Ingénierie Dirigée par les Modèles (IDM) – État de l'art

Benoit Combemale

Ingénierie Dirigée par les Modèles (IDM)
État de l'art

Benoît Combemale
Institut de Recherche en Informatique de Toulouse (UMR CNRS 5505)
benoit.combemale@enseeiht.fr

12 août 2008

Résumé

L'ingénierie dirigée par les modèles (IDM), ou *Model Driven Engineering* (MDE) en anglais, a permis plusieurs améliorations significatives dans le développement de systèmes complexes en permettant de se concentrer sur une préoccupation plus abstraite que la programmation classique. Il s'agit d'une forme d'ingénierie générative dans laquelle tout ou partie d'une application est engendrée à partir de modèles. Un modèle est une abstraction, une simplification d'un système qui est suffisante pour comprendre le système modélisé et répondre aux questions que l'on se pose sur lui. Un système peut être décrit par différents modèles liés les uns aux autres. L'idée phare est d'utiliser autant de langages de modélisation différents (*Domain Specific Modeling Languages* – DSML) que les aspects chronologiques ou technologiques du développement du système le nécessitent. La définition de ces DSML, appelée *métamodélisation*, est donc une problématique clé de cette nouvelle ingénierie. Par ailleurs, afin de rendre opérationnels les modèles (pour la génération de code, de documentation et de test, la validation, la vérification, l'exécution, etc.), une autre problématique clé est celle de la *transformation de modèle*.

Nous proposons dans ce document une présentation des principes clés de cette nouvelle ingénierie. Nous introduisons dans un premier temps la notion de modèle, les travaux de normalisation de l'OMG, et les principes de généralisation offerts à travers les DSML (section 1). Nous détaillons ensuite les deux axes principaux de l'IDM. La métamodélisation d'abord, dont le but est d'assurer une définition correcte des DSML (section 2). Nous illustrons cette partie par la définition de SIMPLEPDL, un langage simple de description de procédé de développement. Nous présentons ensuite les principes de la transformation de modèle et les outils actuellement disponibles (section 3). Nous concluons enfin par une discussion sur les limites actuelles de l'IDM (section 4).

1 Les modèles au coeur du développement de système

> « Pour un observateur A, M est un modèle de l'objet O, si M aide A à répondre aux questions qu'il se pose sur O »[42].

1.1 Les principes généraux de l'IDM

Suite à l'approche objet des années 80 et de son principe du « tout est objet », l'ingénierie du logiciel s'oriente aujourd'hui vers l'ingénierie dirigée par les modèles (IDM) et le principe du « tout est modèle ». Cette nouvelle approche peut être considérée à la fois en *continuité* et en *rupture* avec les précédents travaux [11, 12]. Tout d'abord en continuité car c'est la technologie objet qui a déclenché l'évolution vers les modèles. En effet, une fois acquise la conception des

systèmes informatiques sous la forme d'objets communicant entre eux, il s'est posé la question de les classifier en fonction de leurs différentes origines (objets métiers, techniques, etc.). L'IDM vise donc, de manière plus radicale que pouvaient l'être les approches des *patterns* [24] et des *aspects* [35], à fournir un grand nombre de modèles pour exprimer séparément chacune des préoccupations des utilisateurs, des concepteurs, des architectes, etc. C'est par ce principe de base fondamentalement différent que l'IDM peut être considérée en rupture par rapport aux travaux de l'approche objet.

Alors que l'approche objet est fondée sur deux relations essentielles, « InstanceDe »et « HériteDe », l'IDM est basée sur un autre jeu de concepts et de relations. Le concept central de l'IDM est la notion de *modèle* pour laquelle il n'existe pas à ce jour de définition universelle. Néanmoins, de nombreux travaux s'accordent à un relatif consensus d'une certaine compréhension. A partir des travaux de l'OMG[1], de Bézivin *et al.* [13] et de Seidewitz [52], nous considérerons dans la suite de cette thèse la définition suivante d'un modèle.

Définition (*Modèle*) Un modèle est une abstraction d'un système, modélisé sous la forme d'un ensemble de faits construits dans une intention particulière. Un modèle doit pouvoir être utilisé pour répondre à des questions sur le système modélisé.

On déduit de cette définition la première relation majeure de l'IDM, entre le modèle et le système qu'il représente, appelée *représentationDe* dans [3, 10, 52], et nommée μ sur la figure 1.

Notons que même si la relation précédente a fait l'objet de nombreuses réflexions, il reste toutefois difficile de répondre à la question « qu'est ce qu'un bon modèle ? »et donc de formaliser précisément la relation μ. Néanmoins un modèle doit, par définition, être une abstraction pertinente du système qu'il modélise, c.-à-d. qu'il doit être suffisant et nécessaire pour permettre de répondre à certaines questions en lieu et place du système qu'il représente, exactement de la même façon que le système aurait répondu lui-même. Ce principe, dit de *substituabilité*, assure que le modèle peut se substituer au système pour permettre d'analyser de manière plus abstraite certaines de ses propriétés [42].

Définition (*Principe de substituabilité*) Un modèle doit être suffisant et nécessaire pour permettre de répondre à certaines questions en lieu et place du système qu'il est censé représenté, exactement de la même façon que le système aurait répondu lui-même.

Sur la figure 1, nous reprenons l'exemple utilisé dans [23] qui s'appuie sur la cartographie pour illustrer l'IDM. Dans cet exemple, une carte est un modèle (une représentation) de la réalité, avec une intention particulière (carte routière, administrative, des reliefs, etc.).

La notion de modèle dans l'IDM fait explicitement référence à la notion de langage bien défini. En effet, pour qu'un modèle soit productif, il doit pouvoir être manipulé par une machine. Le langage dans lequel ce modèle est exprimé doit donc être clairement défini. De manière naturelle, la définition d'un langage de modélisation a pris la forme d'un modèle, appelé *métamodèle*.

Définition (*Métamodèle*) Un métamodèle est un modèle qui définit le langage d'expression d'un modèle [46], c.-à-d. le langage de modélisation.

La notion de métamodèle conduit à l'identification d'une seconde relation, liant le modèle et le langage utilisé pour le construire, appelée *conformeA* et nommée χ sur la figure 1.

En cartographie (cf. figure 1), il est effectivement indispensable d'associer à chaque carte la description du « langage »utilisé pour réaliser cette carte. Ceci se fait notamment sous la forme d'une légende explicite. La carte doit, pour être utilisable, être conforme à cette légende. Plusieurs cartes peuvent être conformes à une même légende. La légende est alors considérée comme un modèle représentant cet ensemble de cartes (μ) et à laquelle chacune d'entre elles doit se conformer (χ).

Ces deux relations permettent ainsi de bien distinguer le langage qui joue le rôle de système, du (ou des) métamodèle(s) qui jouent le rôle de modèle(s) de ce langage.

[1] *The Object Management Group (OMG)*, cf. `http://www.omg.org/`.

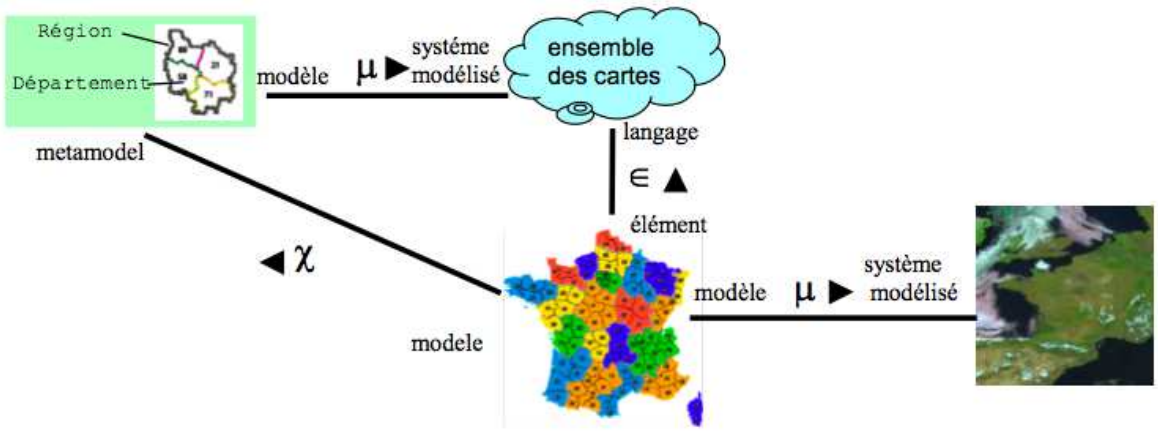

FIG. 1 – Relations entre système, modèle, métamodèle et langage [23]

C'est sur ces principes de base que s'appuie l'OMG pour définir l'ensemble de ses standards, en particulier UML (*Unified Modeling Language*) [48, 49] dont le succès industriel est unanimement reconnu.

1.2 L'approche MDA

Le consensus sur UML fut décisif dans cette transition vers des techniques de production basées sur les modèles. Après l'acceptation du concept clé de métamodèle comme langage de description de modèle, de nombreux métamodèles ont émergés afin d'apporter chacun leurs spécificités dans un domaine particulier (développement logiciel, entrepôt de données, procédé de développement, etc.). Devant le danger de voir émerger indépendamment et de manière incompatible cette grande variété de métamodèles, il y avait un besoin urgent de donner un cadre général pour leur description. La réponse logique fut donc d'offrir un langage de définition de métamodèles qui prit lui-même la forme d'un modèle : ce fut le *métamétamodèle* MOF (*Meta-Object Facility*) [46]. En tant que modèle, il doit également être défini à partir d'un langage de modélisation. Pour limiter le nombre de niveaux d'abstraction, il doit alors avoir la propriété de *métacircularité*, c.-à-d. la capacité de se décrire lui-même.

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

C'est sur ces principes que se base l'organisation de la modélisation de l'OMG généralement décrite sous une forme pyramidale (cf. figure 2). Le monde réel est représenté à la base de la pyramide (niveau $M0$). Les modèles représentant cette réalité constituent le niveau $M1$. Les métamodèles permettant la définition de ces modèles (p. ex. UML) constituent le niveau $M2$. Enfin, le métamétamodèle, unique et métacirculaire, est représenté au sommet de la pyramide (niveau $M3$).

L'approche consistant à considérer une hiérarchie de métamodèles n'est pas propre à l'OMG, ni même à l'IDM, puisqu'elle est utilisée depuis longtemps dans de nombreux domaines de l'informatique. Chaque hiérarchie définit un *espace technique* [38, 6, 7]. Nous distinguons par exemple le *modelware* (espace technique des modèles), le *grammarware* (espace technique des grammaires définies par les langages tels que BNF[2] ou EBNF[3]), le *BDware* (espace technique des bases de données), etc.

[2] *Backus-Naur form*
[3] *Extended BNF*

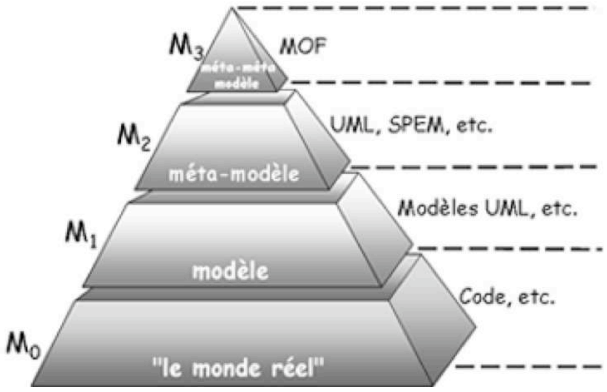

FIG. 2 – Pyramide de modélisation de l'OMG [5]

Définition (*Espace technique*) Un espace technique est l'ensemble des outils et techniques issus d'une pyramide de métamodèles dont le sommet est occupé par une famille de (méta)métamodèles similaires [23].

L'OMG a défini le MDA (*Model Driven Architecture*) en 2000 [54] pour promulguer de bonnes pratiques de modélisation et exploiter pleinement les avantages des modèles. En 2003, les membres ont adopté la dernière version de la spécification [41] donnant une définition détaillée de l'architecture. Cette approche vise à mettre en valeur les qualités intrinsèques des modèles, telles que pérennité, productivité et prise en compte des plateformes d'exécution. Le MDA inclut pour cela la définition de plusieurs standards, notamment UML, MOF et XMI[4].

Le principe clé et initial du MDA consiste à s'appuyer sur le standard UML pour décrire séparément des modèles pour les différentes phases du cycle de développement d'une application. Plus précisément, le MDA préconise l'élaboration de modèles (cf. figure 3) :

- d'exigence (*Computation Independent Model* – CIM) dans lesquels aucune considération informatique n'apparait,
- d'analyse et de conception (*Platform Independent Model* – PIM),
- de code (*Platform Specific Model* – PSM).

L'objectif majeur du MDA est l'élaboration de modèles pérennes (PIM), indépendants des détails techniques des plate-formes d'exécution (J2EE, .Net, PHP, etc.), afin de permettre la génération automatique de la totalité des modèles de code (PSM) et d'obtenir un gain significatif de productivité.

Le passage de PIM à PSM fait intervenir des mécanismes de transformation de modèle (cf. section 3) et un modèle de description de la plateforme (*Platform Description Model* – PDM). Cette démarche s'organise donc selon un cycle de développement « en Y »propre au MDD (*Model Driven Development*) (cf. figure 3).

Le MDA a fait l'objet d'un grand intérêt dans la littérature spécialisée. Nous citons entre autre les ouvrages de X. Blanc [8] et de A. Kleppe [37] qui ont inspirés cette section.

1.3 Les langages dédiés de modélisation

De la même façon que l'arrivée de la programmation par objet n'a pas invalidé les apports de la programmation structurée, le développement dirigé par les modèles ne contredit pas les apports de la technologie objet. Il est donc important de ne pas considérer ces solutions comme antagonistes mais comme complémentaires.

[4]XMI, *XML Metadata Interchange*, est un format d'échange basé sur XML pour les modèles exprimés à partir d'un métamodèle MOF.

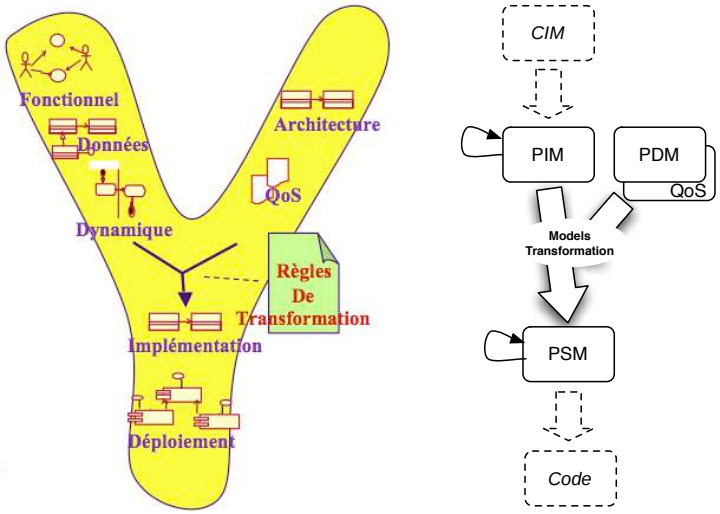

FIG. 3 – MDA : Un processus en Y dirigé par les modèles

Toutefois un point de divergence entre ces deux approches concerne l'intégration de paradigmes. Initialement, la technologie objet se voulait aussi une technologie d'intégration car il était théoriquement possible de représenter de façon uniforme les processus, les règles, les fonctions, etc. par des objets. Aujourd'hui, on revient à une vision moins hégémonique où les différents paradigmes de programmation coexistent sans donner plus d'importance à l'un ou à l'autre [11].

Un point important est alors de séparer clairement les approches IDM du formalisme UML, et de l'utilisation qui en est faite dans le MDA. En effet, non seulement la portée de l'IDM est plus large que celle d'UML mais la vision de l'IDM est aussi très différente de celle d'UML, parfois même en contradiction. UML est un standard assez monolithique obtenu par consensus *a maxima*, dont on doit réduire ou étendre la portée à l'aide de mécanismes comme les profils [49, §18]. Ces mécanismes n'ont pas tous la précision souhaitable et mènent parfois à des contorsions dangereuses pour « rester »dans le monde UML.

Au contraire, l'IDM favorise la définition de langages de modélisation dédiés à un domaine particulier (*Domain Specific Modeling Languages* – DSML) offrant ainsi aux utilisateurs des concepts propres à leur métier et dont ils ont la maîtrise. Ces langages sont généralement de petite taille et doivent être facilement manipulables, transformables, combinables, etc.

Selon ces principes, la définition d'un système complexe fait généralement appel à l'utilisation de plusieurs DSML ayant des relations entre eux, restreignant ainsi l'ensemble des combinaisons valides des modèles conformes à ces différents métamodèles (c.-à-d. construits à l'aide de ces différents DSML). Il est ainsi récemment apparu la nécessité de représenter ces différents DSML et les relations entre eux. Une réponse logique à ce besoin a été de proposer des modèles dont les éléments de base sont d'une part les différents DSML et d'autre part les liens exprimant leurs dépendances. Ce type de modèle est appelé *mégamodèle* [14, 22] et est déjà supporté dans l'outil de mégamodélisation AM3 [2].

Définition (*Mégamodèle*) Un mégamodèle est un modèle dont les éléments représentent des (méta)modèles ou d'autres *artefacts* (comme des DSML, des outils, des services, etc.).

Notons par ailleurs que le concept de DSML a donné lieu à la création de nombreux langages qu'il est maintenant urgent de maîtriser (documentation, hiérarchie, etc.) et de pouvoir manipuler facilement (combiner, transformer, etc.). Pour cela, certains *zoos*[5] proposent un recensement, une

[5]Un zoo est (une vue d')un mégamodèle où tous les métamodèles qui le compose ont le même métamétamodèle [58].

documentation et une classification de ces DSML et offrent certaines manipulations, comme de pouvoir les transformer vers différents espaces techniques.

2 La métamodélisation

Comme nous l'avons vu dans la section précédente, l'IDM préconise l'utilisation de petits langages de modélisation, dédiés chacun à un domaine particulier. Ainsi, la première problématique clé de l'IDM est la maîtrise de la définition de ces DSML, dont le nombre ne cesse de croître avec la multiplication des domaines d'application de l'informatique. La métamodélisation, activité correspondant à définir un DSML, doit donc être étudiée et maîtrisée. Pour cela, les premiers travaux ont consisté à définir précisément les différentes composantes d'un langage de modélisation et à offrir les outils permettant de les décrire.

Définition (*Métamodélisation*) La métamodélisation est l'activité consistant à définir le métamodèle d'un langage de modélisation. Elle vise donc à bien modéliser un langage, qui joue alors le rôle de système à modéliser.

Nous détaillons maintenant les principes de la métamodélisation (section 2.1) et présentons ensuite les différents outils et techniques pour la spécification d'un DSML (section 2.2). Ceci est illustré par la définition de SIMPLEPDL, un petit langage de modélisation dédié au domaine des procédés de développement, c.-à-d. un langage permettant de définir des modèles de procédé de développement.

2.1 Qu'est-ce qu'un langage ?

Que ce soit en linguistique (langage naturel) ou en informatique (langage de programmation ou de modélisation), il est depuis longtemps établi qu'un langage est caractérisé par sa *syntaxe* et sa *sémantique*. La syntaxe décrit les différentes constructions du langage et les règles d'agencement de ces constructions (également appelées *context condition* [29]). La sémantique désigne le lien entre un signifiant (un programme, un modèle, etc.), et un signifié (p. ex. un objet mathématique) afin de donner un sens à chacune des constructions du langage. Il y a donc entre la sémantique et la syntaxe le même rapport qu'entre le fond et la forme.

Définition (*Langage*) Un langage (L) est défini selon le tuple $\{S, Sem\}$ où S est sa syntaxe et Sem sa sémantique.

Cette définition est très générale et assez abstraite pour caractériser l'ensemble des langages, quel que soit le domaine. De manière plus précise, dans le contexte de l'informatique, on distingue généralement la syntaxe concrète (*CS* sur la figure 4), manipulée par l'utilisateur du langage, de la syntaxe abstraite (*AS* sur la figure 4) qui est la représentation interne (d'un programme ou d'un modèle) manipulée par l'ordinateur [1]. Dans les langages de programmation, la représentation interne (l'arbre abstrait) est dérivée de la syntaxe concrète. Ainsi, la syntaxe d'un langage de programmation est définie par les syntaxes concrète et abstraite et par un lien, dit de *dérivation* ou d'*abstraction*, entre la syntaxe concrète et la syntaxe abstraite (M_{ca} sur la figure 4). Ce lien d'abstraction permet d'enlever tout le « sucre »syntaxique inutile à l'analyse du programme.

D'autre part, on distingue également le *domaine sémantique* (*SD* sur la figure 4) qui représente l'ensemble des états atteignables (c.-à-d. les états possibles du système). La sémantique d'un langage de programmation est alors donnée en liant les constructions de la syntaxe concrète avec l'état auquel elles correspondent dans le domaine sémantique (M_{as} sur la figure 4).

Définition (*Langage de programmation*) Un langage de programmation (L_p) est défini selon le tuple $\{AS, CS, M_{ca}, SD, M_{cs}\}$ où AS est la syntaxe abstraite, CS est la syntaxe concrète, M_{ca} est le *mapping* de la syntaxe concrète vers sa représentation abstraite, SD est le domaine sémantique et M_{cs} est le *mapping* de la syntaxe concrète vers le domaine sémantique.

Nous citons par exemple les zoos `http://www.eclipse.org/gmt/am3/zoos/` et `http://www.zooomm.org/`.

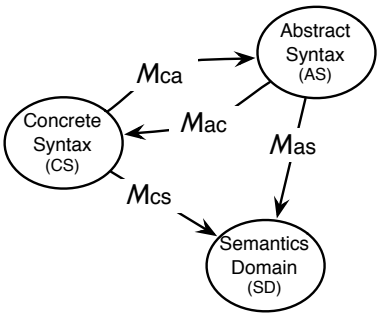

FIG. 4 – Composantes d'un langage

Dans le contexte de l'IDM, la syntaxe abstraite est placée au cœur de la description d'un langage de modélisation. Elle est donc généralement décrite en premier et sert de base pour définir la syntaxe concrète. La définition de la syntaxe concrète consiste alors à définir des décorations (textuelles ou graphiques) et à définir un lien entre les constructions de la syntaxe abstraite et leurs décorations de la syntaxe concrète (M_{ac} sur la figure 4). Ce changement de sens du lien par rapport aux langages de programmation permet d'envisager de définir plusieurs syntaxes concrètes (M^*_{ac}) pour une même syntaxe abstraite et donc d'avoir plusieurs représentations d'un même modèle. Le langage peut alors être manipulé avec différents formalismes mais avec les mêmes constructions et la même représentation abstraite. D'autre part, dans le cadre des langages de modélisation, la sémantique est exprimée à partir des constructions de la syntaxe abstraite par un lien vers un domaine sémantique (M_{as} sur la figure 4). Nous considérerons dans cette thèse qu'un DSML ne peut avoir qu'une seule sémantique (et donc un seul *mapping* vers un domaine sémantique). La définition d'un autre *mapping* engendrera la définition d'un nouveau DSML. Toutefois, un même modèle peut être simultanément ou successivement conforme à plusieurs DSML.

Définition (*Langage de modélisation*) Un langage de modélisation (L_m) est défini selon le tuple $\{AS, CS^*, M^*_{ac}, SD, M_{as}\}$ où AS est la syntaxe abstraite, CS^* est la (les) syntaxe(s) concrète(s), M^*_{ac} est l'ensemble des *mappings* de la syntaxe abstraite vers la (les) syntaxe(s) concrète(s), SD est le domaine sémantique et M_{as} est le *mapping* de la syntaxe abstraite vers le domaine sémantique.

2.2 Outils et techniques pour la spécification d'un DSML

Nous introduisons dans cette partie les techniques, standards, et outils actuellement disponibles pour décrire les différentes parties d'un DSML présentées dans la partie précédente.

2.2.1 Syntaxe abstraite

La syntaxe abstraite (*AS*) d'un langage de modélisation exprime, de manière structurelle, l'ensemble de ses concepts et leurs relations. Les langages de métamodélisation tels que le standard MOF de l'OMG [46], offrent les concepts et les relations élémentaires qui permettent de décrire un métamodèle représentant la syntaxe abstraite d'un langage de modélisation. Pour définir cette syntaxe, nous disposons à ce jour de nombreux environnements et langages de métamodélisation : Eclipse-EMF/Ecore [9], GME/MetaGME [39], AMMA/KM3 [33, 4], XMF-Mosaic/Xcore [15] ou Kermeta [43]. Tous ces langages reposent toutefois sur les mêmes constructions élémentaires (cf. figure 5). S'inspirant de l'approche orientée objet, les langages de métamodélisation objet offre le concept de classe (*Class*) pour définir les concepts d'un DSML. Une classe est composée de propriétés (*Property*) qui la caractérisent. Une propriété est appelée *référence* lorsqu'elle est

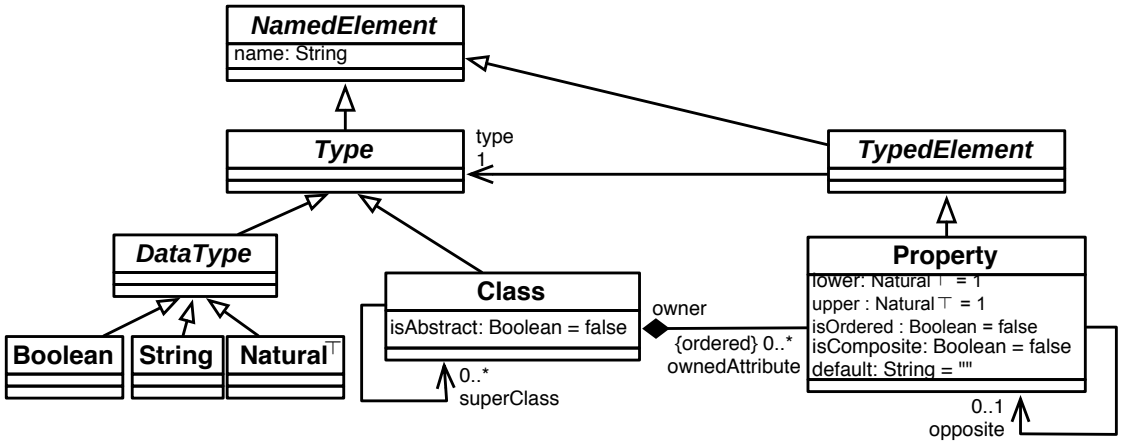

FIG. 5 – Concepts principaux de métamodélisation (EMOF 2.0)

typée (*TypedElement*) par une autre classe, et *attribut* lorsqu'elle est typée par un type de donnée (p. ex. booléen, chaîne de caractère et entier).

Nous illustrons ces concepts par la définition de SIMPLEPDL, un langage simple de description de procédé, que nous utiliserons dans la suite de cette thèse pour illustrer les différentes techniques abordées. Il s'inspire du standard SPEM (*Software & Systems Process Engineering Metamodel*) [45] proposé par l'OMG mais aussi du métamodèle UMA (*Unified Method Architecture*) utilisé par le *plug-in* Eclipse EPF [6] (*Eclipse Process Framework*), dédié à la modélisation de procédé. Il est volontairement simplifié pour ne pas compliquer inutilement les expérimentations.

Pour définir la syntaxe abstraite de *SimplePDL*, nous avons utilisé l'éditeur graphique du projet TOPCASED permettant la description de métamodèles à l'aide du langage de métamodélisation Ecore. Le métamodèle SIMPLEPDL est donné figure 6. Il définit le concept de processus (*Process*) composé d'un ensemble d'activités (*WorkDefinition*) représentant les différentes tâches à réaliser durant le développement. Une activité peut dépendre d'une autre (*WorkSequence*). Une contrainte d'ordonnancement sur le démarrage ou la fin de la seconde activité est précisée (attribut *linkType*) grâce à l'énumération *WorkSequenceType*. Par exemple, deux activités A_1 et A_2 reliées par une relation de précédence de type *finishToStart* signifie que A_2 ne pourra commencer que quand A_1 sera terminée. Enfin, des annotations textuelles (*Guidance*) peuvent être associées aux activités pour donner plus de détails sur leurs réalisations.

La figure 7 donne un exemple de modèle de processus composé de quatre activités. Le développement ne peut commencer que quand la conception est terminée. La rédaction de la documentation ou des tests peut commencer dès que la conception est commencée (*startToStart*) mais la documentation ne peut être terminée que si la conception est terminée (*finishToFinish*) et les tests si le développement est terminé.

La représentation graphique offerte par les langages de métamodélisation ne permet pas de capturer formellement l'ensemble des propriétés du langage (c.-à-d. les *context conditions*). Dans le domaine des langages de programmation, la sémantique axiomatique est basée sur des logiques mathématiques et exprime une méthode de preuve pour certaines propriétés des constructions d'un langage [17]. Celle-ci peut être très générale (e.g. triplet de Hoare) ou restreinte à la garantie de la cohérence de ces constructions (p. ex. le typage). Dans le cadre d'un langage de modélisation, cette seconde utilisation est exprimée par le biais de règles de bonne formation (*Well-Formed Rules* – WFR), au niveau du métamodèle. Ces règles devront être respectées par les modèles

[6] `http://www.eclipse.org/epf/`

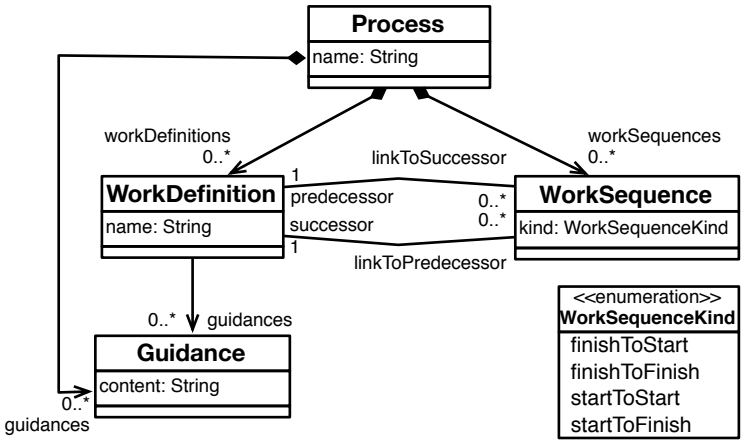

FIG. 6 – Syntaxe abstraite de SIMPLEPDL

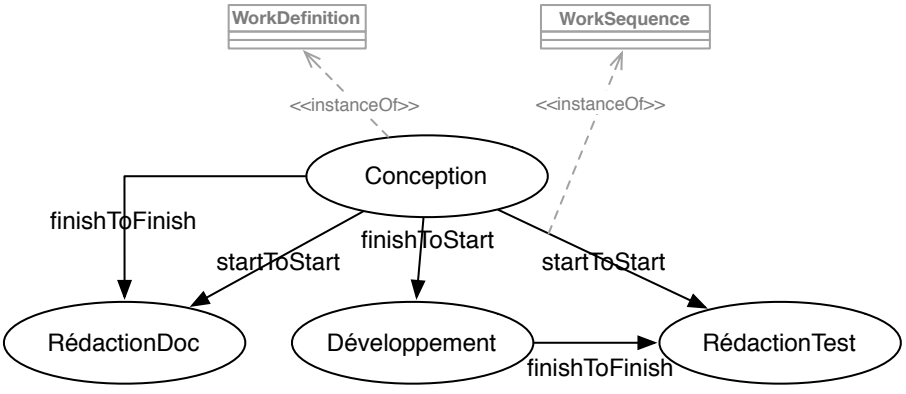

FIG. 7 – Exemple de modèle SIMPLEPDL

conformes à ce métamodèle.

Pour exprimer ces règles, l'OMG préconise d'utiliser OCL (*Object Constraint Language*) [47, 59]. Appliqué au niveau du métamodèle, il permet d'ajouter des propriétés, principalement structurelles, qui n'ont pas pu être capturées par les concepts fournis par le métamétamodèle. Il s'agit donc d'un moyen de préciser la sémantique du métamodèle en limitant les modèles conformes.

Par exemple, pour *SimplePDL*, on peut utiliser la contrainte OCL suivante pour imposer l'unicité du nom des activités dans un processus.

```
context Process inv :
  self . activities ->forAll(a1, a2 : Activity |
    a1 <> a2 implies a1.name <> a2.name)
```

Pour vérifier qu'un modèle respecte ces contraintes, on peut utiliser des vérificateurs OCL tels que Use [51], OSLO[7], TOPCASED, etc.

2.2.2 Syntaxe concrète

Les syntaxes concrètes (*CS*) d'un langage fournissent à l'utilisateur un ou plusieurs formalismes, graphiques et/ou textuels, pour manipuler les concepts de la syntaxe abstraite et ainsi en créer des « instances ». Le modèle ainsi obtenu sera conforme à la structure définie par la syntaxe abstraite. La définition d'une syntaxe concrète consiste à définir un des *mappings* de M_{ac}^{*} (cf. figure 4), $M_{ac} : AS \leftrightarrow CS$, et permet ainsi « d'annoter »chaque construction du langage de modélisation définie dans la syntaxe abstraite par une (ou plusieurs) décoration(s) de la syntaxe concrète et pouvant être manipulée(s) par l'utilisateur du langage.

La définition du modèle d'une syntaxe concrète est à ce jour bien maîtrisée et outillée. Il existe en effet de nombreux projets qui s'y consacrent, principalement basés sur EMF (*Eclipse Modeling Framework*) [20] : GMF (*Generic Modeling Framework*) [27], TOPCASED [57, 21], Merlin Generator [40], GEMS [25], TIGER [56, 19], etc. Si ces derniers sont principalement graphiques, des projets récents permettent de définir des modèles de syntaxe concrète textuelle. Nous citons par exemple les travaux des équipes INRIA Triskell (Sintaks [44, 53]) et ATLAS (TCS [31, 55]) qui proposent des générateurs automatiques d'éditeurs pour des syntaxes concrètes textuelles. Ces approches génératives, en plus de leurs qualités de généricité, permettent de normaliser la construction des syntaxes concrètes.

Nous illustrons l'utilisation du générateur d'éditeur graphique de TOPCASED afin de définir la syntaxe concrète de SIMPLEPDL. Cet outil permet, pour un modèle Ecore donné, de définir une syntaxe concrète graphique et l'éditeur associé[8]. Cette génération d'éditeur s'appuie sur le résultat de la génération de la bibliothèque de sérialisation d'un modèle au format XMI fournie par EMF [9]. La syntaxe concrète est décrite dans un *modèle de configuration* qui offre une grande liberté de personnalisation des éléments graphiques souhaités pour représenter les concepts. Dans TOPCASED, il a été utilisé pour engendrer les éditeurs pour les langages Ecore, UML2, AADL[9] et SAM[10].

Pour SIMPLEPDL, nous avons défini en premier lieu le modèle de notre syntaxe concrète (la figure 8 en présente une version simplifiée). *Activity* et *Guidance* sont définies comme *Node* (boîtes). *Precedes* est définie comme *Edge*, un arc entre deux boîtes. *Process* est représentée comme *Diagram* qui correspond à un paquetage qui contiendra les autres éléments. Définir la syntaxe concrète d'un langage peut nécessiter l'emploi d'éléments additionnels ne correspondant à aucun concept abstrait. Par exemple, ici il est indispensable d'ajouter *GuidanceLink* comme *Edge* pour relier une *Guidance* à une *Activity*. *GuidanceLink* ne correspond pourtant à aucun

[7] Open Source Library for OCL, `http://oslo-project.berlios.de`.

[8] Nous proposons sur le site de documentation de TOPCASED, `http://topcased-mm.gforge.enseeiht.fr`, un tutoriel présentant les fonctionnalités du générateur de manière itérative et sur l'exemple de SIMPLEPDL

[9] *Architecture Analysis & Design Language*, cf `http://www.aadl.info/`

[10] *Structured automata Metamodel*, formalisme à base d'automates hiérarchiques utilisé par Airbus

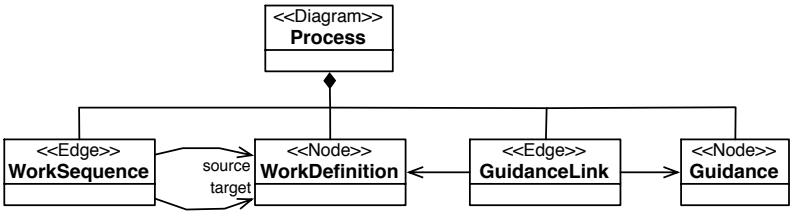

FIG. 8 – Modèle de configuration de la syntaxe concrète de SIMPLEPDL

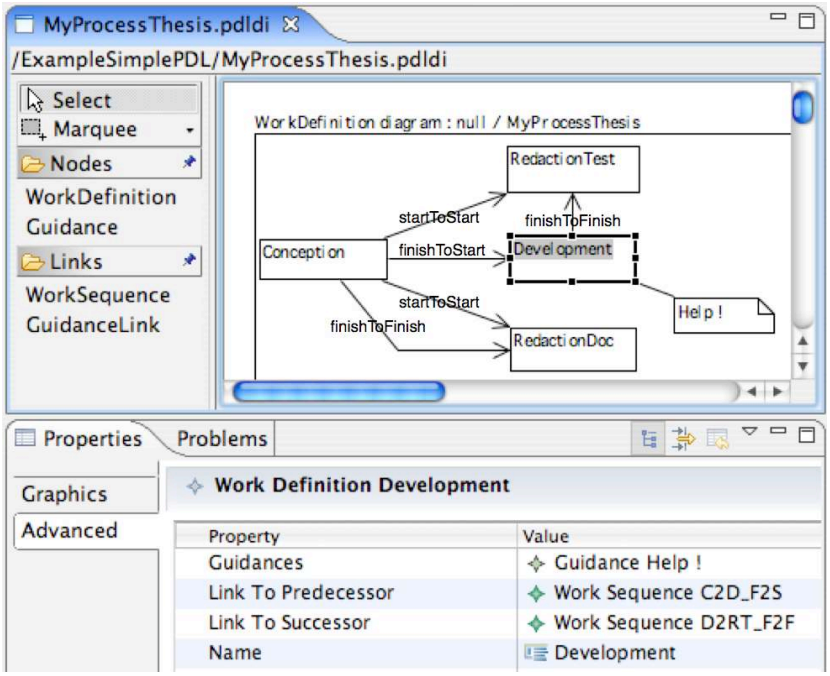

FIG. 9 – Éditeur graphique de SimplePDL généré avec TOPCASED

concept de *SimplePDL*. Il s'agit de « sucre »syntaxique dont la présence est obligatoire pour lier un élément graphique *Guidance* à la boîte représentant l'activité qu'il décrit. Il se traduit par la référence appelée *guidance* de *Activity* (cf. figure 6). Il faut noter que les concepts de la syntaxe abstraite (cf. figure 6) et ceux de la syntaxe concrète (cf. figure 8) sont des concepts différents qui doivent être mis en correspondance. Nous avons employé les mêmes noms quand la correspondance était évidente. La figure 9 présente l'éditeur engendré. Tous les concepts du modèle de configuration sont présents dans la palette. Sélectionner un élément de la palette et le déposer sur le diagramme crée un élément graphique (*node* ou *edge*) et instancie, selon le modèle de configuration, la métaclasse correspondante du métamodèle SIMPLEPDL. Le modèle de configuration permet également de préciser la représentation graphique des différents éléments. Par exemple *WorkSequence* connecte deux *WorkDefinition* avec une flèche du côté de la cible.

2.2.3 Sémantique

Définir la sémantique d'un langage revient à définir le domaine sémantique et le *mapping* M_{as} entre la syntaxe abstraite et le domaine sémantique ($AS \leftrightarrow SD$). Le domaine sémantique définit l'ensemble des états atteignables par le système, et le *mapping* permet d'associer ces états aux

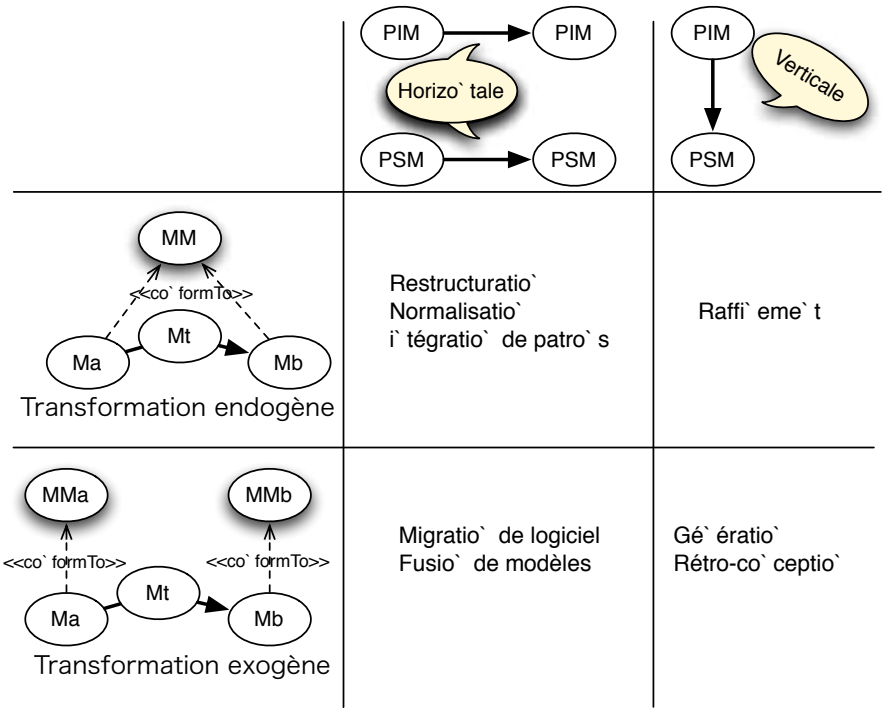

FIG. 10 – Types de transformation et leurs principales utilisations

éléments de la syntaxe abstraite.

Dans le contexte de l'IDM, au même titre que les autres éléments d'un langage de modélisation, la définition du domaine sémantique et du *mapping* prend la forme de modèle [30, 28, 29]. La sémantique des langages de modélisation est à ce jour rarement défini et fait actuellement l'objet d'intenses travaux de recherche. Des informations complémentaires sur cet aspect sont données dans la thèse de Benoît Combemale [16].

3 La transformation de modèle

La deuxième problématique clé de l'IDM consiste à pouvoir rendre opérationnels les modèles à l'aide de transformations. Cette notion est au centre de l'approche MDA et plus généralement de celle des DSML. En effet, l'intérêt de transformer un modèle *Ma* en un modèle *Mb* que les métamodèles respectifs *MMa* et *MMb* soient identiques (transformation *endogène*) ou différents (transformation *exogène*) apparaît comme primordial (génération de code, *refactoring*, migration technologique, etc.) [11].

D'autre part, l'approche MDA repose sur le principe de la création d'un modèle indépendant de toute plateforme (PIM) pouvant être raffiné en un ou plusieurs modèle(s) spécifique(s) à une plateforme (PSM). Les méthodes de transformation sont là aussi indispensables pour changer de niveau d'abstraction (transformation *verticale*), dans le cas du passage de PIM à PSM et inversement, ou pour rester au même niveau d'abstraction (transformation *horizontale*) dans le cas de transformation PIM à PIM ou PSM à PSM [26]. Ces différentes classes de transformation sont reprises sur la figure 10 en indiquant leurs cas d'utilisation.

Enfin, la transformation de modèle est également utilisée dans la définition des langages de modélisation pour établir les *mappings* et des traductions entre différents langages. Ces différentes classes de transformation sont résumées sur la figure 11.

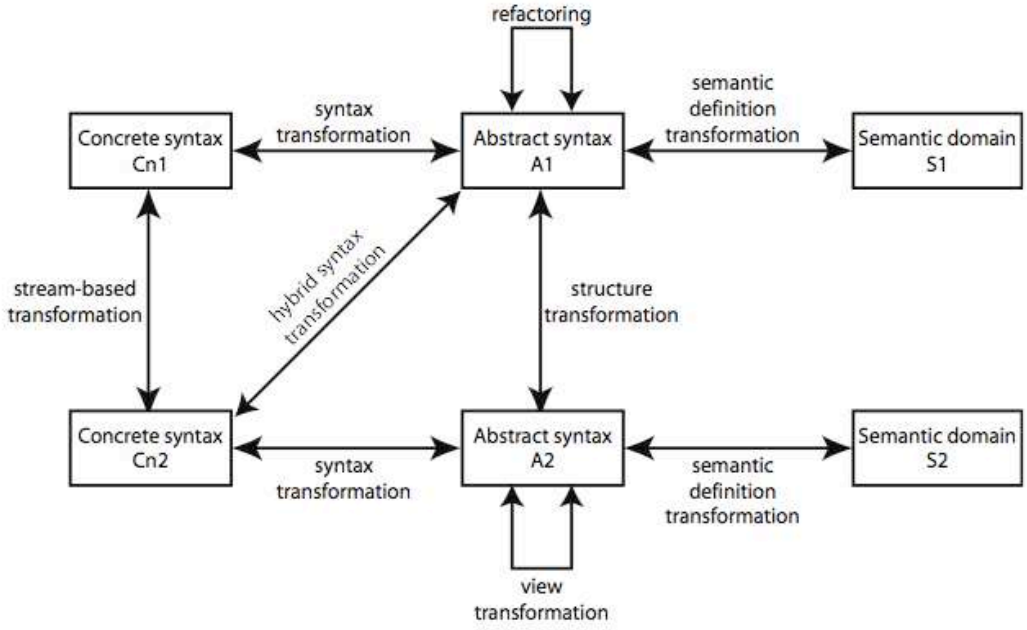

FIG. 11 – Classes de transformations dans la définition d'un DSML [36]

On comprend donc pourquoi le succès de l'IDM repose en grande partie sur la résolution du problème de la transformation de modèle. Cette problématique a donnée lieu ces dernières années à de nombreux travaux académiques, industriels et de normalisation que nous présentons ci-après.

3.1 Historique

Les travaux réalisés dans le domaine de la transformation de modèle ne sont pas récents et peuvent être chronologiquement classés selon plusieurs générations en fonction de la structure de donnée utilisée pour représenter le modèle [5] :

- *Génération 1 : Transformation de structures séquentielles d'enregistrement.* Dans ce cas un script spécifie comment un fichier d'entrée est réécrit en un fichier de sortie (p. ex. des scripts Unix, AWK ou Perl). Bien que ces systèmes soient plus lisibles et maintenables que d'autres systèmes de transformation, ils nécessitent une analyse grammaticale du texte d'entrée et une adaptation du texte de sortie [26, 5].
- *Génération 2 : Transformation d'arbres.* Ces méthodes permettent le parcours d'un arbre d'entrée au cours duquel sont générés les fragments de l'arbre de sortie. Ces méthodes se basent généralement sur des documents au format XML et l'utilisation de XSLT[11] ou XQuery[12].
- *Génération 3 : Transformation de graphes.* Avec ces méthodes, un modèle en entrée (graphe orienté étiqueté) est transformé en un modèle en sortie.
  Ces approches visent à considérer l' « opération »de transformation comme un autre modèle (cf. figure 12) conforme à son propre métamodèle (lui-même défini à l'aide d'un langage de métamodélisation, par exemple le MOF). La transformation d'un modèle Ma (conforme à son métamodèle MMa) en un modèle Mb (conforme à son métamodèle MMb) par le modèle Mt peut donc être formalisée de la manière suivante :

  $$Mb^* \leftarrow f(MMa^*, MMb^*, Mt, Ma^*)$$

[11] *XSL (eXtensible StyleSheet Language) Transformation*, cf. `http://www.w3.org/TR/xslt`.
[12] *An XML Query Language*, cf. `http://www.w3.org/TR/xquery/`.

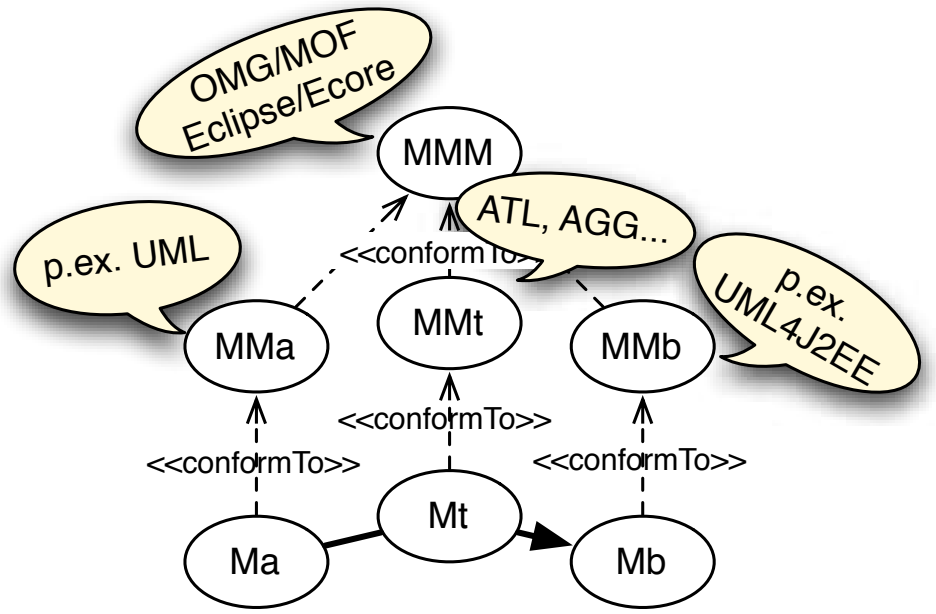

FIG. 12 – Principes de la transformation de modèle

Notons qu'il est possible d'avoir plusieurs modèles d'entrées (Ma^*) et de sorties (Mb^*).

Cette dernière génération a donné lieu à d'importants travaux de recherche et à la proposition de diverses approches [18]. Celles-ci peuvent être caractérisées à partir de critères comme le paradigme pour la définition des transformations, les scénarios de transformation, la directivité des transformations établies, le nombre de modèles sources et cibles, la traçabilité, le langage de navigation utilisé, l'organisation et l'ordonnancement des règles, etc. [32].

3.2 Standards et langages pour la transformation de modèle

De nombreux langages sont à ce jour disponibles pour écrire des transformations de modèle de génération 3. On retrouve d'abord les langages généralistes qui s'appuient directement sur la représentation abstraite du modèle. On citera par exemple l'API[13] d'EMF [9] qui, couplée au langage Java, permet de manipuler un modèle sous la forme d'un graphe. Dans ce cas, c'est à la charge du programmeur de faire la recherche d'information dans le modèle, d'expliciter l'ordre d'application des règles[14], de gérer les éléments cibles construits, etc.

Afin d'abstraire la définition des transformations de modèle et rendre transparent les détails de mise en œuvre, l'idée a été de définir des DSML dédiés à la transformation de modèle. Cette approche repose alors sur la définition d'un métamodèle dédié à la transformation de modèle et des outils permettant d'exécuter les modèles de transformation. Nous citerons par exemple ATL (*ATLAS Transformation Language*) [34] que nous utilisons tout au long de cette thèse. Il s'agit d'un langage hybride (déclaratif et impératif) qui permet de définir une transformation de modèle à modèle (appelée *Module*) sous la forme d'un ensemble de règle. Il permet également de définir des transformations de type modèle vers texte (appelée *Query*). Une transformation prend en entrée un ensemble de modèles (décrits à partir de métamodèles en Ecore ou en KM3).

[13] *Application Programming Interface* ou Interface de programmation.

[14] Une règle est l'unité de structuration dans les langages de transformation de modèle.

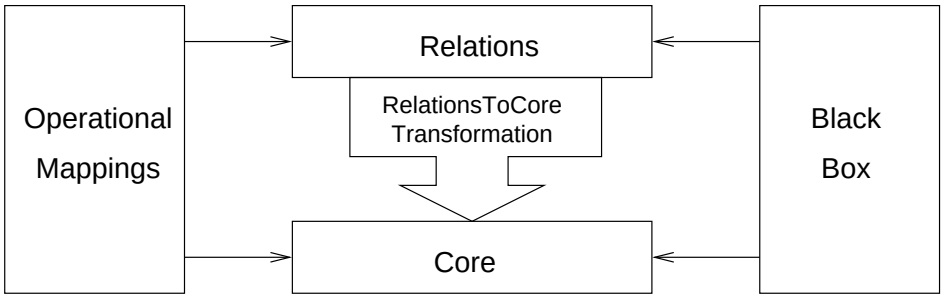

FIG. 13 – Architecture du standard QVT [50]

Afin de donner un cadre normatif pour l'implantation des différents langages dédiés à la transformation de modèle, l'OMG a défini le standard QVT (*Query/View/Transformation*) [50]. Le métamodèle de QVT est conforme à MOF et OCL est utilisé pour la navigation dans les modèles. Le métamodèle fait apparaître trois sous-langages pour la transformation de modèles (cf. figure 13), caractérisés par le paradigme mis en œuvre pour la définition des transformations (déclaratif, impératif et hybride). Les langages *Relations* et *Core* sont tous deux déclaratifs mais placés à différents niveaux d'abstraction. L'un des buts de *Core* est de fournir une base pour la spécification de la sémantique de *Relations*. La sémantique de *Relations* est donnée comme une transformation de *Relations* vers *Core*. Il est parfois difficile de définir une solution complètement déclarative à un problème de transformation donné. Pour adresser cette question, QVT propose deux mécanismes pour étendre *Relations* et *Core* : un troisième langage appelé *Operational Mappings* et un mécanisme d'invocation de fonctionnalités de transformation implémentées dans un langage arbitraire (boîte noire ou *black box*). *Operational Mappings* étend *Relations* avec des constructions impératives et des constructions OCL avec effets de bord.

4 Discussion et synthèse

Nous avons introduit dans ce document les principes généraux de l'IDM, c'est-à-dire la métamodélisation d'une part et la transformation de modèle d'autre part. Ces deux axes constituent les deux problématiques clé de l'IDM sur lesquelles la plupart des travaux de recherche se concentrent actuellement.

Les premiers résultats en métamodélisation ont permis d'établir les concepts (*modèle* et *métamodèle*) et relations (*représentationDe* et *conformeA*) de base dans une architecture dirigée par les modèles. Malgré tout, les modèles sont actuellement construits à partir de DSML décrits principalement par leur structure. Nous avons vu que si la définition des syntaxes abstraites et concrètes était maîtrisée et outillée. Il est aussi possible de vérifier structurellement la conformité d'un modèle par rapport à son DSML. Cependant, nous n'avons pas abordé dans ce document les aspects liés à la sémantique d'exécution qui font actuellement l'objet d'intenses travaux de recherche [16]. Notons qu'il n'y a toutefois pas de consensus sur la portée de la relation *conformeA*, en particulier si elle ne doit prendre en compte que la conformité *structurelle* (c.-à-d. le modèle respecte les *context conditions* de la syntaxe) ou une conformité plus large incluant la sémantique et vérifiant aussi la cohérence comportementale du modèle. Une formalisation précise de cette relation n'est à ce jour pas définie et la description précise de la sémantique des DSML reste une étape déterminante pour le passage à une approche complètement dirigée par les modèles (c.-à-d. une approche où les modèles seront nécessaires et suffisants pour décrire entièrement un système – logiciel par exemple – et dans laquelle, le recours à la programmation sera intégralement transparent pour le concepteur).

Les techniques de transformation de modèle, clé du succès de l'IDM afin de pouvoir rendre opérationnels les modèles, sont issues de principes qui sont bien antérieurs à l'IDM. Malgré tout,

les travaux récents de normalisation et d'implantation d'outils ainsi que les nouveaux principes de l'IDM ont permis de faire évoluer ces techniques. Les transformations s'expriment maintenant directement entre les syntaxes abstraites des différents DSML et permettent ainsi de se concentrer sur les concepts et de gagner alors en abstraction. Malgré tout, des travaux sont encore nécessaires afin de formaliser ces techniques et pouvoir ainsi valider et vérifier les transformations écrites. Par exemple, les transformations permettant de générer du code à partir d'un modèle sont assimilables à une compilation qu'il est indispensable dans certains cas de certifier. D'autre part, si QVT offre un cadre très large pour la description de transformation de modèle en couvrant une large partie des approches possibles, ce n'est généralement pas le cas des implantations actuellement disponibles. En effet, la plupart des langages n'implantent qu'une partie du standard QVT, prévu grâce aux niveaux de conformité définis par l'OMG [50, §2].

Références

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] The ATLAS MegaModel Management (AM3) Project Home Page. `http://www.eclipse.org/gmt/am3`, 2007. INRIA ATLAS.

[3] Colin Atkinson and Thomas Kuhne. Model-Driven Development : A Metamodeling Foundation. *IEEE Software*, 20(5) :36–41, 2003.

[4] INRIA ATLAS. KM3 : Kernel MetaMetaModel. Technical report, LINA & INRIA, Nantes, August 2005.

[5] Jean Bézivin. *La transformation de modèles*. INRIA-ATLAS & Université de Nantes, 2003. Ecole d'Eté d'Informatique CEA EDF INRIA 2003, cours #6.

[6] Jean Bézivin, Frédéric Jouault, Peter Rosenthal, and Patrick Valduriez. Modeling in the Large and Modeling in the Small. In U. Aßmann, M. Aksit, and A. Rensink, editors, *Model Driven Architecture, European MDA Workshops : Foundations and Applications, MDAFA 2003 and MDAFA 2004, , June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004, Revised Selected Papers*, volume 3599 of *Lecture Notes in Computer Science*, pages 33–46, Twente, The Netherlands, 2005. Springer.

[7] Jean Bezivin and Ivan Kurtev. Model-based Technology Integration with the Technical Space Concept. In *Metainformatics Symposium*, Esbjerg, Denmark, 2005. Springer-Verlag.

[8] Xavier Blanc. *MDA en action*. EYROLLES, March 2005.

[9] Franck Budinsky, David Steinberg, and Raymond Ellersick. *Eclipse Modeling Framework : A Developer's Guide*. Addison-Wesley Professional, 2003.

[10] Jean Bézivin. In Search of a Basic Principle for Model Driven Engineering. *CEPIS, UPGRADE, The European Journal for the Informatics Professional*, V(2) :21–24, 2004.

[11] Jean Bézivin. Sur les principes de base de l'ingénierie des modèles. *RSTI-L'Objet*, 10(4) :145–157, 2004.

[12] Jean Bézivin. On the unification power of models. *Software and System Modeling (SoSym)*, 4(2) :171–188, 2005.

[13] Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *Proceedings of the 16th IEEE international conference on Automated Software Engineering (ASE)*, page 273, San Diego, USA, 2001. IEEE Computer Society Press.

[14] Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. On the Need for Megamodels. In *Proceedings of the OOPSLA/GPCE : Best Practices for Model-Driven Software Development workshop*, 2004.

[15] Tony Clark, Andy Evans, Paul Sammut, and James Willans. Applied Metamodelling – A Foundation for Language Driven Development. version 0.1, 2004.

[16] Benoît Combemale. *Approche de métamodélisation pour la simulation et la vérification de modèle – Application à l'ingénierie des procédés*. PhD thesis, INPT ENSEEIHT, July 2008.

[17] Patrick Cousot. Methods and Logics for Proving Programs. In *Handbook of theoretical computer science (vol. B) : formal models and semantics*, pages 841–994. MIT Press, Cambridge, MA, USA, 1990.

[18] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[19] Karsten Ehrig, Claudia Ermel, Stefan Hänsgen, and Gabriele Taentzer. Towards Graph Transformation Based Generation of Visual Editors Using Eclipse. *Electr. Notes Theor. Comput. Sci*, 127(4), 2005.

[20] The Eclipse Modeling Framework (EMF). `http://www.eclipse.org/emf`, 2007. Eclipse.

[21] Patrick Farail, Pierre Gaufillet, Agusti Canals, Christophe Le Camus, David Sciamma, Pierre Michel, Xavier Crégut, and Marc Pantel. The TOPCASED project : a Toolkit in OPen source for Critical Aeronautic SystEms Design. In *3rd European Congress* EMBEDDED REAL TIME SOFTWARE *(ERTS)*, Toulouse, France, January 2006.

[22] Jean-Marie Favre. Towards a Basic Theory to Model Model Driven Engineering. In *Workshop on Software Model Engineering (WISME), joint event with UML ?2004*, Lisboa, 2004.

[23] Jean-Marie Favre, Jacky Estublier, and Mireille Blay. *L'Ingénierie Dirigée par les Modèles : au-delà du MDA*. Informatique et Systèmes d'Information. Hermes Science, lavoisier edition, February 2006.

[24] Erich Gamma, Richard Helm, and Ralph Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[25] The Generic Eclipse Modeling System (GEMS). `http://sourceforge.net/projects/gems`, 2007.

[26] Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation : The Missing Link of MDA. In A. Corradini, H. Ehrig, H. Kreowski, and G. Rozenberg, editors, *Proceedings of the First International Conference on Graph Transformation (ICGT)*, volume 2505 of *Lecture Notes in Computer Science*, pages 90–105, Barcelona, Spain, October 2002. Springer.

[27] The Graphical Modeling Framework (GMF). `http://www.eclipse.org/gmf`, 2007. Eclipse.

[28] David Harel and bernhard Rumpe. Modeling Languages : Syntax, Semantics and All That Stuff, Part I : The Basic Stuff. Technical report, Mathematics & Computer Science, Weizmann Institute Of Science, Weizmann Rehovot, Israel, August 2000.

[29] David Harel and Bernhard Rumpe. Meaningful Modeling : What's the Semantics of "Semantics" ? *Computer*, 37(10) :64–72, 2004.

[30] Jan Hendrik Hausmann. *Dynamic Meta Modeling – A Semantics Description Technique for Visual Modeling Languages*. PhD thesis, University of Paderborn, 2005.

[31] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. TCS : a DSL for the specification of textual concrete syntaxes in model engineering. In S. Jarzabek, D. Schmidt, and T. Veldhuizen, editors, *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 249–254, Portland, Oregon, USA, October 2006. ACM.

[32] Frédéric Jouault. *Contribution à l'étude des langages de transformation de modèles*. PhD thesis, Université de Nantes, September 2006.

[33] Frédéric Jouault and Jean Bézivin. KM3 : a DSL for Metamodel Specification. In *Proceedings of the IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, volume 4037 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2006.

[34] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Satellite Events at the MoDELS 2005 Conference, Proceedings of the Model Transformations in Practice Workshop*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138, Montego Bay, Jamaica, 2005. Springer.

[35] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, Jyväskylä, Finland, June 1997.

[36] Anneke Kleppe. MCC : A Model Transformation Environment. In *Proceedings of the First European Conference Model Driven Architecture - Foundations and Applications (ECMDA-FA)*, pages 173–187, Bilbao, Spain, July 2006.

[37] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained. The Model Driven Architecture : Practice and Promise*. Addison-Wesley, 2003.

[38] Ivan Kurtev, Jean Bézivin, and Mehmet Aksit. Technological Spaces : An Initial Appraisal. In *CoopIS, DOA'2002 Federated Conferences, Industrial track*, Irvine, 2002.

[39] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason IV, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The Generic Modeling Environment. In *Proceedings of the IEEE Workshop on Intelligent Signal Processing (WISP)*, Budapest, Hungary, May 2001.

[40] Merlin generator. `http://sourceforge.net/projects/merlingenerator`, 2007.

[41] Joaquin Miller and Jishnu Mukerji. *Model Driven Architecture (MDA) 1.0.1 Guide*. Object Management Group, Inc., June 2003.

[42] Marvin Minsky. Matter, mind, and models. *Semantic Information Processing*, pages 425–432, 1968.

[43] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-Languages. In L. Briand and C. Williams, editors, *Proceedings of the 8th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278, Montego Bay, Jamaica, October 2005. Springer.

[44] Pierre-Alain Muller, Frédéric Fondement, Franck Fleurey, Michel Hassenforder, Rémi Schneckenburger, Sébastien Gérard, and Jean-Marc Jézéquel. Model-driven analysis and synthesis of textual concrete syntax. *Journal of Software and Systems Modeling (SoSyM)*, 2008. Online first.

[45] Object Management Group, Inc. *Software Process Engineering Metamodel (SPEM) 1.1*, January 2005.

[46] Object Management Group, Inc. *Meta Object Facility (MOF) 2.0 Core Specification*, January 2006. Final Adopted Specification.

[47] Object Management Group, Inc. *Object Constraint Language (OCL) 2.0 Specification*, May 2006.

[48] Object Management Group, Inc. *Unified Modeling Language (UML) 2.1.2 Infrastructure*, November 2007. Final Adopted Specification.

[49] Object Management Group, Inc. *Unified Modeling Language (UML) 2.1.2 Superstructure*, November 2007. Final Adopted Specification.

[50] Object Management Group, Inc. *Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification, version 1.0*, April 2008.

[51] Mark Richters and Martin Gogolla. Validating UML Models and OCL Constraints. In A. Evans, S. Kent, and B. Selic, editors, *Proceedings of the 3rd International Conference UML'00 – The Unified Modeling Language*, volume 1939 of *Lecture Notes In Computer Science*, pages 265–277. Springer, October 2000.

[52] Ed Seidewitz. What models mean. *IEEE Software*, 20(5) :26–32, 2003.

[53] The Sintaks Project Home Page. `http://www.kermeta.org/sintaks`, 2007. INRIA TRISKELL.

[54] Richard Soley. *Model Driven Architecture (MDA), Draft 3.2*. Object Management Group, Inc., November 2000.

[55] TCS. `http://www.eclipse.org/gmt/tcs/`, 2007. INRIA ATLAS.

[56] Tiger. `http://tfs.cs.tu-berlin.de/~tigerprj`, 2007.

[57] Toolkit in OPen source for Critical Application & SystEms Development. `http://www.topcased.org/`, 2007. TOPCASED Consortium.

[58] Eric Vépa, Jean Bézivin, Hugo Brunelière, and Frédéric Jouault. Measuring Model Repositories. In *Proceedings of the Model Size Metrics Workshop at the MoDELS/UML 2006 conference*, Lecture Notes In Computer Science, Genoava, Italy, 2006. Springer.

[59] Jos Warmer and Anneke Kleppe. *The Object Constraint Language : Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., 2003.