plus qu'une simple juxtaposition d'attributs et le fait de couvrir les attributs indépendamment les uns des autres n'est pas suffisant pour obtenir un bon ensemble de modèles pour le test d'une transformation de modèles. En effet, dans un modèle, les valeurs des attributs sont combinées dans des objets et des structures d'objets complexes. Cette section définit un formalisme permettant d'exprimer ces combinaisons et un framework permettant de vérifier si un ensemble de modèles présente des structures et combinaisons de valeurs attendues. Ce framework permet de représenter facilement ce que l'on attend d'un ensemble de modèles utilisés pour tester une transformation. La section suivante utilise ce framework pour définir un ensemble de critères de test spécifiques aux transformations de modèles.

Exemple et discussion

Reprenons l'exemple de la transformation *SMFlatten* introduit dans la section précédente. A l'issue de la phase de partitionnement nous avons obtenu les partitions de la figure 5.4 pour chacun des attributs du méta-modèle de la figure 5.3. Ces partitions permettent de spécifier que les modèles de test doivent contenir des représentants de chacune des classes d'équivalence. Par exemple, dans le cas de la propriété *ownedState* de la classe *Composite*, la partition sur les multiplicités impose que l'on teste la transformation avec des modèles contenant des états composites vides, des états composites contenant un seul état et des états composites contenant plusieurs états. La figure 5.7 montre trois modèles qui permettent de satisfaire cette exigence. Ces trois modèles sont intuitivement insuffisants pour valider la transformation. En effet, lors de la mise à plat des machines à états, les transitions entrantes et sortantes des états composites sont distribuées sur leurs états internes. Ce qui est donc intéressant lors du test est de vérifier que quelque soit le nombre d'états internes (0, 1 ou plusieurs) et quelque soit

le nombre de transitions entrantes et sortantes, la transformation fonctionne correctement. On souhaite donc voir apparaître dans les modèles de test des combinaisons de choix de valeurs pour les attributs *ownedState*, *incomingTransition* et *outgoingTransition*. Il faudra donc par exemple avoir des modèles de test contenant un état composite avec plusieurs transitions entrantes, aucune transition sortante et un seul état interne.

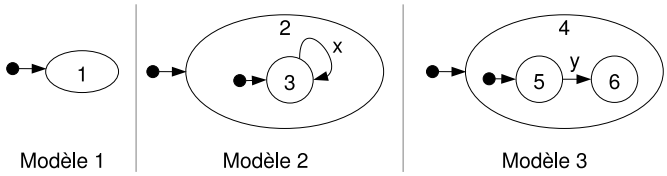


FIG. 5.7 – Modèles de test permettant de couvrir la partition de la propriété *Composite : :ownedState*.

L'exemple précédent met en évidence le fait qu'il est nécessaire de combiner les contraintes relatives à plusieurs attributs du méta-modèle d'entrée pour construire des données de test pertinentes. Dans ces conditions, la solution la plus simple consiste à recommander de réaliser toutes les combinaisons de valeurs possibles des attributs du méta-modèle dans les modèles de test. Cependant cette approche est insatisfaisante pour deux raisons :

- Premièrement, le fait de combiner les valeurs de tous les attributs du méta-modèle d'entrée conduit à un trop grand nombre de combinaisons. Sur l'exemple du méta-modèle de machines à état, étant donné les partitions de la figure 5.4, la combinaison des classes d'équivalence pour les valeurs et les multiplicités de chaque attribut imposerait de faire apparaître 1944 (3*1*1*3*2*3*3*2*2*3) combinaisons de valeurs dans les modèles de test. Indépendamment du fait que ce nombre soit très grand au vue de la taille et de la simplicité du méta-modèle considéré, il est intuitivement évident qu'un certain nombre de combinaison sont bien moins importantes que d'autres pour le test. Au vue de la transformation à tester, il est intuitivement beaucoup moins important de combiner les différentes valeurs possibles pour le *label* d'un état et le nombre de transitions qui entre et sort de cet état. En pratique il est donc nécessaire de bien choisir les attributs dont on souhaite combiner les valeurs afin d'éviter une explosion combinatoire inutile.
- Deuxièmement, le fait de combiner les valeurs de tous les attributs n'est pas nécessairement suffisant pour obtenir de bonnes données de test. En effet, le fait d'imposer une contrainte pour chacun des attributs du méta-modèle ne permet pas de combiner les valeurs des attributs de plusieurs instances d'une même classe dans un modèle de test. Sur l'exemple du méta-modèle de machine à états, aucune des 1944 combinaisons obtenues à partir des partitions de la figure 5.4 n'impose qu'un modèle de test ne contienne plus d'un état composite. Pour valider la transformation permettant de mettre à plat des machines à état il est pourtant intuitivement clair qu'il faudrait des machines à états contenant plusieurs états composites. Pour choisir un bon ensemble de modèles de test il est donc nécessaire de pouvoir exprimer des contraintes portant sur plusieurs objets d'un modèle

d'entrée.

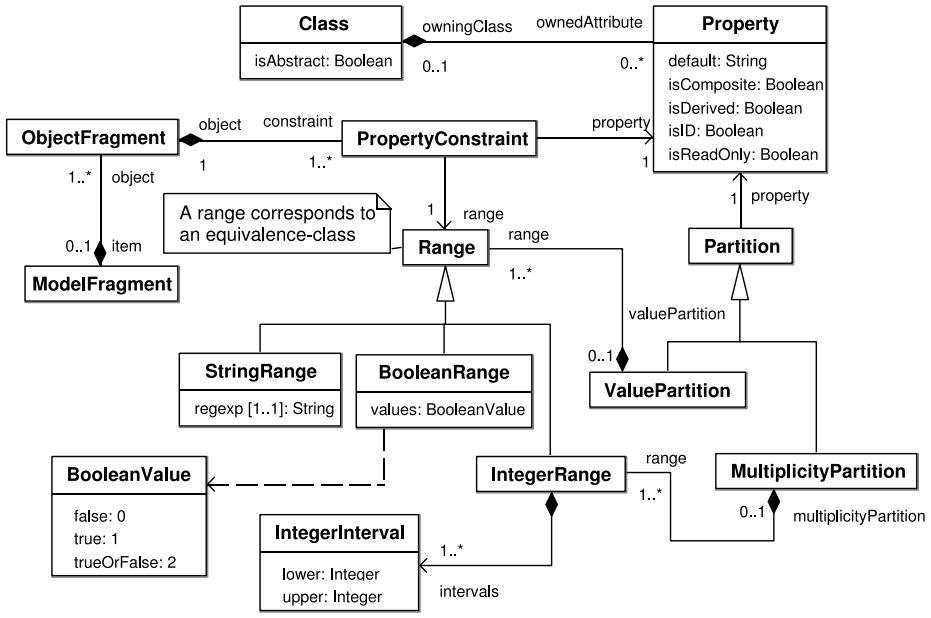


FIG. 5.8 – Méta-modèle de partitions et de fragments de modèle.

Un framework pour représenter les combinaisons de valeurs

A partir de ces deux constatations, nous proposons un framework permettant d'exprimer simplement un ensemble de combinaisons et de vérifier qu'un ensemble de modèles présente les combinaisons de valeurs attendues. Le méta-modèle de la figure 5.8 présente la structure de ce framework. On retrouve dans ce méta-modèle un ensemble de classes permettant de représenter les partitions associées à chaque propriété d'un méta-modèle. Les classe *Class* et *Property* représentées sur le diagramme sont issues du méta-méta-modèle utilisé (EMOF dans notre cas). La classe abstraite *Partition* fait référence à une *Property*. Les classes concrètes *ValuePartition* et *MultiplicityPartition* permettent de représenter respectivement les partitions sur la valeur et sur la multiplicité d'une propriété. Chaque partition est constituée d'un ensemble de classes d'équivalences représentées par la classe abstraite *Range*. Suivant le type de la propriété à laquelle se rapporte une partition, les classes concrètes *BooleanRange*, *IntegerRange* et *StringRange* permettent de représenter les classes d'équivalences.

La figure 5.9 présente un sous-ensemble des partitions définies pour le méta-modèle de machines à états (figure 5.4) sous forme de diagramme d'objets. La partie gauche de la figure représente des objets appartenant au méta-modèle de machines à état : les classes *State* et *Transition* et une propriété appartenant à chacune de ces classes. La droite de la figure représente les instances du méta-modèle de partitions de la figure 5.8 permettant de représenter la partition sur la valeur de la propriété *event* de la classe

Transition et la partition sur la multiplicité de la propriété *incomingTransition* de la classe *State*. De la même manière que pour les partitions de ces deux propriétés, le méta-modèle de partition de la figure 5.8 permet de modéliser le résultat du partitionnement du méta-modèle d'entrée de toute transformation.

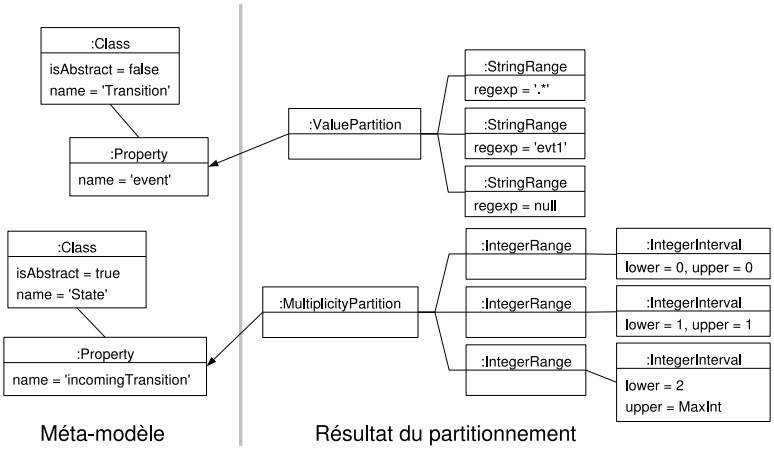


FIG. 5.9 – Modèle de partitionnement conforme au méta-modèle de la figure 5.8.

Une fois les partitions définies pour l'ensemble des propriétés du méta-modèle d'entrée d'une transformation, l'intérêt du méta-modèle de la figure 5.8 est de permettre, au travers des notions de *PropertyConstraint*, *ObjectFragment* et *ModelFragment*, la définition précise des combinaisons de valeurs pertinentes pour le test. Pour chaque combinaison de valeurs que l'on souhaite voir apparaître dans les modèles de test, il faut définir un fragment de modèle. Un fragment de modèle est composé d'un ensemble de fragments d'objets et chaque fragment d'objet se compose d'un ensemble de contraintes portant sur les valeurs de ses propriétés. Chaque contrainte (*PropertyConstraint*) associe une propriété du méta-modèle d'entrée de la transformation avec une classe d'équivalence (*Range*) appartenant à la partition associée à cette propriété. Pour qu'un objet dans un modèle soit conforme à un fragment d'objet il faut que cet objet présente des valeurs appropriées pour chacune des propriétés qui sont contraintes par le fragment d'objet. En d'autres termes, il faut que les valeurs de propriété de l'objet appartiennent aux classes d'équivalences associées par les contraintes appartenant au fragment d'objet. Enfin pour qu'un modèle soit conforme à un fragment de modèle, il faut que ce modèle contienne au moins un objet conforme à chacun des fragments d'objets contenu dans le fragment de modèle.

La figure 5.10 présente un exemple de fragment de modèle. Ce fragment de modèle comporte deux fragments d'objets : un fragment qui correspond à une instance de la classe *CompositeState* et un fragment qui correspond à une instance de la classe *State*. Chacun de ces fragments d'objets comporte plusieurs contraintes portant sur leurs propriétés. L'interprétation de ce fragment de modèle est que l'on souhaite que les modèles de test intègrent au moins un modèle qui contienne à la fois :

– Un état composite contenant plusieurs états internes et une seule transition entrante.
– Une état numéroté 0 et comportant une transition entrante et plusieurs transitions sortantes.

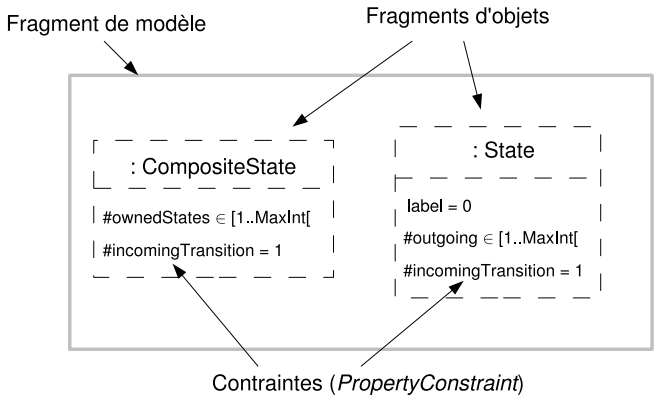


FIG. 5.10 – Exemple de fragment de modèle.

Afin de représenter facilement des fragments de modèles, dans la suite nous utilisons une notation textuelle. Chaque fragment de modèle est noté par $MF\{of_1, ..., of_n\}$ où les of_i correspondent au fragments d'objets. Chaque fragment d'objet est noté par $NomDeClasse(c_1, ..., c_n)$ où les c_i sont les contraintes portant le les propriétés de l'objet. Le fragment de modèle repésenté sur la figure 5.10 est donc noté :

$$MF\{CompositeState(\sharp ownedStates > 0, \sharp inTrans. = 1),$$
$$State(label = 0, \sharp outTrans. > 0, \sharp inTrans. = 1)\}$$

Le formalisme des fragments d'objets et de modèles permet d'exprimer formellement les combinaisons de valeurs et les structures d'objets qui doivent apparaître dans les modèles de tests. Cependant, le problème de créer de bons ensembles de fragments de modèles demeure. En effet, lors de la création des fragments des modèles la pertinence des combinaisons de valeurs choisies est déterminante. Dans cette optique, la section 5.3 propose un ensemble de critères permettant d'obtenir automatiquement des ensembles de fragments de modèles. Une fois qu'un ensemble de fragments de modèles est constitué, la sous-section suivant détaille un processus itératif permettant de construire un ensemble de modèles de test.

5.2.4 Sélection des modèles de tests

La Figure 5.11 présente le processus de sélection de modèles de tests associé au framework décrit dans la section précédente. L'intérêt de ce processus est qu'il permet non seulement de qualifier un ensemble de modèles de test mais aussi de l'améliorer. On dispose initialement du méta-modèle d'entrée de la transformation de modèle à

tester et un ensemble initial de modèles de test (en blanc sur la figure). Le processus se décompose en quatre phases successives :

- La première phase du processus (1) est le partitionnement. Un modèle de partition conforme au méta-modèle de partition est construit à partir du méta-modèle d'entrée de la transformation. Une première version du modèle de partition peut être construit automatiquement en utilisant les politiques de partitionnement par défaut détaillées dans la section précédente. Par la suite, et si cela est nécessaire, le testeur peut enrichir les partitions par défaut.
- La seconde phase (2) est la création des fragments de modèles. Un ensemble de fragments de modèles est construit pour représenter les combinaisons de valeur et multiplicités qui devront apparaître dans les modèles de tests. Dans la suite de ce chapitre (Section 5.3) nous proposons un ensemble de stratégies pour créer automatiquement des ensembles de fragments de modèle.
- La troisième phase (3) consiste à vérifier que les modèles de test couvrent les fragments de modèles construits. Pour chaque fragment de modèle, il faut rechercher un modèle dans l'ensemble des modèles de test qui contienne des objets correspondant à chaque fragment d'objet du fragment de modèle. A l'issue de cette phase, l'ensemble de fragment de modèle est divisé en deux : d'un coté les fragments de modèle couverts par les modèles de test et de l'autre les fragments de modèle non-couverts.
- La quatrième phase (4) consiste à améliorer l'ensemble de modèles de test. L'ensemble de fragments de modèles qui n'ont pas été couverts par les modèles de tests fournit une information précieuse au testeur pour améliorer son ensemble de modèles de test.

Les phases (3) et (4) du processus doivent être répétées itérativement jusqu'à ce que tous les fragments de modèles soit couverts par au moins un modèle de test.

La suite de ce chapitre s'intéresse plus particulièrement aux étapes 2 et 4 de ce processus. La section 5.3 propose un ensemble de critères pour la création des fragments de modèles et la section 5.4 discute de l'automatisation de la phase d'amélioration des modèles de test.

5.3 Exemple de critères de test

Comme il a été discuté précédemment, le problème de construire un bon ensemble de fragments de modèles ne peut être résolu au moyen de stratégies simples. En effet, le fait de couvrir les valeurs des propriétés indépendamment les unes des autres n'est pas suffisant et l'idée d'exiger toutes les combinaisons possibles est déraisonnable. Dans ce contexte, il faut trouver un compromis entre ces deux extrêmes afin d'assurer la sélection de modèles de test de qualité à partir d'un nombre raisonnable de fragments de modèles.

Dans le cas général, trouver des combinaisons d'objets et de valeurs pertinentes est difficile sans une bonne connaissance de la transformation à tester. Pourtant, il reste dans le méta-modèle d'entrée de la transformation une quantité d'informations sur la

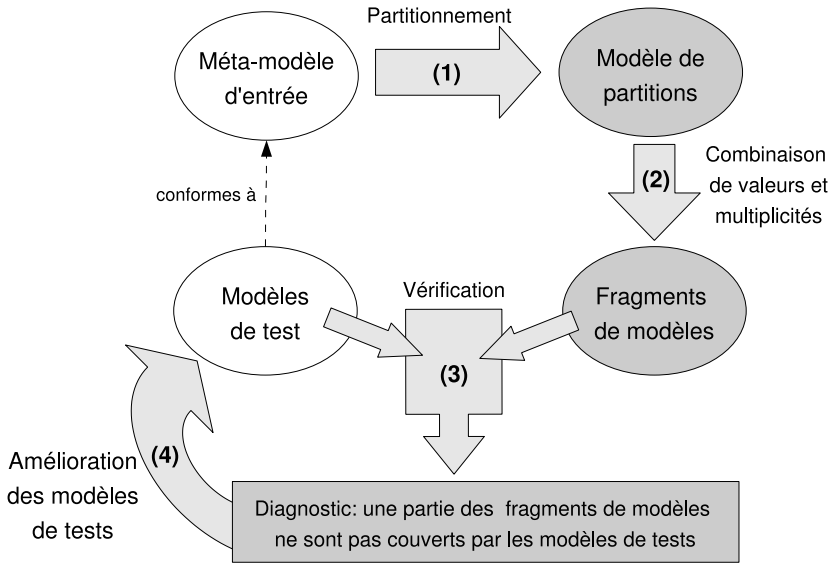


FIG. 5.11 – Processus de sélection des modèles de tests.

structuration de son domaine d'entrée que nous n'avons pas exploité jusque là. En effet, un méta-modèle est plus qu'une simple juxtaposition de propriétés : les propriétés sont encapsulées dans des classes qui ont entre elles des relations tels que l'héritage par exemple. Dans cette section nous proposons d'utiliser ces informations pour définir des critères pour la création de fragments des modèles. Comme le montre le processus présenté dans la section précédente, de tels critères ont un impact direct sur la sélection des modèles de test : ce sont des *critères de test*.

Pour être en accord avec les conclusions des sections précédentes, chacun des critères que nous proposons assure au moins la couverture des classes du méta-modèle d'entrée et la couverture des partitions associées à chacune de ses propriétés.

Couverture des classes : Chaque classe concrète du méta-modèle d'entrée doit être utilisée au moins une fois dans les fragments de modèles. Cette contrainte a pour but d'assurer que les modèles de test incluent au moins une instance de chaque classe concrète du méta-modèle d'entrée.

Couverture des partitions : Toutes les classes d'équivalences pour toute les partitions associées aux propriétés du méta-modèle d'entrée doivent être utilisées au moins dans un fragment de modèle. Cette contrainte permet d'assurer, comme le préconise les techniques de test par partitions, que l'on ait choisi au moins une valeur dans chacune des classes d'équivalences définies.

La figure 5.12 présente une formalisation en OCL de ces deux contraintes.

Couverture des classes

```
Class.allInst.forAll{ C |
 ModelFragment.allInst.exists{ MF |
  MF.object.exists{ OF |
   OF.constraint.exists{ PC |
    PC.property.owningClass == C
}}}}
```

Couverture des partitions

```
Partition.allInst.forAll{ P |
  P.range.forAll{ R |
   ModelFragment.allInst.exists{ MF |
    MF.object.exists{ OF |
     OF.constraint.exists{ PC |
      PC.range == R
}}}}}
```

FIG. 5.12 – Couverture des classes et des partitions en OCL.

5.3.1 Critères simples

Pour commencer, cette section définit deux critères simples qui permettent d'assurer la couverture des classes et des partitions en combinant les contraintes de deux manières différentes. Le premier critère, appelé *AllRanges*, n'ajoute aucune contrainte par rapport à la couverture des partitions définies dans la section précédente. Le second critère, appelé *AllPartition*, est un peu plus fort. L'idée de ce critère est d'assurer que des combinaisons de valeurs pour une même propriété se retrouvent dans les modèles de test. Il impose pour cela que pour chaque propriété, il existe un modèle de test qui couvre la partition associée à cette propriété. La figure 5.13 formalise ces deux critères en OCL.

Critère *AllRanges*

```
Partition.allInst.forAll{ P |
 P.range.forAll{ R |
  ModelFragment.allInst.exists{ MF |
   MF.object.size == 1
   MF.object.one.constraint.size == 1
   MF.object.one.constraint.one.range == R
}}}
```

Critère *AllPartitions*

```
Partition.allInst.forAll{ P |
 ModelFragment.exists{ MF |
  P.range.forAll{ R |
   MF.object.exists{ OF |
    OF.constraint.size == 1
    OF.constraint.one.range == R
}}}}
```

FIG. 5.13 – Critères de couverture des *Ranges* et des *Partitions*.

Afin d'illustrer ces critères, les figures 5.14 et 5.15 présentent une partie des fragments d'objets engendrés par ces deux critères pour l'exemple des machines à états. Les partitions considérées pour créer ces fragments de modèles sont celles données à titre d'exemple sur la figure 5.4. Pour chacun des critères seuls les fragments de modèles relatifs aux propriétés des classes *Transition* et *AbstractState* sont représentés. Dans le cas du critère *AllRanges* (Figures 5.14) chaque fragment de modèle est constitué d'un unique fragment d'objet qui ne porte qu'une seule contrainte. Dans le cas du critère *AllPartitions* (Figure 5.15), un fragment de modèle est défini pour chaque propriété du méta-modèle. Ce fragment de modèle contient un fragment d'objet par classe d'équivalence de la partition associée à la propriété considérée.

Dans la pratique, il probable que ces critères soient insuffisants par assurer la sélection de modèles de test pertinents. Cependant leur utilisation peut permettre de créer un premier ensemble de fragments de modèle qui doit être complété soit par le testeur

$MF\{Transition(event = "")\}$,
$MF\{Transition(event = "evt1")\}$,
$MF\{Transition(event \in .+)\}$,
$MF\{Transition(\sharp source = 1)\}$,
$MF\{Transition(\sharp target = 1)\}$,
$MF\{AbstractState(label = 0)\}$,
$MF\{AbstractState(label = 1)\}$,
$MF\{AbstractState(label \geq 2)\}$,
...

FIG. 5.14 – Exemple de fragments de modèles pour le critère *AllRanges*.

$MF\{Transition(event = ""), Transition(event = "evt1"),$
$Transition(event \in .+)\}$,
$MF\{Transition(\sharp source = 1)\}$,
$MF\{Transition(\sharp target = 1)\}$,
$MF\{AbstractState(label = 0), State(label = 1), State(label \geq 2)\}$,
...

FIG. 5.15 – Exemple de fragments de modèles pour le critère *AllPartitions*.

soit en utilisant d'autres critères. Les sections suivantes présentent des critères plus forts.

5.3.2 Combinaisons classe par classe

Dans les méta-modèles, les propriétés sont encapsulées dans des classes. En raison de cette structuration, et étant donné la façon dont les méta-modèles sont conçus, il est naturel que les propriétés d'une même classe aient des relations sémantiques plus fortes que les propriétés de deux classes différentes. Afin de tirer partie de cette constatation, dans cette section nous proposons quatre critères qui consistent à combiner classe par classe les valeurs et les multiplicités des propriétés. Ces critères diffèrent d'une part par le nombre de combinaisons qu'ils requièrent et d'autre part par la façon dont ces combinaisons sont groupées dans les fragments de modèles.

Le première chose à faire est de définir comment sont choisies les combinaisons de valeurs qui devront être couvertes par les modèles de test. La figure 5.16 présente deux stratégies possibles. La première (*OneRangeCombination*) consiste à créer un ensemble de combinaisons pour les propriétés d'une classe qui permette de couvrir au moins une fois chaque classe d'équivalence des partitions qui lui sont associées. La seconde (*AllRangesCombination*) est plus forte car elle consiste à créer toute les combinaisons possibles de valeurs et multiplicités pour les propriétés d'une classe.

La définition de la stratégie *AllRangesCombination* sur la figure 5.16 utilise une opération *getCombinations* (ligne 2) qui permet de calculer l'ensemble des combinaisons de classe d'équivalences possibles à partir d'un ensemble de partitions. La signature de cette opération est :

```
getCombinations(partitions : Sequence<Partition>) : Set<Sequence<Range>>
```

A titre d'exemple, si l'on considère les partitions $P = \{P_1, P_2\}$ et $Q = \{Q_1, Q_2, Q_3\}$ le résultat de l'opération *getCombinations* pour la sequence de partitions $[P, Q]$ est $\{[P_1, Q_1], [P_1, Q_2], [P_1, Q_3], [P_2, Q_1], [P_2, Q_2], [P_2, Q_3]\}$.

OneRangeCombination

```
Range.allInst.forAll{ R |
 ObjectFragment.allInst.exist{ OF |
  OF.constraint.exists{ PC |
   PC.range = R
}}}
```

AllRangesCombination

```
Class.allInst.forAll{ C |
 getCombinations(
        Partition.allInst.select{ P |
  C.ownedAttribute.contains(P.property)}
 ).forAll{ RSet |
   ObjectFragment.allInst.exists{ OF |
    RSet.forAll{ R |
     OF.constraint.exists{ PC |
      PC.range == R and
      PC.property == R.partition.property
}}}}}}
```

FIG. 5.16 – Deux stratégies de création des combinaisons.

OneMFPerClass

```
Class.allInst.forAll{ C |
 ModelFragment.allInst.select{ MF |
  MF.object.forAll{ OF |
   C.ownedAttribute.size ==
      OF.constraint.size and
   C.ownedAttribute.forAll{ P |
    OF.constraint.exists{ PC |
     PC.property == P
 }}}}.size == 1
}
```

OneMFPerCombination

```
ModelFragment.allInst.forAll{ MF |
 MF.object.size == 1 and
 Class.allInst.exists{ C |
  MF.object.forAll{ OF |
   C.ownedAttribute.size ==
      OF.constraint.size and
   C.ownedAttribute.forAll{ P |
    OF.constraint.exists{ PC |
     PC.property == P
}}}}}
```

FIG. 5.17 – Deux stratégies pour la création des fragments de modèles.

Une fois l'ensemble des combinaisons de valeurs à couvrir définie, la figure 5.17 présente deux stratégies différentes pour la création des fragments de modèles. La première stratégie (*OneMFPerClass*) consiste à créer un seul fragment de modèle par classe du méta-modèle. Ce fragment de modèle contient alors toute les combinaisons relatives à une classe. L'avantage de cette stratégie est de sélectionner des modèles de test qui contiennent toutes les combinaisons de valeurs retenues pour une classe. En revanche, son inconvénient est de conduire à définir de très gros fragments de modèles dans le cas ou un grand nombre de combinaisons sont retenues pour une classe. Afin de palier à ce problème, la seconde stratégie que nous proposons (*OneMFPerCombination*) consiste à créer un fragment de modèle par combinaison de valeurs. En utilisant cette stratégie, chaque fragment de modèle ne contient qu'un fragment d'objet.

Étant donné que l'on a défini deux stratégies pour choisir un ensemble de combinaisons et deux stratégies pour grouper ces combinaisons dans des fragments de modèles,

on obtient quatre critères de test différents. Ces quatre critères sont définis sur la Figure 5.18 en fonction des stratégies décrites en OCL sur les figures 5.16 et 5.17.

$Comb\sum$	`OneRangeCombination and OneMFPerCombination`
$Comb\prod$	`AllRangesCombination and OneMFPerCombination`
$Class\sum$	`OneRangeCombination and OneMFPerClass`
$Class\prod$	`AllRangesCombination and OneMFPerClass`

FIG. 5.18 – Quatre critères de combinaison classe par classe.

$MF\{AbstractState(label = 0, \sharp inTrans. = 0, \sharp outTrans. = 0)\},$
$MF\{AbstractState(label = 1, \sharp inTrans. = 1, \sharp outTrans. = 1)\},$
$MF\{AbstractState(label \geq 2, \sharp inTrans. \geq 2, \sharp outTrans. \geq 2)\},$
...

FIG. 5.19 – Exemple de fragments de modèles pour le critère $Comb\sum$.

$MF\{$
$AbstractState(label = 0, \sharp inTrans. = 0, \sharp outTrans. = 0),$
$AbstractState(label = 1, \sharp inTrans. = 1, \sharp outTrans. = 1),$
$AbstractState(label \geq 2, \sharp inTrans. \geq 2, \sharp outTrans. \geq 2)$
$\}$ ...

FIG. 5.20 – Exemple de fragments de modèles pour le critère $Class\sum$.

Afin d'illustrer les différences entre ces quatre critères, les figures 5.19, 5.20 et 5.21 présentent des exemples de fragments de modèles engendrés par les critères $Comb\sum$, $Class\sum$ et $Comb\prod$. L'exemple utilisé est toujours celui des machines à états. Pour chaque critère, seuls les fragments de modèles associés à la classe *AbstractState* sont représentés. Cette classe comporte trois propriétés, *label*, *incomingTransition* (*inTrans.*) et *outgoingTransition* (*outTrans.*) auxquelles ont été associées des partitions constituées de trois classes d'équivalences (cf Figure 5.4). En ce qui concerne les critères $Comb\sum$ et $Class\sum$, les combinaisons de valeurs attendues sont les mêmes et consistent a couvrir une fois chaque classe d'équivalence pour chaque propriété. Comme le montre les figures 5.19 et 5.20 trois fragments d'objets sont nécessaires pour cela. La différence entre ces deux critères réside dans la façon dont ces fragments d'objets sont encapsulés dans les fragments de modèles. Dans le cas du critère $Comb\sum$ un fragment de modèle est créé pour chaque fragment d'objet, alors que dans le cas du critère $Class\sum$ seul un fragment de modèle contenant l'ensemble des fragments de modèles relatifs à la classe *AbstractState* est défini.

De la même manière, les critères $Comb\prod$ et $Class\prod$ ne diffèrent que par la façon dont les fragments d'objets sont agencés dans les fragments de modèles. La figure 5.21 montre les fragments de modèles obtenus pour le critère $Comb\prod$. Les 27 fragments de modèles correspondent aux 27 combinaisons possibles pour les 3 classes d'équivalences

de chacune des partitions associées aux trois propriétés de la classe *AbstractState*.

$$
\begin{array}{l}
MF\{AbstractState(label = 0, \sharp inTrans. = 0, \sharp outTrans. = 0)\},\\
MF\{AbstractState(label = 1, \sharp inTrans. = 0, \sharp outTrans. = 0)\},\\
MF\{AbstractState(label \geq 2, \sharp inTrans. = 0, \sharp outTrans. = 0)\},\\
MF\{AbstractState(label = 0, \sharp inTrans. = 1, \sharp outTrans. = 0)\},\\
MF\{AbstractState(label = 1, \sharp inTrans. = 1, \sharp outTrans. = 0)\},\\
MF\{AbstractState(label \geq 2, \sharp inTrans. = 1, \sharp outTrans. = 0)\},\\
MF\{AbstractState(label = 0, \sharp inTrans. \geq 2, \sharp outTrans. = 0)\},\\
MF\{AbstractState(label = 1, \sharp inTrans. \geq 2, \sharp outTrans. = 0)\},\\
MF\{AbstractState(label \geq 2, \sharp inTrans. \geq 2, \sharp outTrans. = 0)\},\\
MF\{AbstractState(label = 0, \sharp inTrans. = 0, \sharp outTrans. = 1)\},\\
MF\{AbstractState(label = 1, \sharp inTrans. = 0, \sharp outTrans. = 1)\},\\
MF\{AbstractState(label \geq 2, \sharp inTrans. = 0, \sharp outTrans. = 1)\},\\
MF\{AbstractState(label = 0, \sharp inTrans. = 1, \sharp outTrans. = 1)\},\\
MF\{AbstractState(label = 1, \sharp inTrans. = 1, \sharp outTrans. = 1)\},\\
MF\{AbstractState(label \geq 2, \sharp inTrans. = 1, \sharp outTrans. = 1)\},\\
MF\{AbstractState(label = 0, \sharp inTrans. \geq 2, \sharp outTrans. = 1)\},\\
MF\{AbstractState(label = 1, \sharp inTrans. \geq 2, \sharp outTrans. = 1)\},\\
MF\{AbstractState(label \geq 2, \sharp inTrans. \geq 2, \sharp outTrans. = 1)\},\\
MF\{AbstractState(label = 0, \sharp inTrans. = 0, \sharp outTrans. \geq 2)\},\\
MF\{AbstractState(label = 1, \sharp inTrans. = 0, \sharp outTrans. \geq 2)\},\\
MF\{AbstractState(label \geq 2, \sharp inTrans. = 0, \sharp outTrans. \geq 2)\},\\
MF\{AbstractState(label = 0, \sharp inTrans. = 1, \sharp outTrans. \geq 2)\},\\
MF\{AbstractState(label = 1, \sharp inTrans. = 1, \sharp outTrans. \geq 2)\},\\
MF\{AbstractState(label \geq 2, \sharp inTrans. = 1, \sharp outTrans. \geq 2)\},\\
MF\{AbstractState(label = 0, \sharp inTrans. \geq 2, \sharp outTrans. \geq 2)\},\\
MF\{AbstractState(label = 1, \sharp inTrans. \geq 2, \sharp outTrans. \geq 2)\},\\
MF\{AbstractState(label \geq 2, \sharp inTrans. \geq 2, \sharp outTrans. \geq 2)\},\\
...
\end{array}
$$

FIG. 5.21 – Exemple de fragments de modèles pour le critère $Comb\prod$.

5.3.3 Utilisation de l'héritage

Les critères de la section précédente se contente de combiner les valeurs des propriétés d'une seule classe. En pratique, comme le montre l'exemple présenté dans le début de ce chapitre cela peut être insuffisant. En effet, pour tester la mise a plat de machines à états, nous avons identifié le besoin d'avoir des modèles de test contenant des états composites ayant à la fois un nombre variable d'états interne, et un nombre variable de transitions entrantes et sortantes. Or dans le méta-modèle de machine à états que nous utilisons, la seule propriété de la classe *CompositeState* est *ownedState* car les propriété *incomingTransition* et *outgoingTransition* sont héritées de la classe *AbstractState*. Cette section propose quatre nouveaux critères analogues à ceux de la section précédente qui combinent non seulement les valeurs des propriétés propre à une classe mais aussi les valeurs des propriétés héritées par cette classe.

Les noms de ces quatre nouveaux critères sont obtenus en préfixant le nom du critère de combinaison classe par classe correspondant par *IF* (pour *Inheritance Flattened*). Les

$MF\{Composite(label = 0, \sharp inTrans. = 0, \sharp outTrans. = 0, \sharp ownedState = 0)\},$
$MF\{Composite(label = 0, \sharp inTrans. = 0, \sharp outTrans. = 0, \sharp ownedState = 1)\},$
$MF\{Composite(label = 0, \sharp inTrans. = 0, \sharp outTrans. = 0, \sharp ownedState \geq 2)\},$
$MF\{Composite(label = 0, \sharp inTrans. = 0, \sharp outTrans. = 1, \sharp ownedState = 0)\},$
$MF\{Composite(label = 0, \sharp inTrans. = 0, \sharp outTrans. = 1, \sharp ownedState = 1)\},$
$MF\{Composite(label = 0, \sharp inTrans. = 0, \sharp outTrans. = 1, \sharp ownedState \geq 2)\},$
$MF\{Composite(label = 0, \sharp inTrans. = 1, \sharp outTrans. = 1, \sharp ownedState = 0)\},$
$MF\{Composite(label = 0, \sharp inTrans. = 1, \sharp outTrans. = 1, \sharp ownedState = 1)\},$
$MF\{Composite(label = 0, \sharp inTrans. = 1, \sharp outTrans. = 1, \sharp ownedState \geq 2)\},$
...

FIG. 5.22 – Exemple de fragments de modèles pour le critère $IFComb\prod$.

quatre critères obtenus sont donc $IFComb\sum$, $IFComb\prod$, $IFClass\sum$ et $IFClass\prod$.

Remarquons qu'une conséquence du fait de tenir compte des propriétés héritées est qu'il devient inutile de créer les fragments de modèles relatifs aux classes abstraites. En effet, tant que l'on ne tenait pas compte de l'héritage il était nécessaire de considérer les classes abstraites pour couvrir les partitions relatives à leurs propriétés. Dans le cas où l'on tient compte des propriétés héritées, les partitions liées aux propriétés des classes abstraites sont couvertes dans les fragments d'objets se rapportant à leurs sous-classes concrètes.

La figure 5.22 présente des fragments de modèle obtenues pour la classe *Composite* du méta-modèle de machines à états en utilisant le critère $IFComb\prod$. Seul un sous-ensemble des fragments de modèles relatifs à la classe *Composite* sont représentés. La classe *Composite* possède une propriété propre et en hérite trois de la classe *AbstractState*. Pour chacune de ces 4 propriétés la partition associée contient 3 classes d'équivalences ce qui conduit à un total de 81 (3^4) combinaisons pour la classe *Composite*.

5.3.4 Comparaison des critères et discussion

Cette section revient sur les critères définis dans les trois sous-sections précédentes et discute de leur pertinence pour la sélection de modèles de test. La Figure 5.23 représente les relations de subsumption entre les critères. On dit qu'un critère subsume un autre critère si tout ensemble de modèles de test vérifiant le premier critère vérifie aussi le second. De par cette définition, cette relation est transitive et permet de comparer la "force" des critères les uns par rapport aux autres. L'idée est que plus un critère est fort et plus le pouvoir de détection d'erreur des modèles de tests correspondant est élevé. Ainsi, on retrouve sans surprise au bas de la figure 5.23 les critères simples *AllRanges* et *AllPartitions* puis en remontant les différents critères qui tiennent compte de la structuration des propriétés en classes et de la relation d'héritage.

La Figure 5.24 présente pour chacun des critères le nombre de fragments de modèles, de fragments d'objets et de contraintes engendrés par chacun des critères sur l'exemple du méta-modèle de machine à état. Ces chiffres ont été calculés à partir des partitions présentées précédemment sur la figure 5.4. Les nombres de fragments de mo-

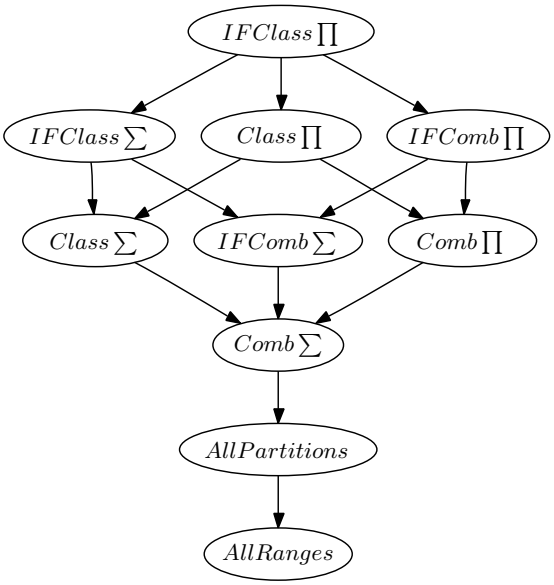


FIG. 5.23 – Relation de subsumption entre les critères.

dèle correspondent au nombre maximal de modèles de test permettant d'atteindre un critère. Il s'agit bien d'un nombre maximal car un modèle de test peut aisément couvrir plusieurs fragments de modèles si ceux-ci contiennent peu de fragments d'objets. Le nombre de fragments d'objets par rapport au nombre de fragments de modèle donne une estimation de la complexité minimale des modèles de test permettant d'atteindre un critère. Enfin, le nombre de contraintes permet d'estimer globalement la complexité de l'ensemble des modèles de test permettant de vérifier un critère. A titre de comparaison, la dernière ligne du tableau de la figure 5.24 donne les chiffres obtenus pour le critère *AllCombinations* discuté dans le début de ce chapitre et consistant à combiner systématiquement toutes les classes d'équivalence des partitions de toutes les propriétés du méta-modèle d'entrée.

Il est intéressant de noter que tous les critères proposés permettent de réduire de façon significative le nombre de contraintes et de fragments d'objet par rapport au critère *AllCombinations*. En effet, tous les critères proposés conduisent à un nombre de contraintes et de fragments d'objets qui croît linéairement avec le nombre de classes du méta-modèle d'entrée de la transformation à tester. Dans le cas du critère *AllCombinations*, cette croissance est exponentielle ce qui limite grandement les possibilités de passage à l'échelle de ce critère. Parmi les critères proposés, les critères de la forme $\star\prod$ présentent une complexité exponentielle avec le nombre de propriétés par classe dans le méta-modèle d'entrée. Ce nombre étant généralement faible et peu corrélé à la taille des méta-modèles, ce n'est pas réellement un obstacle à l'applicabilité de ces critères. Les autres critères ont une complexité linéaire avec le nombre de propriétés du méta-modèle d'entrée.

Critère	Frag. de modèles	Frag. d'objets	Contraintes
$AllRanges$	23	23	23
$AllPartitions$	10	10	23
$Comb\sum$	11	11	28
$Comb\prod$	64	64	236
$Class\sum$	4	11	28
$Class\prod$	4	64	236
$IFComb\sum$	9	9	42
$IFComb\prod$	381	381	2115
$IFClass\sum$	3	9	42
$IFClass\prod$	3	381	2115
AllCombinations	.	**1944**	**19440**

FIG. 5.24 – Comparaison des critères sur l'exemple des machines à états.

5.4 Génération de modèles

Cette section s'intéresse à la génération automatique de modèles de tests pour les transformations de modèles. Les critères présentés dans la section précédente ont pour objectif de vérifier si un ensemble de modèles de test est suffisant. Pour cela, ils identifient des fragments de modèles à couvrir. En pratique, la donnée de ces fragments de modèles peut être utilisée par le testeur comme une assistance à l'écriture ou à l'amélioration d'un ensemble de modèles de test. Au delà de cette utilisation manuelle de l'information, nous nous intéressons dans cette section à l'automatisation du processus de génération de modèles de tests. Nous proposons pour cela deux techniques de génération différentes. La première est basée sur un algorithme itératif qui construit les objets d'un modèle à partir de fragments. La seconde utilise un algorithme évolutionniste que nous avons initialement mis au point pour le test de programmes orienté-objets. Ces deux propositions constituent un premier pas vers l'automatisation de la génération de modèles mais des travaux d'expérimentation complémentaires, de mise au point et de validation restent nécessaires pour construire une solution satisfaisante au problème.

Cette section est organisée comme suit. La section 5.4.1 détaille les enjeux de la génération de modèles et les motivations de ce travail. La section 5.4.2 propose un algorithme itératif et détaille la validation expérimentale de cet algorithme. La section 5.4.3 rappelle le principe des algorithmes bactériologiques et détaille leur application à la génération de modèles. Enfin la section 5.4.4 conclut cette section.

5.4.1 Problématique et motivations

Afin d'illustrer les discussions de cette section, nous utilisons, une fois de plus l'exemple présenté dans la section 5.2.1. La transformation à tester est la mise-à-plat de machines à états conformes au méta-modèle rappelé par la figure 5.26. L'utilisation du partitionnement a permis d'associer des partitions à chaque propriété de ce méta-modèle (la figure 5.4). Grâce au framework et aux critères proposés dans les sections 5.2 et 5.3 les classes d'équivalences de ces partitions sont combinées pour former des frag-

ments d'objets et de modèles. La figure 5.25 présente trois ensembles de cinq fragments de modèles. Les deux premiers ensembles de fragments de modèles ont été obtenu en construisant aléatoirement cinq fragments de modèles très simples (contenant chacun un seul fragment d'objet). Le troisième ensemble correspond à cinq fragments de modèles produits par le critère $IFClass \prod$. L'objectif de la phase de génération de modèles est de produire des modèles qui permettent de couvrir des ensembles de fragments de modèles comme ceux-ci.

$MF\{AbstractState(label = 0)\}$, $MF\{Composite(\sharp ownedState = 0)\}$, $MF\{Transition(\sharp source = 1)\}$, $MF\{Transition(event \in .+)\}$, $MF\{AbstractState(\sharp outTrans. = 1)\}$
$MF\{Composite(\sharp ownedState \geq 2)\}$, $MF\{AbstractState(label = 1)\}$, $MF\{AbstractState(\sharp inTrans. \geq 2)\}$, $MF\{Transition(event ='')\}$, $MF\{State(isFinal = true)\}$
$MF\{AbstractState(label = 1, \sharp inTrans. = 1, \sharp outTrans. = 0),$ $State(isFinal = true, isInitial = false)\}$, $MF\{Transition(event \in .+, \sharp source = 1, \sharp target = 1)\}$, $MF\{AbstractState(label = 1, \sharp inTrans. \geq 2, \sharp outTrans. = 1),$ $State(isFinal = false, isInitial = true)\}$, $MF\{Transition(event =' ev1', \sharp source = 1, \sharp target = 1)\}$, $MF\{AbstractState(label \geq 2, \sharp inTrans. \geq 2, \sharp outTrans. = 0),$ $Composite(\sharp ownedState = 1)\}$,

FIG. 5.25 – Trois ensembles de cinq fragments de modèles.

Le premier ensemble de fragments de modèles de la figure 5.25 spécifie que l'ensemble des modèles de test doit contenir :

- Un état dont l'étiquette est 0.
- Un état composite vide.
- Une transition ayant un état source.
- Une transition portant un évènement quelconque.
- Un état ayant une transition sortante.

A partir de cet ensemble de fragments, il est possible de vérifier si un ensemble de modèles est satisfaisant. Il suffit pour cela de rechercher au moins un modèle contenant chaque fragment de modèle. Cependant, le fait de générer des modèles dans le but de couvrir ces fragments n'est pas direct dans la mesure où il n'existe pas de solution unique. La figure 5.27 présente, par exemple, deux ensembles de modèles construits manuellement dans le but de satisfaire ces cinq propriétés. Le premier ensemble compte un seul modèle (A.1) qui concentre tous les éléments demandés. Le second ensemble compte trois modèles : B.1 qui couvre la première propriété, B.2 qui couvre la seconde et B.3 qui couvre les trois dernières.

La génération automatique de modèles de test, comme ceux de la figure 5.27, pose deux problèmes : les modèles générés doivent être valides (conformes au méta-modèles et aux contraintes qui lui sont associées) et les modèles générés doivent être minimaux (en nombre et en taille).

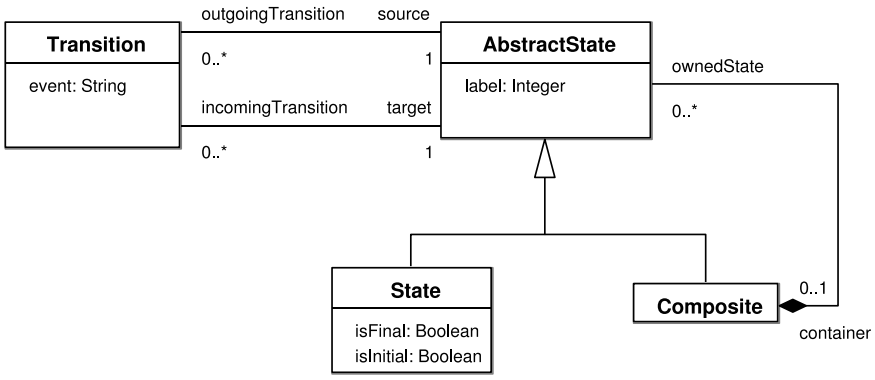


FIG. 5.26 – Méta-modèle de machines à états composites.

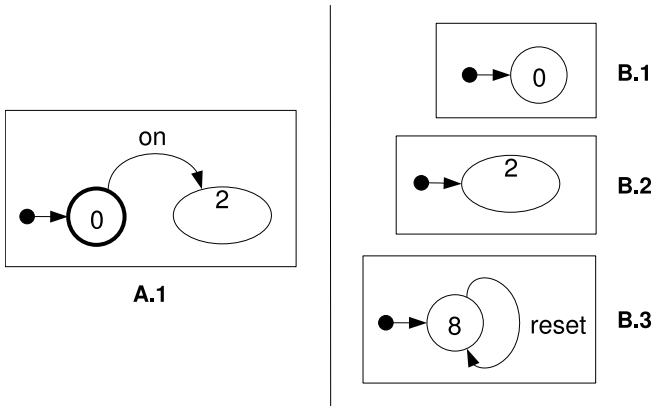


FIG. 5.27 – Exemple d'ensembles de modèles conformes au méta-modèle de la figure 5.26 permettant de couvrir les cinq premiers fragments de modèles de la figure 5.25.

Validité des modèles générés

Le premier problème est que les modèles de test doivent être conformes à un méta-modèle donné et aux éventuelles contraintes qui lui sont associées. Les fragments de modèles ne peuvent pas être directement utilisés pour le test car il ne constituent pas des modèles valides vis-à-vis du méta-modèle : il faut les compléter. Pour être conformes à un méta-modèle il est nécessaire que tous les objets d'un modèle respectent les types et les multiplicités des propriétés du méta-modèle. Si l'on considère le méta-modèle de la figure 5.26 cela signifie par exemple que chaque transition d'un modèle doit avoir un état source et un état cible. Pour créer un modèle valide contenant une transition, il est donc nécessaire que ce modèle contienne au moins un état.

Pour être conforme à un méta-modèle, un modèle doit vérifier les contraintes associées à ce méta-modèle. Ces contraintes sont généralement exprimées en OCL (d'autres langages déclaratifs ou impératifs peuvent également être utilisés) et doivent être vérifiées par les modèles de test. Par exemple, dans le cas du méta-modèle d'automate, pour qu'un automate soit valide il faut qu'il ait un seul état initial. Ce type de contrainte doit être pris en compte lors de la génération automatique de modèles.

En pratique, il est possible d'assurer par construction que la structure d'un modèle est en accord avec les structures décrites par un méta-modèle. Dans un environnement de modélisation comme Kermeta, ce niveau de conformité est en grande partie garanti par l'environnement. En effet, la gestion des associations et compositions et le typage fort permettent de s'assurer que tout objet possède une méta-classe et que les types des valeurs de ses propriétés sont correctes. La seule contrainte qui n'est pas assurée par l'environnement est le respect des bornes inférieures des multiplicités des propriétés. Les deux techniques que nous présentons dans la suite permettent d'assurer la tenue de cette contrainte.

En ce qui concerne le respect des contraintes associées au méta-modèle, le problème est plus difficile à résoudre. Ces contraintes étant quelconques, elles sont impossible à assurer par construction dans le cas général. La solution que nous proposons est, dans la première technique, un filtrage à posteriori des modèles de test et, un filtrage des modèles au cours de la génération dans la seconde.

Taille et nombre de modèle générés

En plus des deux exemples de la figure 5.27, un grand nombre d'ensembles solutions peut être construit. Ces ensembles diffèrent principalement par le nombre et la taille des modèles qui les composent. En effet, pour couvrir un ensemble de fragments de modèles, deux stratégies extrêmes peuvent être envisagées :

- La première est de créer un modèle de test par fragment de modèle à couvrir. Dans ce cas l'ensemble de modèles obtenu comportera un grand nombre de modèles (autant que de fragment de modèles) et ces modèles seront de petite taille.
- La seconde est de couvrir tous les fragments de modèles dans un unique modèle de test. Dans ce cas, un seul modèle de test est généré mais ce modèle à une taille importante. C'est cette stratégie qui a été utilisée pour créer le modèle A.1 de la figure 5.27.

Pour le bon déroulement de la phase de test, il faut trouver un compromis entre ces deux stratégies. D'une part, il faut minimiser le nombre de modèles de test afin de réduire les coûts liés, entre autres, à la production de l'oracle. D'autre part, il faut que les modèles de test soit aussi petits que possible pour faciliter la localisation des erreurs détectées par un modèle de test (diagnostique). Les deux techniques que nous proposons permettent d'ajuster ce compromis.

5.4.2 Algorithme de génération itératif

La première technique que nous proposons pour la génération de modèles est un algorithme qui construit des modèles en itérant sur les fragments de modèles à couvrir. L'idée de cet algorithme est d'utiliser des fragments de modèles comme germes et de leur ajouter un minimum d'éléments pour les transformer en des modèles valides. On assure ainsi une taille raisonnable pour les modèles générés. Cet algorithme a été implanté[1] et étudié expérimentalement.

Définition de l'algorithme

Le principe de l'algorithme est présenté en pseudo-code sur le listing de la figure 5.28. Les entrées de l'algorithme sont un méta-modèle (avec les partitions qui lui sont associées) et un ensemble de fragments de modèles à couvrir. La sortie de l'algorithme est un ensemble de modèles, conforme au méta-modèle donné, qui couvre l'ensemble des fragments de modèles donnés. Le principe de l'algorithme est de construire l'ensemble résultat en couvrant un à un les fragments de modèle en entrée.

Le pseudo-code du listing de la figure 5.28 utilise deux primitives importantes : *trouver* et *choisir*. La primitive *trouver* permet d'obtenir un ou plusieurs objets vérifiant un critère. Ces objets peuvent être des objets existants (si le modèle contient des objets répondant au critère demandé) ou des nouveaux objets. La primitive *trouver* correspond donc, soit à une recherche d'objets existants, soit à une création d'objets. La primitive *choisir* est utilisée pour choisir des valeurs pour les propriétés dont le type est simple. Aux lignes 12 et 14, le choix des valeurs est fait dans la classe d'équivalence spécifiée par la contrainte CT.

L'algorithme commence par créer un modèle (ligne 5) puis ajoute des objets dans ce modèle afin de couvrir des fragments de modèles (lignes 6 à 24). Pour chaque fragment de modèle à couvrir, il faut identifier ou ajouter des objets correspondant à chaque fragment d'objet dans le modèle (ligne 9). Les valeurs des propriétés de ces objets sont choisies en accord avec les contraintes contenues dans les fragments d'objet (ligne 10 à 21). Lorsque le modèle atteint une taille limite, il est mis en conformité avec le méta-modèle (ligne 27 à 35) puis ajouté à l'ensemble résultat (ligne 36). La mise en conformité consiste à vérifier et à compléter le modèle pour qu'il soit valide par rapport aux contraintes de multiplicités du méta-modèle. Tant qu'il reste des fragments de modèle non-couverts, le processus recommence (ligne 4).

[1] L'implantation de l'algorithme et l'étude de cas ont été réalisés par Erwan Brottier au cours d'un stage de Master 2 Pro à France Télécom R&D

```
1  Entrée : méta-modèle (MM),
2           ensemble de fragments de modèles à couvrir
3  Sortie : ensemble de modèles
4  tq il reste des fragments non couverts faire
5   créer un modèle M
6   tq la limite pour M n'est pas atteinte faire
7    choisir un fragment de modèle MF non couvert
8    pour tous les fragments d'objets OF de MF faire
9     trouver un objet O correspondant à OF .
10    pour chaque contrainte CT de OF portant sur une propriété P faire
11     si CT contraint une valeur alors -- cas d'une ValuePartition
12      choisir une valeur et l'affecter à P
13     sinon -- cas d'une MultiplicityPartition
14      choisir une cardinalité N
15      si le type de P est Classe alors
16       trouver N objets du type de P et
17       les associer à O par la propriété P
18      sinon choisir N valeurs du type de P et les associer à O
19      fsi
20     fsi
21    fpour
22    ajouter O à M
23   fpour
24  ftq
25  marquer MF comme couvert
26  -- compléter le modèle M pour qu'il soit conforme à MM
27  pour tout objet O de M
28   pour toute propriété nulle P de O ayant une multiplicité N..M
29    si le type de P est Classe alors
30     trouver N objets du type de P et
31     les associer à O par la propriété P
32    sinon choisir N valeurs du type de P et les associer à O
33    fsi
34   fpour
35  fpour
36  ajouter M au résultat
37 ftq
```

FIG. 5.28 – Algorithme de génération de modèles en pseudo-code.

L'algorithme proposé comporte des points de variations pour lesquels différentes stratégies sont utilisables. En fonction des stratégies utilisées, les modèles produits peuvent être très différents bien qu'ils assurent toujours la couverture des même fragments de modèles. Les principaux points de variations sont :

limite de taille des modèles Cette limite est le critère qui détermine (à la ligne 6 du listing) si un modèle a atteint une taille suffisante pour être ajouté dans l'ensemble résultat. Cette limite peut être exprimée en terme de nombre d'objets contenus dans le modèle M ou en terme de nombre de fragments de modèles couverts par M. Cette stratégie permet d'ajuster le compromis entre la taille et le nombre de modèles générés par l'algorithme. En effet, si l'on fixe une taille limite faible pour les modèles, l'algorithme va produire un grand nombre de petits modèles. A l'inverse si la limite de taille est grande, on obtiendra un petit nombre de gros modèles.

réutilisation/création d'objets La primitive *trouver* est utilisée à trois reprises dans l'algorithme (lignes 9, 16 et 30). A chaque fois que cette primitive est utilisée il est possible de chercher un objet existant dans le modèle ou de créer un nouvel objet. Deux stratégies extrêmes peuvent être envisagées : la création systématique d'objets ou la réutilisation systématique d'objets (lorsque cela est possible, ie. lorsqu'il existe des objets répondants aux critères demandés). Pour chacune des utilisations de la primitive *trouver* il est possible d'utiliser une stratégie différente. Par exemple, on pourra créer des objets lorsque ces objets correspondent à des fragments d'objets (ligne 9) puis réutiliser les objets existants pour compléter le modèle (ligne 16 et 30). L'impact sur les modèles générés de l'utilisation de différentes stratégies est illustré grâce à un exemple dans la section suivante.

classe à instancier Ce point de variation concerne également la primitive *trouver*. Lorsqu'un nouvel objet doit être créé, si le type de cet objet est une classe abstraite ou une classe ayant des sous-classes, il est nécessaire de choisir la classe qui doit être instanciée. Dans beaucoup de cas, les contraintes portent sur un petit nombre de propriétés qui peut appartenir à une classe abstraite ou à une classe ayant des sous-classes. Lors de l'instanciation, un choix doit donc être fait parmis les sous-classes. Plusieurs stratégies peuvent être envisagées : un choix aléatoire, le choix de la classe la moins dérivée ou encore le choix de celle qui présente le plus petit nombre de propriétés. Ces choix influent directement sur les types des objets contenus dans les modèles générés.

choix de valeurs La mise en oeuvre de la primitive *choisir* (lignes 12, 14, 18 et 32) peut également donner lieu à différentes stratégies. Dans le cas des lignes 12 et 14, la valeur doit être choisie dans une classe d'équivalence particulière définie par la contrainte CT. Dans le cas des lignes 18 et 32, le choix de valeur est libre. Pour l'ensemble de ces choix, il faut définir des politiques qui peuvent être : un choix aléatoire, un choix de valeurs uniques, un choix cyclique dans un ensemble de valeurs, etc. Dans le cas, par exemple, du choix concernant une multiplicité (ligne 14) on peut choisir la plus petite multiplicité admissible afin de minimiser la complexité des modèles générés.

Évaluation expérimentale

Pour évaluer la pertinence de cet algorithme pour la génération de modèles de test, il a été implanté, et deux études de cas ont été menées. La première étude porte sur la génération de diagrammes de classes pour le test d'une transformation permettant de générer des schémas de bases de données. La seconde étude porte sur la transformation permettant de mettre à plat des machines à états. Ces deux études ont permis de mettre au point l'algorithme mais aussi de mettre en évidence ses limitations relatives à la prise en compte des contraintes associées à un méta-modèle.

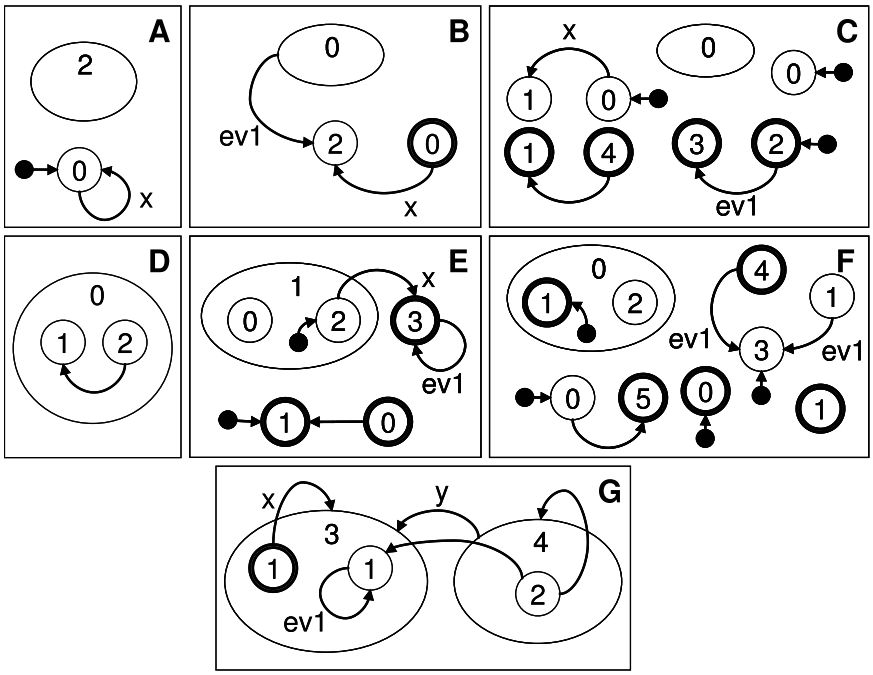


FIG. 5.29 – Exemple de modèles générés automatiquement par l'algorithme de la figure 5.28.

A titre d'exemple, la figure 5.29 présente un ensemble de modèles générés automatiquement pour couvrir les fragments de modèles de la figure 5.25. Pour chacun des modèles générés, la politique de limite de taille a été fixée a la couverture de cinq fragments de modèles. A partir d'un groupe de cinq fragments de modèle de la figure 5.25, un seul modèle est donc généré pour chaque exécution de l'algorithme. Les différents modèles présentés correspondent à l'utilisation de différentes stratégies pour la création/réutilisation des objets. Les stratégies employées pour le choix des classes à instancier et les valeurs sont des choix aléatoires.

Les modèles A, B et C ont été générés à partir des cinq premiers fragments de modèles de la figure 5.25. Ces trois expériences diffèrent par la politique de création/-réutilisation des objets qui a été utilisée. Pour le modèle A les objets existants ont été réutilisés au maximum, pour le modèle B un objet a été créé pour chaque fragment

d'objet à couvrir et enfin, pour le modèle C aucun objet n'a été réutilisé. Les modèles D, E et F ont été générés en utilisant les mêmes politiques que les modèles A, B et C mais en utilisant la seconde liste de fragments de modèles de la figure 5.25. Enfin, le modèle G correspond au modèle généré pour la dernière liste de fragments de modèles de la figure 5.25 et en utilisant la politique de réutilisation maximale des objets.

Les différents modèles générés mettent en évidence la grande influence des politiques utilisées pour les points de variation de l'algorithme. L'utilisation de politiques de réutilisation des objets permet d'obtenir des modèles comportant peu d'objets mais beaucoup de dépendances entre ces objets, alors que la politique de création d'objets systématique produit des modèles comportant plus de groupes d'objets indépendants les uns des autres. L'utilisation de stratégies différentes permet d'ajuster la morphologie des modèles générés en fonction des besoins : pour la minimisation des données de test il peut être avantageux d'avoir des modèles contenant le minimum d'objets alors que pour le diagnostic il peut être intéressant que chaque fragment de modèle soit couvert par un groupe d'objet isolé.

Bien que l'algorithme permette de générer des modèles conformes a un méta-modèle, les observations issues des deux études de cas réalisées sont en accord sur le fait que la non prise en compte des contraintes associées au méta-modèle est une limitation importante. On peut remarquer, par exemple, sur les modèles de la figure 5.29 que les modèles B et D n'ont pas d'états initiaux, que les modèles C et F ont plusieurs états initiaux et que le modèle G comporte des transitions d'un état interne vers son état composite ce qui n'a probablement pas de sens. Une solution à ce problème serait de filtrer les modèles générés afin de garder uniquement ceux qui vérifient les contraintes statiques associées au méta-modèle. Cependant, cette solution a l'inconvénient d'être très peu efficace.

Afin de palier cette limitation, la section suivante propose d'utiliser un algorithme évolutionniste. Cet algorithme a pour but de rechercher des modèles permettant d'assurer à la fois la couverture des fragments de modèles et la tenue des contraintes statiques associées à un méta-modèle.

5.4.3 Algorithme pseudo-aléatoire

La littérature sur la génération automatique de test présente un certain nombre de travaux utilisant les algorithmes génétiques [JHD96, Jon98, PHP99, MMS01, WBS01]. La plupart de ces travaux ne sont applicables que dans le cas où les données de test sont des types simples (valeurs numériques) et si un ensemble d'objectifs de test a été préalablement défini. L'algorithme génétique permet alors de générer pour chaque objectif de test une donnée de test couvrant cet objectif. Le travail présenté ici est original dans la mesure où l'objectif de test utilisé est global. En effet, on ne cherche plus ici à générer un cas de test par objectif de test mais un ensemble de cas de test, aussi petit que possible, assurant un critère de test global tel que : le score de mutation, la couverture de fonctions ou d'instructions...

Les limites de l'algorithme génétique dans ce domaine sont identifiées dans [BTLJ00] dans le cadre du test unitaire. Ces résultats sont confirmés dans [BFJT02b, BFJT02a,

BFJT05, BFJLT05] dans le cadre du test système. Ces travaux présentent un algorithme original, dérivé de l'algorithme génétique et adapté aux besoins de la génération de test : l'algorithme bactériologique. L'avantage de cet algorithme est de permettre d'optimiser non pas un cas de test (comme c'est le cas lorsque l'on utilise un algorithme génétique) mais un ensemble de cas de test.

Dans [BFJT05] nous avons utilisé un algorithme bactériologique pour optimiser les tests de programmes dont la structure du domaine d'entrée est décrite par une grammaire. Ce que nous proposons ici est d'étendre la technique pour l'appliquer lorsque le domaine d'entrée du programme à tester est décrit par un méta-modèle. La suite de cette section rappelle les principes des algorithmes bactériologiques et détaille leur application à la génération de modèles.

Les algorithmes bactériologiques

Les algorithmes bactériologiques s'inspirent du processus naturel d'adaptation des bactéries [Ros95]. Naturellement, les bactéries s'adaptent grâce aux mutations survenant lors de la division cellulaire afin de coloniser au mieux leur milieu. L'idée de l'algorithme bactériologique est d'utiliser ce phénomène afin de résoudre des problèmes dont la solution est un ensemble d'éléments (ou *bactéries*) choisis dans un vaste espace de recherche et satisfaisant une propriété.

La figure 5.30 présente le processus d'évolution bactériologique. L'algorithme bactériologique manipule deux ensembles de bactéries : le bouillon bactériologique qui contient la population courante, et la "mémoire" dans laquelle la solution est construite. Cette solution est construite itérativement lors de l'exécution de l'algorithme. L'algorithme se termine lorsque l'ensemble solution atteint l'objectif fixé ou qu'un nombre maximal d'itérations est atteint.

Chaque itération de l'algorithme peut se diviser en quatre phases. Une phase de classement (1) permet d'ordonner les bactéries du bouillon de la meilleure à la moins bonne. Pour cela l'algorithme se fonde sur une *fonction d'utilité* permettant d'évaluer la qualité des bactéries les unes par rapport au autres. Une fois l'utilité des bactéries évaluée, la meilleure est éventuellement ajoutée à la solution (2). Une bactérie est ajoutée à la solution si sa contribution est suffisante, c'est-à-dire si sa mémorisation augmente suffisamment la qualité de la solution ; on utilise pour cela une *fonction de mémorisation*. Pour éviter la surpopulation dans le bouillon, on le filtre afin d'enlever les bactéries les plus mauvaises (3), celles qui ont une utilité nulle par exemple. Enfin, avant de recommencer un cycle, la phase de mutation (4) ajoute au bouillon un ensemble de bactéries nouvelles construites en mutant les meilleures bactéries du bouillon (grâce à un *opérateur de mutation*).

Afin d'appliquer un algorithme bactériologique, il est nécessaire de modéliser le problème à résoudre en terme de bactéries. Dans la suite on note $\mathbb{B}$ l'espace de recherche, c'est-à-dire l'ensemble des bactéries conformes au modèle utilisé. Une fois le problème ainsi modélisé, quatre fonctions restent à définir :

- Une *fonction d'utilité* $F : 2^{\mathbb{B}} \rightarrow \mathbb{R}^+$
  La fonction d'utilité est construite à partir de l'objectif à atteindre et permet

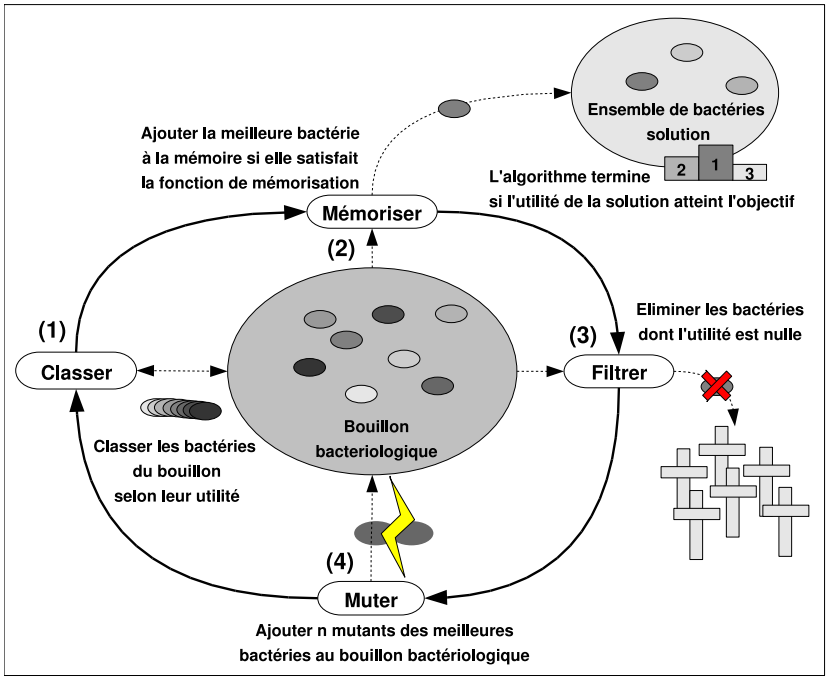


FIG. 5.30 – Principe des algorithmes bactériologiques.

d'évaluer la qualité d'un ensemble de bactéries vis-à-vis de cet objectif. L'algorithme bactériologique s'attache à maximiser cette fonction avec l'ensemble de bactéries le plus petit possible. A partir de cette fonction d'utilité globale, on définit une fonction f permettant d'évaluer la qualité d'une bactérie par rapport à un ensemble de bactéries :

$$f : \mathbb{B} \times 2^{\mathbb{B}} \rightarrow \mathbb{R}^{+}$$

$$f(B, b) = F(B \cup b) - F(B)$$

Cette fonction est utilisée dans l'algorithme pour évaluer l'utilité relative des bactéries du bouillon, c'est-à-dire leur utilité par rapport aux bactéries déjà mémorisées. Lors de la phase de classement, c'est l'utilité relative des bactéries du bouillon qui permet de les ordonner.
Il est important pour la convergence de l'algorithme que F soit croissante par l'ajout d'une bactérie pour que f soit positive, c'est-à-dire que l'ajout d'une bactérie à un ensemble ne puisse pas diminuer la qualité de cet ensemble vis-à-vis de l'objectif :

$$\forall B \in 2^{\mathbb{B}}, \forall b \in \mathbb{B}, F(B) \leqslant F(B \cup b)$$

- Une *fonction de mémorisation* $S : \mathbb{B} \rightarrow \{\text{vrai}, \text{faux}\}$
  La fonction de mémorisation a pour paramètre une bactérie candidate à la mémorisation. Elle retourne "vrai" si la bactérie candidate doit être mémorisée. Dans la

pratique on peut, par exemple, décider de mémoriser une bactérie si l'utilité relative de la meilleure bactérie n'a pas augmenté au cours de n dernières itérations de l'algorithme. Une autre solution est de mémoriser les bactéries si leur utilité relative est supérieure à un seuil.
La fonction de mémorisation assure un compromis entre la vitesse de convergence de l'algorithme et la minimisation de l'ensemble solution. En effet, mémoriser rapidement les bactéries assure une évolution rapide du bouillon mais tend à produire une solution de taille importante.

- Une *fonction de filtrage* $K : 2^{\mathbb{B}} \rightarrow 2^{\mathbb{B}}$
  La fonction de filtrage permet de réduire la taille du bouillon bactériologique en tuant les bactéries considérées comme inutiles. Dans la pratique, une heuristique simple est d'éliminer les bactéries dont l'utilité relative est nulle. On peut également décider de supprimer les bactéries qui ont un âge supérieur à une limite.
  Cette fonction doit éliminer suffisamment de bactéries afin de limiter l'espace mémoire nécessaire à l'exécution de l'algorithme mais pas trop afin de ne pas perdre d'informations utiles dans le bouillon.
- Un *opérateur de mutation* $M : \mathbb{B} \rightarrow \mathbb{B}$
  L'opérateur de mutation permet de créer une bactérie mutante à partir d'une bactérie en modifiant légèrement les données qui la composent. Cet opérateur dépend évidemment du modèle de bactérie adopté.
  Pour être efficace et assurer la convergence de l'algorithme quelque soit les conditions initiales, l'opérateur de mutation doit permettre par applications successives d'explorer de proche en proche l'ensemble $\mathbb{B}$ tout entier.

Algorithmes bactériologiques et génération de tests

Cette section présente l'application des algorithmes bactériologiques au problème de la génération/optimisation de tests indépendamment du critère de test à satisfaire. Ayant été conçue pour l'optimisation de suites de tests, l'application des algorithmes bactériologiques à ce problème est immédiate. Ainsi, comme le montre le tableau 5.31, l'espace de recherche correspond au domaine d'entrée du programme à tester et les bactéries correspondent aux cas de tests. Le but est de générer une suite de tests (ensemble de bactéries) de taille raisonnable satisfaisant au mieux un critère de test. La fonction d'utilité est ainsi logiquement dérivée du critère de test à satisfaire.

Algorithme bactériologique		*Optimisation de tests*
Espace de recherche	$\leftrightarrow$	Domaine d'entrée du programme
Ensemble de bactéries	$\leftrightarrow$	Suite de tests
Bactérie	$\leftrightarrow$	Cas de test
Fonction d'utilité	$\leftrightarrow$	Critère de test

FIG. 5.31 – Génération de test et algorithme bactériologique.

La figure 5.32 résume la méthodologie de génération/optimisation de suites de tests

au moyen d'un algorithme bactériologique. La fonction d'utilité est définie afin d'évaluer la pertinence d'un ensemble de tests vis-à-vis du critère de test à satisfaire. Si, par exemple, le critère de test est la couverture du code, on pourra utiliser pour la fonction d'utilité, le pourcentage d'instructions couvertes par un ensemble de cas de tests. L'opérateur de mutation est défini à partir du domaine d'entrée du programme sous test. Il permet, à partir d'un cas de test, de produire un nouveau cas de test "proche" du premier. Si, par exemple, l'entrée du programme sous test est un entier, l'opérateur de mutation pourra être d'ajouter plus ou moins un à une donnée de test pour obtenir une donnée de test mutée.

La technique proposée peut être utilisée à la fois pour la génération de suites de tests, si la population initiale de cas de tests est générée aléatoirement, et pour l'optimisation d'une suite de tests, si la population initiale est construite avec les cas de tests d'une suite de tests existante.

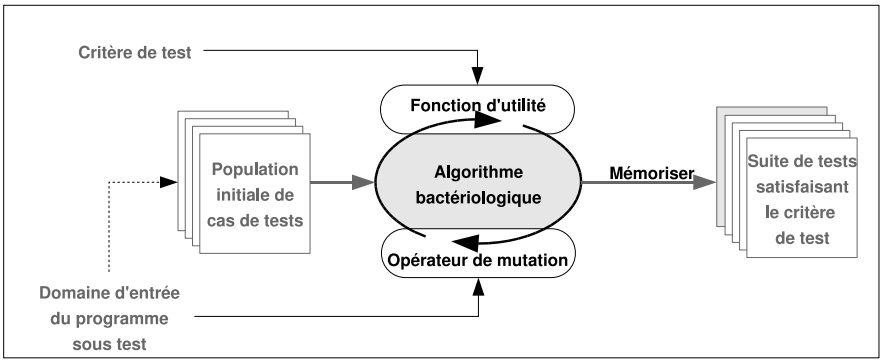


FIG. 5.32 – Optimisation automatique de tests avec un algorithme bactériologique.

Remarque 5.1 *La fonction d'utilité permettant de guider la recherche de bon cas de tests dans le domaine d'entrée du programme, il est nécessaire, afin d'assurer la convergence rapide de l'algorithme, qu'elle soit à valeur réelle. En effet, si la fonction d'utilité ne prend que deux valeurs (vrai si le critère est satisfait et faux dans le cas contraire), le classement de bactéries, ne satisfaisant pas le critère, n'apporte aucune information. La technique s'apparente alors à de la génération aléatoire de tests.*

Pour un algorithme bactériologique, les fonctions de mémorisation et de filtrage assurent respectivement la construction de la solution et l'efficacité de l'algorithme. Dans le cadre de la génération de suite de tests, la fonction de mémorisation assure un compromis entre la vitesse de convergence de l'algorithme et la taille de la suite de tests obtenue. La fonction de filtrage assure, quant à elle, que la taille du bouillon reste raisonnable en supprimant à chaque itération les bactéries inutiles. Cette fonction doit minimiser la taille du bouillon (plus le bouillon contient de cas de tests et plus l'algorithme est complexe) sans toutefois perdre des informations utiles.

Dans les premiers travaux [BTLJ00] utilisant l'algorithme bactériologique, la fonction de mémorisation consiste à mémoriser un cas de test si son utilité relative est strictement positive. Dans les travaux plus récents, la fonction de mémorisation est durcie afin de conduire à des suites de test plus petites et ne mémorise les bactéries que si leur utilité est supérieure à un seuil de mémorisation.

Initialement, la fonction de filtrage est inspirée des algorithmes génétiques. Elle consiste à éliminer toutes les bactéries de la génération précédente à l'exception de la meilleure. Or, l'expérience montre que cette fonction de filtrage conduit à des pertes d'information importantes. Une fonction de filtrage plus "laxiste" n'éliminant que les cas de test d'utilité nulle est donc préférable. Le problème de la fonction de filtrage dans le contexte de la génération de suites de test revient à un problème de minimisation de suites de test et un grand nombre d'heuristiques existant dans ce domaine peuvent être appliquées (cf section 2.2.4 de l'état de l'art).

Application à la génération de modèles

Afin de pouvoir utiliser un algorithme bactériologique pour la génération de modèles de test, il nécessaire de définir la modélisation des cas de tests, une fonction d'utilité et un opérateur de mutation. L'objectif est de générer un ensemble de modèles conformes à un méta-modèle et qui couvre un ensemble de fragments de modèles. Chaque cas de test (ou bactérie) est donc un modèle conforme au méta-modèle d'entrée de la transformation à tester. L'objectif a atteindre est de couvrir un ensemble de fragments de modèles, il est donc naturel d'utiliser le nombre de fragments de modèles couverts comme fonction d'utilité. La tableau de la figure 5.33 détaille la correspondance entre le problème de la génération de modèles et les notions de l'algorithme bactériologique.

Algorithme bactériologique		*Génération de modèles*
Espace de recherche	$\leftrightarrow$	Méta-modèle d'entrée
Ensemble de bactéries	$\leftrightarrow$	Ensemble de modèles
Bactérie	$\leftrightarrow$	Modèle
Fonction d'utilité	$\leftrightarrow$	Nb de fragments de modèles couverts

FIG. 5.33 – Algorithme bactériologique et génération de modèles.

Lors de l'exécution d'un algorithme bactériologique, le rôle de l'opérateur de mutation est primordial car il est l'unique source d'évolution pour les bactéries. Pour faire évoluer des modèles dans le but de couvrir un maximum de fragments de modèles, nous proposons d'utiliser deux opérateurs de mutation. Ces deux opérateurs réutilisent l'algorithme de génération de modèles proposé dans la section 5.4.2 pour construire de nouveaux modèles à partir d'un modèle existant. Le premier a pour but d'assurer une bonne exploration de l'ensemble des modèles en effectuant des mutations élémentaires sur les modèles. Le second a pour objectif de diriger les mutations vers la couverture de nouveaux fragments de modèles. Ces deux opérateurs sont donc complémentaires.

Pour chaque utilisation de la mutation au cours de l'exécution de l'algorithme bactériologique, l'opérateur à utiliser doit être choisi aléatoirement.

Le premier opérateur consiste à effectuer des modifications élémentaires sur le modèle à muter. Ces modifications peuvent être la création d'un objet, la suppression d'un objet ou la modification d'une propriété d'un objet. La figure 5.34 présente un ensemble d'applications de cet opérateur de mutation sur une machine à états. Le modèle initial est la machine à état O.

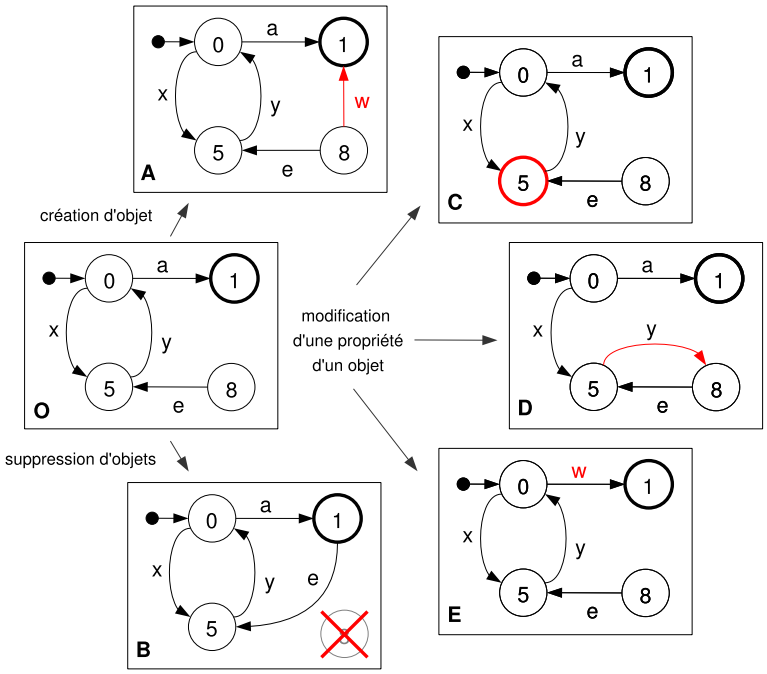


FIG. 5.34 – Exemple de mutations élémentaires sur un modèle.

Le modèle A est obtenu par l'ajout d'un objet (une transition). Lors de l'ajout d'un objet, une classe concrète du méta-modèle est choisie et instanciée. Les valeurs des propriétés du nouvel objet sont choisies et affectées en utilisant le principe de l'algorithme de génération de modèle de la section 5.4.2 (lignes 28 à 34 sur le listing de la figure 5.28). Le modèle B est obtenu par la suppression d'un objet (l'état 8). De la même manière que pour l'ajout d'un objet, l'algorithme de la section 5.4.2 est utilisé une fois l'objet supprimé afin d'assurer que le nouveau modèle est conforme au méta-modèle (lignes 27 à 35 sur le listing de la figure 5.28). Que ce soit pour l'ajout ou la suppression d'objets, afin d'ajouter un minimum d'objets parasites au modèle muté, la politique de création/réutilisation d'objet la plus adaptée est la réutilisation systématique d'objets existant.

Les modèles C, D et E ont été construits par la modification d'une propriété d'un objet du modèle O. Pour le modèle C c'est la propriété *isFinal* de l'état 5 qui a été mise a *true*. Pour le modèle D c'est la valeur de la propriété *target* de la transition y qui a

été modifié de l'état 0 à l'état 8. Enfin, pour le modèle E c'est la valeur de la propriété *event* de la transition entre les états 0 et 1 qui a changé. Ici encore, une fois que la modification de valeur pour une propriété a été faite il est nécessaire de compléter le modèle afin d'assurer sa validité vis-à-vis du méta-modèle en utilisant l'algorithme de la section 5.4.2 (lignes 27 à 35 sur le listing de la figure 5.28).

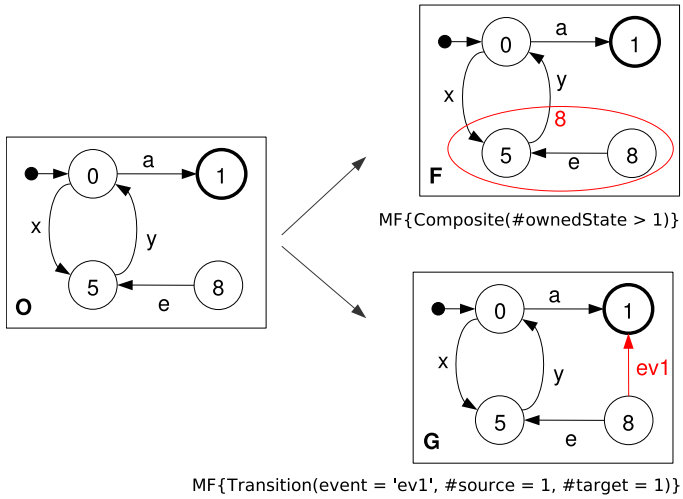


FIG. 5.35 – Exemple de mutations dirigées par des fragments de modèles.

Le second opérateur de mutation que nous proposons a pour but de faire évoluer les modèles vers la couverture de nouveaux fragments de modèles. Cet opérateur de mutation correspond exactement aux lignes 7 à 23 du listing de la figure 5.28. L'idée est de choisir aléatoirement un fragment de modèle non couvert et d'ajouter les objets correspondants à ce fragment dans le modèle. La figure 5.35 présente deux exemples de mutation basés sur des fragments de modèles. Ici encore, afin de faire croître le moins possible la taille de modèles, la politique utilisée est de réutiliser au maximum les objets existants.

Quel que soit l'opérateur de mutation utilisé, après la mutation d'un modèle, un nouveau modèle est créé. On fait l'hypothèse que le modèle initial est conforme à son méta-modèle et qu'il vérifie toutes les contraintes associées à ce méta-modèle. Par construction et grâce à l'utilisation de l'algorithme définie dans la section 5.4.2, les modèles mutés sont conformes à leur méta-modèle. Cependant, il est possible que les modèles mutés ne vérifient plus les contraintes associées au méta-modèle. Dans ce cas le modèle muté doit être rejeté et une autre mutation doit être effectuée. Cette procédure simple permet de s'assurer que les nouveaux modèles sont non seulement conformes au méta-modèle mais aussi aux contraintes qui lui sont associées. La mutation étant la seule source d'évolution des modèles dans l'algorithme bactériologique, le fait de garantir cette propriété pour l'opérateur de mutation permet d'assurer que l'ensemble de modèles produits seront parfaitement conformes au méta-modèle.

Le dernier élément nécessaire à l'exécution de l'algorithme bactériologique est un

ensemble de modèles initiaux. Ces modèles initiaux constituent le germe qui évolue ensuite grâce à l'opérateur de mutation. Ces modèles initiaux doivent être conformes au méta-modèle d'entrée de la transformation à tester et vérifier l'ensemble des contraintes associées à ce méta-modèle. Deux solutions sont envisageables pour obtenir cet ensemble initial. La première est de demander au testeur de le fournir. La seconde est d'utiliser l'algorithme proposé dans la section 5.4.2. Cet algorithme permet d'obtenir des modèles qui couvrent l'ensemble des fragments de modèles mais ne permet pas d'assurer la tenue des contraintes associées au méta-modèle. En filtrant l'ensemble de modèles obtenu grâce au contraintes associées au méta-modèle, on obtient un ensemble de modèles qui peut être utilisé comme germe. L'avantage de cette dernière technique est qu'elle est complétement automatisable.

5.4.4 Conclusion

Nous avons proposé dans cette section deux techniques pour la génération automatique de modèles de test à partir de fragments de modèles. La première se base sur un algorithme itératif qui vise à couvrir un à un les fragments de modèles. Cet algorithme a été implanté et évalué expérimentalement. Le résultat de cette étude est la mise en évidence d'une limitation importante de l'algorithme : les contraintes associées au méta-modèle ne sont pas prise en compte.

La seconde approche que nous proposons est l'utilisation d'un algorithme évolutionniste. Cette approche se base sur des travaux antérieurs sur la génération automatique de données de test pour lesquelles nous avons mis au point les algorithmes bactériologiques. L'objectif de cette approche est de permettre de tenir compte des contraintes associées aux méta-modèles. L'algorithme itératif conçu dans l'approche initiale est réutilisé pour construire l'opérateur de mutation indispensable aux algorithmes bactériologiques. La chaîne de génération de modèles est en cours d'implantation en Kermeta mais nous ne disposons, pour l'instant, d'aucun résultat expérimental. Seule une validation expérimentale soigneuse permettra d'affiner et de valider les techniques de génération de modèles proposées dans cette section.

5.5 Conclusion et perspectives

La validation des transformations de modèles est primordiale pour que les techniques de développement basées sur les modèles tiennent leurs promesses en terme de qualité du logiciel produit. Les travaux présentés dans ce chapitre ne sont qu'un premier pas dans cette direction. Bien que les transformations de modèle soient des programmes, nous avons montré qu'il est avantageux de développer des techniques de validation spécifiques aux transformations de modèles. En effet, les techniques de test existantes sont, d'une part peu adaptées à la complexité des données manipulées par les transformations de modèles et, d'autre part incapables d'utiliser les éléments de spécification (comme les méta-modèles) propres au transformations de modèles.

Dans ce contexte, nous nous sommes intéressés au test "boite noire" de transformation de modèles. La technique que nous proposons s'appuie sur le méta-modèle

d'entrée de la transformation à tester. Le méta-modèle d'entrée d'une transformation fournit une description formelle de son domaine d'entrée. Pour sélectionner des modèles de test l'idée est de "couvrir" ce domaine d'entrée. Nous proposons pour cela :

- un framework permettant de représenter les structures d'objets attendues dans les modèles de tests : les fragments d'objets et de modèles.
- un ensemble de critères permettant de construire des fragments de modèles à partir du méta-modèle d'entrée de la transformation à tester.

L'utilisation du framework et des critères permet de qualifier un ensemble de modèles de test pour le test d'une transformation. La donnée des fragments de modèles à couvrir permet également d'assister le testeur lors de l'écriture ou de l'optimisation d'un ensemble de modèles de test. Dans la suite de ces travaux il est nécessaire d'évaluer et de comparer la capacité de détection d'erreurs des différents critères proposés. Au besoin, deux directions complémentaires peuvent permettre d'enrichir la technique proposée. La première est d'augmenter l'expressivité des fragments de modèles afin de permettre de représenter des relations entre les fragments d'objets qu'ils contiennent. La seconde est la définition de nouveaux critères de tests qui s'appuient sur d'autres éléments de spécifications que le méta-modèle d'entrée.

Dans un second temps, nous avons détaillé deux propositions pour la génération automatique de modèles. Le problème de la génération automatique de données de test est un problème qui a donné lieu à beaucoup de travaux dans le domaine du test de logiciel. Le problème est d'autant plus difficile pour le test de transformation de modèle que les données de test sont complexes. Le premier algorithme que nous avons proposé est un algorithme itératif qui ne permet pas de tenir compte des contraintes statiques associées au méta-modèle. De ce fait, son application en pratique semble difficile. La seconde proposition que nous faisons est l'utilisation d'un algorithme évolutionniste : un algorithme bactériologique. Nous avons développé cet algorithme dans des travaux antérieurs pour la génération de test pour des systèmes orientés-objets. L'idée de réutiliser cet algorithme avec des opérateurs de mutation adaptés à la génération de modèle semble prometteuse. La validation expérimentale de cette technique devrait permettre de la mettre au point et de la valider.

En plus de problèmes discutés en détail dans ce chapitre, c'est-à-dire le critère de test et la génération de modèles, le test de transformation de modèles reste un domaine pour lequel un effort de recherche est nécessaire. D'une part, le problème de l'oracle constitue, entre autres, une piste de recherche importante du fait de la complexité des données manipulées par les transformations de modèles[2]. D'autre part, le développement et la normalisation de langages spécifiques à la transformation de modèles (comme QVT) devraient permettre de développer des techniques de test structurelles (ou "boite blanche") adaptées aux transformations de modèles.

[2]La thèse de Jean-Marie Mottu, commencée en septembre 2005, s'intéresse à ce problème

Chapitre 6

Conclusion et perspectives

Les travaux présentés dans cette thèse s'inscrivent dans un contexte où la taille et la complexité des logiciels augmentent alors que les contraintes de temps, de développement, de qualité, de maintenance et d'évolution sont toujours plus fortes. Pour répondre à cette tendance, l'ingénierie des modèles apparaît comme une évolution prometteuse des techniques de génie logiciel. Cependant, le succès d'une approche de développement est conditionné par l'existence de techniques permettant d'assurer la qualité du logiciel produit. Comme nous l'avons montré dans cette thèse, le fait d'utiliser les modèles de manière productive nécessite de disposer d'environnements de modélisation bien formalisés et de techniques pour valider les programmes de transformation de modèles. Nos travaux constituent un pas dans la direction d'une ingénierie des modèles fiable et ouvrent de nombreuses perspectives dans ce domaine.

6.1 Contribution

Chronologiquement, le premier point auquel nous nous sommes intéressés dans le but de fiabiliser les processus de développement basés sur les modèles est la validation de transformations de modèles. L'utilisation de transformations de modèles étant un élément central pour l'application de l'ingénierie dirigée par les modèles, leur validation nous paraissait comme l'élément déterminant pour la qualité du logiciel produit. Au cours des réflexions que nous avons menées, nous avons constaté que les techniques et outils existants pour définir des modèles et des transformations de modèles n'offraient pas les outils nécessaires à la mise en place de techniques de validation rigoureuses. En effet, pour réaliser un dispositif complet de validation pour des transformations de modèles nous avons identifié le besoin de disposer :

- d'éléments de spécification de la transformation (par exemple des pré-conditions et post-conditions).
- de la structure de la transformation (par exemple pour permettre des vérifications ou des analyses statiques).
- de la spécification complète des ses entrées/sorties : structure, sémantique et contraintes associées.

Tant que les modèles étaient utilisés comme de simples représentations du logiciel, le fait de définir les langages de modélisation grâce à des formalismes hétérogènes (MOF, OCL et le langages naturel dans le cas d'UML par exemple) était suffisant. Cependant, lorsque les modèles sont définis pour être exploités directement par une machine, il est indispensable de disposer de langages de modélisation et d'environnements sémantiquement bien définis et offrant les outils nécessaires à une démarche de qualité. Cette constatation nous a amené à travailler sur la définition d'un environnement de méta-modélisation.

Dans le chapitre 3 nous avons détaillé la réalisation de l'environnement de méta-modélisation Kermeta. Cet environnement est centré autour du langage Kermeta qui est utilisé pour définir formellement et de manière unifiée les différents artefacts de l'ingénierie des modèles : méta-modèles, contraintes, sémantique et transformations. Les points forts du langage Kermeta sont son expressivité, ses concepts dédiés à l'ingénierie des modèles, son puissant système de type statique et sa compatibilité avec les langages et outils de modélisation existants. Kermeta est aujourd'hui un projet open-source qui compte une douzaine de développeurs actifs et un nombre croissant d'utilisateurs.

Dans le chapitre 4 nous avons présenté quatre études de cas réalisées pour valider les qualités du langage Kermeta. Les études de cas que nous avons choisies couvrent une large variété d'applications, à la fois académiques et industrielles, de l'ingénierie des modèles : la définition de langage dédiés, la transformation de modèles, la composition et le tissage des modèles et le traitement automatique d'exigences logicielles. Le travail que nous avons réalisé sur ces études de cas nous a permis de valider le langage Kermeta et en particulier de mettre en évidence ses qualités pour la conception, la réutilisation, la validation et l'évolution de code d'ingénierie des modèles.

Dans le chapitre 5 nous avons présenté les travaux que nous avons menés pour la validation de transformations de modèles. Notre contribution dans ce domaine est un framework permettant de définir des critères de test en utilisant le méta-modèle d'entrée de la transformation à tester. En utilisant ce framework nous avons proposé un ensemble de critères et des algorithmes pour la génération automatique de modèles de test. Les premiers algorithmes que nous avons proposés sont limités dans la mesure où ils ne permettent pas de tenir compte des contraintes associées aux méta-modèles. C'est la raison pour laquelle nous avons proposé une seconde technique utilisant un algorithme bactériologique.

6.2 Perspectives

Les travaux menés au cours de cette thèse ouvrent un certain nombre de perspectives à la fois liées au langage Kermeta et dans le domaine du test de programmes de transformations de modèles. Les paragraphes suivants détaillent trois perspectives dans des domaines différents.

Au cours de la réalisation des études de cas, nous avons fréquemment intégré du code directement dans les méta-modèles afin d'encapsuler de la sémantique ou des traitements. Cette possibilité offerte par Kermeta est un avantage indéniable mais présente

un inconvénient : il y a un risque de pollution des méta-modèles par un grand nombre de traitements dont la pertinence peut être discutable. Afin d'éviter cela, dans le domaine de la méta-modélisation, il existe des opérateurs qui permettent de créer de nouveaux packages par composition d'un package existant avec un ensemble d'éléments nouveaux (l'opérateur *merge* utilisé pour la construction d'UML par exemple). Malheureusement, ces opérateurs sont uniquement structurels et leur implantation dans Kermeta nécessite de leur donner une sémantique opérationnelle. Dans le domaine de la programmation, des opérateurs comme les classboxes [BDNW05] ont été définis pour permettre d'enrichir ou de redéfinir un ensemble de classes avec une portée locale. L'intégration d'un mécanisme de ce genre dans Kermeta fournirait un très bon moyen de garder les avantages de l'encapsulation sans risquer une pollution des méta-modèles utilisés.

Le langage Kermeta permet de spécifier les principaux éléments d'un langage dédié : sa structure (syntaxe abstraite), des contraintes et sa sémantique opérationelle. Nous avons également travaillé sur un langage permettant d'associer une syntaxe textuelle concrète à une syntaxe abstraite puis d'assurer la génération de parseurs et de "pretty-printer" [MFF+06]. Dans des travaux futurs nous souhaitons construire un environnement complet pour la définition et l'utilisation de langages de modélisations dédiés. Les deux principaux problèmes de ce domaine sont la spécification de syntaxes graphiques et la génération d'environnements de développement et d'exécution personalisable et intuitifs. L'interopérabilité entre les différents langages dédiés et donc la cohérence générale d'un projet utilisant plusieurs langages dédiés serait assurée par les fondations communes des différents langages dans l'environnement Kermeta.

Dans le domaine de la transformation de modèles, les travaux que nous avons présentés ne sont qu'un premier pas vers un processus de test complet et automatisé des transformations de modèles. Une étude comparative du pouvoir de détection d'erreurs des critères de test proposés reste nécessaire afin d'identifier les critères qui offrent le meilleur compromis entre le pouvoir de détection d'erreurs et la taille des suites de test. L'algorithme de génération de modèles basé sur l'utilisation d'un algorithme bactériologique semble être une technique prometteuse pour la génération automatique de modèles de test. Nous sommes actuellement en train de l'évaluer par la pratique afin d'ajuster les opérateurs de mutation utilisés et les paramètres de l'algorithme. Le problème de l'oracle mérite également une attention particulière en raison de la complexité des sorties d'une transformation de modèle.

Une évolution possible des techniques de test de transformations de modèles est une spécialisation pour cibler différents types de transformations de modèles. En pratique, en fonction des besoins on utilise en effet différents types de transformations dont les caractéristiques peuvent être exploitées lors du test. Par exemple :

- un "refactoring" est une transformation dans laquelle la structure d'un modèle change sans que sa sémantique ne soit affectée (transformation iso-fonctionelle).
- une migration est également une transformation iso-fonctionelle mais pour laquelle les formalismes d'entrée et de sortie sont différents.
- un rafinnement est une transformation dans laquelle les éléments en entrée sont conservés et au cours de laquelle de nouveaux éléments sont ajoutés au modèles.
- etc ...

D'un point de vue pratique, le fait de développer des techniques de test ad-hoc pour chaque transformation de modèles n'est pas rentable. A l'autre extrême, il semble peu probable que des technique générales puisse suffire à la validation de toute transformation de modèles. Il est donc nécessaire de chercher un compromis entre ces deux extrêmes.

Annexe A

Sémantique du langage Kermeta

Cette annexe présente la sémantique naturelle du langage Kermeta présenté au chapitre 3. Le principe de la sémantique naturelle est de représenter l'exécution des programmes comme un système logique. En pratique, le comportement d'un programme impératif s'exprime par la modification d'un état. Le premier élément pour définir la sémantique de Kermeta est donc de choisir une représentation pour cet état. Ensuite, des règles de déduction correspondant à chaque construction du langage permettent de décrire l'exécution des programmes.

A.1 Structure de données et notations

Cette section définit les structures de données et notations que nous utilisons dans la suite.

L'état interne d'un programme Kermeta est une structure composée de références et d'objets. Tous les objets Kermeta sont alloués dans un unique espace mémoire. Les références sont soit des propriétés d'objets soit des variables. Pour représenter l'état d'exécution, nous utiliserons un tuple $S = (\Gamma, \Omega, \rho)$ dans lequel Γ est un ensemble de variables, Ω est un ensemble d'objets et $\alpha : V \rightarrow O$ un ensemble de relations entre variables et objets. En plus de cette représentation il est nécessaire de définir une notation pour décrire les références inter-objets. Ces références sont la manifestation des propriétés, associations et compositions définies sur les classes (cf section 3.3.1).

En Kermeta, chaque objet possède un ensemble de propriétés qui peuvent définir ou non une relation de composition, être ou non multiples et avoir ou non une propriété opposée. Afin d'améliorer la lisibilité des règles de déductions présentées, dans la suite nous avons choisi une représentation compacte pour ces différentes situations :

- $o \xrightarrow{p} x$: l'objet o possède une propriété p dont la valeur est x.
- $o \xrightarrow{\diamond p} x$: l'objet o possède une propriété composite p dont la valeur est x.
- $o \xrightarrow{\diamond} x$: l'objet o est contenu par l'objet x. Cette relation impose qu'il existe une propriété composite de x qui fasse référence à o ($\exists q / x \xrightarrow{\diamond q} o$).
- $o \xrightarrow{p} \{x_1, \ldots, x_n\}$: l'objet o possède une propriété multiple p qui fait référence a l'ensemble d'objets $\{x_1, \ldots, x_n\}$.

– $o \overset{p,q}{\longleftrightarrow} x$: l'objet o possède une propriété p dont la valeur est x. La propriété p possède une propriété opposée q. La valeur de la propriété q de l'objet x est o.
– ...

Dans les règles de déduction, nous avons besoin de spécifier de nouveaux états par la modification d'états existants. Nous utilisons pour cela la notation $s \vdash m_1, \ldots, m_n \mapsto s'$ qui se lit : l'état s' est obtenu en appliquant les modifications m_1 à m_n à l'état s. En pratique, les modifications peuvent être la modification de la valeur d'une variable ou la modification de la valeur d'une propriété d'un objet. On écrira par exemple :
– $os \vdash \rho(v) = x \mapsto s'$ pour exprimer le fait que l'état s' est obtenu à partir de l'état s par la modification de la valeur de la variable v.
– $os \vdash o \xrightarrow{p} x \mapsto s'$ pour spécifier que dans s' la valeur de la propriété p de l'objet o est x.

Les derniers point de notation que nous abordons dans cette section sont l'évaluation d'une expression dans un état et la modification d'un état par une instruction. Du fait que Kermeta ne distingue pas les notions d'instruction et d'expressions, toute les expression Kermeta conduisent potentiellement à la modification de l'état. On notera :
– $s \vdash e \rightarrow o$: pour représenter le fait que l'expression e a la valeur o dans l'état s.
– $s \vdash i \rightsquigarrow s'$: pour représenter que dans l'état s, l'exécution de l'instruction i conduit à un état s'.
– $s \vdash e \rightarrow o \rightsquigarrow s'$: pour représenter, en utilisant les deux notation précédentes, que l'évaluation de l'expression e rend la valeur o et conduit à l'état s' :

A.2 Sémantique des expressions

Cette section détaille la sémantique de chaque construction du langage Kermeta.

A.2.1 Structures de contrôles

Les structures de contrôle de Kermeta sont la séquence (ou bloc), la conditionelle et la boucle. Leur sémantique est classique des langages impératifs. La règle A.1 précise la sémantique des blocs Kermeta. Les instructions du bloc sont exécutées séquentiellement et la valeur retournée est la valeur retournée par la dernière instruction du bloc.

$$\frac{s_0 \vdash e_1 \rightarrow o_1 \rightsquigarrow s_1, \ldots, s_{n-1} \vdash e_n \rightarrow o_n \rightsquigarrow s_n}{s_0 \vdash \text{do } e_1, \ldots, e_n \text{ end} \rightarrow o_n \rightsquigarrow s_n} \tag{A.1}$$

Les règles A.2 et A.3 détaillent l'interprétation des expressions conditionnelles de Kermeta. La règle A.2 correspond à l'exécution du bloc "then" lorsque la condition est vraie et la règle A.3 correspond à l'exécution du bloc "else" dans le cas où la condition est fausse.

$$\frac{s \vdash \mathit{cond} \rightarrow \mathit{true} \rightsquigarrow s_0 \ , \ s_0 \vdash e_1 \rightarrow o_1 \rightsquigarrow s_1, \ s_0 \vdash e_2 \rightarrow o_2 \rightsquigarrow s_2}{s \vdash \text{if } \mathit{cond} \text{ then } e_1 \text{ else } e_2 \text{ end} \rightarrow o_1 \rightsquigarrow s_1} \tag{A.2}$$

$$\frac{s \vdash cond \rightarrow false \rightsquigarrow s_0,\ s_0 \vdash e_1 \rightarrow o_1 \rightsquigarrow s_1,\ s_0 \vdash e_2 \rightarrow o_2 \rightsquigarrow s_2}{s \vdash \text{if } cond \text{ then } e_1 \text{ else } e_2 \text{ end} \rightarrow o_2 \rightsquigarrow s_2} \tag{A.3}$$

Les boucles Kermeta comportent une partie initialisation, une condition d'arrêt et un corps. La règle A.4 présente l'initialisation d'une boucle : l'expression d'initialisation est évaluée puis, tant que la condition d'arrêt est fausse, le corps de la boucle est exécuté. Les règles A.5 et A.6 détaillent respectivement l'exécution d'une itération et l'arrêt de la boucle.

$$\frac{s \vdash init \rightsquigarrow s',\ s' \vdash while(\neg cond, e) \rightsquigarrow s''}{s \vdash \text{from } init \text{ until } cond \text{ loop } exp \text{ end} \rightarrow \perp \rightsquigarrow s''} \tag{A.4}$$

$$\frac{s \vdash cond \rightarrow true \rightsquigarrow s',\ s' \vdash e \rightsquigarrow s'',\ s'' \vdash while(cond, e) \rightsquigarrow s'''}{s \vdash while(cond, e) \rightsquigarrow s'''} \tag{A.5}$$

$$\frac{s \vdash cond \rightarrow false \rightsquigarrow s'}{s \vdash while(cond, e) \rightsquigarrow s'} \tag{A.6}$$

A.2.2 Variables

La définition de variable locales permet d'ajouter de nouvelles variables dans l'environnement d'exécution. En Kermeta, les variables sont déclarées avec un type. Par défaut les variables sont initialisées avec la valeur *void* (ou $\perp$) mais il est possible de spécifier une expression d'initialisation. La règle A.7 présente la déclaration d'une variable sans code d'initialisation. La règle A.8 présente l'initialisation de la variable définie par une expression.

$$\frac{s = (\Gamma, \Omega, \rho),\ s' = (\Gamma', \Omega, \rho'),\ \Gamma' = \Gamma \cup \{v\},\ \rho' = \rho \cup \{v \mapsto \perp\}}{s \vdash \text{var } v : \tau_v \rightarrow \perp \rightsquigarrow s'} \tag{A.7}$$

$$\frac{s \vdash e \rightarrow o \rightsquigarrow s',\ s' \vdash \text{var } v : \tau_v \rightsquigarrow s'',\ s'' \vdash \rho(v) = o \mapsto s'''}{s \vdash \text{var } v : \tau_v \text{ init } e \rightarrow \perp \rightsquigarrow s'''} \tag{A.8}$$

L'ensemble des variables définies dans l'environnement d'exécution peuvent être lues par les programmes. La lecture d'une variable n'entraîne pas de modification de l'état. La règle A.9 détaille la sémantique de l'accès à une variable.

$$\frac{s = (\Gamma, \Omega, \rho),\ v \in \Gamma,\ \rho(v) = o}{s \vdash v \rightarrow o} \tag{A.9}$$

A.2.3 Affectation et cast

En Kermeta, il est possible d'affecter les variables et les propriétés des objets. Comme le montre la règle A.10 l'affectation des variables est classique : la partie droite de l'affectation est évaluée et une référence à l'objet résultat est associée à la variable.

$$\frac{s \vdash e \rightarrow o \rightsquigarrow s',\ s' \vdash \rho(v) = o \mapsto s''}{s \vdash v := e \rightarrow o \rightsquigarrow s''} \tag{A.10}$$

Les règles A.11 et A.12 détaillent le "cast dynamique" disponible en Kermeta. Cette construction permet de réaliser une affectation conditionnée par le type de la variable en partie gauche et le type dynamique de l'objet retourné par l'expression en partie droite. Si les types sont compatibles (cas de la règle A.11) alors l'affectation a lieu normalement. Dans le cas contraire, la valeur $\perp$ est affectée à la variable (cas de la règle A.12). Nous ne l'avons pas détaillé dans la suite, mais cette construction est également disponible pour l'affectation de propriétés.

$$\frac{s \vdash e \rightarrow o \rightsquigarrow s',\ s' \vdash \rho(v) = o \mapsto s'',\ \tau_o \preceq \tau_v}{s \vdash v := e \rightarrow o \rightsquigarrow s''} \tag{A.11}$$

$$\frac{s \vdash e \rightarrow o \rightsquigarrow s',\ s' \vdash \rho(v) = \perp \mapsto s'',\ \tau_o \not\preceq \tau_v}{s \vdash v := e \rightarrow \perp \rightsquigarrow s''} \tag{A.12}$$

L'affectation des propriétés des objets est un point intéressant de la sémantique de Kermeta. Les propriétés étant utilisées à la fois pour représenter des références en objets, des associations et des compositions, lors de l'affectation il est nécessaire d'assurer la cohérence globale des objets manipulés.

La règle A.13 présente le déroulement général de l'affectation d'une propriété : les expressions en partie gauche et droite de l'affectation sont évaluées puis la valeur de la propriété p de l'objet o est modifiée. Nous avons utilisé la notation $o.p \leftarrow o'$ pour représenter la modification de la propriété. En fonction de la nature de la propriété p, les règles A.14 à A.20 précisent le déroulement de la modification.

$$\frac{\begin{array}{c} s \vdash lhs \rightarrow o \rightsquigarrow s'\ \ s' \vdash rhs \rightarrow o' \rightsquigarrow s'', \\ s'' \vdash o.p \leftarrow o' \rightsquigarrow s''' \end{array}}{s \vdash lhs.p := rhs \rightarrow o' \rightsquigarrow s'''} \tag{A.13}$$

La règle A.14 précise la sémantique de l'affectation d'une référence simple. Dans ce cas, l'affectation consiste simplement à mettre à jour la référence associée à la propriété.

$$\frac{\begin{array}{c} s \vdash o \xrightarrow{p} x \\ s \vdash o \xrightarrow{p} o' \mapsto s' \end{array}}{s \vdash o.p \leftarrow o' \rightsquigarrow s'} \tag{A.14}$$

La règle A.15 montre l'affectation d'une propriété composite. Dans ce cas, le conteneur des objets doit être mis à jour afin d'assurer que chaque objet n'ait qu'un seul conteneur. Si un objet x était associé par la propriété p de l'objet o alors son conteneur est mis à $\perp$. Le conteneur de l'objet o' est modifié et devient o (lors du changement de conteneur, si o' avait un conteneur c alors la relation entre o' et c est détruite).

$$\frac{\begin{array}{c} s \vdash o \xrightarrow{\diamond p} x \\ s \vdash x \xrightarrow{\diamond} \perp, o \xrightarrow{p} o', o' \xrightarrow{\diamond} o \mapsto s' \end{array}}{s \vdash o.p \leftarrow o' \rightsquigarrow s'} \tag{A.15}$$

La règle A.16 spécifie la sémantique de l'affectation d'une propriété participant à une association. En Kermeta les associations sont représentées par deux propriétés opposées (ici p et q). Comme le montre la règle, l'affectation d'une des propriété provoque la mise à jour de l'autre extrémité de l'association. De cette manière, l'affectation conserve l'intégrité des associations.

$$\frac{s \vdash o \xleftrightarrow{p,q} x,\ s \vdash y \xleftrightarrow{p,q} o' \qquad s \vdash x \xleftrightarrow{q,p} \bot, y \xleftrightarrow{p,q} \bot, o \xleftrightarrow{p,q} o' \mapsto s'}{s \vdash o.p \leftarrow o' \rightsquigarrow s'} \tag{A.16}$$

Les règles A.17 et A.18 présentent la sémantique de l'affectation d'une propriété composite participant à une association. Ici encore, l'intégrité de l'association et les contraintes liées à la composition sont assurées par l'affectation.

$$\frac{s \vdash o \xleftrightarrow{\diamond p,q} x,\ s \vdash y \xleftrightarrow{\diamond p,q} o' \qquad s \vdash x \xleftrightarrow{q,\diamond p} \bot, x \xrightarrow{\diamond} \bot, y \xleftrightarrow{\diamond p,q} \bot, o \xleftrightarrow{\diamond p,q} o', o' \xrightarrow{\diamond} o \mapsto s'}{s \vdash o.p \leftarrow o' \rightsquigarrow s'} \tag{A.17}$$

$$\frac{s \vdash o \xleftrightarrow{p,\diamond q} x,\ s \vdash y \xleftrightarrow{p,\diamond q} o' \qquad y \xrightarrow{\diamond} \bot, y \xleftrightarrow{p,\diamond q} \bot, x \xleftrightarrow{\diamond q,p} \bot, o \xleftrightarrow{p,\diamond q} o', o \xrightarrow{\diamond} o' \mapsto s'}{s \vdash o.p \leftarrow o' \rightsquigarrow s'} \tag{A.18}$$

Enfin, les règles A.19 et A.20 détaillent l'affectation de propriétés dont les propriété opposées sont multiples. La cas de l'affectation de propriété multiple n'est pas traité ici car en Kermeta ce type d'affectation n'est pas valable.

$$\frac{s \vdash \{o_1, \ldots, o_n\} \xleftrightarrow{p,q} x,\ s \vdash \{y_1, \ldots, y_n\} \xleftrightarrow{p,q} o' \qquad s \vdash x \xleftrightarrow{q,p} \{o_1, \ldots, o_n\} \setminus \{o_i\}, o' \xleftrightarrow{q,p} \{o_i, y_1, \ldots, y_n\} \mapsto s'}{s \vdash o_i.p \leftarrow o' \rightsquigarrow s'} \tag{A.19}$$

$$\frac{s \vdash \{o_1, \ldots, o_i, \ldots, o_n\} \xleftrightarrow{p,\diamond q} x,\ s \vdash \{y_1, \ldots, y_n\} \xleftrightarrow{p,\diamond q} o' \qquad s \vdash x \xleftrightarrow{q,p} \{o_1, \ldots, o_n\} \setminus \{o_i\}, o' \xleftrightarrow{q,p} \{o_i, y_1, \ldots, y_n\}, o_i \xrightarrow{\diamond} o' \mapsto s'}{s \vdash o_i.p \leftarrow o' \rightsquigarrow s'} \tag{A.20}$$

A.2.4 Appel de propriétés et d'opérations

La lecture des propriétés et l'appel d'opération Kermeta est classique de ce qui existe pour la lecture des attributs et l'appel de méthode dans les langages orientés-objets.

La règles A.21 détaille la lecture d'une propriété de multiplicité 1. Dans ce cas la valeur retournée est directement la valeur de la propriété. La règle A.22 présente la lecture d'une propriété multiple. Dans ce cas, la valeur renvoyée est une collection contenant les valeurs de la propriété.

$$\frac{s \vdash exp \rightarrow o \rightsquigarrow s',\ s' \vdash o \xrightarrow{p} x}{s \vdash exp.p \rightarrow x \rightsquigarrow s'} \tag{A.21}$$

$$\frac{s \vdash exp \rightarrow o \rightsquigarrow s',\ s' \vdash o \xrightarrow{p} \{x_1, \ldots, x_n\}}{s \vdash exp.p \rightarrow \{x_1, \ldots, x_n\} \rightsquigarrow s'} \tag{A.22}$$

La règle A.23 présente le déroulement d'un appel d'opération. En premier lieu, l'expression correspondant à la cible de l'appel est évaluée. Ensuite les expressions correspondant aux paramètres effectifs sont évaluées. Dans la règle $ops(\tau_o)$ représente l'ensemble des opérations disponibles sur le type de l'objet o. Pour évaluer l'opération, un nouvel état (ou contexte) est créé à partir du type τ_o ($s_n \vdash init(\tau_o) \mapsto s_m$). Cet état contient les variables disponibles dans le type de o. La liaison entre les paramètres formels et les paramètres effectifs est alors réalisée en créant les références dans l'état s_m (En Kermeta tous les appels sont réalisés par référence et non par valeur). La variable *result* est initialisée à $\perp$. Le corps de l'opération place le résultat de l'appel d'opération dans cette variable *result*.

$$\frac{\begin{array}{c} s \vdash exp \rightarrow o \rightsquigarrow s_0,\ s_0 \vdash e_1 \rightarrow o_1 \rightsquigarrow s_1, \ldots, s_{n-1} \vdash e_n \rightarrow o_n \rightsquigarrow s_n, \\ m(p_1, \ldots, p_n) \in ops(\tau_o),\ s_n \vdash init(\tau_o) \mapsto s_m, \\ s_m \vdash \rho(p_1) = e_1, \ldots, \rho(p_n) = e_n,\ \rho(result) = \perp \mapsto s'_m \\ s'_m \vdash body(m) \rightsquigarrow s''_m,\ s''_m \vdash \rho(result) \rightarrow o' \end{array}}{s \vdash exp.m(e_1, \ldots, e_n) \rightarrow o'} \tag{A.23}$$

A.2.5 Fonctions

La dernière construction de Kermeta que nous considérons ici est l'application de fonctions. La particularité des fonctions Kermeta est qu'elles sont évaluées dans l'environnement de leur définition. Lors de la définition d'une fonction, il est nécessaire de stocker l'environnement courant. Afin de représenter cela, nous notons f_s une fonction f définie dans l'état s. La règle A.24 présente la définition et l'application d'une fonction. La fonction est définie dans l'état s est appelée dans l'état s_o. La première étape est l'évaluation des expressions correspondant aux paramètres effectifs. Ensuite, l'état dans lequel le corps de la fonction est évalué est créé à partir de l'état dans lequel la fonction a été définie en liant les paramètres formels de la fonction aux paramètres effectifs calculés.

$$\frac{\begin{array}{c} s_f \vdash \text{function } \{p_1 : \tau_1, \ldots, p_n : \tau_n \mid exp\} \rightarrow f_s \\ s_0 \vdash e_1 \rightarrow o_1 \rightsquigarrow s_1, \ldots,\ s_{n-1} \vdash e_n \rightarrow o_n \rightsquigarrow s_n, \\ s_f \vdash \rho(p_1) = e_1, \ldots, \rho(p_n) = e_n \mapsto s'_f \\ s'_f \vdash exp \rightarrow o \end{array}}{s_0 \vdash f_s(e_1, \ldots, e_n) \rightarrow o} \tag{A.24}$$

Glossaire

Contrat (Design by Contracts) Dans le domaine de la programmation orientée-objet, les contrats sont des pré-conditions et post-conditions associées aux opérations et des invariants associés au classes. Initialement le terme et la méthodologie "Design by Contracts" ont été introduits par Bertrand Meyer et implantés dans le langage Eiffel. La section 3.3.6 montre comment les contrats sont implantés dans Kermeta.

Critère de test Critère qui doit être vérifié par les données de test utilisées pour valider un programme. La section 2.2.2 donne une définition et des exemples de critères de test.

Diagnostic Activité qui consiste à localiser les erreurs détectées lors du test d'un programme. Dans la section 2.2.5 nous présentons des techniques de diagnostic semi-automatiques.

Donnée de test Donnée d'entrée d'un programme utilisé pour le test.

Essential MOF (EMOF) Essential Meta-Object Facility est un langage pour la définition de méta-modèles standardisés par l'Object Management Group (OMG). La section 2.1.3 présente le langage EMOF dans le détail.

Ingénierie des modèles L'ingénierie des modèles regroupe les activités d'ingénierie relatives à la définition et la manipulation de modèles : définition de langages de modélisation, écriture de transformation de modèles, etc.

Ingénierie Dirigée par les Modèles (IDM) L'ingénierie dirigée par les modèles regroupe les techniques de génie logiciel qui s'appuient sur l'utilisation de modèles.

Meta-modèle Un méta-modèle est le modèle d'un langage de modélisation (cf. section 2.1.2).

Meta-meta-modèle Un méta-méta-modèle est le modèle d'un langage permettant la spécification de méta-modèles (cf. section 2.1.2).

Meta-Object Facilities (MOF) Famille de langages pour la définition de méta-modèles standardisés par l'Object Managment Group (OMG).

Modèle Un modèle est une représentation d'un système qui peut se substituer à ce système dans un but précis. D'un point de vue pratique, un modèle est un graphe d'objets conforme à un méta-modèle c'est-à-dire appartenant à un langage de méta-modélisation (cf. section 2.1.2).

Modèle de test Un modèle de test est une donnée de test pour un programme de transformation de modèles.

Object Constraint Language (OCL) OCL est un langage de contrainte défini par l'Object Managment Group (OMG). La section 2.1.4 présente les grandes caractéristiques de ce langage.

Oracle Dans le contexte du test de logiciel, l'oracle est une fonction qui permet de vérifier si le comportement observé d'un programme est conforme à son comportement attendu. La section 2.2.3 donne une définition plus formelle de cette fonction.

Suite de test Une suite de test est un ensemble de données de test.

Test de logiciel Le test de logiciel est une technique de validation dont l'objectif est de détecter des erreurs dans les programmes. La section 2.2 présente le test de logiciel dans le détail.

Transformation de modèles Une transformation de modèles est un programme dont les entrées et les sorties sont des modèles.

Bibliographie

[ABDL02] G. Antoniol, L. Briand, M. DiPenta, and Y. Labiche. A case study using the round-trip strategy for state-based class testing. In *Proceedings of ISSRE'02 (Int. Symposium on Software Reliability Engineering)*, Annapolis, MD, USA, 2002. IEEE Computer Society.

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[AFGC03] A. Andrews, R. France, S. Ghosh, and G. Craig. Test adequacy criteria for UML design models. *Journal of Software Testing, Verification and Reliability*, 13(2) :95–127, april-june 2003.

[AHLW95] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. *Proc. of the 6th IEEE International Symposium on Software Reliability Engineering*, pages 143–151, October 1995.

[And] AndroMDA. Andromda. http ://www.andromda.org/.

[ANT] ANTLR. Another tool for language recognition. http ://www.antlr.org/.

[AP04] Marcus Alanen and Ivan Porres. Coral : A metamodel kernel for transformation engines. In D. H. Akerhurst, editor, *Proceedings of the Second Europena Workshop on Model Driven Architecture (MDA)*, pages 165–170, Canterbury, Kent CT2 7NF, United Kingdom, Sep 2004. University of Kent.

[BDNW05] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes : Controlling visibility of class extensions. *Computer Languages, Systems and Structures*, 31(3-4) :107–126, may 2005.

[Bei90] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 2nd edition edition, 1990.

[BFJLT05] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. Automatic test cases optimization : a bacteriologic algorithm. *IEEE Software*, 22(2) :76–82, mars 2005.

[BFJT02a] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. Automatic test cases optimization using a bacteriological adaptation model : Application to .net components. *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, 2002.

[BFJT02b] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. Genes and bacteria for automatic test cases optimization in the .net environment. In *procedings of the Thirteenth International Symposium on Software Reliability engineering (ISSRE)*, november 2002.

[BFJT05] B. Baudry, F. Fleurey, J-M. Jézéquel, and Y. Le Traon. From genetic to bacteriological algorithms for mutation-based testing. *Software, Testing, Verification & Reliability journal (STVR)*, 15(2) :73–96, juin 2005.

[BFLT06] Benoit Baudry, Franck Fleurey, and Yves Le Traon. Improving test suites for efficient fault localization. In *In proceedings of the 28th International Conference on Software Engineering (ICSE 06)*. ACM, 2006. selection : 9%.

[BFS+06] Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry, and Yves Le Traon. Metamodel-based test generation for model transformations : an algorithm and a tool. In *ISSRE'06*. IEEE, 2006.

[BGM91] Gilles Bernot, Marie Claude Gaudel, and Bruno Marre. Software testing based on formal specifications : a theory and a tool. *Softw. Eng. J.*, 6(6) :387–405, 1991.

[BGN+04] Sven Burmester, Holger Giese, Jörg Niere, Matthias Tichy, Jörg P. Wadsack, Robert Wagner, Lothar Wendehals, and Albert Zündorf. Tool integration at the meta-model level within the fujaba tool suite. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3) :203–218, August 2004.

[BHSPLB03] Franck Barbier, Brian Henderson-Sellers, Annig Le Parc-Lacayrelle, and Jean-Michel Bruel. Formalization of the whole-part relationship in the unified modeling language. *IEEE Trans. Softw. Eng.*, 29(5) :459–470, 2003.

[Bin99] Robert V. Binder. *Testing Object-Oriented Systems : Models, Patterns, and Tools*. Addison Wesley, 1999.

[BL02] L. Briand and Y. Labiche. A uml-based approach to system testing. *Journal of Software and Systems Modeling*, 2002.

[BR88] Alan Bawden and Jonathan Rees. Syntactic closures. In *LFP '88 : Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 86–95, New York, NY, USA, 1988. ACM Press.

[Bra03] Peter Braun. Metamodel-based Integration of Tools. In *Proceeding of ESEC/FSE 2003, TIS 2003 Workshop on Tool Integration in System Development*, 2003.

[BRST05] Jean Bézivin, Bernhard Rumpe, Andy Schürr, and Laurence Tratt. Model transformations in practice workshop, 2005.

[BTLJ00] Benoit Baudry, Yves Le Traon, Hann Vu Le, and Jean-Marc Jézéquel. Building trust into oo components using a genetic analogy. In *ISSRE'2000 (International Symposium on Software Reliability Engineering 2000)*, San Jose, CA, 2000.

[CAR] CAROLL. The carroll research program. http ://www.carroll-research.org/.

[CL02] Y. Cheon and G.T. Leavens. A simple and practical approach to unit testing : The jml and junit way. In *ECOOP*, 2002.

[Cla76] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transaction on Software Enginnering*, 2(3) :215–22, 1976.

[CM98] C. Consel and R. Marlet. Architecturing software using a methodology for language development. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of the 10th International Symposium on Programming Language Implementation and Logic Programming*, volume 1490 of *LNCS*, pages 170–194, Pisa, Italy, septembre 1998.

[Com] Compuware. Optimalj. http ://www.compuware.com/products/optimalj/.

[Dav03] James Davis. Gme : the generic modeling environment. In *OOPSLA '03 : Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 82–83, New York, NY, USA, 2003. ACM Press.

[DF93] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *FME '93 : Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, pages 268–284, London, UK, 1993. Springer-Verlag.

[DLS78] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection : Help for the practicing programmer. *IEEE Computer*, April 1978.

[DTKG+05] Trung T. Dinh-Trong, Nilesh Kawane, Sudipto Ghosh, Robert B. France, and Anneliese Amschler Andrews. A tool-supported approach to testing uml design models. In *ICECCS*, pages 519–528, 2005.

[Dum] Cedric Dumoulin. Modtransf. http ://www.lifl.fr/west/modtransf/.

[ECo] ECore. The eclipse modeling framework project home page. http ://www.eclipse.org/emf.

[EFB+05] J. Estublier, J-M. Favre, J. Bézivin, L. Duchien, R. Marvie, S. Gérard, B. Baudry M. Bouzhegoub, J-M. Jézéquel, M. Blay, and M. Riveil. Action Spécifique CNRS sur l'Ingénierie Dirigée par les Modèles. Rapport de synthèse 1.1.2, CNRS, janvier 2005.

[EGdL+05] Karsten Ehrig, Esther Guerra, Juan de Lara, Laszló Lengyel, Tihamér Levendovszky, Ulrike Prange, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Model transformation by graph transformation : A comparative study. In *MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005)*, 2005.

[EGR01] Sebastian G. Elbaum, David Gable, and Gregg Rothermel. The impact of software evolution on code coverage information. In *ICSM*, pages 170–179, 2001.

[EHJS01] J. Eagan, M. J. Harrold, J. Jones, and J. Stasko. Visually encoding program test information to find faults in software. Technical report, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, June 2001.

[EMF] EMF. The eclipse modeling framework project home page. http ://www.eclipse.org/emf.

[EMR00] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. Technical report, Computer Science Departmeent, Oregon State University, February 2000.

[EMR02] S. Elbaum, A.G. Malishevsky, and G. Rothermel. Test case prioritization : A family of empirical studies. *proceedings of IEEE Transactions on Software Engineering*, pages 159–182, February 2002.

[FSB04] Franck Fleurey, Jim Steel, and Benoit Baudry. MDE and validation : Testing model transformations. In *Proc. of the SIVOES-Modeva workshop, SIVOES (Specification Implementation and Validation Of Embedded Systems)-MoDeVa (Model Design and Validation)*, Rennes, novembre 2004.

[GB] Erich Gamma and Kent Beck. http ://junit.org.

[GBR03] Martin Gogolla, Jörn Bohling, and Mark Richters. Validation of uml and ocl models by automatic snapshot generation. In *UML*, pages 265–279, 2003.

[GFB+03] S. Ghosh, R. B. France, C. Braganza, N. Kawane, A. Andrews, and O. Pilskalns. Test adequacy assessment for UML design model testing. In *International Symposium on Software Reliability Engineering*, 2003.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, 1995.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80 : the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[GSC+04] J. Greenfield, K. Short, S. Cook, S. Kent, and J. Crupi. *Software Factories : Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley ; 1st edition, 2004. *ISBN : 0471202843*.

[GSCK04] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories : Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, August 2004.

[HGS93] Mary Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3) :270–285, July 1993.

[HJGP99] Wai-Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac'h. Umlaut : an extendible uml transformation framework. In *Automated Software Engineering, ASE'99*, Florida, October 1999.

[HL03] Reiko Heckel and Marc Lohmann. Towards model-driven testing. *Electr. Notes Theor. Comput. Sci.*, 82(6), 2003.

[Hub] Richard Hubert. Arcstyler – the architectural ide for mda. http ://www.io-software.com/.

[JHD96] B. Jones, H.Sthamer, and D.Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal 11*, pages 299–306, September 1996.

[JHS01] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization for fault localization. *Proceedings of the Workshop on Software Visualization*, 2001.

[JHS02] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. *Proceedings of the 24th International Conference on Software Engineering*, 2002.

[JM99] T. Jéron and P. Morel. Test generation derived from model-checking. In N. Halbwachs and D. Peled, editors, *CAV'99, Trento, Italy*, pages 108–122. Springer-Verlag, July 1999.

[Jon98] B. F. Jones. A strategy for using genetic algorithms to automate branch and fault-based testing. *The Computer Journal*, 2(41) :98–107, 1998.

[Jéz96] Jean-Marc Jézéquel. *Object-oriented software engineering with Eiffel*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

[Kö4] J.M. Küster. Systematic validation of model transformations. In *Proceedings of Workshop in Software Model Engineering associated to UML'04*, Lisbon, Portugal, 2004.

[Ker] Kermeta. The kermeta project home page. http ://www.kermeta.org.

[KF06] Jacques Klein and Franck Fleurey. Tissage d'aspects comportementaux. In *Langages et Modèles à Objets : LMO'06*, Nimes, France, mars 2006.

[KLPU04] Nikolai Kosmatov, Bruno Legeard, Fabien Peureux, and Mark Utting. Boundary coverage criteria for test generation from formal models. In *ISSRE '04 : Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 139–150, Washington, DC, USA, 2004. IEEE Computer Society.

[Kor92] B. Korel. Dynamic method for software test data generation. *Software Testing, Verification and Reliability*, 2(4) :203–13, 1992.

[KSLB03] Gabor Karsai, Janos Sztipanovits, Ákos Lédeczi, and Ted Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1) :145–164, 2003.

[LBM+01] Akos Lédeczi, Arpad Bakay, Miklos Maroti, Peter Volgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gabor Karsai. Composing domain-specific design environments. *Computer*, 34(11) :44–51, 2001.

[LPU02] Bruno Legeard, Fabien Peureux, and Mark Utting. Automated boundary testing from z and b. In *FME '02 : Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, pages 21–40, London, UK, 2002. Springer-Verlag.

[LS05] Michael Lawley and Jim Steel. Practical declarative model transformation with tefkat. In *Model Transformations In Practice Workshop*, Montego Bay, Jamaica, octobre 2005.

[LZG05] Y. Lin, J. Zhang, and J. Gray. A testing framework for model transformations. *Model-driven Software Development - Research and Practice in Software Engineering, Springer*, 2005.

[Mey] Bertrand Meyer. Eiffel agents. http ://archive.eiffel.com/doc/manuals/language/agent/agent.pdf.

[Mey92] B. Meyer. Applying "design by contract". *Computer (IEEE)*, pages 40–51, October 1992.

[MFF+06] Pierre-Alain Muller, Franck Fleurey, Frédéric Fondement, Michel Hassenforder, Rémi Schnekenburger, Sébastien Gérard, and Jean-Marc Jézéquel. Model-driven analysis and synthesis of concrete syntax. In *Proceedings of MODELS/UML'2006*, volume to be published of *LNCS*, pages –. Springer, octobre 2006.

[MFJ05] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In S. Kent L. Briand, editor, *Proceedings of MODELS/UML'2005*, volume to be published of *LNCS*, pages –, Montego Bay, Jamaica, octobre 2005. Springer.

[MFV+05] Pierre-Alain Muller, Franck Fleurey, Didier Vojtisek, Zoé Drey, Damien Pollet, Frédéric Fondement, Philippe Studer, and Jean-Marc Jézéquel. On executable meta-languages applied to model transformations. In *Model Transformations In Practice Workshop*, Montego Bay, Jamaica, octobre 2005. to appear.

[Mil04] Russell Miles. *AspectJ Cookbook*. O'Reilly Media, Inc., 2004.

[MMS01] C.C. Michael, G. Mcgraw, and M.A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12) :1085–1110, December 2001.

[MMSC98] I. MacColl, L. Murray, P. Strooper, and D. Carrington. Specification-based class testing : A case study. In IEEE Computer Society, editor, *2nd International Conference on Formal Engineering Methods (ICFEM'98)*, 1998.

[MS] Mia-Software. Mia-transformation. http ://www.mia-software.com/.

[MS76] W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Transaction on Software Enginnering*, 2(3) :223–226, 1976.

[MSFB05] P.-A. Muller, P. Studer, F. Fondement, and J. Bézivin. Platform independent web application modeling and development with netsilon. *Journal on Software and Systems Modeling (SoSyM)*, 4(4) :424–442, novembre 2005.

[NF05] Clémentine Nebut and Franck Fleurey. Une méthode de formalisation progressive des exigences basée sur un modèle simulable. In *Langages et Modèles à Objets : LMO'05 (L'Objet logiciel, bases de données, réseaux, RSTI série l'Objet Vol. 11 Nř 1-2/2005)*, pages 145–158, Bern, Switzerland, février 2005.

[NFLJ03] Clémentine Nebut, Franck Fleurey, Yves Le Traon, and Jean-Marc Jézéquel. Requirements by contracts allow automated system testing. In *Proc. of the 14th. IEEE International Symposium on Software Reliability Engineering (ISSRE'03)*, 2003.

[NFLTJ06] Clémentine Nebut, Franck Fleurey, Yves Le Traon, and Jean-Marc Jézéquel. Automatic test generation : A use case driven approach. *IEEE Trans. on Software Engineering*, 32(3) :to appear, mars 2006.

[OA99] A.J. Offutt and A. Abdurazik. Generating tests from uml specifications. In *Proceedings of UML'99 (Unified Modeling Language)*, Fort Collins, CO, USA, 1999.

[OB88] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31(6) :676–686, 1988.

[(OM97] Object Management Group (OMG). Meta Object Facility (MOF) specification. OMG Document ad/97-08-14, septembre 1997.

[(OM02] Object Management Group (OMG). MOF 2.0 Query/Views/Transformations RFP. OMG Document ad/2002-04-10, octobre 2002.

[(OM03] Object Management Group (OMG). The object constraint language (OCL), 2003. http ://www.omg.org/docs/ptc/03-08-08.pdf.

[OMGa] OMG. Meta object facility 1.4 specification. http ://www.omg.org/cgi-bin/doc ?formal/2002-04-03.

[OMGb] OMG. Meta object facility 1.4 specification. http ://www.omg.org/cgi-bin/doc ?formal/2005-09-01.

[OMGc] OMG. Meta object facility 2.0 specification. http ://www.omg.org/cgi-bin/doc ?ptc/2004-10-15.

[Omo] Omondo. Eclipseuml omondo. http ://www.omondo.com.

[PHP99] R. Pargas, M.J. Harrold, and R. Peck. Test-data generation using genetic algorithms. *Journal of software testing, Verification and Reliability*, 9 :263–283, September 1999.

[PJT+02] S. Pickin, C. Jard, Y. Le Traon, T. Jéron, J.-M. Jézéquel, and A. Le Guennec. System test synthesis from uml models of distributed software. In LNCS, editor, *Formal Techniques for Networked and Distributed Systems - FORTE 2002*, Houston, Texas, November 2002.

[Pol05] Damien Pollet. *Une architecture pour les transformations de modèles et la restructuration de modèles UML*. PhD thesis, Univeristy of Rennes 1, Rennes (France), 2005.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice–Hall, Englewood Cliffs, NJ, 1991.

[Rev96] N. Revault. *Principes de méta-modélisation pour l'utilisation de canevas d'applications à objets (MétaGen et les frameworks)*. PhD thesis, Paris, France, Nov. 1996.

[RFG+05] Raghu Reddy, Robert France, Sudipto Ghosh, Franck Fleurey, and Benoit Baudry. Model composition - a signature-based approach. In *Aspect Oriented Modeling (AOM) Workshop*, Montego Bay, Jamaica, octobre 2005. to appear.

[RG00] Mark Richters and Martin Gogolla. Validating UML models and OCL constraints. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 265–277. Springer, 2000.

[RG02] Mark Richters and Martin Gogolla. OCL : Syntax, semantics, and tools. In Tony Clark and Jos Warmer, editors, *Object Modeling with the OCL : The Rationale behind the Object Constraint Language*, pages 42–68. Springer, 2002.

[Ros95] M.L. Rosenzweig. *Species Diversity In Space and Time*. Cambridge University Press, 1995.

[RSBP95] N. Revault, H.A. Sahraoui, G. Blain, and J.-F. Perrot. A metamodeling technique : The métagen system. In *Tools 16 : Tools Europe'95*, pages 127–139. Prentice Hall, Mar. 1995.

[RUCH99] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization : An empirical study. *procedings of International Conference on Software Maintenance*, page 179, September 1999.

[RUCH01] Gregg Rothermel, Roland Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10) :929–948, October 2001.

[RW03] Matthew J. Rutherford and Alexander L. Wolf. A case for test-code generation in model-driven systems. In *GPCE '03 : Proceedings of the second international conference on Generative programming and component engineering*, pages 377–396, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[SA00] Zhong Shao and Andrew W. Appel. Efficient and safe-for-space closure conversion. *ACM Trans. Program. Lang. Syst.*, 22(1) :129–161, 2000.

[SGLS75] Gerald Jay Sussman and Jr. Guy Lewis Steele. Scheme : An interpreter for extended lambda calculus. Technical Report AI Lab Memo AIM-349, MIT AI Lab, December 1975.

[SK97] Janos Sztipanovits and Gabor Karsai. Model-integrated computing. *Computer*, 30(4) :110–111, 1997.

[SKB+95] Janos Sztipanovits, Gabor Karsai, Csaba Biegl, Ted Bapty, Ákos Lédeczi, and Amit Misra. Multigraph : an architecture for model-integrated computing. In *ICECCS*, pages 361–368, 1995.

[SL04] Jim Steel and Michael Lawley. Model-based test driven development of the tefkat model-transformation engine. In *Proceedings of ISSRE04 (International Conference on Software Reliability Engineering)*, St Malo, France, novembre 2004.

[SLTM91] Kari Smolander, Kalle Lyytinen, Veli-Pekka Tahvanainen, and Pentti Marttiin. Metaedit : a flexible graphical environment for methodology modelling. In *CAiSE '91 : Proceedings of the third international conference on Advanced information systems engineering*, pages 168–193, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[Sof] Objecteering Software. Objecteering. http ://www.objecteering.com/.

[Sol] Pathfinder Solutions. Pathmate : Transformation engine. http ://www.pathfindermda.com/.

[StOs00a] R. Soley and the OMG staff. MDA Model-Driven Architecture, novembre 2000. Online presentation http ://www.omg.org/mda/presentations.htm.

[StOS00b] R. Soley and the OMG Staff. Model-Driven Architecture. OMG Document, novembre 2000.

[TH00] Dave Thomas and Andy Hunt. *Programming Ruby : A Pragmatic Programmer's Guide*. Addison-Wesley, 2000.

[TR03] Juha-Pekka Tolvanen and Matti Rossi. Metaedit+ : defining and using domain-specific modeling languages and code generators. In *OOPSLA '03 : Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 92–93, New York, NY, USA, 2003. ACM Press.

[Tra] Laurence Tratt. Qvteclipse. http ://qvtp.org/downloads/qvtp-eclipse/.

[UQ] UMT-QVT. http ://umt-qvt.sourceforge.net/.

[USE] USE. A uml-based specification environment. http ://www.db.informatik.uni-bremen.de/projects/USE/.

[vABM97] Lionel van Aertryck, Marc Benveniste, and Daniel Le M&#233 ;tayer. Casting : A formally based software test generation method. In *ICFEM*

'97 : Proceedings of the 1st International Conference on Formal Engineering Methods, page 101, Washington, DC, USA, 1997. IEEE Computer Society.

[vDKV00] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages : An annotated bibliography. *ACM SIGPLAN Notices*, 35(6) :26–36, juin 2000.

[VJ04] D. Vojtisek and J.-M. Jézéquel. MTL and Umlaut NG - engine and framework for model transformation. *ERCIM News 58*, 58, juillet 2004.

[W3C05a] W3C. XQuery 1.0 : An XML Query Language Version 2.0, novembre 2005. Candidate Recommendation.

[W3C05b] W3C. XSL transformations (XSLT) version 2.0, avril 2005. Working Draft 4.

[WBS01] Joachim Wegener, André Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information & Software Technology*, 14(43) :841–854, 2001.

[Xac] Xactium. Xmf-mosaic. http ://www.xactium.com.

Table des figures

Publications

Journaux internationaux

1. Clémentine Nebut, Franck Fleurey, Yves Le Traon, Jean-Marc Jézéquel. **Automatic Test Generation : A Use Case Driven Approach**. *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 140-155, Mar. 2006.
2. Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. **From genetic to bacteriological algorithms for mutation-based testing**. *Software, Testing, Verification & Reliability journal* (STVR), 15(2) :73–96, 2005.
3. Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. **Automatic test cases optimization : a bacteriologic algorithm**. *IEEE Software*, 22(2) :76–82, March 2005.

Conférences internationales

1. Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry and Yves Le Traon. **Metamodel-based test Generation for Model Transformations : an Algorithm and a Tool**. Accepted for publication at the *IEEE ISSRE 2006* conference.
2. Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. **Model-Driven Analysis and Synthesis of Concrete Syntax**. Accepted for publication at the *MoDELS/UML 2006* conference (Acceptance Ratio : 28%).
3. Benoit Baudry, Franck Fleurey, and Yves Le Traon. **Improving test suites for efficient fault localization**. in proceedings of the 28th *International Conference on Software Engineering (ICSE 06)*, ACM, 2006. (Acceptance Ratio : 9%)
4. Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. **Weaving executability into object-oriented meta-languages**. In proceedings of *MODELS/UML'2005*, LNCS, Montego Bay, Jamaica, October 2005. Springer. (Acceptance Ratio : 27%)
5. Franck Fleurey, Benoit Baudry, and Yves Le Traon. **From testing to diagnosis : An automated approach**. In Proc. 19th *IEEE International Conference on Automated Software Engineering (ASE'04)*, Linz, October 2004. (Acceptance Ratio : 27%)

6. Clémentine Nebut, Franck Fleurey, Yves Le traon, and Jean-Marc Jézéquel. **Requirements by contracts allow automated system testing**. In Proc. of the 14th. *IEEE International Symposium on Software Reliability Engineering (ISSRE'03)*, Denver (CO) 2003. (Acceptance Ratio : 21%)
7. Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. **Genes and bacteria for automatic test cases optimization in the .net environment**. In Proceedings of *ISSRE02 (International Symposium on Software Reliability Engineering)*, November 2002. (Acceptance Ratio : 45%)
8. Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. **Automatic test cases optimization using a bacteriological adaptation model : Application to .net components**. In Proceedings of *ASE02 (Automated Software Engineering)*, October 2002. (Acceptance Ratio : 40%)

Workshops internationaux

1. B. Baudry, T. Dinh-Trong, J.-M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. Le Traon. **Challenges for Model Transformation Testing**. Proceedings of *IMDT workshop* in conjunction with ECMDA'06, Bilbao, Spain, 2006.
2. Pierre-Alain Muller, Franck Fleurey, Zoé Drey, Damien Pollet, Frédéric Fondement and Philippe Studer. **On Executable Meta-Languages applied to Model Transformations**. *Model Transformation In Practice (MTIP) workshop* held in conjunction with MODELS/UML 2005 conference, Montego Bay, Jamaica, October, 2005.
3. Trung Dinh-Trong, Sudipto Ghosh, Robert France, Benoit Baudry and Franck Fleurey. **A Taxonomy of Faults for UML Designs**. *Model Design and Validation (MoDeVa) workshop* held in conjunction with MODELS/UML 2005 conference, Montego Bay, Jamaica, October, 2005.
4. Benoit Baudry, Franck Fleurey, Robert France and Raghu Reddy. **Exploring the Relationship between Model Composition and Model Transformation**. *Aspect Oriented Modeling (AOM) workshop* held in conjunction with MODELS/UML 2005 conference, Montego Bay, Jamaica, October, 2005.
5. Raghu Reddy, Robert France, Sudipto Ghosh, Franck Fleurey and Benoit Baudry. **Model Composition - A Signature-Based Approach**. *Aspect Oriented Modeling (AOM) workshop* held in conjunction with MODELS/UML 2005 conference, Montego Bay, Jamaica, October, 2005.
6. Franck Fleurey, Jim Steel and Benoit Baudry. **MDE and validation : Testing model transformation**. In Procedings of the *SIVOES-Modeva workshop*, Rennes, November 2004.
7. Clémentine Nebut, Franck Fleurey, Yves Le Traon and Jean-Marc Jézéquel **A requirement-based approach to test product families**. In Proc. of the *5th*

workshop on Product Families Engineering (PFE-05) LNCS. Springer Verlag, 2003.

8. Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel and Yves Le Traon. **Computational intelligence for testing .net components**. In Proceedings of Microsoft Summer Research Workshop September 2002.

Conférences francophones

1. Jacques Klein and Franck Fleurey. **Tissage d'aspects comportementaux**. *Langages et Modèles à Objets : LMO'06*, Nimes, France, 2006.
2. Clémentine Nebut and Franck Fleurey. **Une méthode de formalisation progressive des exigences basée sur un modèle simulable**. *Langages et Modèles à Objets : LMO'05*, pages 145–158, Bern, Switzerland, February 2005.

Résumé

Une des conditions pour que l'ingénierie des modèles tienne ses promesses en terme de productivité et de qualité, est d'assurer, (1) la correction des modèles, et (2) la correction des transformations utilisées. La contribution de cette thèse s'articule autour de ces deux axes. Pour améliorer la qualité des modèles, nous proposons l'utilisation d'un coeur sémantique unique pour la spécification de langages de modélisation. Au coeur de la plateforme d'ingénierie des modèles développée par l'équipe Triskell, le langage Kermeta a été validé dans des contextes variés tels que la transformation de modèles, la modélisation orientée-aspects et la définition de langages dédiés. Pour améliorer la qualité des transformations, nous proposons une technique pour la sélection et la génération de modèles de test. Cette technique tire avantage du fait qu'une transformation manipule des données décrites par un méta-modèle. Ces travaux constituent un premier pas vers une ingénierie des modèles fiable.

Abstract

One of the conditions for model-driven developement to deliver its promises in terms of quality and productivity is to ensure, (1) the quality of the models and (2) the correctness of model tansformations that are used. The contribution of this work follows both directions. To improve the quality of models, we propose an original language called Kermeta for the specification of modeling languages. Kermeta is the core of the model-oriented platform developped in the Triskell team. It has been validated with a wide range of applications including model transformation, model conposition and weaving and the specification of dedicated modeling languages. To improve the quality of model transformations, we propose a technique for the generation and the selection of test models. This technique takes advantage of the fact that models processed by a model transformation are described by meta-models. Overall, this work is a first step towards reliability of model-driven developement.