HAL
archives-ouvertes.fr

Ingénierie Dirigé par les Modèles : du design-time au runtime

Jean-Marc Jézéquel

▶ **To cite this version:**

Jean-Marc Jézéquel. Ingénierie Dirigé par les Modèles : du design-time au runtime. Génie Logiciel - Ingéniérie dirigée par les modèles, Revue Génie Logiciel - AFCET, 2010. <inria-00504666>

HAL Id: inria-00504666
https://hal.inria.fr/inria-00504666

Submitted on 21 Jul 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.

Ingénierie Dirigée par les Modèles : du *design-time* au *runtime*

Jean-Marc Jézéquel

Résumé : Les modèles sont utilisés depuis longtemps en sciences et en ingénierie comme un outil fondamental de gestion de la complexité. La modélisation permet de séparer des préoccupations en abstrayant des aspects spécifiques de la réalité pour des objectifs précis. Cette approche est devenue relativement populaire ces dernières années pour faire face à des préoccupations d'analyse et de conception, en s'appuyant notamment sur des langages de modélisation de la famille d'UML. Bien sûr, la séparation des préoccupations est d'un intérêt limité si on ne peut pas recomposer ces préoccupations automatiquement. La composition automatique de modèles permet en effet de gérer des variations de préoccupations de conception par exemple dans le contexte d'opérations de maintenance ou dans celui de l'ingénierie des lignes de produits. Allant au-delà de la résolution de cette problématique dans les phases de conception, nous montrons comment la composition de modèles peut aussi être utilisée pendant l'exécution d'un système pour spécifier et gérer des systèmes logiciels dynamiquement adaptatifs, ici conceptualisés comme des lignes de produits logiciels dynamiques. Montrant comment l'IDM à l'exécution peut aider les ingénieurs à maîtriser la complexité des systèmes adaptatifs tout en offrant un haut degré d'automatisation et de validation.

Mots clés : ingénierie dirigée par les modèles, modélisation, ligne de produits, variabilité, auto-adaptation

1. INTRODUCTION

La société d'aujourd'hui dépend de plus en plus de logiciels devant être disponibles quasiment sans interruption et devant s'adapter en permanence aux différentes conditions environnementales ainsi qu'aux changements d'exigences [2]. Ces systèmes, qu'on peut qualifier de dynamiquement adaptatifs, présentent de hauts degrés de variabilité fonction à la fois des besoins utilisateurs et des fluctuations de leur contexte d'exécution. Du point de vue de la gestion de la variabilité, les systèmes dynamiquement adaptatifs peuvent être conceptualisés comme des lignes de produits dynamiques dans lesquelles la variabilité serait résolue à l'exécution [3]. En effet, la notion de lignes de produits logiciels permet de mettre en évidence dans une famille de logiciels les parties communes à tous les produits de la ligne et les parties variables, différentes d'un produit à l'autre. Par exemple, tous les téléphones portables d'une ligne de produits donnée pourraient accepter le protocole GSM, tandis que seuls certains d'entre eux seraient équipés en plus de modules de communication Wifi et/ou 3G. On parle de *lignes de produits* logiciels lorsque la décision de configurer tel ou tel téléphone portable de tel ou tel module de communication est prise lors de la conception [9]. On parlera de *lignes de produits dynamiques* si cette décision est prise au moment de l'exécution pour, par exemple, répondre dynamiquement à un changement de contexte lié à la disponibilité de tel ou tel réseau de communication.

Un problème bien connu des approches lignes de produits, et donc hérité dans les lignes de produits dynamiques, est que le nombre de configurations possibles d'une ligne de produits croît de manière combinatoire avec le nombre de points de variations (aussi appelés *dimensions de variabilité*, comme dans notre exemple le mode de communication d'un téléphone portable), et pour chaque point de variation avec le nombre de variantes (GSM, Wifi, ou 3G). Dans le pire des cas où toutes les variantes seraient indépendantes les unes des autres, le nombre total de configurations possibles pour *n* variantes serait 2^n.

Dans les systèmes adaptatifs, la notion de mode de fonctionnement peut ainsi être vue comme une configuration particulière de leur conceptualisation sous forme de ligne de produits dynamique. Une difficulté supplémentaire de ces lignes de produits dynamiques consiste à s'assurer que le passage d'une configuration donnée à une autre (i.e. le changement de mode) peut se faire de manière maîtrisée. Or, pour n variantes, si on a 2^n modes, on obtient potentiellement jusqu'à (2^n x 2^{n-1}) transitions entre ces modes, chacune de ces transitions devant être programmée et validée avec le niveau de rigueur nécessaire [2].

Dans cet article, nous proposons de maîtriser la complexité des systèmes dynamiquement adaptatifs à l'aide de modèles capturant de manière fine la décomposition de lignes de produits dynamiques suivant leurs dimensions de variabilité. Ces

modèles sont ensuite classiquement exploités au moment de la conception à des fins de génération de code et de validation, mais aussi à l'exécution pour piloter la reconfiguration d'un système en fonction des changements de son environnement [1].

2. MODÉLISATION : DÉCOMPOSITION ET COMPOSITION

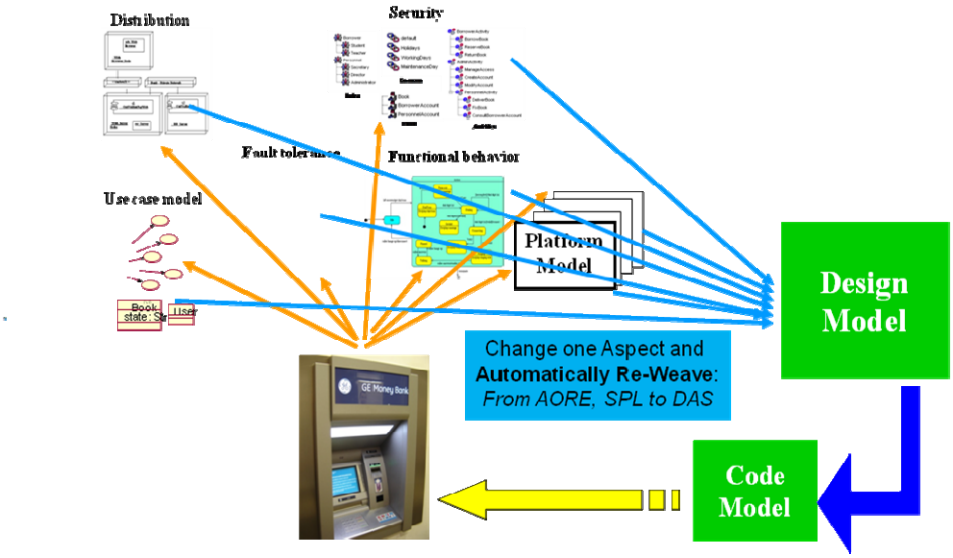

Figure 1 : La modélisation est orientée-aspect par définition

En informatique comme dans les autres sciences, on s'appuie de plus en plus sur la modélisation pour essayer de maîtriser la complexité des systèmes informatiques, tant pour produire le logiciel (conception) que pour le valider (test). Par définition, la modélisation est en effet l'utilisation efficace d'une représentation simplifiée d'un aspect de la réalité pour un objectif donné. Si la modélisation peut-être vue comme la séparation des différents besoins fonctionnels et préoccupations extra-fonctionnelles (telles que sécurité, fiabilité, efficacité, performances, ponctualité, flexibilité, etc.) issus des exigences, la conception du logiciel consiste donc réciproquement à fusionner (ou tisser) ces différents aspects dans un modèle de conception détaillée qui pourra donner lieu à de la génération de code [6] ou de la génération de tests [7].

Cette activité de conception, qui vise donc à intégrer un ensemble de préoccupations variées dans un tout cohérent, reste encore aujourd'hui le plus souvent de nature artisanale, même si certains canevas logiciels peuvent aider les développeurs en fournissant des solutions pré-câblées dans des domaines bien maîtrisés (J2E, Spring, etc.). Ceci implique que lorsqu'on veut choisir une alternative de conception dans l'une des dimensions de variabilité d'un logiciel, par exemple le choix d'une nouvelle plate-forme ou encore d'un nouveau modèle de tolérance aux défaillances, le concepteur artisanal doit en quelque sorte rejouer sa démarche de conception en tenant compte de ces nouvelles préoccupations. Un des enjeux de l'ingénierie dirigée par les modèles [5] est bien sûr d'aider à automatiser cette activité de conception en fournissant le moyen de composer automatiquement des modèles représentant des aspects nécessairement partiels des solutions à mettre en œuvre. Puisqu'un modèle est par définition l'abstraction d'un aspect de la réalité (parler de modélisation orientée aspect est en cela un pléonasme), la composition de modèles peut être vue comme du tissage d'aspects au niveau des modèles, ce qui est probablement le bon niveau d'abstraction pour une ingénierie des modèles fiable et efficace [3,6].

Si cette composition de modèles se fait au moment de la conception, alors on obtient une approche efficace pour dériver automatiquement des produits dans une ligne de produits [9]. Si maintenant cette composition se fait au moment de l'exécution, cela donne un cadre de haut niveau pour reconfigurer des systèmes dynamiquement adaptatifs.

3. PRINCIPE DU PILOTAGE DES SYSTÈMES ADAPTATIFS PAR LES MODÈLES

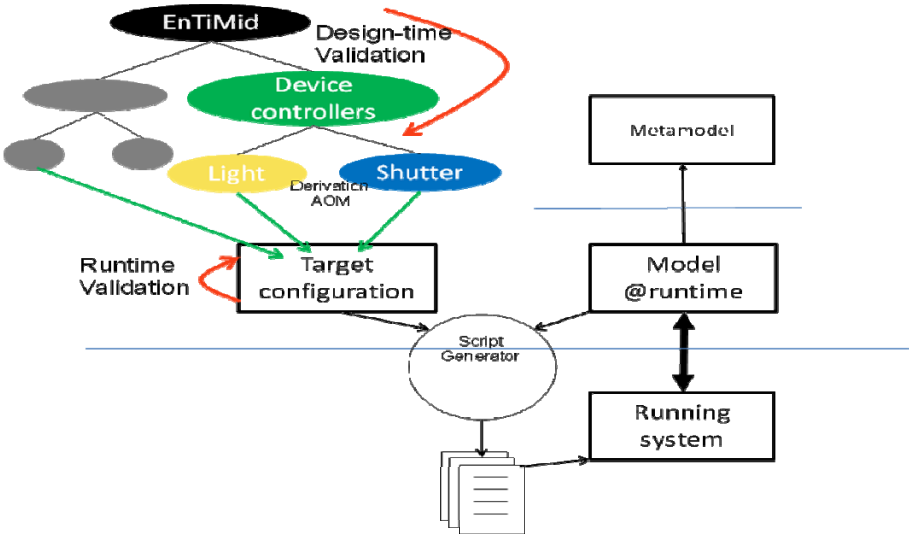

Figure 2 : Principe de l'utilisation du tissage de modèles au runtime pour piloter la reconfiguration dynamique

La figure 2 décrit le principe de la solution que nous avons proposée dans le projet DiVA [1] pour piloter la reconfiguration dynamique d'une application à base de composants sur une plate-forme d'exécution OSGi. À un instant donné, le système en cours d'exécution est composé d'un ensemble de *bundles* OSGi dont on a rendu les interconnexions explicites et manipulables par des scripts de reconfiguration. Cette configuration de composants est abstraite sous forme d'un modèle de composants explicite existant au runtime à la manière d'une couche de réflexion (Model@runtime). La différence avec les couches de réflexion classiques existant dans les architectures à objets ou à composants est qu'ici on a un véritable modèle (défini par un métamodèle) dont la connexion avec le système en cours d'exécution se fait par un lien de causalité explicite : à chaque fois que la configuration des composants du système en cours d'exécution change, le modèle mis à jour par un mécanisme classique d'observation.

À l'autre bout de la chaîne, la variabilité de l'application est capturée sous la forme d'un diagramme de *features*, comme cela est effectué classiquement dans le contexte des lignes de produits logiciels. Par exemple, les appareils contrôlés dans un système de domotique peuvent être une combinaison de lampes et de volets roulants. Chaque feuille du diagramme de *features* représente un atome de variabilité et est associé à un fragment d'architecture, qui se présente comme un *aspect* vis-à-vis du modèle de composants existant au runtime. En effet, ce fragment d'architecture décrit quels composants logiciels doivent être déployés lorsqu'on veut utiliser une caractéristique particulière (par exemple la commande d'un volet roulant), ainsi que la manière dont ces composants doivent être connectés entre eux et avec le reste de l'architecture. Comme le reste de l'architecture est par définition mouvante, cette interconnexion est décrite de manière déclarative grâce à un *pattern* d'architecture, ce qu'on appelle un point de coupure (ou *pointcut*) dans les approches orientées aspect. Ce *pointcut* capture toutes les informations nécessaires pour facilement « brancher » notre fragment d'architecture (jouant ici le rôle de ce qu'on appelle un *advice* en orienté aspect) sur l'architecture de base. Comme le montre la figure 3, un tel modèle d'aspect se compose de trois parties :

• Un modèle *d'advice*. Ce fragment d'architecture spécifie ce qui est nécessaire pour réaliser la variante associée. Ce modèle n'est pas nécessairement conforme au métamodèle de base : par exemple dans le cas de ces diagrammes de composants, il peut rester des ports non connectés. Le modèle de base devra apporter les éléments manquants nécessaires pour rendre le modèle d'*advice* cohérent lors du tissage d'aspects. Plus formellement, on montre que le métamodèle du modèle de l'*advice* doit être le plus haut super-type (au sens de [8]) du métamodèle du modèle de base qui en contient tous les concepts mais qui relâche toutes les contraintes.
• Un modèle de *pointcut*. Ce fragment d'architecture précise les composants et les liaisons que le modèle aspect attend du modèle de base, c'est à dire la spécification de l'endroit où l'aspect doit être tissé. Plus le modèle de *pointcut* est précis, plus l'ensemble des lieux potentiels où l'aspect peut être tissé sera petit, et vice versa. Par exemple, s'il n'est pas spécifié dans le modèle de *pointcut*, le type d'un composant pourra s'adapter à n'importe quel composant de l'architecture de base, indépendamment de son type réel.
• Un protocole de composition. Celui-ci décrit la façon d'intégrer le modèle d'*advice* dans le modèle de *pointcut*, et donc

de contextualiser le tissage de l'aspect dans le modèle de base (non développé ici).

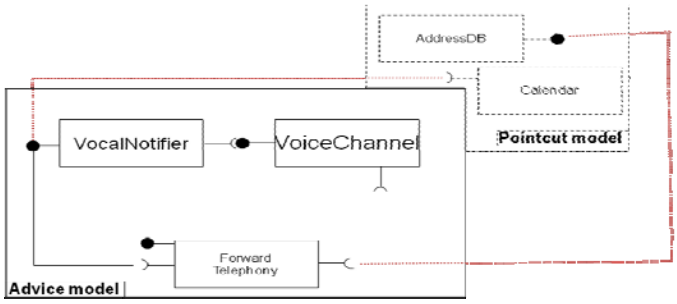

Figure 3 : Modèle d'un aspect de type « composant »

Lors de l'exécution du système adaptatif, un certain nombre de capteurs sont déployés pour contrôler les changements dans l'environnement ou les changements d'exigences des utilisateurs. Ces événements sont fournis à un moteur de raisonnement qui décidera, si une reconfiguration s'avère nécessaire, quel est le nouvel ensemble de variantes qui sont requises pour répondre à la nouvelle situation.

Cet ensemble de variantes est alors composé dynamiquement par notre tisseur d'aspects SmartAdapter pour produire le modèle souhaité de l'architecture de composants, qu'on appellera le modèle cible de la reconfiguration. Un certain nombre de validations sont ensuite effectuées sur ce modèle cible, afin de, par exemple, vérifier que des invariants importants du système sont respectés. Dans le cas contraire, il est demandé au raisonneur de chercher une autre configuration de variantes qui sont à nouveau composées en un modèle cible jusqu'à l'obtention d'un modèle satisfaisant.

La dernière étape consiste à comparer le modèle cible de la reconfiguration avec le modèle actuel de l'architecture de composants, et d'après la différence entre ces deux modèles de synthétiser le script de reconfiguration permettant de passer du source à la cible (en utilisant les techniques classiques de génération de code de l'ingénierie des modèles), et finalement d'exécuter ce script de reconfiguration sur la plate-forme d'exécution afin d'obtenir la reconfiguration décidée par le raisonneur.

4. IMPLANTATION AVEC KERMETA ET SMARTADAPTER

Ce pilotage de la reconfiguration dynamique d'une application à base composants par du tissage de modèles à l'exécution a été réalisé dans l'environnement Kermeta [4], qui est une plate-forme MDE (Model-Driven Engineering) pour la construction des environnements de développement riches autour de méta-modèles en utilisant un paradigme orienté aspects. Kermeta a été conçu pour étendre facilement les méta-modèles avec des fonctionnalités diverses (telles que la sémantique statique, sémantique dynamique, transformations de modèles, la connexion à la syntaxe concrète textuelle ou graphique, etc.), éventuellement exprimées dans des langages hétérogènes. Un méta-langage comme le MOF (Meta Object Facility) supporte en effet déjà une définition orientée-objet de méta-modèles en termes de packages, classes, propriétés et les opérations, ainsi que des constructions plus orientées modèles comme les compositions et les associations entre les classes. MOF ne comprend cependant pas de concepts pour la définition des contraintes ou pour la sémantique opérationnelle. Kermeta peut donc être vu comme une extension de MOF avec un langage impératif pour spécifier des contraintes et des corps d'opérations au niveau d'un méta-modèle.

Le langage d'action de Kermeta est spécialement conçu pour traiter des modèles. Il est impératif et inclut des structures de contrôle classique comme des blocs, des conditionnels et des boucles. Comme le MOF spécifie des structures orientées objet (classes, propriétés et opérations), Kermeta implante les mécanismes orientés objet traditionnels pour l'héritage multiple et la redéfinition de comportements avec une sémantique de liaison dynamique (pour éviter les conflits d'héritage multiple, un mécanisme de sélection simple est disponible dans Kermeta). Comme la plupart des langages orientés objets modernes, Kermeta est statiquement typé, avec support de la généricité et fournit aussi des mécanismes de réflexion et de gestion d'exceptions. Les expressions Kermeta sont fonctionnellement très similaires à celles du langage OCL (Object

Constraint Language) tel que standardisé par l'OMG. En particulier, Kermeta inclut des fermetures lexicales similaires aux itérateurs de collections d'OCL (comme *each, collect, select,* etc.). La librairie standard de Kermeta contient aussi toutes les opérations définies dans le standard OCL. Cet alignement entre Kermeta et OCL permet à des contraintes OCL décrites avec la syntaxe textuelle du standard d'être directement importées et évaluées dans Kermeta. Des pré-conditions et des post-conditions peuvent être ainsi définies pour les opérations de métamodèles et des invariants peuvent être associés à des classes. La machine virtuelle Kermeta possède un mode exécution spécifique qui permet de surveiller ces contrats et de signaler toute violation.

Comme Kermeta est une extension du MOF, un métamodèle MOF peut réciproquement être vu comme un programme orienté objet valide qui se contente de déclarer des packages, des classes, des attributs etc. mais ne fait rien. Kermeta peut-être alors utilisés pour insuffler la vie dans ce métamodèle en introduisant de manière incrémentale des *aspects* pour gérer les préoccupations de sémantique statique, de sémantique dynamique, ou de transformation de modèles.

À cet égard, une des caractéristiques de Kermeta est l'opérateur de composition statique *require* qui permet d'étendre un métamodèle existant avec des nouveaux éléments tels que des propriétés, des opérations, des contraintes ou des classes. Cet opérateur permet la définition de ces différents aspects dans des unités séparées (et donc développables et testables de manière indépendante ou même en parallèle) et de les intégrer automatiquement et à la demande dans le métamodèle. Cette composition est effectuée statiquement et le métamodèle composé subit une vérification de typage afin de vérifier l'intégration correcte de toutes les unités avant toute exécution. Ce mécanisme permet de faciliter la réutilisation de métamodèles préexistants ou de couper de gros métamodèle en petites unités plus réutilisables. La conséquence pratique de l'utilisation de cet opérateur *require* est qu'une métaclasse qui identifie un concept du domaine peut être étendue sans aucune modification physique directe du métamodèle de référence, assurant ainsi l'interopérabilité avec n'importe quel outil de la galaxie Eclipse capable de travailler avec ce métamodèle.

L'environnement Kermeta est lui-même bâti suivant ces principes, et dispose dans l'environnement Eclipse d'un éditeur spécialisé, d'un interpréteur, d'un débogueur, ainsi que d'un compilateur qui permet de fournir à l'utilisateur final des archives Java (JAR) n'ayant aucun lien de dépendance avec l'environnement de développement Kermeta.

Kermeta est donc un outil pour construire des outils pour construire du logiciel. Dans l'exemple de ce projet, Kermeta a été utilisé pour construire le tisseur de modèles génériques SmartAdapter [3]. Ce dernier prend en entrée un modèle de base (défini par métamodèle arbitraire) et un modèle d'aspects (défini par un super type du métamodèle du modèle de base), et produit en sortie un nouveau modèle dans lequel l'aspect a été tissé dans le modèle de base. SmartAdapter fonctionne donc avec n'importe quel métamodèle en appliquant l'algorithme suivant. SmartAdapter commence par traduire le *pointcut* de l'aspect en un ensemble de règles DROOLS qui sont exécutées sur le modèle de base et qui retournent donc un ensemble de morphismes entre les éléments de modèles du *pointcut* et des éléments de modèles du modèle de base (opération d'unification). Les éléments du modèle de base non impactés sont recopiés tels quels dans le modèle résultat, les éléments unifiés entre le modèle de base, le *pointcut* et *l'advice* sont copiés une seule fois dans le modèle résultat et finalement les éléments supplémentaires présents dans *l'advice* sont ajoutés en utilisant les règles décrites dans le protocole de composition. SmartAdapter est un programme Kermeta qui est compilé sous forme d'un jar déployé dans l'architecture cible OSGi, et appelé à chaque fois que la production d'un nouveau modèle de configuration a été décidée par le raisonneur à la suite d'un changement de contexte. Nos premières expériences de tissage d'aspects dans des modèles à l'exécution font apparaître des temps d'exécution de l'ordre de quelques millièmes de seconde, ce qui semble tout à fait raisonnable compte tenu des applications visées.

5. CONCLUSION

Nous avons proposé dans cet article d'utiliser une approche inspirée des lignes de produits dynamiques pour maîtriser la complexité de la conception des systèmes adaptatifs, en particulier eu égard à l'explosion combinatoire de leur nombre de configurations possibles vis-à-vis du nombre de variantes disponibles. Chaque variante est associée au modèle d'un fragment d'architecture de composants, qui se présente comme un modèle d'aspects. Obtenir une nouvelle configuration du système revient à choisir un certain nombre d'aspects et à les tisser à la demande lors de l'exécution pour obtenir un modèle de la configuration de l'architecture cible. Après un ensemble de validations qui permettent un retour en arrière (roll-back) simple et gratuit, le modèle d'architecture cible est comparé au modèle d'architecture de la configuration courante, ce qui permet de générer de manière sûre et efficace un script de reconfiguration.

Cette approche permet de ne plus avoir à représenter toutes les configurations possibles d'un système adaptatif et d'obtenir celles-ci par le tissage au runtime des aspects les plus adaptés à la demande. Cette approche, outillée dans l'environnement Kermeta, est utilisée dans le cadre du projet DiVA pour aider à la gestion de crise dans les aéroports ainsi que pour les aspects adaptatifs d'une application de CRM. Nous l'utilisons aussi pour gérer la configuration et l'adaptation automatique d'une plate-forme logicielle pour la domotique dans le domaine du maintien à domicile de personnes dépendantes.

Remerciements
Je tiens à remercier O. Barais, B. Baudry, B. Morin et G. Nain qui sont les principaux contributeurs aux idées présentées ci-dessus, ainsi que le reste de l'EPI Triskell pour avoir permis de transformer la vision en réalité.

BIBLIOGRAPHIE

[1] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey et Arnor Solberg : *Models at runtime to support dynamic adaptation* ; IEEE Computer, pages 46--53, octobre 2009.
[2] Brice Morin, Olivier Barais, Grégory Nain et Jean-Marc Jézéquel : *Taming dynamically adaptive systems with models and aspects* ; in 31st International Conference on Software Engineering (ICSE'09), Vancouver, Canada, mai 2009.
[3] Brice Morin, Jacques Klein, Olivier Barais et Jean-Marc Jézéquel : *A generic weaver for supporting product lines* ; in International Workshop on Early Aspects at ICSE'08, Leipzig, Allemagne, mai 2008.
[4] Pierre-Alain Muller, Franck Fleurey et Jean-Marc Jézéquel : *Weaving executability into object-oriented meta-languages* ; in S. Kent L. Briand, réd., Proceedings of MODELS/UML'2005, *volume* 3713 of *LNCS*, pages 264-278, Montego Bay, Jamaïque, octobre 2005, Springer.
[5] Jean-Marc Jézéquel, Sébastien Gérard et Benoît Baudry : *L'ingénierie dirigée par les modèles*, chapitre Le génie logiciel et l'IDM : une approche unificatrice par les modèles, Lavoisier, Hermes-science, 2006.
[6] Jean-Marc Jézéquel : *Model-driven design and aspect weaving* ; Journal of Software and Systems Modeling *(*SoSyM*)*, 7(2), pp. 209-218, mai 2008.
[7] Simon Pickin, Claude Jard, Thierry Jéron, Jean-Marc Jézéquel et Yves Le Traon : *Test synthesis from UML models of distributed software* ; IEEE Transactions on Software Engineering, 33(4), pp. 252-268, avril 2007.
[8] Jim Steel et Jean-Marc Jézéquel : *On model typing* ; Journal of Software and Systems Modeling (SoSyM*)*, 6(4), pp. 401-414, decembre 2007.
[9] Tewfik Ziadi et Jean-Marc Jézéquel : *Software Product Lines*, chapitre Product Line Engineering with the UML: Deriving Products, pages 557-586. -- Number ISBN: 978-3-540-33252-7. Springer Verlag, 2006