HAL
open science

Développement d'Applications à Grande Echelle par Composition de Méta-Modèles

German Eduardo Vega Baez

To cite this version:

German Eduardo Vega Baez. Développement d'Applications à Grande Echelle par Composition de Méta-Modèles. Génie logiciel [cs.SE]. Université Joseph-Fourier - Grenoble I, 2005. Français. NNT : . tel-00011325

HAL Id: tel-00011325

https://theses.hal.science/tel-00011325

Submitted on 9 Jan 2006

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.

UNIVERSITE JOSEPH FOURIER – GRENOBLE I

THESE

pour obtenir le grade de

DOCTEUR de l'Université Joseph Fourier de Grenoble

(arrêtés ministériels du 5 juillet 1984 et du 30 mars 1992)

Discipline : Informatique

présentée et soutenue publiquement par

Germàn Eduardo VEGA BAEZ

le 8 décembre 2005

Développement d'Applications à Grande Echelle par Composition de Méta-Modèles

Directeur de thèse :

Jacky ESTUBLIER

JURY :

Yves Ledru, Professeur à l'Université Joseph Fourier	(Président)
Jean-Marc Jézéquel, Professeur à l'Université de Rennes 1	(Rapporteur)
Laurence Duchien, Professeur à l'Université de Lille 1	(Rapporteur)
Joëlle Coutaz, Professeur à l'Université Joseph Fourier	(Examinateur)
Roland Balter, ScalAgent Distributed Technologies	(Examinateur)
Jacky Estublier, Directeur de recherche au CNRS	(Directeur de thèse)

Thèse préparée au sein du Laboratoire LSR – Equipe ADELE

Quand le ciel créa le temps, il en créa suffisamment

Proverbe Mongol

A Adriana, Sergio et Esteban

Remerciements

Je remercie tout d'abord les membres du jury qui m'ont fait l'honneur de participer à ma soutenance de thèse. Je tiens à remercier plus particulièrement mes rapporteurs Mme L. Duchien, professeur à l'Université de Lille 1, et M. J-M. Jézéquel, professeur à l'Université de Rennes 1, pour leurs commentaires et principalement pour l'intérêt qu'ils ont porté sur mes travaux.

Je tiens à exprimer mes plus sincères remerciements à Jacky Estublier, qui m'a accueilli dans son équipe de recherche, et qui, dans son rôle de directeur de thèse, m'a toujours aidé et conseillé, spécialement dans la phase finale de cette thèse.

Merci à tous les membres de l'équipe Adèle. Merci pour leur soutien et pour tous les bons moments que nous sommes passés ensemble. Merci à Anca, Tam et Stéphane avec qui j'ai partagé le quotidien du travail, toujours dans la bonne entente et la bonne humeur. Merci à Sonia et Vincent, toujours présents et prêts à me collaborer. Merci à tous pour votre amitié.

Une reconnaissance spéciale à Jean Marie Favre. Nos longues conversations métaphysiques sur la nature des modèles et du langage ont changé pour toujours ma vision du développement logiciel, et surement contribué à façonner les travaux de cette thèse.

Une pensée chaleureuse à mes amis colombiens à Grenoble : Pilar, Constanza, Mario, Claudia et quelques autres, pour leur amitié et l'aide qu'ils m'ont apportées au cours de ces dernières années.

Je voudrais adresser ma gratitude à ma famille, à mes parents et me frères, qui m'ont toujours soutenu, malgré la distance. Finalement, un grand merci à mes plus chers, mon épouse, Adriana, et mes enfants, Sergio et Esteban, pour leur inépuisable amour.

Résumé

Parmi les approches de génie logiciel, l'*Ingénierie Dirigée par les Modèle*s (IDM) vise à fournir un cadre qui permet de s'attaquer à la complexité croissante du développement des logiciels dans un paradigme unificateur : elle conçoit l'intégralité du cycle de vie du logiciel comme un processus de production, de raffinement itératif et d'intégration de modèles.

Les travaux de cette thèse se placent dans cette mouvance. Nous nous intéressons particulièrement à la problématique de la gestion de la complexité et de la diversité de domaines métier impliqués dans les applications de grande taille, et nous proposons de l'approcher sous une optique IDM.

Un domaine est un champ d'expertise, potentiellement partagé par de multiples applications. La connaissance et le savoir-faire développé au sein de ce domaine deviennent des atouts majeurs. Cette expertise peut être formalisée, et surtout réutilisé, sous la forme d'un langage de modélisation dédié, un *Domain Specific Language* (DSL).

Nous proposons une démarche basée sur la description d'un système par divers modèles exprimés dans des langages de modélisation dédiés différents. La composition de modèles permet de modéliser des applications complexes couvrant plusieurs domaines simultanément.

L'originalité de notre démarche est que, d'une part, chaque DSL est spécifié par un méta-modèle suffisamment précis pour pouvoir construire, de façon semi-automatique, une machine virtuelle spécialisée du domaine ; c'est cette machine virtuelle qui permet l'exécution des modèles associés. D'autre part, il est possible de composer ces méta-modèles pour définir de nouveaux domaines plus complexes. La composition de méta-modèles permet d'améliorer la modularité, d'obtenir de taux de réutilisation plus importants, et surtout d'obtenir des fonctionnalités plus vastes et sophistiquées que celles des domaines composés.

Mots clés: Ingénierie dirigée par les modèles, langages dédiés de modélisation, DSL, domaine, conception dirigée par le domaine, méta-modélisation, modèles exécutables, composition de méta-modèles.

Composing Domain-Specific Meta-Models for Large-Scale Software Engineering

Abstract

Keywords: MDA, MDSE, MDD, DSL, meta models, executable models, meta model composition, model composition.

Model Driven Software Engineering (MDSE) is a Software Engineering approach that addresses the ever increasing complexity of software development and maintenance through a unified conceptual framework in which the whole software life cycle is seen as a process of model production, refinement and integration.

This thesis contributes to this MDSE trend. We focus mainly on the issues raised by the complexity and diversity of the domains of expertise involved in large size software applications, and we propose to address these issues in an MDSE perspective.

A domain is an expertise area, potentially shared by many different software applications. The knowledge and know-how in a domain are major assets. This expertise can be formalized and reused when captured by a Domain Specific Language (DSL).

We propose an approach in which the target system is described by different models, written in different DSL. In this approach, composing these different models allows for modeling complex application covering simultaneously different domains.

Our approach is an original contribution in that each DSL is specified by a meta model precise enough to build, in a semi automatic way, a domain virtual machine; it is this virtual machine that interprets the domain models. Then, it is possible to compose these meta models to define new and more complex domains. Meta model composition increases modularity and reuse, and allows building domain with much larger functional scope than possible with traditional approaches.

Table des Matières

1. INTRODUCTION. 15
   1.1 Contexte Général. 15
   1.2 Les applications aux multiples facettes. 16
   1.3 Les domaines specialises. 18
   1.4 Notre Proposition. 19
   1.5 Plan de la thèse. 19
2. ETAT DE L'ART. 21
   2.1 L'Ingénierie Dirigée par les Modèles. 21
      2.1.1 Le concept de modèle. 22
         La nature des modèles. 23
         Modèles statiques et dynamiques. 23
         Modèles descriptifs et spécification. 23
         Modèles et séparation des préoccupations. 24
         Précision des modèles. 24
         Modèles et niveaux d'abstraction. 25
      2.1.2 Les concepts de méta-modèle et conformité. 25
         Langages de modélisation. 25
         Méta-modèle et relation de conformité. 26
         L'architecture de méta-modélisation. 28
      2.1.3 Synthèse. 29
   2.2 Ingénierie des Domaines. 29
      2.2.1 Le concept de domaine. 29
      2.2.2 Ingénierie des domaines. 30
         Analyse du domaine. 31
         Conception et implémentation du domaine. 32
         Ingénierie des applications. 33

2.2.3 Domaines et applications. 33
2.2.4 Un regard basé sur l'IDM. 33
2.2.5 Synthèse. 35
2.3 Les Lignes de Produits Logiciels. 35
2.3.1 Approches Génératives. 36
2.3.2 Approches générative basée sur diagrammes de Features. 37
Dérivation d'un produit. 38
Un regard basé sur l'IDM. 39
2.3.3 Approches générative basée sur UML. 40
Dérivation d'un produit. 41
Un regard basé sur l'IDM. 41
2.3.4 Approches générative basée sur des langages dédiés. 42
2.3.5 Synthèse. 44
2.4 Conclusion. 44
3. DOMAINES EXECUTABLES. 47
3.1 Spécification du Domaine. 47
3.1.1 L'analyse du domaine. 48
Le domaine de la Gestion de Configurations. 48
Une vision Ligne de Produits. 49
Une vision par séparation des préoccupations. 51
3.1.2 La conceptualisation du domaine. 53
3.1.3 Spécification du méta-modèle. 56
3.2 Les Modèles Exécutables. 58
3.3 Machine Virtuelle du Domaine. 60
3.3.1 Représentation de l'état de l'exécution. 60
Le lien de conformité. 61
Correspondance avec la conceptualisation. 62
3.3.2 Dynamique de l'exécution. 63
Le lien de conformité. 64
Réification des concepts absorbés. 65
Le choix du langage de méta-programmation. 66
3.4 Composants et Couche de Médiation. 67
3.4.1 Composants et fédération de composants. 68

3.4.2 La couche de médiation. 70
3.5 Variations de la Réalisation. 74
3.5.1 Substitution de composants. 74
3.5.2 Les options de réalisation. 75
3.6 La notion d'Application. 76
3.7 Conclusions. 76
4. COMPOSITION DE DOMAINES. 79
4.1 Intégration des Points de Vue. 79
4.2 Un scénario de composition. 81
Domaine de Gestion de Procédés. 81
Fonctionnalités du domaine composite. 84
4.3 Composition Conceptuelle. 85
4.4 Composition des Méta-Modèles. 87
4.5 Composition de Modèles. 90
4.6 Composition de Machines Virtuelles. 91
4.6.1 Représentation de l'état de l'exécution. 91
4.6.2 Dynamique de l'exécution. 93
4.7 Exécution Concrète. 96
Spécification de la configuration. 96
Exécution de l'application concrète. 97
4.8 Composition Multi-Niveaux. 98
4.9 Conclusion. 99
5. CONCLUSION ET PERSPECTIVES. 101
5.1 Applications à Multiples Domaines. 101
5.2 L'amalgame conceptualisation/Méta-Modèle. 102
5.3 La Machine Virtuelle du Domaine. 103
5.4 La Composition de Méta-Modèles. 104
6. BIBLIOGRAPHIE. 107

Table des Figures

figure 1. Modèle et Représentation [Fav04] 22
figure 2. Langage de modélisation 26
figure 3. Méta-modèle et relation de conformité 27
figure 4. Architecture de méta-modélisation MDA® 28
figure 5. Processus de développement dirigé par les domaines [LO00] 31
figure 6. Ingénierie des domaines, concepts de base 34
figure 7. Modèles impliqués dans un domaine 34
figure 8. Diagramme de Features [BTR05].....37
figure 9. Approches par features, regard IDM 39
figure 10. Dérivation à partir du diagramme d'activité UML étendu [CA05].....40
figure 11. Approches UML étendu, regard IDM 41
figure 12. Approches génératives basés sur des langages dédiés [CE00].....43
figure 13. Approches basées sur langages dédiés, regard IDM 43
figure 14. Diagramme de features - Objets Versionnés [KG04].....49
figure 15. Diagramme de features - Relations versionnées [KG04] 50
figure 16. Décomposition par domaines 52
figure 17. Diagramme de Features du Domaine de Produit.....53
figure 18. Relation Conceptualisation – Langage [GPvS05].....54
figure 19. Modèle du Domaine de Produit.....55
figure 20. Contraintes du modèle du Domaine de Produit.....57
figure 21. Outil de modélisation du domaine de produit 57
figure 22. Démarche globale d'Ingénierie du Domaine.....58
figure 23. Méta-Modèle d'exécution du Domaine de Produit 61
figure 24. Relation Conceptualisation – Langage - Exécution 63
figure 25. Machine Virtuelle du Domaine de Produit.....63
figure 26. Lien de conformité - `Revision` 64

figure 27. Lien de conformité - `AttributeDefinition`.....64
figure 28. Composition par coordination 69
figure 29. Relation conceptualisation – exécution concrète.....70
figure 30. Architecture d'exécution du domaine.....71
figure 31. Contrat de Coordination, en AspectJ.....72
figure 32. Environnement Mélusine : gestion de composants 74
figure 33. Environnement Mélusine : gestion d'options.....76
figure 34. Composition conceptuelle 80
figure 35. Modèle de Procédé 82
figure 36. Modèle du Domaine de Gestion de Procédé 83
figure 37. Démarche globale de composition 85
figure 38. Composition de méta-modèles statiques 87
figure 39. Méta-modèle statique de la composition.....88
figure 40. Composition de Modèles.....90
figure 41. Méta-modèle du domaine composite.....92
figure 42. Extension de Méta-Modèle, en AspectJ 93
figure 43. Composition de Machines Virtuelles, en AspectJ.....94
figure 44. Composition de Machines Virtuelles, en AspectJ.....95
figure 45. Environnement Mélusine : gestion de domaines composites.....96
figure 46. Exécution d'une application composite.....97
figure 47. Composition multi-niveaux 98

1. INTRODUCTION.

1.1 CONTEXTE GENERAL.

Le génie logiciel cherche à faire du développement des logiciels un processus industriel, tout en s'attaquant à la complexité croissante des logiciels de grande taille. L'objectif est d'améliorer la qualité et la fiabilité du produit logiciel, tout en augmentant la productivité et l'efficacité de son processus de production.

Même si la maîtrise de la complexité reste le souci principal du génie logiciel, le développement logiciel doit faire face actuellement à de nouveaux défis. Comme le remarquent Clark et al. [CES+04], ces défis sont souvent une combinaison de trois facteurs principaux : complexité, diversité et évolution.

La complexité des logiciels de grande taille est évidemment liée aux besoins, plus nombreux et complexes, propres aux domaines d'application plus sophistiqués auxquels on s'intéresse. Cependant on observe qu'elle devient aussi, et de plus en plus, liée à la complexité de la plate-forme technologique de base. Ceci est particulièrement visible dans le cas des applications réparties, où les problèmes d'infrastructure (tels que la distribution, la sécurité ou les défaillances) accroissent considérablement la complexité propre au domaine métier.

Un deuxième facteur à considérer est l'énorme diversité et hétérogénéité auxquelles doivent faire face les développeurs de logiciel. Cette diversité se manifeste sur plusieurs axes. Tout d'abord, tout logiciel de grande taille comprend une multiplicité de domaines, métiers et techniques, qu'il faut maîtriser ; ceci implique une forte spécialisation des développeurs, et nécessite des mécanismes pour réconcilier les besoins de différentes spécialités.

A cette diversité vient s'ajouter l'hétérogénéité des outils, artefacts et technologies d'implémentation qui doivent être mise en œuvre dans tout développement substantiel. Dans les diverses phases du cycle de vie du logiciel, le développeur doit produire un grand nombre d'artefacts de diverse nature, se confronter à la variété d'outils, souvent incompatibles, mis à sa disposition, et assurer l'intégration et la cohérence de ces différents artefacts.

Un dernier facteur, intimement lié à la diversité, est l'évolution. Chacun des axes de diversité peut se révéler être une dimension d'évolution du logiciel. Les exigences des utilisateurs évoluent, les domaines couverts par l'application changent au fur et à mesure que le cahier des charges évolue, les outils de développement changent, l'infrastructure technologique devient obsolète et se renouvelle sans cesse. La maîtrise de la complexité devient alors imbriquée à la gestion de l'évolution.

Toute approche de génie logiciel doit tenir compte de ce contexte, et doit permettre de s'attaquer à la complexité des domaines visés, tout en gardant la flexibilité pour faire évoluer les solutions informatiques, même en face d'une infrastructure technologique mouvante.

Parmi les approches de génie logiciel, l'*Ingénierie Dirigée par les Modèle*s (IDM) vise à fournir un cadre qui permet d'aborder toute cette problématique dans un paradigme unificateur [JGM+05] : elle conçoit l'intégralité du cycle de vie du logiciel comme un processus de production, de raffinement itératif et d'intégration de modèles.

L'utilisation de modèles (exprimés dans plusieurs formalismes) permet de capturer la structure des différents domaines d'une application, à différents niveaux d'abstraction, qu'ils soient des domaines métiers ou techniques. Elle permet aussi de formaliser des artefacts qui n'étaient jusqu'à maintenant exprimés que de manière informelle, pour pouvoir les réutiliser et les rendre opérationnels.

Cette formalisation permet aussi d'expliciter les règles qui définissent la cohérence entre ces différents modèles, et la manière de les transformer ou de les composer au cours du développement. La définition explicite de ces règles et transformations permet de capitaliser l'expertise et le savoir-faire des concepteurs et développeurs sous forme de procédés semi-automatisés qui rendent le processus de développement plus agile et plus réactif aux évolutions de l'environnement.

Cette vision unificatrice est séduisante, elle promet la convergence des résultats d'un grand nombre de disciplines, couvrant un vaste spectre de recherche en génie logiciel. Néanmoins, il nous semble clair que l'IDM ne pourra progresser que si elle se concrétise en des démarches capables d'intégrer les approches conceptuelles avec les pratiques et les contraintes industrielles.

Les travaux de cette thèse s'orientent dans ce sens. Nous nous intéressons particulièrement à la problématique de la gestion de la complexité et de la diversité des domaines impliqués dans les applications de grande taille, et nous proposons de l'approcher sous une optique IDM, en nous appuyant sur des résultats éprouvés dans la pratique, notamment ceux issus des approches basées sur les *Lignes des Produits Logiciels* et les *Langages Dédiés de Modélisation* (DSL)[1].

1.2 LES APPLICATIONS AUX MULTIPLES FACETTES.

L'abstraction est l'un des mécanismes clé pour faire face à la complexité croissante des systèmes. L'abstraction nous permet « d'extraire les caractéristiques

[1] Domain Specific Languages

essentielles d'un système, selon la perspective particulière d'un certain observateur. Les deux idées clés sont d'ignorer certains détails jugés non-essentiels, et de définir un contexte dans lequel l'abstraction prend du sens »[1].

Le concept d'abstraction est fortement lié à la notion de modèle. En effet, un modèle « …représente la réalité pour un objectif donné ; le modèle est une abstraction de la réalité dans le sens où il ne peut pas représenter tous les aspects de cette réalité. Ceci nous permet de traiter le monde d'une façon simplifiée, en évitant toute sa complexité…»[2]. Trouver des mécanismes d'abstraction efficaces est donc critique à toute démarche d'ingénierie dirigée par les modèles.

Cependant, il faut tenir compte du fait que toute abstraction est relative à un point de vue et à une finalité donnée. L'abstraction nous permet donc, aussi, de séparer les préoccupations des divers acteurs impliqués dans la modélisation. Les « bons » mécanismes d'abstraction seraient ainsi dépendants des différentes visions d'un système.

Par exemple, la croissante complexité des plates-formes technologiques à conduit à séparer les préoccupations dites fonctionnelles d'une application de ses aspects non fonctionnels. L'IDM s'est, dès ses origines, intéressé à cette séparation, et a proposé de spécifier une application par des modèles séparés pour sa partie métier et sa partie implémentation dépendante de la plate-forme, PIM et PSM[3] respectivement. Il est clair que les mécanismes d'abstractions adéquats pour modéliser la partie métier d'une application ne sont pas nécessairement les plus appropriés pour spécifier son infrastructure.

L'abstraction ne fait pas disparaître la complexité du système, elle nous permet simplement de l'appréhender selon les préoccupations particulières d'un acteur privilégié. Or, cette complexité est « proportionnelle au nombre de vues distinctes, indépendantes et simultanément significatives du système »[4], qui correspondent aux différentes préoccupations adressées.

Nous pensons donc qu'un formalisme de modélisation unique, même avec des mécanismes d'abstraction très puissants, ne peut répondre à tous les besoins, souvent contradictoires, inhérents aux applications complexes de grande taille. A notre avis, une approche IDM voulant faire face à cette complexité doit embrasser cette diversité, et permettre de définir une grande variété de formalismes de modélisation, avec ses mécanismes d'abstraction spécifiques, pour répondre aux préoccupations des différents acteurs de la modélisation.

Cette hétérogénéité de modèles et formalismes de modélisation n'est pas sans risque, et entraîne sa propre complexité. Des nombreuses questions restent ouvertes, en particulier sur la façon de composer, dans une vision intégrée, les modèles représentant

---

[1] [CES+04] p. 6.

[2] J. Rothenberg. – *The Nature of Modelling*. Cité par [Bez05] p. 174.

[3] Platform Independent Model, Platform Specific Model.

[4] [Cop00] p. 41.

tous les points de vue d'une application, ainsi que sur l'intégration des formalismes de modélisation eux-mêmes.

L'action spécifique du CNRS sur l'IDM [BBB+05] a en effet identifié la maîtrise de cette hétérogénéité comme un des verrous scientifiques à lever pour l'éventuel succès de l'IDM. Les travaux de cette thèse visent spécifiquement à contribuer à cet axe de recherche.

1.3 LES DOMAINES SPECIALISES.

Toute abstraction est définie dans un contexte particulier, ce contexte détermine la portée de la réutilisation des mécanismes d'abstraction : « Des abstractions très spécifiques peuvent être utilisées dans moins de produits, mais contribuent beaucoup plus à leur développement. Des abstractions plus générales peuvent être utilisées dans plus de produits, mais contribuent moins à leur développement »[1].

Nous sommes confrontés donc à un paradoxe. D'une part nous sommes convaincus que la valeur d'une abstraction augmente avec sa spécificité vis à vis d'une préoccupation particulière, et que les formalismes de modélisation doivent fournir les mécanismes d'abstraction le plus adaptés à chaque facette du système. D'autre part, il peut ne pas être viable de construire des environnements de modélisation trop spécialisés, car il n'est pas possible de réaliser les économies d'échelle suffisantes.

L'issue de ce paradoxe est de se rendre compte que les applications ne sont pas isolées, mais font souvent partie d'un ensemble plus vaste : une *Famille de Systèmes* [Par01]. Soit parce qu'il s'agit des variantes d'un même système, soit parce qu'il s'agit d'évolutions ou simplement parce qu'elles partagent un nombre important de préoccupations.

Les familles de systèmes constituent donc un contexte à la fois spécifique et ample, couvrant un nombre significatif d'applications apparentées, idéal pour définir des abstractions à la fois riches et hautement réutilisables. Elles sont aussi le point de départ pour le développement de frameworks à composants spécifiques qui adressent les besoins propres aux membres de la famille, et qui peuvent devenir leur plate-forme cible de génération de code par excellence.

Les langages dédiés de modélisation (DSL) sont un moyen effectif de garantir la réutilisation à l'intérieur d'une famille : ils définissent un formalisme permettant de spécifier une application à partir des concepts propres à la famille ; ils permettent aussi la réutilisation des connaissances et du savoir-faire métier des experts de la famille, ainsi que la réutilisation de frameworks spécialisés ; et, finalement, ils rendent possible le développement d'outils spécialisés de modélisation, d'analyse et de génération de code.

Les bénéfices des familles de systèmes et des langages de modélisation dédiés ont déjà été prouvés dans la pratique par les différentes démarches des *Lignes de Produits Logiciels*. Malgré cela, les lignes de produits logiciels sont restées, jusqu'à

[1] [GS03] p. 25

présent, limitées à des domaines d'applications dans lesquels la notion de gamme de produit est bien enracinée, et pour lesquels le logiciel s'apparente souvent au produit matériel final. La transposition de ces résultats à la pratique courante du génie logiciel reste encore un défi.

Toutefois, la forte affinité mise en évidence entre l'IDM, les approches basées sur les familles de systèmes, et les langages dédiés est encourageante ; Nous nous intéressons particulièrement à cette convergence, et l'explorons dans cette thèse.

1.4 NOTRE PROPOSITION.

L'Ingénierie Dirigée par les Modèles correspond à un changement de paradigme dans lequel le code source n'est plus considéré comme un élément central, mais au contraire comme un élément dérivé à partir du « tissage » et de la fusion de multiples autres modèles plus abstraits. Chaque modèle correspond à une vision partielle du logiciel, et correspond généralement à des points de vue ou à des métiers différents.

Chacun des ces points de vue est potentiellement partageable par de multiples applications, et peut être considéré en soi comme un domaine d'expertise. La connaissance et le savoir-faire développé au sein de ce domaine deviennent des atouts majeurs. Cette expertise peut être formalisée, et surtout réutilisé, sous la forme d'un langage de modélisation dédié, un DSL.

Nous proposons donc une démarche d'Ingénierie Dirigée par les Modèles basée sur la description d'un système par divers modèles exprimés dans des langages de modélisation dédiés différents. La composition de modèles permet de modéliser des applications complexes couvrant plusieurs domaines simultanément.

L'originalité de notre démarche est que, d'une part, ces DSL sont suffisamment précis pour permettre l'exécution des modèles, et d'autre part, qu'il est possible de les composer pour définir de nouveaux domaines plus complexes. La composition de DSL permettant d'améliorer la modularité, d'obtenir de taux de réutilisation plus importants, et surtout d'étendre leur portée.

1.5 PLAN DE LA THESE.

Outre l'introduction, ce rapport est organisé en trois grands chapitres :

- Le chapitre 2 présente les concepts à la base de notre démarche : l'ingénierie dirigée par les modèles, l'ingénierie des domaines, les langages dédiés de modélisation et les Lignes de Produits Logiciels. L'objectif de ce chapitre est de présenter chacune de ces approches, pour essayer de mettre en évidence les relations existantes entre elles.
- Le chapitre 3 présente le concept fondamental de notre proposition : le domaine exécutable. Notre approche se base sur l'existence de divers points de vue sur un système. Chaque point de vue correspond à un domaine d'expertise particulier. Notre proposition consiste à articuler ce savoir-faire autour d'un pivot central : le méta-modèle du domaine. Ce chapitre détaille notre proposition, et décrit comment tous les autres éléments du domaine viennent se rattacher au méta-modèle du domaine.

- Le chapitre 4 se concentre sur le problème de la composition de domaines. L'objectif est de définir de nouveaux domaines, plus complexes, à partir de domaines préalablement validés et testés.

Enfin, le chapitre 5 propose une synthèse des principales idées de notre proposition, et reprend certaines de nos réflexions ; dans le but de mettre en avant nos contributions principales, d'identifier les questions ouvertes et les perspectives du travail.

2. ETAT DE L'ART.

Notre démarche se situe à la convergence de plusieurs approches du génie logiciel : l'ingénierie dirigée par les modèles, l'ingénierie des domaines, les langages dédiés de modélisation et les Lignes de Produits Logiciels. L'objectif de ce chapitre est de présenter chacune de ces approches, pour essayer de mettre en évidence les relations existantes entre elles.

Avant de commencer nous voulons faire deux précisions. En premier lieu, nous ne cherchons pas à couvrir l'intégralité de ces sujets vastes et variés ; nous nous concentrons donc sur les concepts que nous jugeons nécessaires pour comprendre notre proposition. En second lieu, nous allons utiliser les concepts de l'IDM pour aborder tous les thèmes ; nous pensons que cette vision unifiée permet de mieux déceler les synergies entre les diverses approches, au risque de donner un regard biaisé des sujets.

2.1 L'INGENIERIE DIRIGEE PAR LES MODELES.

L'Ingénierie Dirigée par les Modèles (IDM) vise à fournir un cadre qui permet d'aborder toute la problématique du génie logiciel dans un paradigme unificateur [JGM+05] : elle conçoit l'intégralité du cycle de vie du logiciel comme un processus de production, de raffinement itératif et d'intégration de modèles.

L'utilisation de modèles (exprimés dans plusieurs formalismes) permet de capturer la structure des différents domaines d'une application, à différents niveaux d'abstraction, qu'ils soient des domaines métiers ou techniques. Elle permet aussi d'expliciter les règles qui définissent la cohérence entre ces différents modèles, et la manière de les transformer ou les composer au cours du développement.

Nous pouvons entrevoir dans cette description générale que trois notions sont au cœur de la démarche : la notion de modèle, la notion de manipulation de modèles et la notion de formalisme. Nous allons essayer de les préciser par la suite, et de montrer comment elles peuvent fournir un cadre unificateur pour s'attaquer aux différentes problématiques du génie logiciel.

2.1.1 Le concept de modèle.

Le terme modèle est très ancien et il a plusieurs acceptions, même si on se limite au domaine de l'informatique. Il n'existe donc pas de définition universelle de ce qu'est un modèle. Il existe cependant un relatif consensus dans toutes les définitions sur une caractéristique fondamental [Lee00] : « Un modèle implique une représentation … un modèle imite, ressemble ou est utilisé à la place de quelque chose d'autre »[1].

Dans ce même ordre d'idées, Ludewig [Lud03] propose trois critères pour caractériser un modèle, dans le contexte plus particulier du génie logiciel :

- Critère de représentation : il existe un objet ou phénomène « original » qui est représenté par le modèle ;
- Critère de réduction : toutes les propriétés de l'original ne sont pas représentées dans le modèle ; mais d'un autre côté, le modèle représente au moins certaines propriétés de l'original ;
- Critère de pragmatisme : le modèle peut remplacer l'original pour un propos donné.

Ces définitions font intervenir trois concepts : le modèle, le système original et la relation entre les deux. Un point très important à signaler est que les notions de système original et modèle sont relatives : ils sont deux rôles complémentaires basés sur la relation liant un modèle au système qu'il modélise [BBB+05].

Ces notions sont synthétisées par le diagramme de classes[2] de la figure 1, extrait de [Fav04]. Notons que rien n'est dit sur la nature des modèles et des systèmes modélisés, ni sur la nature exacte de la relation de représentation. C'est justement la généralité de cette définition qui lui confère son caractère intégrateur.

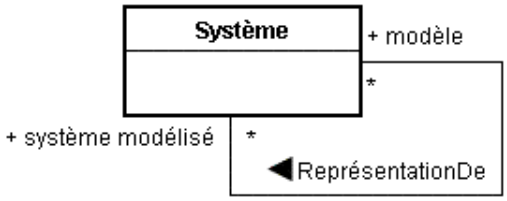

figure 1. Modèle et Représentation [Fav04]

Néanmoins, si nous voulons comprendre le rôle que peut jouer l'ingénierie dirigée par les modèles dans le cadre concret du génie logiciel, il faut préciser plus en détail la nature des modèles utilisés tout au long du cycle de vie logiciel, les divers usages de ces modèles, ainsi que les possibles intentions du concepteur au moment de construire un modèle.

---

[1] [Lee00] p. 182

[2] L'usage d'UML pour représenter les concepts même de l'IDM a donné lieu à l'idée du Mega Modèle IDM [BJV04][Fav04]. Comme tout modèle, nous utilisons ces diagrammes dans un but très pragmatique : faire appel à la familiarité du lecteur avec UML pour essayer de préciser les concepts présentés dans le texte.

La nature des modèles.

La définition que nous avons présentée ne dit rien sur la nature des modèles et systèmes modélisés, elle ne se restreint pas donc aux modèles informatiques. Il peut s'agir d'un modèle mental dans la tête d'un concepteur, d'instructions informelles pour mettre en place une activité ou encore d'une documentation utilisée principalement pour décrire un système.

Favre [Fav04] fait la distinction entre système physique, système abstrait et système numérique. Cette distinction est significative dans la mesure où elle identifie les modèles auxquels on s'intéresse dans le cadre de l'IDM : des modèles numériques, informatiques, susceptibles d'être traités et exploités automatiquement.

Si nous voulons automatiser le traitement des modèles, nous avons alors besoin de les formaliser. Certains auteurs font cette précision dans la définition même de la notion de modèle : « Un modèle est une description d'un système écrite dans un langage bien-défini »[1]. Cette précision nous ramène donc à la question des formalismes de modélisation, qui sera traitée dans la prochaine section.

Modèles statiques et dynamiques.

Un autre critère de classification des systèmes regarde la nature de leur évolution. Bézivin [Bez05] fait la différence entre système statique et dynamique, en fonction de sa capacité à changer dans le temps.

Il utilise cette distinction pour remarquer l'usage rependu en informatique de modèles statiques de systèmes dynamiques : le modèle en soi ne change pas[2], mais il représente l'évolution du système modélisé dans le temps.

Cette distinction permet aussi de raisonner sur la notion d'exécution. Un système dynamique peut être représenté par l'exécution d'un programme : le comportement de l'exécution représente le comportement du système réel. Le code source du programme est, de sa part, une représentation statique de la dynamique de l'exécution.

Modèles descriptifs et spécification.

Nous pouvons aussi préciser la nature de la relation de représentation. Par exemple, Ludewig [Lud03] distingue entre modèle descriptif et modèle normatif : le premier décrit le système modélisé, tandis que le deuxième spécifie les caractéristiques du système modélisé.

Il insiste sur le fait que c'est une propriété de la relation de représentation et non du modèle lui-même. Par exemple, le cahier des charges d'un système décrit les besoins des utilisateurs et, simultanément, spécifie les fonctionnalités à développer.

---

[1] Kleppe A. et al. - *MDA Explained: The Model Driven Architecture. Practice and Promise.* - 2003. Cité par [BBB+05] p.6.

[2] Ou du moins il ne le fait pas dans la même échelle de temps que le système.

En général, un nouveau système est construit à partir d'une spécification, et un modèle descriptif est construit à partir d'un système existant. Néanmoins, il est possible de réaliser des modèles descriptifs d'un système qui n'existe pas : par exemple, une prévision est la description d'une réalité qui n'existe pas encore.

Seidewitz [Sei03] utilise cette distinction pour définir les notions de correction et validité. Un modèle descriptif est correct si toute réponse obtenue par le modèle est effectivement constatée dans le système. Un système modélisé est valide, si toutes les propriétés définies par la spécification sont satisfaites par le système.

Modèles et séparation des préoccupations.

La notion de modèle est liée à la notion d'abstraction. La nature exacte de la relation de représentation est donc relative à un point de vue et à une finalité donnée ; ceci correspond aux critères énoncés auparavant de réduction et pragmatisme.

Il est donc possible de produire, et de travailler, simultanément avec plusieurs modèles d'un même système, en fonction de diverses perspectives des différents acteurs impliqués. La séparation des préoccupations est donc une caractéristique intrinsèque de l'ingénierie dirigée par les modèles, qui mène naturellement à l'idée de modélisation orientée par aspect.

Dans le cas des modèles descriptifs, ces différents aspects sont liés dans le système modélisé, et la combinaison des différentes facettes est toujours envisageable [Bez05]. Dans le cas des modèles de spécification, il faut assurer la cohérence de tous ces modèles, et la construction du système modélisé revient à effectuer la composition et le tissage de tous les aspects [JGM+05].

Précision des modèles.

Lee [Lee00] définit la précision d'un modèle comme une mesure du degré ou granularité de son abstraction. La précision peut être réduite en éliminant des détails non essentiels ou en faisant recours à des descriptions qualitatives et non quantitatives.

Il faut remarquer que le degré de précision d'un modèle est indépendant de sa formalité et sa justesse. Un modèle informel et vague peut très bien être une description juste d'un système, pour une tâche donnée.

Dans le cas des modèles de spécification, la précision du modèle est beaucoup plus importante car elle détermine la validité du système. En effet, la validité du modèle est toujours relative à une spécification donnée.

Par exemple, Mellor[1] classifie les modèles selon leur degré de précision en esquisses, plans et modèles exécutables. L'usage attendu est cependant différent dans chaque cas. Une esquisse n'est pas ni précise ni complète, elle sert à évaluer des nouvelles idées. Un plan est suffisamment précis pour servir de modèle de spécification,

[1] Mellor S. et al. - *MDA Distilled. Principles of Model Driven Architecture*. - 2004. Cite par [Fav04] p. 21

mais ne couvre qu'un aspect du système. Un modèle exécutable est suffisamment précis et complet pour pouvoir servir à dériver un système exécutable.

Cette séparation en trois niveaux est clairement arbitraire, mais elle met simplement en évidence le fait que le degré de précision du modèle de spécification dépend de l'usage attendu.

Modèles et niveaux d'abstraction.

Les systèmes et modèles informatiques peuvent aussi être classifiés en termes du degré d'abstraction des détails, de la solution technologique, et la plate-forme d'exécution. Par exemple, le standard MDA® (Model Driven Architecture) fait la différence entre modèles indépendants et dépendants de la plate-forme, PIM et PSM[1] respectivement.

Le MDA® envisage le cycle de développement comme une transformation progressive d'un modèle PIM métier, qui spécifie un système indépendamment de la technologie, dans un modèle PSM, qui décrit comment le système peut être implémenté pour une technologie particulière. Bien que cette idée paraisse intuitive, la notion de plate-forme technologique est assez vague et très dépendante du contexte, si bien que cette distinction nette entre PIM et PSM est très controversée.

Néanmoins, l'idée qui nous paraît important à souligner est que nous pouvons réfléchir au développement logiciel comme une série de raffinements entre divers modèles d'un même système à différents niveaux d'abstraction.

2.1.2 Les concepts de méta-modèle et conformité.

On vient de voir que dans l'IDM on s'intéresse à des modèles informatiques susceptibles d'être traités et exploités automatiquement. On s'intéresse, par exemple, à pouvoir réaliser de transformations entre spécifications de différents niveaux d'abstraction ou à composer des modèles représentant divers aspects d'un système.

Cette automatisation passe nécessairement par une formalisation des modèles. Pour être manipulé par une machine, un modèle doit être écrit dans un langage précis, et ce langage doit être clairement défini. Nous introduisons donc dans la suite la notion de langage de modélisation, et nous nous intéressons à la façon de définir un langage.

Langages de modélisation.

Pour introduire la notion de langage de modélisation nous faisons appel à la définition de langage utilisé dans la théorie des langages de programmation. En effet un langage est un ensemble (au sens mathématique du terme) de phrases. Dans le cas des langages informatiques, il s'agit souvent d'un ensemble infini qui n'a pas de matérialisation en soi.

Cette définition est transposée dans [BBB+05] aux concepts de base de l'IDM, et on définit un langage de modélisation comme un ensemble de modèles (ou plus

[1] Platform Independent Model, Platform Specific Model

précisément comme un ensemble de systèmes jouant le rôle de modèles). Dans la figure 2, adaptée de [Fav04b], nous intégrons cette définition aux concepts définis précédemment.

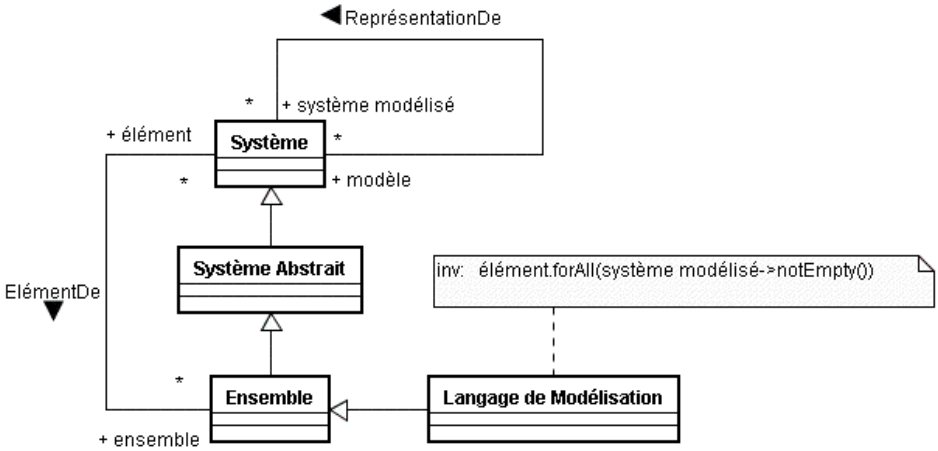

figure 2. Langage de modélisation

Nous faisons remarquer que cette définition garde un caractère très général. En particulier, on ne dit rien sur la nature du langage de modélisation. Par exemple, basé sur cette définition, nous pouvons dire qu'un ensemble de documents décrivant un système est un langage de modélisation.

Comme nous l'avons fait avec la notion de modèle, il faut préciser la nature des langages de modélisation à considérer dans le cadre particulier de l'IDM ; pour cela, nous nous intéressons à la manière de définir un langage de modélisation.

Méta-modèle et relation de conformité.

Afin de définir un langage de modélisation nous devons spécifier tous les aspects du langage que nous voulons formaliser. Le langage de modélisation devient donc un sujet de modélisation. Comme tout système complexe, un langage peut être représenté par plusieurs modèles, descriptifs ou normatifs, de divers niveau d'abstraction.

Cette idée introduit la notion de méta-modèle. Un méta-modèle est défini simplement comme le modèle d'un langage de modélisation [Fav04b] [BBB+05]. A nouveau, cette définition reste très ouverte. Par exemple, une grammaire est un méta-modèle car elle est un modèle d'un aspect du langage, similairement, un document de spécification du langage est un méta-modèle, autant qu'un manuel pour le programmeur.

Les usages et applications des langages de modélisation sont très divers, il est alors très difficile d'énumérer tous les aspects possibles à formaliser. Clark et al. [CES+04] récapitulent les aspects essentiels : une notation pour la construction des modèles dans le langage, qui est défini par une syntaxe concrète ; une description du vocabulaire des concepts et de la façon de les combiner pour créer un modèle bien-formé, qui est défini par une syntaxe abstraite ; une spécification de la sémantique du langage ; des correspondances avec d'autres langages et des mécanismes d'extension du langage.

Malgré la diversité de méta-modèles d'un langage de modélisation, tous partagent une propriété commune : ils caractérisent les modèles du langage. Ceci donne lieu à l'identification d'une nouvelle relation, qui est très important en pratique, la conformité d'un modèle vis-à-vis d'un modèle du langage (son méta-modèle).

Un point clé de l'IDM est de mettre l'accent sur cette relation. En effet, comme le remarque [BBB+05], le fait qu'un modèle soit conforme à un méta-modèle « permet d'assurer d'un point de vue théorique mais surtout opérationnel qu'il est correctement construit et donc qu'il est envisageable de lui appliquer des transformations automatisées »[1].

Notons qu'en fonction de la précision du méta-modèle cette relation peut devenir très opérationnelle. Par exemple, la conformité d'un modèle par rapport à la syntaxe abstraite du langage peut être validée par un éditeur guidé par la syntaxe. Remarquons aussi que la notion de conformité est relative à un méta-modèle donné ; par exemple, un programme peut être conforme à une grammaire et ne pas l'être du point de vue du typage.

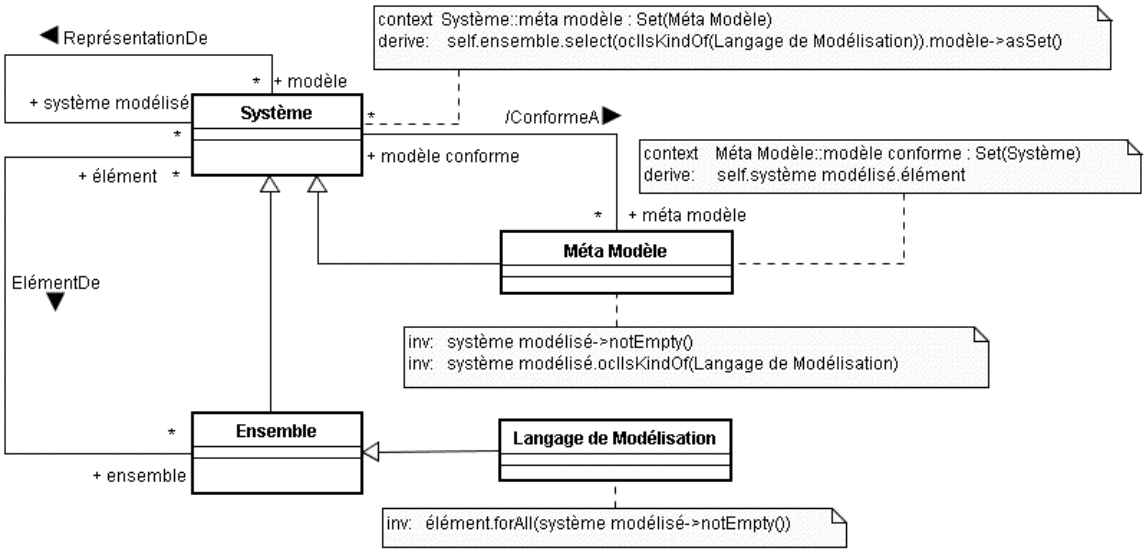

figure 3. Méta-modèle et relation de conformité

La figure 3 synthétise l'ensemble des concepts et relations que nous avons présentées : modèle, méta-modèle, représentation et conformité ; ils constituent le cœur de la démarche de l'IDM. Cependant, si on se limite à ces concepts, on peut se demander quelle est la différence avec les travaux sur la théorie de langages, autre qu'une certaine vision unificatrice.

Le point manquant est la capacité de pouvoir définir de multiples langages de modélisation et de les faire cohabiter dans une architecture unifiée, pour permettre la transformation, composition, comparaison, etc. des différents modèles, issus de ces divers langages.

[1] [BBB+05] p. 7

L'architecture de méta-modélisation.

De la même façon qu'un modèle est conforme à un méta-modèle du langage de modélisation, il est possible de considérer le méta-modèle comme un modèle conforme à un deuxième méta-modèle. Ceci permet d'envisager un méta-modèle unique pour décrire tous les méta-modèles.

Ce méta-modèle unique, appelé le méta méta-modèle, est essentiel pour pouvoir définir tous les langages de modélisation dans un cadre unifié. La relation de conformité donne donc lieu à une architecture qui permet de faire face à la diversité des méta-modèles, et les structurer dans un cadre qui permet d'envisager la définition de mécanismes de comparaison, transformation et intégration de modèles, appartenant à différents langages.

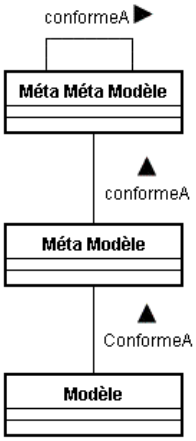

figure 4. Architecture de méta-modélisation MDA®

L'architecture de méta-modélisation la plus connue est celle du MDA®, montrée dans la figure 4 : les modèles sont conformes aux méta-modèles, et tous les méta-modèles sont conformes à un méta méta-modèle unique, qui est conforme à lui même. Le méta-modèle le plus connu dans le MDA® est celui d'UML, mais il existe d'autre méta-modèles standardisés, tels que CWM, SPEM ou EDOC. Le méta méta-modèle unique est le MOF (Meta-Object Facility).

L'existence d'un méta méta-modèle unique est particulièrement importante du point de vue de l'outillage et de la démarche. La possibilité de décrire tous les aspects d'un méta-modèle dans un langage unique est capitale pour assurer l'interopérabilité entre outils. La possibilité de concevoir des outils capables d'être paramétrés par un méta-modèle améliore aussi notablement la flexibilité des environnements de modélisation.

Notons que, dans cette architecture, l'existence d'un méta méta-modèle unique implique l'existence d'un langage de méta-modélisation unifié. De nombreuses questions se posent autour des caractéristiques d'un tel langage, et des facilités disponibles pour spécifier tous les aspects d'un langage de modélisation. Au minimum, il doit fournir les facilités pour décrire de façon précise la syntaxe concrète, la syntaxe abstraite, la sémantique et un format d'échange de modèles [CES+04] [GS+04].

Nous voulons remarquer aussi que ce n'est pas la seule architecture de méta-modélisation possible. La relation de conformité est définie d'une façon tellement

générale qu'il est possible de proposer d'autres architectures pour des contextes plus spécialisés ou avec des propriétés différentes, par exemple [AES01] [AK02].

2.1.3 Synthèse.

L'approche de l'Ingénierie Dirigée par les Modèles repose sur deux relations fondamentales : la représentation et la conformité. La relation de représentation est liée au pouvoir d'abstraction et description des modèles. La relation de conformité est liée à la notion de langage de modélisation et à l'automatisation du traitement des modèles.

La définition de ces concepts est très ouverte, car l'IDM a une vocation unificatrice qui promeut la convergence des résultats d'un grand nombre de disciplines de tout le spectre du génie logiciel.

Cependant, cette vision doit être précisée dans des contextes plus concrets. Nous allons le faire dans la suite de ce chapitre pour l'usage de langages dédiés dans les approches de lignes de produits. Nous allons aussi le faire dans les chapitres 3 et 4 pour notre propre proposition.

2.2 INGENIERIE DES DOMAINES.

La problématique consistant à identifier, capturer et structurer la connaissance d'un domaine pour la rendre opérationnelle et réutilisable a été abordée dans le cadre des approches d'ingénierie des domaines, et par la communauté de recherche en génie logiciel qui s'intéresse à la réutilisation systématique.

L'idée d'ingénierie des domaines part d'un constat simple : quand on développe un composant réutilisable, on a une idée, plus ou moins explicite, de l'ampleur de la réutilisation possible ; basé sur des expériences passées ou sur les besoins anticipés des futurs systèmes envisagés.

Tant que cette notion de « domaine de réutilisation » reste implicite, il est impossible de juger systématiquement sur la « réutilisabilité » d'un composant ou de son adéquation aux besoins d'une application donnée. L'ingénierie des domaines s'intéresse alors principalement à la formalisation du concept de domaine de réutilisation.

2.2.1 Le concept de domaine.

La notion de domaine, comme celle de modèle, a de multiples significations en informatique. Le terme domaine peut être associé à plusieurs interprétations [Har02] :

- Un champ d'expertise dans un domaine métier ;
- Un champ de connaissances avec une terminologie commune.
- Une collection de problèmes : le domaine du problème ;
- Une collection d'applications : le domaine de la solution.

Ces diverses interprétations correspondent en fait à deux points de vue différents du concept de domaine [Sim97] :

- Dans une première vision, le domaine signifie le « monde réel » ; cette vision se concentre alors sur la modélisation de ses phénomènes et de ses processus. L'idée d'ingénierie des domaines, dans ce cas, s'apparente à la modélisation conceptuelle. C'est le cas, par exemple, quand on parle du « modèle du domaine » dans les approches d'analyse orientée objets. C'est cette vision qui sous-tend les trois premières interprétations énumérées précédemment ;
- Dans une deuxième vision, issue de travaux sur la réutilisation, le domaine est caractérisé par un ensemble des systèmes qui partagent un nombre suffisant de problèmes communs, et vraisemblablement une solution commune, pour mériter d'être considérés ensemble. C'est l'idée de « domaine de réutilisation » que nous avons évoqué préalablement.

Cette vision de domaine de réutilisation est centrée sur la notion de famille de systèmes, elle correspond à la dernière interprétation énumérée. Dans cette vision, l'ingénierie des domaines consiste à la production d'une infrastructure de réutilisation qui peut comporter, par exemple, une architecture de référence, des modèles génériques ou des bibliothèques de composants.

Nous pouvons penser que la première vision est suffisante pour assurer la réutilisation : le modèle du domaine capture l'essence des concepts du problème, il est très stable par rapport aux évolutions des besoins des applications et il est hautement réutilisable [Eva04].

Néanmoins, ceci sous estime l'importance d'autres facteurs dans le succès d'une initiative de réutilisation [Gri95] : « … beaucoup d'organisations … naïvement associent réutilisation avec objets. Elles adoptent la technologie d'objets, en espérant qu'elle va assurer automatiquement la réutilisation. Elles sont souvent déçues … sans un programme explicite de réutilisation qui inclut une organisation de support, des procédés, guides et un certain état d'esprit, la réutilisation ne sera pas réussie. »[1].

C'est justement ce dernier constat qui a motivé le développement de l'ingénierie des domaines, et postérieurement des approches des lignes de produits logiciels.

2.2.2 Ingénierie des domaines.

Afin de promouvoir la réutilisation effective et systématique, les approches basées sur les familles de systèmes proposent de séparer le développement logiciel en deux processus différents : l'ingénierie des domaines et l'ingénierie des applications.

L'ingénierie des domaines consiste à développer les artefacts réutilisables qui seront utilisés pour la construction des applications ; on parle alors de développement pour la réutilisation. L'ingénierie des applications consiste à utiliser la base d'artefacts partagés pour la construction d'une application particulière ; on parle de développement par la réutilisation.

[1] [Gri95] p. 1

La structuration concrète de ces deux procédés de développement, les phases à effectuer et les artefacts à produire, change en fonction des différentes démarches spécifiques proposées dans l'industrie et la recherche. Cependant, toutes les démarches sont suffisamment proches pour pouvoir les illustrer à l'aide d'un exemple.

La figure 5 présente un schéma de la proposition de processus de développement dans le projet ESAPS [LO00]. Nous pouvons identifier les trois phases principales dans l'ingénierie des domaines : l'analyse du domaine, la conception du domaine et l'implémentation.

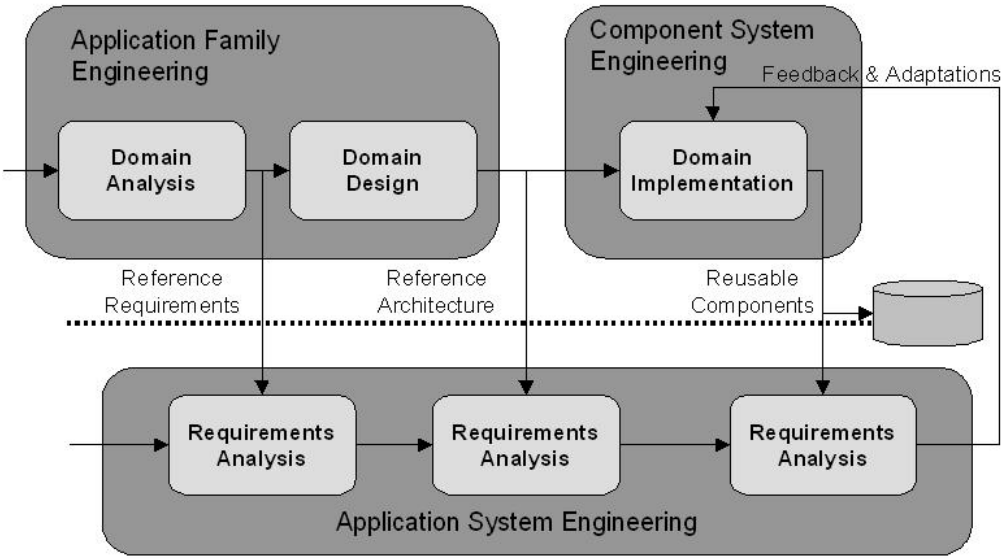

figure 5. Processus de développement dirigé par les domaines [LO00]

Analyse du domaine.

L'analyse du domaine est peut être la phase la plus distinctive des démarches d'ingénierie des domaines. L'analyse du domaine est une activité centrée sur la réutilisation. L'objectif principal de l'analyse du domaine est d'établir le périmètre fonctionnel du domaine, afin de pouvoir déterminer explicitement la portée de la réutilisation des composants.

Dans cette optique, l'activité principale de cette phase est l'analyse de *Portée, Caractéristiques Communes et Variabilité* (SCV)[1] [CHW98]. Elle détermine les fonctionnalités inclues dans le domaine, les fonctionnalités partagées par toutes les applications et les fonctionnalités qui distinguent et caractérisent chaque application.

Le résultat de cette analyse est souvent nommé le modèle du domaine ; pour cette raison on parle aussi de modélisation du domaine. Les formalismes et notations utilisés pour la modélisation du domaine varient en fonction des différentes démarches concrètes proposées. Un formalisme très utilisé dans le cadre des lignes de produits est le diagramme de features, qui sera présenté dans la section §2.3.2.

[1] Scope, Commonality and Variability

L'objectif de ce modèle est de spécifier les limites du domaine ; il n'a aucune relation avec le modèle « conceptuel » du domaine, généralement utilisé dans les méthodes d'analyse orientée objet. Cette dualité dans la terminologie est la source d'innombrables confusions et difficultés à comprendre la démarche.

Ces deux modèles du domaine ne sont pas contradictoires, il s'agit simplement de deux aspects d'un même système. Il est même possible de les intégrer dans un modèle unique, qui capture globalement les concepts du domaine, et simultanément identifie les parties qui peuvent varier entre une application et autre, une telle approche est présentée dans la section § 2.3.3

Etonnement, cette phase s'est révélé être un des défis majeur pour l'ingénierie des domaines. Dans le développement traditionnel, une grande partie du contexte est donné par « le projet », il détermine largement la portée du développement. Dans l'ingénierie des domaines ce contexte est beaucoup plus flou ; l'analyse du domaine est vue donc comme un processus progressif et relativement non structuré d'apprentissage et acquisition d'expérience, qui culmine quand les concepts sont suffisamment clairs pour pouvoir être synthétisés et traduits en artefacts réutilisables.

Conception et implémentation du domaine.

La conception et l'implémentation du domaine se concentrent sur la production des artefacts réutilisables qui vont être mis à dispositions des développeurs d'applications du domaine.

L'ampleur et la nature des artefacts produits dans ces phases dépendent largement du degré de maturité de la démarche d'ingénierie des domaines dans l'organisation de développement. Bosch [Bos02] identifie cinq niveaux progressifs de maturité :

- Produits indépendants : l'organisation développe les produits indépendamment. Ces produits ne partagent pas d'artefacts d'une façon planifiée, seulement par coïncidence ;
- Infrastructure standardisée : l'organisation décide de standardiser l'infrastructure sur laquelle un ensemble de produits est développé ;
- Plate-forme : l'infrastructure standardisée est étendue par des composants développés en interne, qui fournissent les fonctionnalités communes à tous les produits du domaine ;
- Famille de produits : l'organisation adopte une démarche d'ingénierie des domaines. Les développeurs d'applications sont prêts à sacrifier la performance ou autres besoins, en contrepartie des bénéfices de la réutilisation ;
- Base de produit configurable : les différences entre les divers produits sont très bien comprises, et le domaine suffisamment mûr, pour pouvoir envisager d'inclure dans la base de composants toutes les fonctionnalités attendues des produits envisagés dans le domaine. Les produits individuels sont dérivés à partir de la base d'artefacts partagés.

Ingénierie des applications.

Dans le cadre des démarches basées sur l'ingénierie des domaines le développement d'applications reste relativement standard. Cependant, on peut imposer certaines contraintes aux développeurs d'applications, afin rendre effective la réutilisation ; par exemple, on peut imposer l'utilisation d'une architecture ou d'un framework spécifiquement conçus pour le domaine.

Pourtant, un point important à souligner, est que la séparation en deux procédés séparés non seulement garantit la réutilisation mais elle permet aussi d'envisager d'importantes opportunités d'automatisation.

En particulier, quand on atteint le niveau le plus élevé de maturité, la base de composants du domaine couvre les fonctionnalités de toutes les variantes prévues. Il est donc possible d'imaginer de décrire les besoins d'une application d'une façon plus déclarative, dans un modèle plus abstrait, et de générer automatiquement l'assemblage des composants respectifs.

2.2.3 Domaines et applications.

Dans la vision du domaine comme un contexte de réutilisation, la relation entre les applications et les domaines dépend largement de la stratégie suivie pour délimiter la frontière des domaines. Czarnecki [CE00] identifie deux grandes catégories de domaines : horizontaux et verticaux.

Dans le cas des domaines horizontaux, les domaines sont délimités pour couvrir une fonctionnalité précise. Ceci correspond à l'idée d'un domaine de connaissance ou d'expertise. Une application est naturellement concernée par plusieurs domaines et les fonctionnalités des divers domaines vont se mélanger à l'intérieur d'une application. Cette stratégie répond à la logique de séparation de préoccupations.

Pour les domaines horizontaux, la démarche d'ingénierie des domaines peut s'appliquer à chaque domaine. La composition des composants réutilisables issus des différents domaines reste dans ce cas à la charge du programmeur de l'application.

Dans le cas des domaines verticaux, les applications sont classifiées en fonction du métier, par exemple banque ou assurances. Ceci correspond à l'idée d'un domaine comme une collection d'applications. C'est la stratégie utilisée dans les lignes de produits logiciels, que nous détaillons dans la section suivante.

Pour les domaines verticaux, les artefacts partagés développés dans l'implémentation du domaine doivent inclure toutes les fonctionnalités correspondantes aux diverses préoccupations des applications du domaine. Dans ce cas, la façon d'intégrer les divers aspects d'une application reste à la charge du développeur de composants réutilisables.

2.2.4 Un regard basé sur l'IDM.

La présentation que nous avons faite de l'ingénierie des domaines était centrée sur la réutilisation, qui est la principale motivation à son origine. Dans cette section, nous portons un regard centré sur les modèles, afin de déceler les rapports possibles entre l'ingénierie des domaines et l'ingénierie dirigée par les modèles.

Dans un premier temps nous allons le faire d'une façon très générique, en faisant recours aux définitions de base de l'IDM ; nous allons donc établir un cadre général qui va nous permettre par la suite d'explorer ces rapports dans le contexte précis des lignes de produits logiciels.

En premier lieu, nous intégrons le concept de domaine aux concepts définis auparavant. Nous avons vu qu'un domaine, dans la vision centrée sur la réutilisation, est un ensemble des systèmes ; nous pouvons donc l'intégrer facilement aux concepts existants, comme le montre la figure 6.

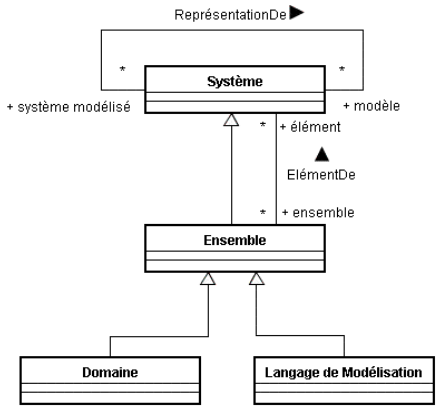

figure 6. Ingénierie des domaines, concepts de base

La question que nous posons ensuite est : quels sont les modèles et les langages de modélisation impliqués ?

Nous avons déjà mentionné le modèle du domaine, qui capture les caractéristiques communes et les variabilités de ses éléments. Nous pouvons aussi supposer que chaque application est modélisée pour spécifier ses caractéristiques particulières. La figure 7 montre un petit scénario qui illustre les divers modèles et méta-modèles impliqués dans un domaine, ainsi que les relations de base existantes.

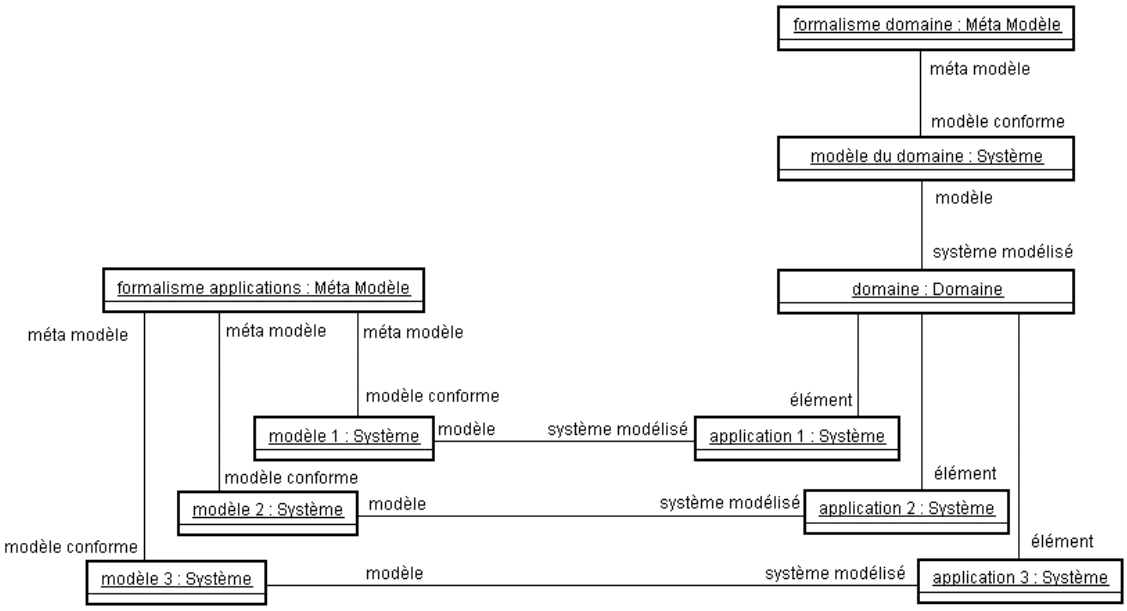

figure 7. Modèles impliqués dans un domaine

Les questions qui restent ouvertes, et auxquelles nous essayerons de répondre dans le contexte précis des approches de lignes de produits sont le suivantes :

a. Existe-t-il une relation entre le modèle du domaine et les modèles des applications ?

b. Existe-t-il une relation entre le formalisme utilisé pour spécifier les applications et celui utilisé pour spécifier le domaine ?

c. Existe-t-il un rapport entre le modèle du domaine et le formalisme utilisé pour décrire les applications ?

2.2.5 Synthèse.

L'ingénierie des domaines est une activité centrée sur la réutilisation. L'objectif principal est de pouvoir déterminer explicitement la portée de réutilisation des composants réutilisables.

Cette idée donne lieu à la notion d'un domaine de réutilisation, qui diffère subtilement de la notion de domaine d'expertise ou de connaissances. La formalisation de ce domaine de réutilisation implique un nouveau procédé de développement séparé, consacré à la production d'une base d'artefacts partagés dédiés au domaine spécifié.

Dans cette démarche le domaine est caractérisé par un modèle, qui capture les caractéristiques communes et les variabilités de ses éléments, et chaque application est caractérisée par un modèle qui spécifie ses particularités.

2.3 LES LIGNES DE PRODUITS LOGICIELS.

Dans la section précédente nous avons présenté les principes généraux des démarches basés sur l'ingénierie des domaines. Les approches de Lignes de Produits Logiciels sont une application réussie de ces principes dans la pratique du développement industriel de logiciel.

L'implantation d'une initiative d'ingénierie des domaines est une tâche coûteuse avec d'énormes répercussions organisationnelles. Les approches de lignes de produits cherchent à augmenter les bénéfices de la réutilisation, et à diminuer les risques, en alignant directement le domaine de réutilisation avec la gamme de produits ciblée par l'entreprise, dans un segment de marché particulier [CN01].

Les lignes de produits mettent en jeu alors des problématiques économiques, organisationnelles et technologiques. Elles changent fondamentalement les pratiques de développement, la structure organisationnelle et la collaboration des équipes de développement [BHJ+03].

Nous n'allons pas traiter ces thèmes dans cette thèse, mais à titre d'exemple voici certaines des considérations pour l'introduction d'une approche de lignes de produits dans une unité de production de logiciel [Bos04] :

- La sélection des fonctionnalités à inclure initialement dans le domaine. Il peut s'agir par exemple des fonctionnalités les plus génériques partagées par tous les projets ou les composants qui nécessitent plus de maintenance ou les nouvelles fonctionnalités communes ;

- L'harmonisation de l'architecture des produits existants. On peut décider de simplement mettre une base de composants à disposition des projets, sans imposer une architecture particulière ; ou une approche progressive, par l'introduction d'un framework pour le domaine par exemple ; ou encore une approche big-bang de migration des applications existantes.
- L'affectation de ressources au développement des composants partagés du domaine. Plusieurs possibilités existent : créer une unité séparée chargée spécifiquement du développement de la plate-forme du domaine, affecter temporairement des ressources provenant des différents projets ou encore répartir la responsabilité entre les équipes existantes ;
- Le financement des coûts de développement de la base de composants partagés par les divers projets impliqués ;
- Les fonctionnalités à inclure dans la base de composants : seulement les fonctionnalités communes à tous les projets ou de composants de base extensibles par un mécanisme de plug-ins ou un composant intégrant toutes les variantes prévues.

L'intérêt principal des lignes de produits logiciels, de notre point de vue, est qu'elles constituent un grand laboratoire de génie logiciel qui a confronté les propositions de l'approche d'ingénierie des domaines aux réalités de la pratique industrielle. Ceci nous rassure sur la viabilité globale de l'approche et nous incite à trouver des rapports avec notre propre proposition.

Les approches de lignes de produits peuvent être appliquées à divers niveaux de maturité, voir section §2.2.2 ; dans le dernier niveau, les différences entre les divers produits sont très bien comprises, et le domaine suffisamment mûr, pour pouvoir envisager d'inclure dans la base de composants toutes les fonctionnalités attendues des produits envisagés dans le domaine.

Dans ce niveau de maturité, il est aussi possible d'envisager l'automatisation de l'assemblage des composants, à partir d'une description plus abstraite des caractéristiques propres au produit. C'est idée a donné lieu aux approches génératives des lignes de produits : le développeur d'applications produit un modèle de l'application, dans un langage approprié au domaine, et le code de l'application est généré automatiquement à partir de ce modèle.

2.3.1 Approches Génératives.

Dans les approches génératives des lignes de produits, la notion clé est la correspondance entre deux espaces [CE00] : l'espace du problème (un ensemble d'abstractions orienté au développeur d'applications) et l'espace de la solution (un ensemble d'abstractions orientées à l'implémentation).

Une façon de raisonner sur cette correspondance est comme une forme de configuration : l'espace du problème représente l'ensemble des configurations possibles du domaine, avec ses caractéristiques distinctives ; et la spécification du produit consiste à faire une sélection cohérente des caractéristiques souhaitées.

Cette vision est une transposition au logiciel de l'idée de gamme de produits dans les lignes de produits manufacturés. Dans cette vision, la correspondance entre espaces consiste à trouver des règles de construction permettant de spécifier un assemblage de composants à partir d'une configuration donnée, on parle alors de dérivation du produit.

Une deuxième façon de penser à cette correspondance est comme une transformation : l'espace du problème est représenté par un langage dédié au domaine, l'espace de la solution par un langage d'implémentation, et la correspondance est une transformation de programmes.

Cette vision permet de rapprocher les idées des lignes de produits aux concepts de l'IDM, en particulière à la transformation entre modèles PIM et PSM dans le MDA®. La principales différence est que dans les approches génératives la transformation concerne toutes les variations du domaine, et non seulement les variations de la plate-forme d'exécution [DSG+03].

Ces deux visions ne sont pas incompatibles. On peut penser que la description de la gamme de produits est un langage dédié à la sélection de produits, et que le langage d'assemblage de composants est un langage d'implémentation [Cza04].

Pour devenir opérationnelles, ces idées générales doivent être concrétisées dans une démarche particulière : il faut choisir le langage dédié approprié au domaine, la technologie de transformation ou configuration, et la technologie ciblée de composants. Nous allons détailler trois grandes catégories d'approches génératives pour illustrer concrètement ces idées.

2.3.2 Approches générative basée sur diagrammes de Features.

Dans l'ingénierie des domaines l'objectif principal de l'analyse du domaine est de déterminer le périmètre du domaine. Dans les approches des lignes de produits un domaine est assimilé à une gamme de produits. Le modèle du domaine dans ce cas peut être vu donc comme une spécification de la gamme de produits.

Une notation très utilisée pour décrire une gamme de produits dans les lignes de produits est le diagramme de features. Un feature est une propriété d'un produit qui capture une caractéristique commune ou permet de distinguer deux produits [CHE04]. Un diagramme de features organise ces propriétés dans un arbre qui identifie les caractéristiques optionnelles, obligatoires et alternatives. Un exemple, extrait de [BTR05], est donné dans la figure 8.

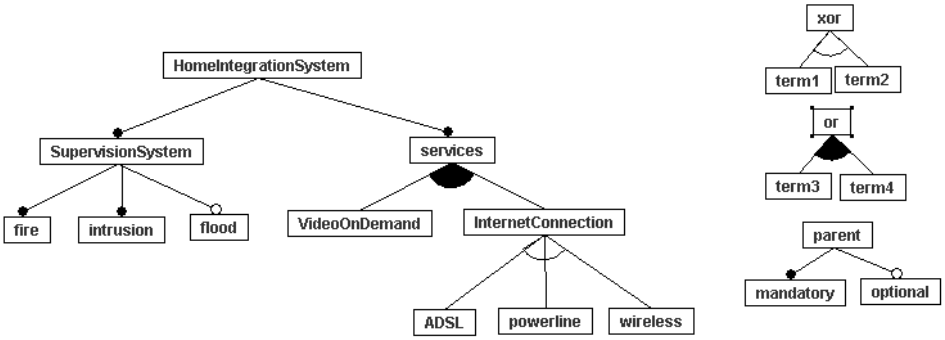

figure 8. Diagramme de Features [BTR05]

A la gauche de la figure, nous pouvons voir un diagramme des features qui décrit toutes les variantes d'un système de domotique (la notation est décrite à la droite de la figure) : le diagramme spécifie qu'un système de gestion de la maison inclut un système de surveillance et un ensemble services ; le système de surveillance inclut la surveillance incendie, vol et optionnellement inondation ; les services offerts sont la vidéo à la demande et/ou l'accès Internet ; l'accès Internet peut se faire par ADSL, sans fil ou réseau électrique.

L'exemple présente les caractéristiques les plus basiques d'un diagramme de features. Ce diagramme est généralement complémenté, entre autres, par des descriptions, des contraintes permettant d'exprimer des incompatibilités entre les options ou des priorités [Rie03]. Le point important à signaler est que le diagramme de features est une spécification abstraite et intuitive, au niveau du domaine du problème, pour à la fois décrire le périmètre de la famille et caractériser ses membres.

Un diagramme de features peut devenir un artefact très opérationnel. Un diagramme de features peut être traduit facilement en une formule logique propositionnelle ; ceci permet, par exemple, de valider les diagrammes, de détecter des inconsistances automatiquement ou de comparer deux diagrammes pour déterminer leur équivalence [Bat05] [Man02]. Il est aussi possible de considérer un diagramme de features comme un système de contraintes, afin de pouvoir exprimer des propriétés plus complexes de la gamme de produits ; par exemple, associer un coût à chaque option pour ensuite déterminer le pourcentage des produits qui dépassent un certain coût [BTR05].

Il est également possible de penser au diagramme de features comme un langage ; en effet, le diagramme décrit l'ensemble de configurations du domaine. Une application peut être très facilement décrite comme une sélection d'options respectant les contraintes spécifiées par le diagramme.

Avec cette vision, les diagrammes de features peuvent devenir le langage dédié qui décrit l'espace du problème dans une approche générative : le concepteur d'applications utilise le diagramme de feaures du domaine pour spécifier une configuration spécifique, et cette configuration est transformée en un assemblage de composants qui implémente les fonctionnalités choisies [CHE04].

Dérivation d'un produit.

La principale difficulté dans cette approche est de pouvoir faire la relation entre un feature et un module logiciel concret ; en général un feature ne correspond pas directement un composant spécifique, mais à une fonctionnalité transversale aux divers composants.

Pour cette raison, plusieurs travaux proposent de matérialiser le feature comme unité de développement logiciel concrète [Gri00] [JSG+04]. Le feature devient ainsi un moyen de séparer les diverses préoccupations du domaine [CRB04]. Le caractère transversal des features a motivé l'utilisation des techniques de Programmation Orientée Aspect pour leur implémentation [LBC05] [AM04].

Le principal intérêt de ces propositions est qu'elles alignent les unités de développement avec les unités de configuration. Il est donc possible de transformer

simplement une configuration décrivant un produit en une application, par le tissage des divers aspects associés à chacun des caractéristiques choisis.

Un regard basé sur l'IDM.

Nous allons maintenant porter un regard centré IDM sur cette démarche, afin de mieux comprendre les relations entre les divers modèles impliqués. La figure 9 synthétise l'ensemble de modèles utilisés.

Les deux principaux modèles sont le diagramme de features et la configuration. Le diagramme de features est une représentation du domaine, et la configuration est une représentation d'une application du domaine.

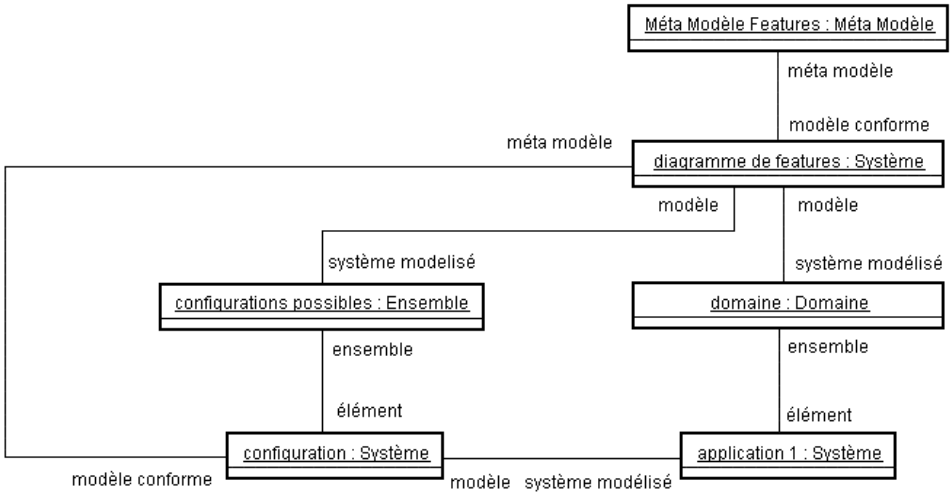

figure 9. Approches par features, regard IDM

Les diagrammes de features sont écrits dans une notation précise. Le langage associé à cette notation à été bien défini et formalisé [CHE04] [FFB02]. Les diagrammes de features sont donc propices à des traitements automatiques, comme nous l'avons signalé

Le diagramme de features se comporte à son tour comme le langage pour décrire l'espace du problème. Pour être plus précis : le diagramme de features est une spécification de l'ensemble de toutes les configurations valides.

Cette interprétation nous permet d'établir une relation directe entre la configuration et le diagramme de features : le diagramme est le méta-modèle de la configuration. Notons que dans ce cas la relation de conformité est très opérationnelle : la configuration peut être confrontée au diagramme pour déterminer sa validité en évaluant les contraintes spécifiées.

Nous faisons aussi remarquer le double rôle joué par le diagramme de features : il est à la fois un modèle du domaine et le méta-modèle du langage de modélisation.

Cette dualité peut s'expliquer par le fait que dans les lignes de produits le domaine est défini comme un ensemble, il existe donc une correspondance naturelle entre la notion de domaine et la notion de langage. Intuitivement, on s'attend à une correspondance entre ces deux ensembles : toute configuration représente une application du domaine, et toute application est décrite par une configuration valide.

2.3.3 Approches générative basée sur UML.

Le diagramme de features fournit un langage simple et puissant pour la dérivation d'applications, basé sur la notion de configuration. Cependant, il est limité dans son rôle de modèle du domaine : la description du domaine se réduit à l'identification des caractéristiques distinctives et ses dépendances.

Une alternative est d'utiliser les méthodes d'analyse et conception orientées objets pour effectuer l'analyse du domaine. Dans ce cas, il est possible de conceptualiser le domaine à un haut niveau d'abstraction par un modèle orienté objet, par exemple un modèle UML.

Les principales difficultés pour utiliser les méthodes orientées objets dans le cadre de lignes de produits sont [CE00] : le manque d'une phase de délimitation du domaine, l'absence d'une distinction entre ingénierie des domaines et ingénierie des applications, et le manque de mécanismes explicites pour distinguer la variabilité entre plusieurs applications.

Les deux premiers points peuvent être abordés en adaptant les méthodologies d'analyse et conception orientées objets aux besoins spécifiques des lignes de produits [Sim97] [Har02]. Le dernier point peut être abordé en proposant des extensions au langage de modélisation pour exprimer les variabilités du domaine.

UML étant un langage extensible, plusieurs travaux ont proposé des extensions du langage adaptées aux lignes de produits [Ma02] [ZHJ03] [CA05]. Ces propositions font appel aux mécanismes d'extension d'UML, tels que les stéréotypes, les étiquettes (tagged values) ou les contraintes.

Ziadi et al. [ZHJ03][ZJ05] proposent par exemple d'utiliser des stéréotypes (tels que <<optional>> ou <<variation>>) et des étiquettes pour identifier les variabilités dans les différents diagrammes ; et d'utiliser des contraintes OCL au niveau du méta-modèle UML pour exprimer les critères de cohérence d'une configuration.

Czarnecki et Antkiewicz [CA05] proposent d'utiliser simultanément un modèle de features et un modèle UML. Les features sont utilisés pour définir des stéréotypes propres à la ligne de produits. Le modèle UML est annoté par ces stéréotypes, pour distinguer les diverses variantes. Par exemple, la partie gauche de la figure 10 présente un diagramme d'activités UML étendu, dans lequel les états et les transitions sont annotés par des stéréotypes associés aux features de la ligne de produits.

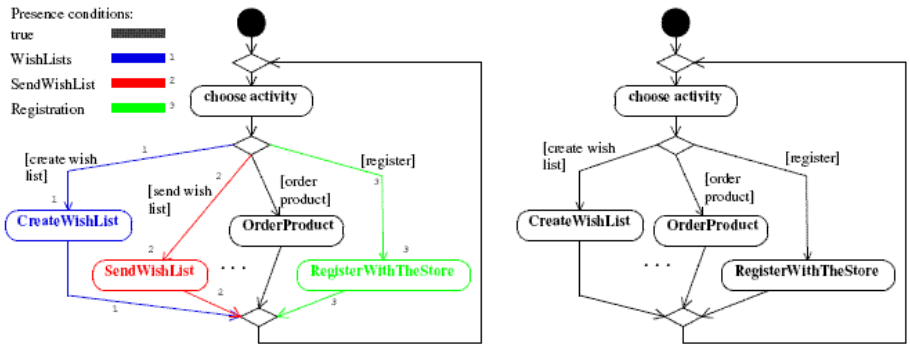

figure 10. Dérivation à partir du diagramme d'activité UML étendu [CA05]

Dérivation d'un produit.

Une fois que nous avons un modèle UML du domaine qui capture toutes les variantes possibles de la ligne de produits, nous pouvons envisager la dérivation automatique d'un produit à partir de ce modèle. Comme dans toutes les approches génératives, il faut spécifier les caractéristiques du produit que nous voulons dériver.

Czarnecki et Antkiewicz [CA05] proposent d'utiliser directement le diagramme de features pour spécifier la configuration du produit. Ziadi et al. [ZHJ03][ZJ05] proposent d'utiliser un *modèle de décision* qui spécifie la valeur des étiquettes et les choix des variantes. Dans les deux cas, à partir de cette spécification, il est possible de raffiner le modèle UML du domaine pour générer automatiquement un modèle UML de l'application, tel que l'illustre la partie droite de la figure 10.

Un regard basé sur l'IDM.

Nous allons essayer de résumer la démarche en nous concentrant sur les divers modèles impliqués ; la figure 11 montre l'ensemble de modèles utilisés.

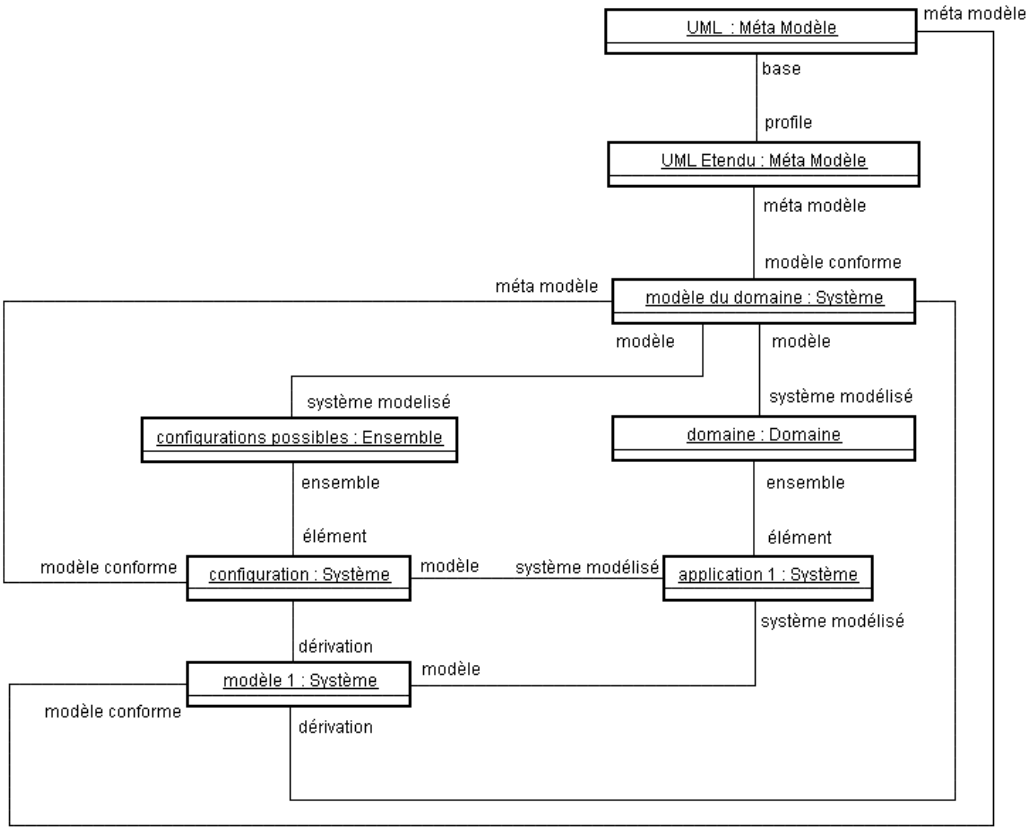

figure 11. Approches UML étendu, regard IDM

Nous pouvons identifier, à nouveau, le double rôle joué par le modèle du domaine : représentation du domaine et méta-modèle du langage de configuration. Cependant, la représentation du domaine est beaucoup plus riche dans cette démarche, puisque le modèle du domaine est écrit dans un langage plus expressif.

Le modèle du domaine est écrit dans un langage bien défini, UML, il est donc possible d'effectuer des traitements automatiques. En particulier, il est possible de

définir la transformation pour générer automatiquement le modèle UML de l'application, montré en bas de la figure. Une des avantages de cette approche est que la transformation est beaucoup plus explicite.

Un point à remarquer est que dans ce cas la génération du produit ne se fait pas par tissage d'aspects, mais par raffinement. Le modèle du domaine est une superposition de tous les aspects de la ligne de produits, et les annotations sont utilisées pour les discriminer.

2.3.4 Approches générative basée sur des langages dédiés.

Dans les deux approches présentées précédemment nous pouvons remarquer que le langage de spécification de l'application est relativement restreint, il s'agit du choix d'une configuration. Le langage proposé aux développeurs d'applications dépend essentiellement du rang de variation du domaine [CE00], ceci peut aller d'un simple « wizard » de configuration jusqu'à un langage de programmation dédié.

Il existe une forte affinité entre les lignes de produits et les langages dédiés. Un langage dédié peut être vu comme la culmination naturelle de la maturité d'une ligne de produits [WL99]. Inversement, le contexte de la ligne de produits permet d'approcher plus facilement la définition d'un langage spécifique [Con04].

Les principaux bénéfices de l'utilisation des langages dédiés dans les approches centrés sur le domaine sont [Thi98] [Wil01] [Wil03] [MHS03] :

- Une notation familière : les langages dédiés proposent une notation naturelle pour le domaine. Ce facteur ne doit pas être sous estimé, car il est directement associé aux gains de productivité attribués aux langages dédiés ;
- Réutilisation de la conception : un langage dédié capture l'ensemble des décisions de conception qui doivent être prises pour construire un membre d'une ligne de produits ;
- Abstraction de haut niveau : un langage dédié propose au concepteur directement des abstractions représentant les concepts du domaine. Ceci se traduit généralement par la concision des spécifications, qui augmente leur compréhensibilité et la confiance dans leur correction. Les langages dédiés sacrifient donc généralité par expressivité dans le domaine ;
- Analyse de la spécification : En restreignant le langage, et en utilisant les connaissances du domaine, on peut effectuer des analyses sur la spécification qui sont indécidables ou impraticables pour une spécification générique ;
- Le bénéfice sans doute le plus important est que l'expert du problème peut écrire directement la spécification du système, sans requérir du tandem expert/programmeur.

L'intégration d'un langage dédié dans une approche générative pour automatiser la génération des applications d'une ligne de produits implique [TS05] : un environnement de modélisation spécialisé pour le langage, un générateur de code et un framework spécifique au domaine.

Remarquons que dans cette approche nous sommes directement dans une vision de transformation entre l'espace du problème et l'espace de la solution : l'espace du problème est représenté par le langage dédié, l'espace de la solution par le framework du domaine, et la correspondance par le générateur de code.

Il existe un grand nombre de variantes de cette démarche, en fonction des technologies utilisées pour spécifier chacun des éléments. La figure 12, extraite de [CE00], place les différentes technologies par rapport à l'espace du problème (à gauche), l'espace de la solution (à droite) ou la transformation (au centre).

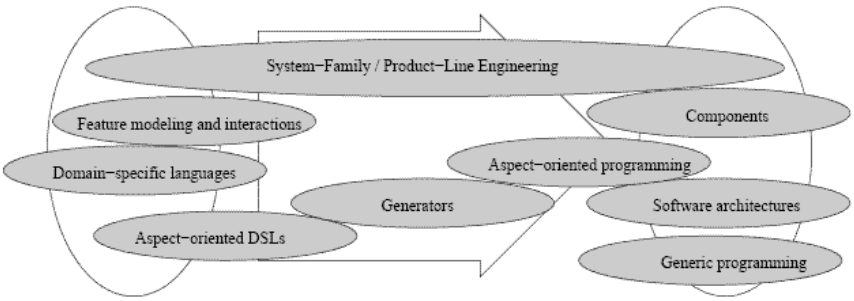

figure 12. Approches génératives basés sur des langages dédiés [CE00]

En général nous pensons que toutes ces variantes ont une structure très similaire aux démarches que nous avons déjà présentées dans les sections précédentes. Nous pensons donc pouvoir généraliser certaines des observations que nous avons faites du point de vue de l'IDM. La figure 13 schématise les patrons que nous avons discernés.

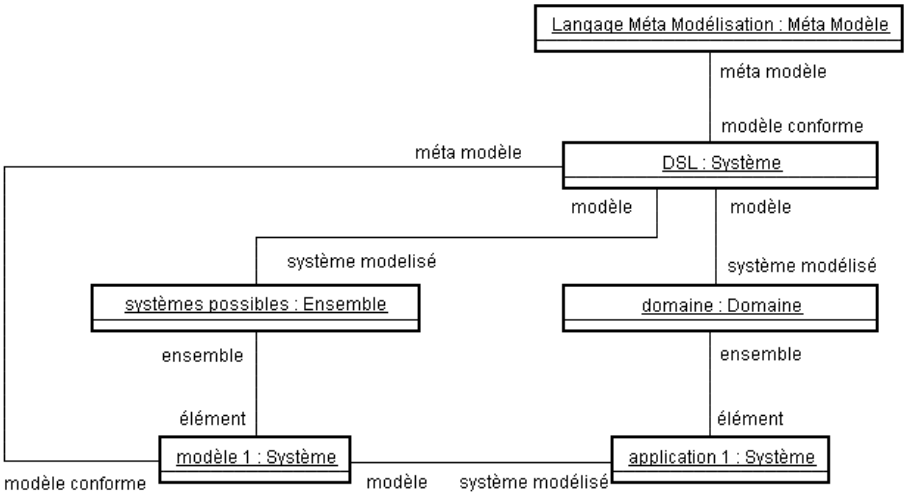

figure 13. Approches basées sur langages dédiés, regard IDM

Notre première observation est que la formalisation du méta-modèle du langage est essentielle dans cette approche. Le méta-modèle spécifie le rang de variabilité à disposition des concepteurs d'applications, et il est indispensable pour la mise en œuvre de l'outillage.

Nous pouvons aussi remarquer que le méta-modèle du langage joue aussi le rôle de modèle du domaine. Comme nous l'avons signalé, intuitivement, le méta-modèle doit être conçu de façon à faire correspondre le langage et le domaine. Une grande partie des bénéfices que nous avons énumérés découlent de cette correspondance.

Un dernier point à souligner est l'importance du langage de méta-modélisation utilisé pour spécifier le méta-modèle. Il doit fournir les mécanismes nécessaires pour pouvoir spécifier à la fois la conceptualisation du modèle et les aspects langagiers du modèle du domaine.

2.3.5 Synthèse.

Les approches de lignes de produits logiciels ont démontré, dans la pratique, les bénéfices en termes de réutilisation de la démarche de l'ingénierie des domaines. Elles ont aussi démontré que, quand le domaine est suffisamment mûr, il est aussi possible d'obtenir d'énormes bénéfices en termes d'automatisation.

Un domaine devient donc à la fois une unité de réutilisation et un mécanisme d'abstraction. La clé de ce double rôle est le modèle du domaine. Le modèle du domaine d'une part représente les concepts du domaine et d'autre part se comporte comme un méta-modèle pour spécifier les applications du domaine.

Il existe plusieurs possibilités pour pouvoir spécifier le modèle du domaine : les diagrammes de features, les extensions à UML et les langages dédiés. Dans tous les cas on dispose d'une représentation abstraite de haut niveau du domaine. Les approches se distinguent principalement dans la flexibilité du langage de modélisation proposé au concepteur d'applications.

2.4 CONCLUSION.

Les travaux de cette thèse se placent dans la mouvance de l'Ingénierie Dirigée par les Modèles. Nous avons donc présenté dans ce chapitre les concepts essentiels de cette approche. Deux relations apparaissent comme fondamentales : la relation de représentation, qui associe le modèle au système modélisé ; et la relation de conformité, qui associe le modèle au méta-modèle du langage de modélisation. Ces deux relations sont liées, respectivement, au pouvoir d'abstraction des modèles et à l'automatisation de leur traitement.

La deuxième fondation de nos travaux est la spécialisation en domaines. Nous nous sommes alors intéressés aux travaux de l'ingénierie des domaines. L'idée qui nous semble essentielle est la séparation du développement en deux procédés distincts : l'ingénierie des domaines, le développement pour la réutilisation ; et l'ingénierie des applications, le développement par la réutilisation.

La convergence entre l'ingénierie dirigée par les modèles et l'ingénierie des domaines a été explorée par les approches génératives des Lignes de Produits Logiciels. Nous avons donc étudié trois propositions concrètes : celle basée sur les diagrammes de features, celle basée sur les extensions d'UML et celle basée sur les langages dédiés d'application.

Nous avons porté un regard « centré modèle » sur ces trois propositions. Ce nouveau regard nous a permis de mieux comprendre le lien entre le modèle du domaine et le modèle de l'application. De façon intéressante, nous avons constaté que dans les approches génératives le modèle du domaine se comporte comme un méta-modèle du modèle de l'application.

Cette observation nous semble capitale, puisqu'elle implique que le modèle du domaine rallie toutes les propriétés des modèles qui nous intéressent : abstraction, spécialisation et automatisation.

C'est sur cette réflexion que nous entamons la présentation de notre proposition. Dans un premier chapitre, nous allons préciser notre notion de domaine, et plus particulièrement l'idée de modèle exécutable ; un deuxième chapitre sera consacré à la composition des domaines.

3. DOMAINES EXECUTABLES.

Notre approche se base sur l'existence de divers points de vue sur un système. Chaque point de vue correspond à un domaine d'expertise particulier. Cette expertise se traduit par de multiples artefacts : la conceptualisation du domaine, des librairies de composants spécialisés, des langages dédiés, l'interfaçage de logiciels patrimoniaux, l'adoption de composants commerciaux sur étagère, etc. Nous voulons formaliser toute cette connaissance pour la rendre explicite, opérationnelle et surtout réutilisable.

Notre proposition consiste à articuler ce savoir-faire autour d'un pivot central : le méta-modèle du domaine. Ce chapitre détaille notre proposition, et décrit comment tous les autres éléments du domaine viennent se rattacher au méta-modèle du domaine.

3.1 SPECIFICATION DU DOMAINE.

Comme nous l'avons déjà évoqué, notre approche s'inscrit dans l'esprit des Familles de Systèmes. Ainsi, une des activités capitales est la spécification du domaine, entendue comme le « …processus par lequel l'information utilisée pour le développement de systèmes logiciels est identifiée, capturée et organisée dans le but de la rendre réutilisable pour la création des nouveaux systèmes » [P-D90][1].

Dans notre démarche, le résultat le plus important de cette activité est le méta-modèle, i.e. la définition d'un langage de modélisation spécialisé au domaine. Toutefois, pour être efficace, ce langage doit être : intuitif, suffisamment expressif pour représenter tous les concepts du domaine, et avoir une sémantique claire et précise [GPvS02].

Beaucoup d'avantages présumés du langage découlent effectivement du rapport étroit entre les concepts du langage et ceux du domaine. Nous proposons donc d'amorcer la construction du domaine par sa conceptualisation, pour pouvoir, à partir de là, extraire et raffiner le méta-modèle du langage respectif. Nous décrivons par la suite les diverses étapes qui conduisent à la spécification du méta-modèle.

---

[1] [P-D90] p. 47

3.1.1 L'analyse du domaine.

Dans le cadre des approches basées sur les familles de systèmes, l'ingénierie des domaines adresse le développement pour la réutilisation. L'établissement d'une initiative d'ingénierie de domaines dans une unité de production logiciel est une tâche coûteuse avec d'énormes répercussions organisationnelles.

En conséquence, une partie importante de l'effort d'analyse du domaine est traditionnellement consacrée à l'évaluation du potentiel de réutilisation, les bénéfices escomptés, les compétences requises ou bien l'évaluation des risques. Ces questions ne seront pas traitées dans cette thèse, et ont été d'ailleurs longuement discutées dans la communauté des Lignes des Produits Logiciels. Ainsi, dans cette section, nous centrons notre intérêt sur le résultat des phases initiales d'analyse : la définition du domaine.

Le but de l'analyse du domaine est d'établir un contexte particulier pour déterminer la portée de la réutilisation. Un domaine est, de ce fait, une construction créée dans l'intention de promouvoir la réutilisation, son périmètre doit être donc défini en fonction de cet objectif, tout en gardant une certaine cohérence et cohésion.

La spécification du périmètre du domaine est, dans les approches de Lignes de Produits Logiciels, le résultat d'une analyse de *Portée, Caractéristiques Communes et Variabilité* (SCV)[1] [CHW98] qui vise à établir la limite de la ligne de produits, les caractéristiques partagées par tous ses membres, et leurs caractéristiques distinctives. Le produit de cette analyse est souvent représenté par un diagramme de features (voir §2.3.2) qui précise le domaine.

Dans notre démarche, nous considérons que la délimitation du périmètre du domaine est un pas indispensable pour entreprendre sa conceptualisation, nous utilisons donc l'analyse du domaine pour raisonner sur les frontières du domaine, et les possibles relations avec d'autres domaines. Pour illustrer cette démarche nous allons présenter par la suite un exemple issu de nos expérimentations industrielles.

Le domaine de la Gestion de Configurations.

Le domaine d'applications qui nous concerne est celui des Systèmes de Gestion de Configurations. Un système de gestion de configurations est généralement utilisé pour la gestion de données complexes, qui évoluent fréquemment par l'action d'un nombre important d'utilisateurs concurrents. La fonctionnalité principale de ces systèmes est de représenter des états intermédiaires, sous forme de versions, et de permettre aux utilisateurs, à tout moment, de pouvoir revenir à une configuration particulière dans le temps.

La gestion de configurations a d'importants champs d'application dans le développement de logiciel et de matériel, la gestion documentaire, la gestion de produits (PDM)[2], entre autres. Les gestionnaires de configurations sont devenus des outils

[1] Scope, Commonality and Variability

[2] Product Data Management

puissants et matures, mais aussi des produits gros, complexes, monolithiques, difficiles à adapter et à réutiliser pour de nouveaux usages [ELV03].

Une vision Ligne de Produits.

Une des difficultés majeures des gestionnaires de configurations est le grand nombre d'options et variantes qui doivent être proposées, pour répondre aux besoins d'un vaste éventail d'usages possibles.

Face à cette grande variabilité, un certain nombre de travaux se sont intéressé aux gestionnaires de configurations avec une vision de ligne de produits. Kovse et Gebauer présentent dans [KG04] une analyse du domaine des systèmes de versionnement. Nous allons nous servir de cette analyse pour détailler le domaine, et pour confronter la démarche de lignes de produits à notre proposition.

Un gestionnaire de configurations gère des objets et des relations entre ces objets. Chaque objet est une instance d'un certain type d'objet, qui détermine son comportement de versionnement. Chaque objet a des attributs, et supporte les opérations de base : création, destruction, copie et création de versions. La notion de type d'objet est donc une caractéristique partagée par tous les membres de cette ligne de produits, si on se réfère au diagramme de features présenté dans [KG04], voir figure 14, il s'agit donc d'un caractéristique obligatoire de la famille.

Le stockage des versions est un des aspects les plus importants de tout gestionnaire de versionnement, car il impacte considérablement la performance du système. Plusieurs possibilités de stockage ont été implantées dans différents systèmes : on peut par exemple stocker une copie de tous les attributs chaque fois qu'on crée une nouvelle version ou stocker seulement les différences par rapport à la version précédente. Ces possibilités sont décrites dans le diagramme par des sous features alternatives du feature « storage ».

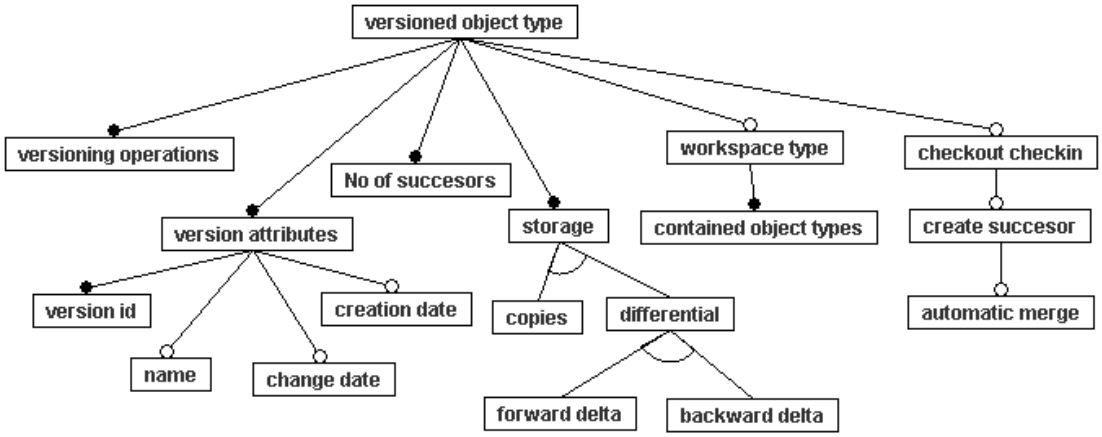

figure 14. Diagramme de features - Objets Versionnés [KG04]

On observe aussi sur le diagramme qu'un gestionnaire de configurations peut, optionnellement, gérer le concept de « workspace ». Un workspace étant un type particulier d'objet qui peut contenir d'autres objets, mais qui ne contient qu'une seule version de chaque objet simultanément.

Un autre aspect très important du domaine de gestion de configurations est la gestion des relations entre objets. En effet, la notion même de configuration est définie en termes d'un ensemble de versions d'objets reliés entre elles. L'analyse du domaine faite par [KG04] concernant la notion de relation est présentée dans la figure 15.

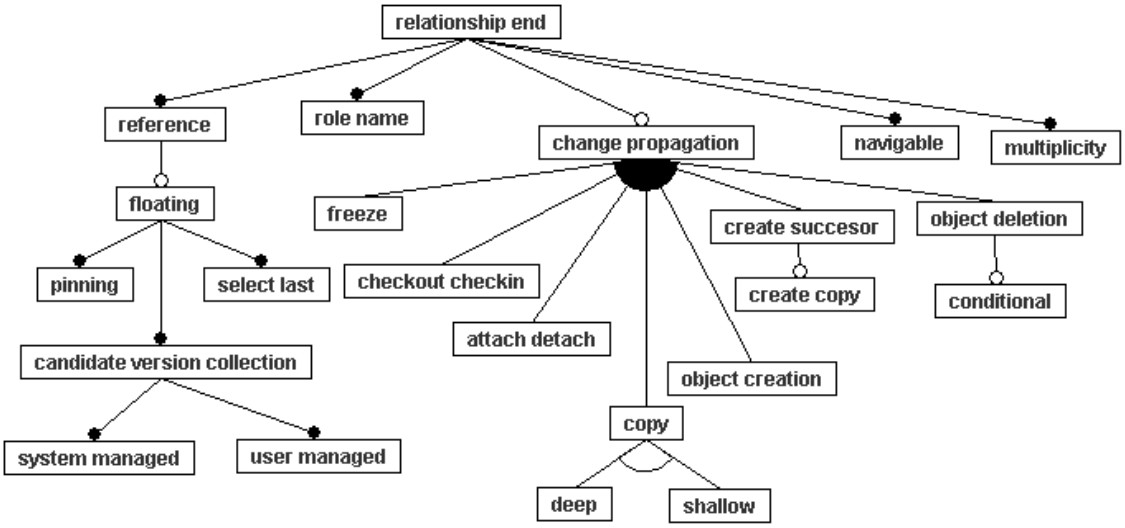

figure 15. Diagramme de features - Relations versionnées [KG04]

Une option liée au concept de relations est la notion de référence flottante : un objet versionné peut référencer un ensemble de versions de l'objet cible de la relation. Quand l'utilisateur veut naviguer sur cette référence, il doit choisir parmi la collection de versions candidates celle qui lui convient ; généralement le système propose par défaut la dernière version.

On voit aussi sur le diagramme que les relations peuvent être utilisées pour propager les opérations. Par exemple, quand on crée une nouvelle version d'un objet, le système peut créer automatiquement de nouvelles versions des objets liés.

Au-delà de la justesse de cette analyse, et bien que certains aspects de la conception puissent être contestés par un expert dans le domaine, ce sur quoi nous voulons insister est le fait que l'analyse du domaine détermine la portée de la ligne de produits.

Les diagrammes de features présentés spécifient d'une façon très succincte l'ensemble de systèmes cible du développement et de la réutilisation. Tout artefact développé au sein de cette ligne de produit doit contribuer à implémenter un des caractéristiques distinctives identifiées. Inversement, un artefact développé pour cette ligne de produits n'est conçu pour être réutilisé qu'à l'intérieur de la même famille, et sera difficilement adaptable hors de ce contexte.

C'est justement cette spécificité qui explique en partie la réussite de l'approche Ligne de Produits dans des domaines bien cernés, mais aussi les difficultés pour l'étendre et la généraliser [OB02] [Omm01].

Cet exemple nous permet aussi de remarquer le fait que les approches de lignes de produits mélangent souvent plusieurs préoccupations. On observe par exemple qu'on

a mélangé les concepts de base avec des préoccupations relatives à leur implémentation qui intéressent souvent des acteurs différents, tels que la façon de stocker les versions.

Cet entremêlement des préoccupations est en partie dû à l'insistance sur la notion de feature, plutôt que sur la notion de point de vue. Ceci s'explique par le fait qu'en général dans les lignes de produits on cherche à définir une gamme de produits, autant qu'à délimiter un domaine de connaissances et d'expertise.

Ce mélange aide aussi à expliquer les difficultés à étendre les approches des lignes de produits. Par exemple, on se rend compte que chaque préoccupation particulière n'est pas traitée dans sa généralité, mais que l'on se limite à une vision partielle qui résout les besoins du groupe d'applications choisi. Ceci explique la difficulté à réutiliser les artefacts développés en dehors de la ligne de produits, même dans des contextes très proches.

Nous pensons qu'une démarche plus centrée sur la séparation des préoccupations peut complémenter l'analyse de domaine, et aider à surmonter les limites de portée propres à la vision ligne de produits.

Une vision par séparation des préoccupations.

L'idée de base est relativement simple : il s'agit, d'une part, de comprendre qu'une ligne de produits peut comporter plusieurs domaines et, d'autre part, que ces domaines peuvent avoir un potentiel de réutilisation plus vaste que la portée de la ligne de produits.

Nous proposons donc d'élargir la perspective de l'analyse du domaine : passer d'une analyse centrée sur la gamme de produits, à une analyse centrée sur un ensemble de domaines de base d'une portée plus étendue. C'est cette idée qui démarque notre approche de celles des lignes de produits, tout en restant dans le cadre général des familles de systèmes.

On retrouve des idées analogues dans les premiers papiers sur l'analyse de domaine ; Prieto-Díaz [P-D90] remarque par exemple que « les domaines peuvent être vus comme appartenant à un réseau, une structure semi-hiérarchique où des domaines simples sont à la base … et des domaines plus complexes se trouvent vers le sommet. La complexité d'un domaine peut être caractérisée par le nombre de domaines interdépendants nécessaires pour le rendre opérationnel »[1]. On peut aussi trouver des propositions similaires dans [TH02] [GW03] et [McG04], mais elles se limitent aux phases amont d'analyse des besoins et cahier des charges. Notre proposition couvre l'ensemble du cycle de vie du domaine.

Dans l'exemple de la Gestion de Configurations, nous avons donc identifié un certain nombre de domaines de base qui font partie de la gestion de configurations [ELV03], mais qui peuvent être réutilisés dans des contextes plus larges ; ils sont présentés dans le schéma de la figure 16. Nous avons définit des domaines de base pour la gestion de workspace et la gestion de ressources. Nous avons également séparé la

[1] [P-D90] p.48

gestion de documents de la gestion d'attributs de produits. Finalement nous avons introduit un domaine de gestion de procédés pour gérer tout ce qui concerne l'automatisation de la gestion de versions.

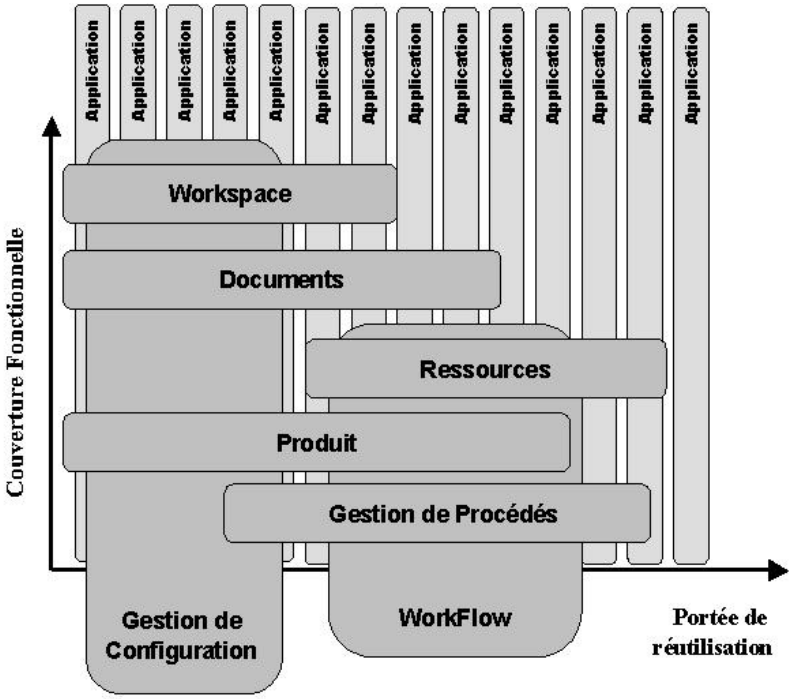

figure 16. Décomposition par domaines

Nous n'allons pas rentrer, en le moment, dans les détails de tous ces domaines, mais nous pouvons déjà remarquer sur la figure plusieurs caractéristiques de notre approche. Tout d'abord, on peut remarquer que les domaines de base (représentés dans la figure 16 par des rectangles horizontaux) ont une portée de réutilisation plus large que la famille visée (représentée dans la figure 16 par la verticale marquée « Gestion de Configuration »), mais aussi qu'ils ont une plus petite couverture fonctionnelle.

Dans notre démarche, donc, il est impératif de pouvoir composer les fonctionnalités des domaines de base, pour pouvoir obtenir les fonctionnalités complètes d'une famille précise d'applications. Nous reportons cette discussion au chapitre suivant, qui sera consacré à la composition de domaines.

Un autre point important à souligner est que ces domaines peuvent être réutilisés pour développer d'autres lignes de produits proches ou avec beaucoup de recouvrements fonctionnels. Par exemple, comme la figure 16 le suggère, nous avons réutilisé dans notre équipe ces mêmes domaines de base pour le développement de systèmes de gestion de procédés [EVL+03] [SE05], et des workflows documentaires en particulier.

En effet, la plupart de systèmes de gestion de procédés utilisent des fonctionnalités de versionnement pour gérer le travail concurrent des participants. Inversement, beaucoup de systèmes de gestion de configuration s'appuient sur des fonctionnalités de procédés pour gérer l'automatisation du versionnement.

Dans le reste de ce chapitre, nous allons nous concentrer sur le développement d'un seul domaine. Nous avons choisi comme exemple le domaine de la gestion d'attributs de produits, car il est relativement facile à comprendre, sans nécessiter une grande expertise en gestion de configurations.

La gestion de produits s'intéresse essentiellement au stockage des attributs des objets versionnés. Ces attributs peuvent être de type simple (texte ou des énumérations de valeurs) ou des références vers d'autres objets. Toutes les versions sont stockées dans un référentiel centralisé, un repository. Les fonctionnalités que nous avons choisies d'inclure dans le domaine sont décrites dans le diagramme de la figure 17, en termes des features définis par [KG04] afin de pouvoir comparer avec les diagrammes précédents.

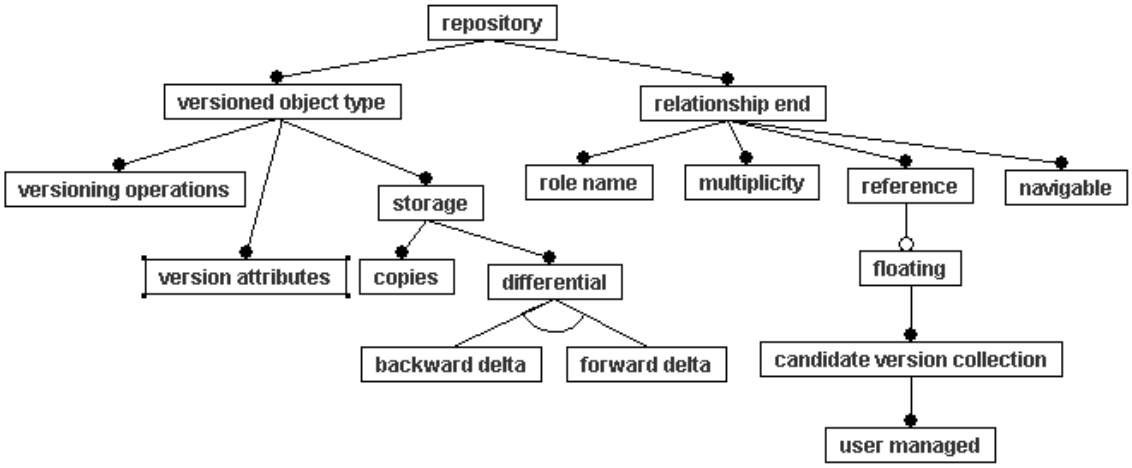

figure 17. Diagramme de Features du Domaine de Produit

On observe qu'on a choisi un sous-ensemble des fonctionnalités totales de la ligne de produits de gestion de configurations présentée auparavant, et c'est dans ce sens qu'on considère que ce domaine a une moindre couverture fonctionnelle. On remarque aussi qu'on a volontairement omis certains features très spécifiques a cette ligne de produits, qui empêcheraient ce domaine d'être réutilisé dans un contexte plus vaste ; c'est le cas par exemple des options de propagation automatique des opérations de versionnment, qui sont souvent liés à l'usage particulier d'un domaine composite.

A ce point dans notre démarche, nous avons choisi le domaine, nous avons examiné d'une façon large son contexte de réutilisation, et nous avons établi son périmètre fonctionnel. A partir de la, nous sommes en mesure de pouvoir entreprendre sa conceptualisation, afin d'aboutir au méta-modèle correspondant.

3.1.2 La conceptualisation du domaine.

L'analyse du domaine nous permet de délimiter la frontière du domaine, de mettre en évidence les aspects qui peuvent varier et de faire apparaître certaines de ses notions clés. C'est un premier pas important, car il permet d'identifier l'étendue du langage visé [Con04]. Néanmoins, il nous reste à identifier les concepts du langage de modélisation, qui vont permettre au concepteur de spécifier les caractéristiques propres à une application dans ce domaine.

Comme nous l'avons mentionné, dans le cas des langages de modélisation dédiés, il est essentiel d'avoir une forte correspondance entre les concepts du langage et ceux du domaine, et donc entre la conceptualisation du domaine et le méta-modèle du langage. Nous voulons proposer dans le langage des primitives de modélisation qui correspondent à des abstractions du domaine.

Nous adhérons donc à la démarche proposée par Guizzardi et al. [GPvS02] de construire le méta-modèle par homomorphisme avec la conceptualisation du domaine. Cette relation est montrée dans le schéma de la figure 18.

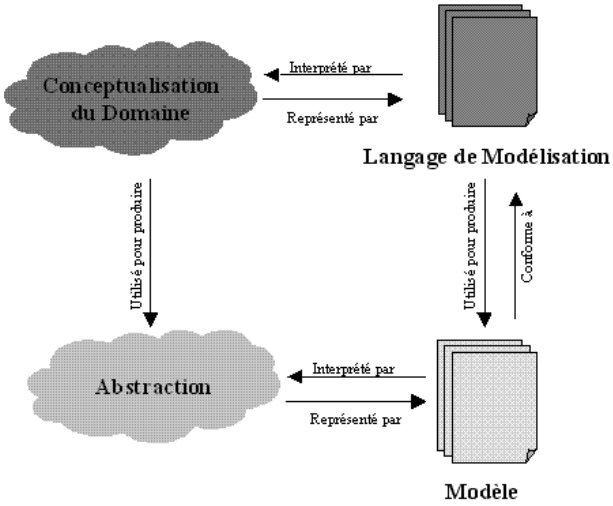

figure 18. Relation Conceptualisation – Langage [GPvS05]

Comme le montre la figure 18, une abstraction dans un domaine est articulée en termes d'une certaine *conceptualisation du domaine*, d'une représentation abstraite de certains aspects du domaine. Cette abstraction peut inclure des aspects structurels, mais aussi comportementaux, des entités modélisées. La conceptualisation et les abstractions sont des constructions qui ne sont que dans la tête des concepteurs ou de la communauté d'experts du domaine. Pour pouvoir devenir opérationnelles, elles doivent être capturées par des artefacts concrets, des modèles et des langages de modélisation [GPvS05].

En utilisant la conceptualisation du domaine comme point de départ pour la construction du méta-modèle, nous essayons de garantir que les modèles développés utiliseront des abstractions pertinentes du domaine. En construisant le méta-modèle par homomorphisme avec cette conceptualisation, nous cherchons à améliorer la compréhension des modèles, dans le sens où la conceptualisation servira de cadre de référence consensuel pour interpréter ces modèles.

Cette vision nous permet aussi d'établir un rapport avec la sémantique du langage. En effet, en sémiotique et linguistique, la sémantique dénote « une forme de correspondance spécifiée entre un représentant et son référent dans le monde réel »[1]. De ce point de vue, la correspondance avec la conceptualisation du domaine définit la sémantique du méta-modèle.

[1] Davis et al. What is Knowledge Representation? dans *AI magazine*, cité par [GPvS05]

En revanche, dans la communauté des langages, la sémantique d'un langage est définit par une correspondance entre les éléments de la syntaxe du langage et un domaine sémantique [HR04], souvent un domaine mathématique sans aucun lien avec la réalité.

Pour différentier ces deux points de vue, nous allons parler de la *sémantique du domaine* pour faire allusion à la relation entre le méta-modèle et le domaine réel, et de *sémantique formelle du langage* pour parler de la sémantique du méta-modèle en tant que langage de programmation.

Le pas suivant dans notre démarche est donc de matérialiser les concepts du domaine. Forcément, ici viennent à l'esprit des rapports avec les travaux sur les ontologies et la représentation des connaissances. Nous n'avons pas exploré ces pistes, car elles sont très éloignées de notre propre expertise.

Nous proposons donc, de façon très pragmatique, de réaliser une modélisation orientée objet des concepts clés qui permettent de définir les abstractions dans le domaine. Ce *modèle du domaine* sera la base pour la suite du développement du domaine.

Dans notre exemple, le domaine de Gestion de Produits, nous avons donc identifié les abstractions clés pour pouvoir modéliser la gestion de versions d'une application. La conceptualisation tourne autour d'un *modèle de données* qui décrit les types de produits qui vont être manipules dans le repository de versionnement. Chaque produit versionné appartient à un *Type de Produit* qui spécifie ses *Attributs*. Chaque attribut a un type : texte, énumération, référence. Un attribut a aussi des propriétés qui définissent leur comportement, vis à vis du versionnement. Le modèle du domaine de produits est montré dans le diagramme de classes de la figure 19.

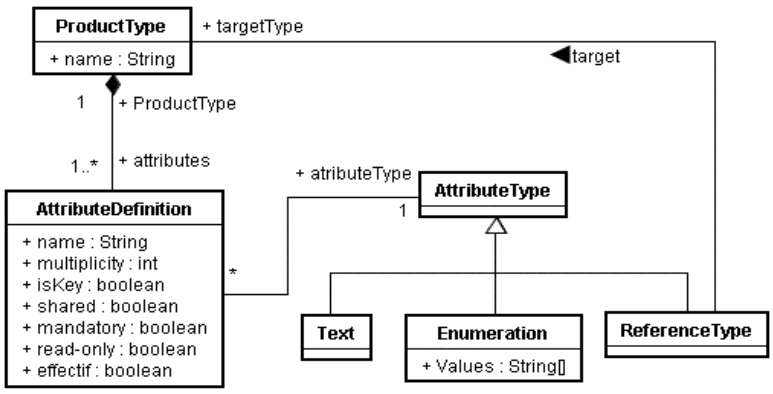

figure 19. Modèle du Domaine de Produit

Un attribut peut, par exemple, être défini comme partagé ; dans ce cas, sa valeur sera la même pour toutes les versions du produit, si on change un attribut partagé, il sera changé dans toutes les versions du même produit. Un attribut peut aussi être défini comme immuable (*read-only*), dans ce cas une fois sa valeur affectée, on ne peut plus la changer. Ces propriétés peuvent se combiner, par exemple, si on déclare un attribut à la fois non partagé et immuable alors la seule façon de changer sa valeur est de créer une nouvelle version du produit, avec la nouvelle valeur de l'attribut.

Les attributs peuvent avoir une multiplicité et, dans le cas de références, peuvent aussi être définis comme effectifs. Une référence effective peut pointer sur un ensemble de versions de l'objet cible de la relation ; ceci correspond à la notion de références flottantes dans la modélisation de [KG04], présentée dans la figure 15.

Un point important à souligner est que nous n'avons pas inclus dans le modèle du domaine certains aspects du domaine ; par exemple, nous ne représentons pas le fait qu'il existe plusieurs variantes de stockage, par copie ou par stockage différentiel. Nous considérons que cet aspect ne concerne pas, dans un premier temps, le concepteur qui veut modéliser les besoins de versionnement d'une application. Clairement, à un moment donné il faudra bien prendre en compte cette variabilité, qui est lié à l'implémentation. Dans notre approche, ces aspects sont considérés au moment de la configuration de la réalisation du domaine (voir §**Erreur ! Source du renvoi introuvable.**).

Nous avons donc à ce stade accompli une conceptualisation du domaine, concrètement représentée par un modèle du domaine. Nous disposons désormais d'un artefact concret : le modèle du domaine, sous forme de diagramme de classes, qui est l'embryon d'un méta-modèle opérationnel.

Avant de continuer une mise en garde s'impose. On peut nous reprocher que le modèle du domaine développé ne couvre pas l'intégralité du domaine, et que nous avons un biais en faveur de la construction d'un langage. C'est le cas. Nous ne cherchons pas à faire une conceptualisation exhaustive du domaine de connaissances, ni à définir tous les concepts d'une façon indépendante de la tâche à accomplir, comme le visent certaines méthodologies d'ingénierie d'ontologies. Le rapport entre notre approche et les approches basées sur les ontologies reste à explorer.

Dans notre démarche, la conceptualisation du domaine a pour objectif de produire un langage dédié de modélisation, qui reprend les concepts du domaine de façon très directe. Le modèle du domaine que nous avons proposé de construire doit donc, en même temps, capturer l'essence du domaine et fournir les constructions de base du méta-modèle du langage.

3.1.3 Spécification du méta-modèle.

Notre objectif final est de pouvoir obtenir un méta-modèle opérationnel du domaine, à partir de la conceptualisation effectuée. Le grand avantage pratique d'utiliser une modélisation orientée objet pour conceptualiser le domaine est que le méta-modèle correspondant peut être obtenu directement, en transformant le modèle du domaine en classes Java ou en éléments d'un framework de méta-modélisation, tel que Ecore.

Une fois les concepts bien identifiés, nous pouvons raffiner et enrichir le modèle du domaine avec des règles de cohérence, afin de préciser les abstractions correctes. Ces règles se convertiront par la suite en validations, qui nous permettrons de déterminer si un modèle est bien formé ou non. On voit ici à nouveau l'intérêt d'avoir une relation directe entre la conceptualisation du domaine et le méta-modèle du langage.

Par exemple, dans le cas du domaine de produit nous pouvons ajouter des règles simples pour décrire les combinaisons valides de propriétés d'attributs. La figure 20 présente certaines de ces contraintes, exprimées en OCL.

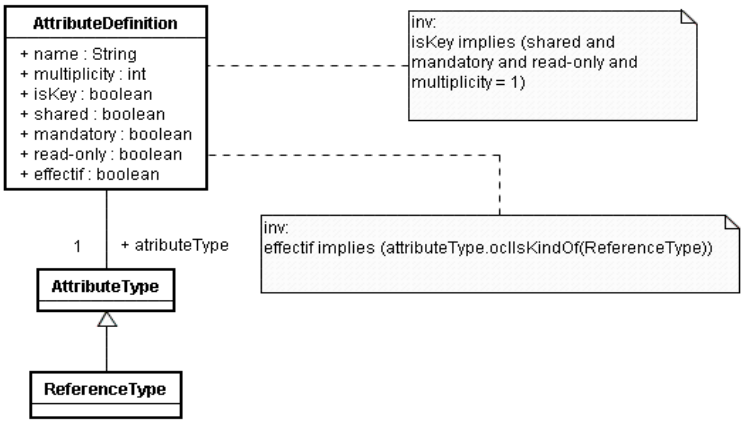

figure 20. Contraintes du modèle du Domaine de Produit

Apres avoir défini un méta-modèle précis, la dernière étape est de spécifier une syntaxe concrète pour le langage, et de fournir aux concepteurs les outils de modélisation appropriés. Pour cela, nous pouvons faire appel aux divers frameworks et environnements de méta-modélisation existants.

Nous n'allons pas nous attarder ici sur ce point, car il n'est pas central à notre démarche. Mais, nous insistons sur le fait que notre approche s'inscrit pleinement dans la mouvance de l'Ingénierie Dirigée par les Modèles, et elle bénéficie ainsi des énormes progrès en termes d'outillage obtenus dernièrement grâce à l'essor de recherche dans ce secteur.

Dans le cas du domaine de gestion de produits nous avons développé un outil spécifique de modélisation, qui permet de façon très simple de construire de modèles de données versionnés, et de valider les contraintes identifiées. Un aperçu de cet outil est montré dans la figure 21.

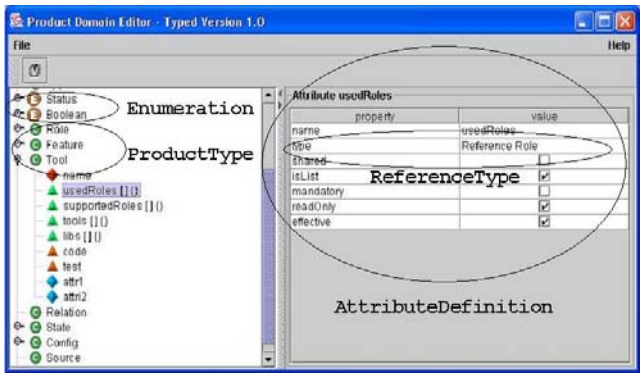

figure 21. Outil de modélisation du domaine de produit

Nous sommes donc au bout de la spécification du domaine. Nous avons choisi de formaliser une certaine expertise métier par un langage de modélisation propre au domaine. Le méta-modèle de ce langage est directement issu d'une conceptualisation qui capture la connaissance du domaine. Il nous permet aussi de fournir des outils pour modéliser les applications sous l'angle privilégié de ce domaine.

Nous ne voulons pas pourtant en rester là. Nous voulons que ces modèles deviennent productifs, et qu'ils puissent être transformés directement en un système exécutable.

3.2 LES MODELES EXECUTABLES.

Nous nous sommes concentrés jusqu'à présent sur la conceptualisation et la définition du périmètre domaine. Ceci nous a permit de spécifier le méta-modèle d'un langage dédié au domaine. Nous nous intéressons maintenant à l'exécution des modèles exprimés dans ce langage de modélisation.

Tout d'abord, nous essaierons de clarifier ce que nous entendons par exécution d'un modèle. Nous avons deux perspectives complémentaires pour réfléchir sur l'exécution d'un modèle : le langage de modélisation et la plate-forme d'exécution.

Si nous nous focalisons sur le langage de modélisation, la notion d'exécution est très liée à la sémantique formelle du domaine. Dans ce cas, il faut spécifier précisément quel est l'état du système et sa dynamique d'évolution.

En revanche, si nous nous centrons sur la plate-forme d'exécution, la notion d'exécution est très liée aux divers artefacts mis en jeu pour supporter l'exécution, à partir de la spécification du modèle. Dans ce cas il faut préciser les composants à développer et, le cas échéant, les systèmes propriétaires ou les composants sur étagère qu'il faut intégrer.

Si nous reprenons notre cas d'étude de la gestion de produits, et nous essayons de donner une notion d'exécution aux modèles de produit versionné, nous devons, par exemple, nous intéresser à la définition précise de la notion de version et à la façon de créer de nouvelles versions. Du point de vue de la plate-forme, il nous faudra aussi définir, entre autres, les caractéristiques du serveur de stockage ou l'éventuelle façon d'interfacer un système propriétaire de PDM.

Ces perspectives correspondent à des préoccupations sur deux axes différents : un axe utilisateur et un axe implémentation [Rev96]. Elles sont également importantes, et doivent pouvoir être intégrées au sein de notre démarche. Nous les considérons comme partie intégrale du processus d'ingénierie du domaine, présenté schématiquement dans la figure 22.

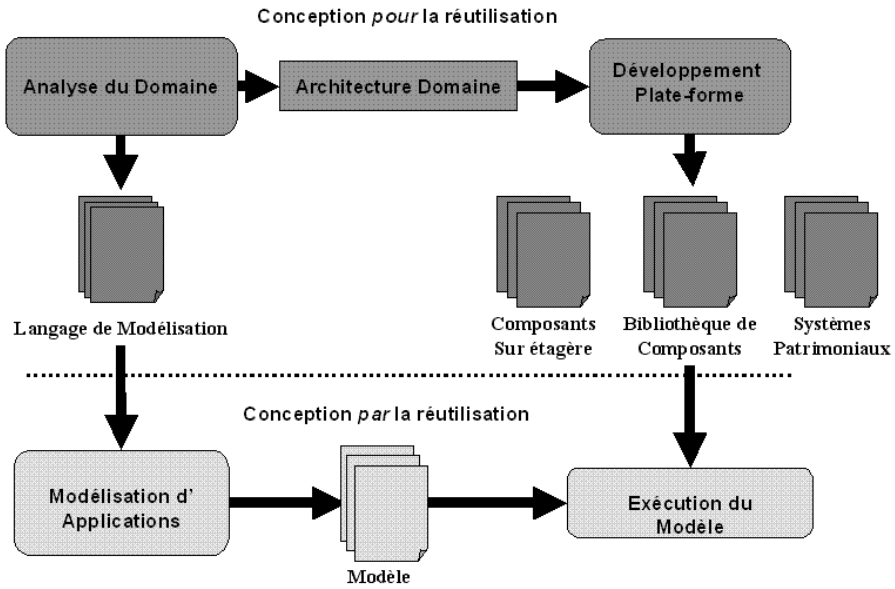

figure 22. Démarche globale d'Ingénierie du Domaine

Nous pouvons implémenter ce schéma d'exécution soit à l'aide d'un générateur de code, soit à l'aide d'un interpréteur. Du point de vue de l'Ingénierie Dirigée par les Modèles, les deux approches sont envisageables. Dans le cas du générateur, nous pouvons faire appel aux techniques de transformation de modèles. Pour l'interprétation, nous pouvons recourir à la sémantique opérationnelle des méta-modèles.

Néanmoins, d'un point de vue pratique, nous devons considérer comment chacune de ces alternatives peut nous permettre de concilier les deux axes de l'exécution d'un modèle : la vision modélisation et la vision plate-forme.

Notre principal critère de choix a été la capacité de répondre à l'évolution du domaine sur ces deux axes. En effet, nous avons, d'un côté, le langage de modélisation, qui est fortement lié à la conceptualisation du domaine, et qui évolue, en conséquence, au fur et à mesure que la compréhension du domaine change. De l'autre côté, nous avons une plate-forme qui doit faire face à l'évolution constante de la technologie, et qui, en plus, comporte des composants sur étagère et patrimoniaux, pour lesquels nous ne contrôlons aucunement leur évolution.

Cette différence en termes de fréquence, et ampleur d'évolution, complique énormément l'approche par transformation. Clairement, cette transformation est dépendante du méta-modèle du domaine et du méta-modèle de la plate-forme cible, et l'évolution fréquente du dernier entraînerait une maintenance continue de la transformation.

Par contre, dans l'approche par interprétation, l'interpréteur est exprimé seulement en termes du méta-modèle du domaine, ce qui facilite sa mise en œuvre [Thi98], ainsi que son écriture, compréhension et maintenance [CES+04].

Nous avons aussi privilégié l'alternative d'interprétation car elle nous paraît une continuation naturelle de la formalisation du méta-modèle du domaine. Le méta-modèle du domaine ne comporte à ce stade que la spécification structurelle du langage, il nous manque sa sémantique formelle de l'exécution. Le développement de l'interpréteur nous permet de spécifier très précisément la sémantique opérationnelle du langage de modélisation.

La difficulté majeure de l'approche par interprétation est son intégration avec la plate-forme d'exécution du domaine. Nous risquons, par exemple, de développer dans l'interpréteur beaucoup de code qui est déjà programmé dans les composants, ce qui nuirait la réutilisation. En revanche, si nous tenons à réutiliser les composants existants, il faut les intégrer à l'interpréteur, et nous risquons d'avoir les mêmes problèmes que dans l'approche par transformation.

Nous proposons d'affronter cette difficulté comme un problème de synchronisation [EJB04]. D'un côté, nous aurons un interpréteur, relativement simple, qui spécifie la sémantique opérationnelle du langage de modélisation domaine, et qui permet d'exécuter directement les modèles. De l'autre côte, nous aurons un système, basé sur des composants réutilisables, s'exécutant en parallèle, et avec son propre état. Une couche intermédiaire de synchronisation assure la coordination entre ces deux exécutions.

Nous parlerons donc dans notre démarche d'*Exécution Abstraite*, d'*Exécution Concrète* et de *Couche de Médiation*.

3.3 MACHINE VIRTUELLE DU DOMAINE.

Nous nous sommes intéressés à l'exécution des modèles, et nous avons proposé de séparer deux visions complémentaires : d'une part, nous allons développer un interpréteur formel du domaine ; d'autre part, nous allons réutiliser un ensemble de composants existants.

L'objectif de l'interpréteur formel est de fournir une sémantique opérationnelle au langage de modélisation. Typiquement, le développement d'un tel interpréteur consiste à associer aux concepts du langage des opérations qui définissent leur comportement [CES+04].

Nous proposons donc de partir du méta-modèle du domaine que nous avons développé, et de l'étendre pour spécifier la dynamique du système. Ce méta-modèle étendu constitue la *Machine Virtuelle du Domaine*, qui va nous permettre d'exécuter directement les modèles du domaine.

3.3.1 Représentation de l'état de l'exécution.

Si nous voulons spécifier précisément la dynamique de l'exécution d'un modèle, forcément il faut s'intéresser à la notion d'état, et à son évolution. Le méta-modèle que nous avons développé jusqu'à maintenant ne nous permet de spécifier que la syntaxe abstraite du langage de modélisation, il faut l'étendre pour tenir compte de la partie concernant l'état de l'exécution.

Nous avons donc au niveau du méta-modèle une partie dite *statique* et une partie dite *dynamique* [BB01]. Ces deux parties se complémentent : « Un bénéfice majeur de la partie statique d'un modèle est qu'elle nous permet de raisonner indépendamment d'une situation particulière. … Le but de la partie dynamique est différent. Elle réagit, comme le système le ferait, dans un contexte particulier, en réponse à un stimulus particulier. Elle peut donc servir à simuler le système ou même le contrôler. La réaction à un événement consiste en une série d'actions appliquées au modèle »[1].

Pour reprendre le cas d'étude du gestionnaire de produits, nous devons donc spécifier en quoi consiste l'état de l'exécution d'un modèle de versionnement, et nous devons étendre le méta-modèle en conséquence, pour tenir compte des concepts dynamiques de l'exécution.

Dans un premier temps, nous allons alors identifier les concepts qui vont nous permettre de représenter l'état dynamique de l'exécution, et nous allons les ajouter au méta-modèle du domaine.

Dans le cas de la gestion de produits, les concepts généralement employés dans le domaine pour raisonner sur l'état du système sont les notions de *produit*, *branche* et *révision*. Le méta-modèle du domaine de produit résultant d'incorporer la partie dynamique est présenté dans la figure 23.

[1] [BB01] p.73

Un produit est l'unité de versionnement, il est associé à un type de produit qui détermine ses caractéristiques. Une version particulière d'un produit est appelée une révision. Les révisions d'un produit sont organisées dans une structure arborescente, chacune des branches de cette structure représente l'évolution du produit dans le temps, il existe donc un ordre temporel entre les révisions d'une même branche

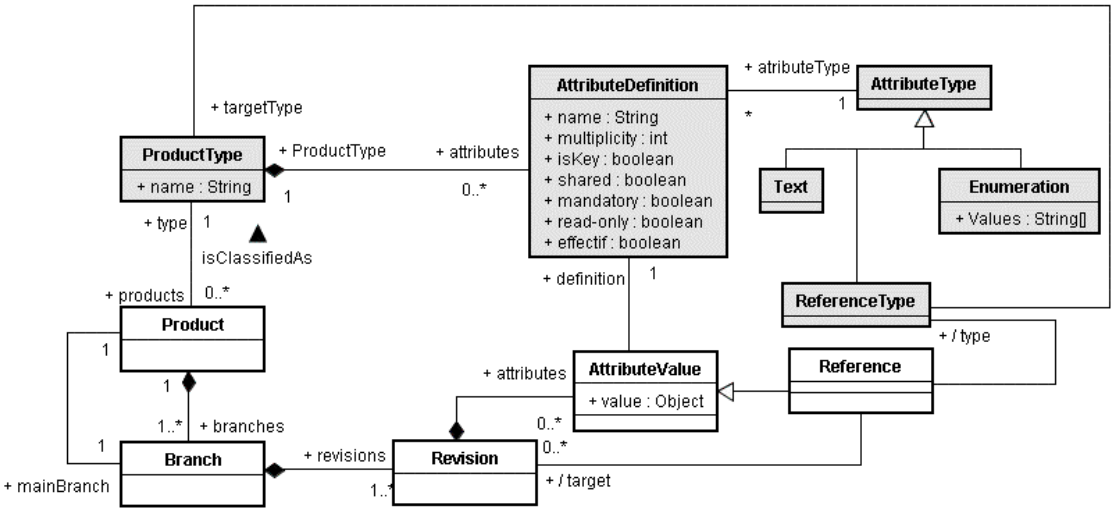

figure 23. Méta-Modèle d'exécution du Domaine de Produit

Le lien de conformité.

Un autre point important à remarquer dans l'exemple de la figure 23 est l'existence d'associations entre les classes de la partie dynamique et celles de la partie statique. Nous pouvons expliquer ces associations comme une réification du lien de conformité qui doit exister entre l'état de l'exécution et la spécification prescrite par le modèle.

Ces associations entre la partie dynamique et la partie statique sont très importantes, car elles vont nous permettre d'assurer que les opérations qui modifient l'état de l'exécution sont conformes à la spécification du modèle.

Par exemple, dans le méta-modèle de la figure 23 nous avons une association entre la class `Product` et la classe `ProductType`. Quand nous allons, par exemple, implémenter la méthode pour modifier un attribut d'une instance de produit, il nous faudra naviguer sur cette association pour vérifier ses caractéristiques. Conceptuellement cette association correspond à la notion de classification : le type de produit caractérise les instances des produits associés.

Nous pouvons discerner dans la figure 23 un pattern très fréquent : en haut une couche qui réifie le modèle, en bas une couche qui réifie l'état de l'exécution du modèle, et des associations pour garantir le lien de conformité entre les deux.

Ce pattern a été, par exemple, recensé par [BKK+99] pour des applications de gestion de versionnement, et par [FR04] pour des lignes de produits basés sur des langages dédiés. Il est aussi très utilisé dans les applications qui visent l'adaptabilité ; dans ce type d'applications il est utilisé pour pouvoir modifier dynamiquement le modèle, et pour que le système prenne en compte ces modifications à la volée.

Il a également été documenté par Yoder et Johnson [YJ02] comme un style architectural, sous le nom de Modèle à Objets Adaptatif, AOM[1]. Curieusement, cette architecture est souvent associé aux systèmes de méta-modélisation orientés objets, par exemple son usage pour la construction d'une machine virtuelle UML est documenté dans [RFB+01]. Ce rapport entre méta-modélisation et architectures AOM est exploré davantage dans [RY01].

Nous avons utilisé systématiquement cette architecture pour le développement des machines virtuelles des domaines, dans nos expérimentations. Nous faisons remarquer que cette architecture découle naturellement de notre démarche. La partie statique du méta-modèle est le résultat de la phase initiale d'analyse du domaine, et la partie dynamique est ajoutée pour définir la machine virtuelle du domaine.

Cette architecture suggère aussi que nous pourrons dans le futur rendre plus modulaire notre démarche en produisant deux artefacts distincts : un méta-modèle structurel et un méta-modèle d'exécution séparé, qui seront tissés pour obtenir la machine du domaine. Ce type de structuration de méta-modèles est utilisé par exemple dans [BB02] et [MFJ05].

Correspondance avec la conceptualisation.

Cet exemple nous permet déjà de faire quelques remarques sur la façon de construire la machine virtuelle du domaine. On observe d'abord que nous avons modélisé très directement les concepts dynamiques du domaine. Nous les avons représentés explicitement par des classes et associations dans le méta-modèle.

Ceci n'est pas strictement nécessaire, beaucoup d'interpréteurs utilisent des représentations internes qui ne correspondent directement aux notions utilisées lors de la conceptualisation, souvent pour des raisons de performance. Cependant, dans notre démarche, nous tenons à maintenir l'homomorphisme entre la conceptualisation du domaine et le méta-modèle du langage, aussi bien pour la partie statique que pour la partie dynamique.

A ce propos, nous partageons l'avis d'Evans [Eva04] : « Si la conception du logiciel … ne correspond pas au modèle du domaine, ce modèle n'a que très peut de valeur, et la correction du logiciel est suspecte. En même temps, des correspondances complexes entre le modèle et l'implémentation sont difficiles à comprendre et, en pratique, impossibles à maintenir. … Concevez le logiciel pour refléter le modèle du domaine de façon très littérale, pour faire en sorte que cette correspondance soit très évidente »[2].

Plus spécifiquement pour notre démarche, le rapport entre la partie dynamique du méta-modèle et la conceptualisation du domaine nous permet de réfléchir à tout moment sur la justesse de l'implémentation de la machine virtuelle. Nous mettons donc

[1] Adaptive Object-Model

[2] [Eva04] p.48

en avant un lien profond entre la sémantique du domaine et la sémantique opérationnelle du langage, comme l'illustre la figure 24.

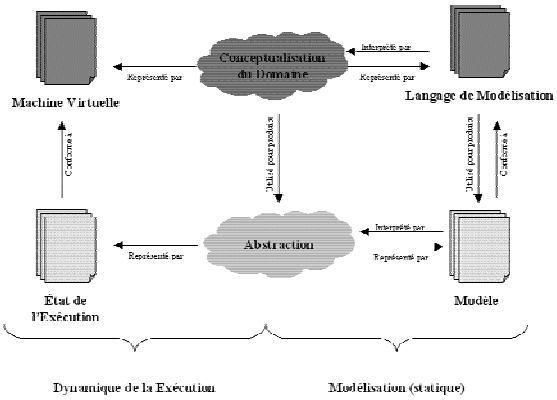

figure 24. Relation Conceptualisation – Langage - Exécution

Il va s'avérer par la suite, que c'est précisément cette relation étroite entre la sémantique du domaine et la sémantique formelle du langage, propre aux langages dédiés, qui va nous permettre d'assurer d'une façon relativement simple et directe la correspondance entre l'exécution abstraite et l'exécution concrète, et qui va nous permettre encore la composition de domaines.

3.3.2 Dynamique de l'exécution.

Après avoir étendu le méta-modèle avec les classes indispensables pour représenter l'état de l'exécution, nous continuons le développement de la machine virtuelle du domaine par les opérations qui vont nous permettre de capturer le comportement dynamique de l'exécution du modèle. Dans notre cas, nous avons choisi de programmer les opérations de la machine virtuelle en Java.

Clairement, la spécification des méthodes à implémenter dépend fortement de la notion d'exécution propre au domaine auquel on s'intéresse. Mais, dans tous les cas, nous les implémentons au niveau des classes dynamiques du méta-modèle.

Dans le cas du domaine de produit la notion d'exécution d'un modèle de versionnement correspond à une spécification précise de la façon dont la structure arborescente des versions d'un produit évolue, ainsi que la valeur des attributs. Les méthodes principales sont présentées brièvement dans la figure 25.

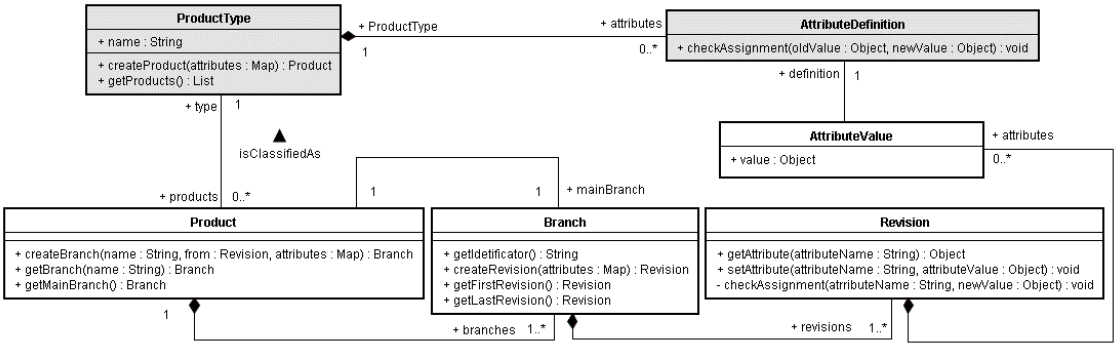

figure 25. Machine Virtuelle du Domaine de Produit

Le lien de conformité.

Nous voulons revenir ici sur la notion de causalité entre le modèle et l'état de l'exécution du modèle, introduite dans la section précédente, pour faire quelques remarques sur notre implémentation dans les machines virtuelles que nous avons construites.

Le lien de conformité se manifeste, dans l'exemple de la figure 25, par une association entre le produit et son type. Cette association doit être naviguée par certaines des méthodes de la machine virtuelle pour assurer la conformité de l'exécution avec la spécification du modèle. Ceci correspond à l'usage du pattern AOM.

Nous allons illustrer l'implémentation de ce pattern en Java par un exemple simple. Le modèle de produit détermine pour chaque attribut son type, et certaines propriétés, telle que l'immutabilité. Ce comportement doit être imposé lors de l'affectation d'une valeur à un attribut. Cette vérification est faite par la méthode `checkAssignment` de la classe `Revision`, transcrite ci-dessous.

```
private void checkAssignment(String attributeName, Object newValue) {
   ProductAttribute attribute =
      getProductType().getAttribute(attributeName);

   if (attribute == null)
      throw new IllegalArgumentException("Invalid attribute);

   attribute.checkAssignment(getAttribute(attribute.getName()),newValue);
}
```

figure 26. Lien de conformité - `Revision`

Pour cela, cette méthode traverse l'association vers le type de produit, et appelle la méthode homologue de la classe `AttributeDefinition`, transcrite dans la figure 27, qui effectue les validations correspondantes.

```
public void checkAssignment(Object oldValue, Object newValue) {
   /*
    * Verify type compatibility
    */
   Class valueType = newValue.getClass();
   if (isText() && !String.class.isAssignableFrom(valueType))
      error(String.class,valueType);
   if (isEnumeration() && !String.class.isAssignableFrom(valueType))
      error(String.class,valueType);
   if (isReference() &&
      !Revision.Reference.class.isAssignableFrom(valueType) )
         error(Revision.Reference.class,valueType);
   /*
    * Verify mutability
    */
   if (!isMutable() && (oldValue != null))
      error("trying to modify immutable attibute ",newValue);

   /*
    * Verify mandatory
    */
   if (isMandatory() && (oldValue != null) && (newValue == null))
      error("trying to set to null mandatory attibute ",newValue);

   /*
    * Verify enumeration values
    */
   if (isEnumeration() && !getType().contains((String)newValue))
      error("invalid Enumeration value ",newValue);
}
```

figure 27. Lien de conformité - `AttributeDefinition`

Cet exemple nous permet de faire quelques observations. Tout d'abord, nous pouvons remarquer que le fait d'avoir séparé nettement la partie statique de la partie dynamique nous permet d'avoir toujours accès à la définition du modèle, sous forme d'instances des classes de la partie statique du méta-modèle.

Nous pouvons entrevoir, également, que le comportement des instances des classes dynamiques, qui représentent l'état effectif de l'exécution, n'est pas complètement déterminé par sa classe, mais il est en partie détermine par les classes statiques.

Nous observons donc que nous faisons cohabiter dans la machine virtuelle trois niveaux de l'architecture de méta-modélisation : le *niveau méta-modèle*, représente par les classes que nous avons qualifiées de statiques, le *niveau modèle*, représenté par les instances de ces classes statiques, et le *niveau exécution de modèle* représenté par les instances des classes que nous avons qualifiées de dynamiques.

Cette cohabitation n'est pas anodine. Elle n'est pas seulement conceptuellement complexe, mais elle complique techniquement la mise en œuvre de la machine virtuelle.

Pour la partie statique, il nous semble que le lien de conformité entre le niveau modèle et le niveau méta-modèle est adéquatement représenté par la notion d'instanciation de Java : nous faisons correspondre le niveau méta-modèle au niveau des classes Java, et le niveau modèle au niveau des objets Java.

En revanche, le lien de conformité entre l'état de l'exécution et le niveau modèle est assuré indirectement, donc beaucoup moins évident : l'état de l'exécution est représenté par des objets Java d'une classe dynamique, et cette classe dynamique délègue une partie de son comportement à une classe statique.

Dans notre exemple concret, nous pouvons également observer que le lien de conformité correspond à une relation de classification, la classe `ProductType` se comporte comme un powertype [HG05] de la classe `Revision` : les instances de `ProductType` classifient les instances de `Revision`.

Ces observations nous rapprochent des langages à méta-classes et nous font nous interroger sur le besoin d'utiliser des facilités de classification dynamique et méta-classes au niveau des langages de méta-programmation. Nous n'avons pas exploré cette idée, mais dans ce sens s'orientent la proposition fait par Gonzalez et Henderson [GH05] d'incorporer le powertype aux frameworks de méta-modélisation, ainsi que les travaux de Razavi et al. [RBY+04] sur le rapport entre le pattern de Modèle à Objets Adaptatif et les langages à méta-classes.

Nous voulons insister sur le fait que l'objectif de la machine virtuelle du domaine est de spécifier le comportement dynamique de l'exécution du modèle, et que la structuration en deux niveaux séparés (partie statique et dynamique) ainsi que l'utilisation systématique des patterns contribuent à la clarté de cette spécification.

Réification des concepts absorbés.

Certaines des classes et méthodes du méta-modèle dynamique sont indépendantes du modèle que nous sommes en train d'exécuter. Leur comportement est identique pour tous les systèmes du domaine, et n'est pas contraint par le modèle. Elles

correspondent aux aspects communs que nous avons identifiés lors de l'analyse du domaine.

Dans le domaine de produit, par exemple, le fait que les versions d'un produit soient structurées dans une structure arborescente en branches n'est pas visible lors de la modélisation, et ne peut pas être contrôlé par le modèle de produit.

Du point de vue du langage de modélisation, si un aspect du comportement interne du système est laissé implicite, on dit qu'il est *absorbé* par les concepts du langage [Ste94]. C'est le cas dans les langages de programmation classiques de certains aspects tels que la pile d'exécution ou le ramasse-miettes, par exemple.

La quantité de détail absorbée par le langage détermine, en partie, sa capacité à exprimer des modèles d'une façon déclarative et concise. Clairement, il y a un compromis entre la quantité d'aspects absorbés par le langage, sa généralité et son pouvoir d'expression. Dans le cas des langages dédiés à un domaine, ce compromis est fait lors de l'analyse des caractéristiques communes et variabilités, c'est à ce point qu'on décide si on va considérer plusieurs variantes d'un certain aspect ou si on va choisir une seule alternative particulière.

On peut constater que la plupart des langages dédiés absorbent un grand nombre d'aspects, et proposent très peut de mécanismes au concepteur pour définir ses propres abstractions [Thi98]. Ceci s'explique par le fait que les caractéristiques communes du domaine fournissent un contexte qui permet de laisser implicite un grand nombre d'aspects du système.

Cependant, ces aspects implicites sont la source d'inexactitudes et ambiguïtés lors de la conception. La machine virtuelle du domaine lève une partie de cette ambiguïté, en fournissant une sémantique précise pour ce comportement implicite. Ceci nous semble particulièrement important dans le cas de langages dédiés, vu le grand nombre d'aspects absorbés.

En effet, nous observons que, au moyen de la machine virtuelle, nous avons réifié ces concepts absorbés et nous avons spécifié leur comportement. Nous considérons donc la machine virtuelle non seulement comme un outil d'exécution, mais aussi comme un artefact conceptuel qui formalise une partie importante de la connaissance du domaine.

Le choix du langage de méta-programmation.

Le développement de la machine virtuelle nous confronte au problème du choix du langage de méta-modélisation à utiliser. La plupart des langages de méta-modélisation, tels que MOF ou Ecore, ne proposent pas directement un support pour la définition d'actions. Nous pouvons donc soit utiliser des extensions, par exemple des langages d'actions comme Xactium [CES+04] ou Kermeta [MFJ05], soit utiliser un langage de programmation générique comme Java.

Nous avons choisi de réifier la conceptualisation du domaine en Java, et programmer la machine virtuelle comme des méthodes de ces classes réifiées. Notre choix à été, en grand partie, motivé par la disponibilité d'environnements de développement performants, et par des besoins propres à notre démarche, comme

l'utilisation des techniques de Programmation Orientée Aspect (voir §3.4.2), qui ne sont pas communément disponibles dans les langages de méta-modélisation.

Rétrospectivement nous pouvons dire que le choix de Java risque de favoriser la vision de la machine virtuelle comme simplement un outil d'exécution, au détriment de la vision que nous voulons mettre en avant d'une spécification de la sémantique opérationnelle du domaine. Néanmoins, ce choix nous a permis d'avancer très rapidement dans nos expérimentations, sans nous confronter aux problèmes liés à l'immaturité des outils de méta-modélisation disponibles quand nous avons démarré nos travaux.

Nous n'avons pas d'état d'âme particulier sur les langages de méta-modélisation, et nous accueillons toute avancée dans l'outillage des environnements de méta-programmation. Au contraire, nous pensons que notre expérience peut devenir un cas d'étude intéressant pour déterminer les caractéristiques et les besoins réels d'un langage exécutable de méta-modélisation, et peut être mis à profit par les développeurs de ce type d'outils.

3.4 COMPOSANTS ET COUCHE DE MEDIATION.

Dans la section précédente, nous nous sommes intéressés à la notion d'exécution des modèles du point de vue du langage de modélisation. Nous avons développé une machine virtuelle du domaine qui spécifie la sémantique opérationnelle du langage, et permet l'*exécution abstraite* des modèles.

La machine virtuelle nous permet de simuler et valider les modèles, au niveau d'abstraction de la conceptualisation du domaine. Cependant, ceci n'est pas suffisant pour avoir un système exploitable, nous devons considérer en détail d'autres aspects qui sont liés à la plate-forme d'exécution.

Par exemple, dans le cas du domaine de produit, à l'aide de la machine virtuelle que nous avons développée, nous sommes capables d'écrire facilement un programme pour simuler la création de révisions d'un produit et son évolution dans le temps, ce qui nous permet de valider le modèle de produit. Toutefois, si nous voulons avoir un système utilisable en pratique, il faut s'intéresser à des aspects plus détaillés de l'exécution, tels que le stockage des versions ou fournir un outil pour permettre aux utilisateurs d'interagir à distance avec un repository centralisé des versions, voir même à la réplication et répartition de ce repository.

Les services de base pour la construction de ce système sont fournis par une plate-forme d'exécution spécifique au domaine. Elle comporte un framework à composants qui adresse les besoins spécifiques des membres du domaine, ainsi que, éventuellement, des adaptateurs pour s'interfacer avec des composants sur étagère ou des systèmes patrimoniaux. Comme nous l'avons signalé dans la section §3.2, le développement de cette plate-forme fait partie du processus d'ingénierie du domaine.

Dans cette section nous nous intéressons à la structuration de la plate-forme d'exécution, l'intégration des logiciels préexistants, de même qu'à l'interdépendance avec la machine virtuelle du domaine.

3.4.1 Composants et fédération de composants.

Contrairement à la sémantique opérationnelle du langage qui peut se définir dans l'espace du problème (au même niveau d'abstraction que la conceptualisation du domaine), la plate-forme d'exécution doit se préoccuper d'un grand nombre de détails appartenant à l'espace de la solution et à l'infrastructure technologique.

De plus, si nous considérons des domaines avec une vaste portée de réutilisation, il est probable que la fonctionnalité du domaine recouvre partiellement celle des systèmes logiciels existants, que nous souhaiterions faire inter-opérer avec le logiciel développé expressément pour le domaine.

Une approche classique à base de composants permet de tenir compte des aspects technologiques, structurer l'espace de la solution, et nous permet de remonter le niveau d'abstraction, pour pouvoir raisonner sur le système en termes d'un ensemble des composants qui interagissent au moyen d'interfaces abstraites.

Dans cette perspective, quand nous parlons de *l'exécution concrète* d'un modèle, nous faisons donc référence effectivement à l'exécution d'un assemblage de composants en concordance avec la spécification du modèle.

L'approche à composants nous permet aussi d'intégrer de façon homogène les systèmes tiers et patrimoniaux: la fonctionnalité fournie par le logiciel existant est considérée comme l'implémentation d'une interface abstraite qui représente son rôle dans le système intégré [WSG+05].

L'incorporation des composants tiers et patrimoniaux dans l'exécution concrète du système augmente la réutilisation à l'intérieur du domaine, mais introduit des nouvelles difficultés [EB04] [EJB04] :

- Ces composants ne sont pas conçus pour participer à un assemblage, ils ne sont pas conscients du rôle qui sont supposés jouer dans le système intégré. Il faut donc pouvoir altérer leur comportement sans les modifier directement, ce qui nécessite l'usage de techniques d'adaptation et instrumentation ;
- Ces composants ont été conçus pour travailler de façon autonome, ils possèdent un état interne qu'ils font évoluer de leur propre initiative, en réponse à des changements dans l'environnement ou à des interactions directes avec l'utilisateur. Certains de ces changements d'état ont un lien avec l'état du système global, qui doit être notifie ;
- Ces composants n'ont pas été conçus pour travailler ensemble. Il est donc possible que les mêmes concepts soient partagés par plusieurs d'entre eux, avec des représentations de données différentes. Le système global doit assurer la cohérence de ces données partagées.

Cette problématique complexe a été abordée, dans notre équipe de recherche, notamment dans la thèse de J. Villalobos [Vill03] qui a proposé une architecture basée sur la notion de coordination comme moyen de composition. Cette architecture introduit une couche de coordination qui est chargé d'assurer la cohérence de l'état interne des composants et le contrôle global du système intégré.

Nous proposons donc d'utiliser cette architecture pour structurer les composants de la couche d'exécution concrète d'un domaine, comme l'illustre l'exemple de la figure 28.

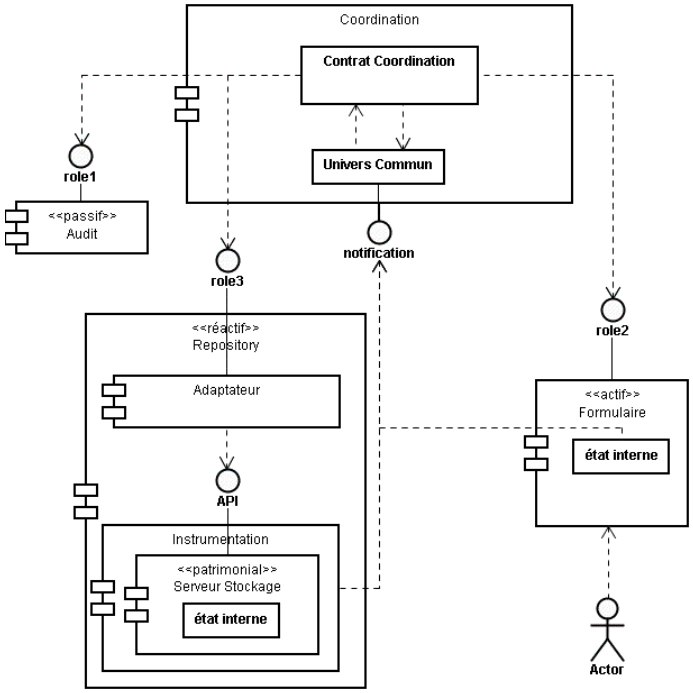

figure 28. Composition par coordination

Dans cette architecture, les concepts partagés par les composants sont matérialisés dans la couche de coordination par un état partagé, dénommé l'Univers Commun, utilisé comme moyen de synchronisation. Quand l'état interne d'un composant change, il doit notifier la couche de synchronisation qui maintient l'état partagé, ce changement peut entraîner le déclenchement d'un contrat de coordination qui assure le contrôle global de l'application, en invoquant les réactions à cet événement dans tous les composants impliqués. Un ensemble de composants régi par un coordinateur commun est appelé une fédération de composants.

Dans le cas de composants préexistants, il faut recourir à des adaptateurs pour implémenter la réaction aux événements, et à l'instrumentation pour effectuer les notifications de changement de l'état interne [EB04].

La figure 28 donne un exemple de cette proposition par des outils du domaine de produit. Le repository est responsable de la base de données de toutes les révisions de produits. L'utilisateur peut créer de nouveaux produits et révisions ou modifier les attributs des révisions existantes à l'aide d'un composant de gestion de formulaires. Clairement, ces deux composants partagent des données ; néanmoins, la représentation interne de ces données n'est pas nécessairement la même : dans un cas elle est optimisée pour le besoin de persistance, dans l'autre pour le besoin d'affichage. Ils doivent forcement être coordonnés.

Dans l'exemple, l'outil interactif est le composant actif, qui a l'initiative : il notifie la couche de coordination en réponse aux modifications de son état interne par les actions de l'utilisateur. La couche de coordination répercute ce changement sur l'état

partagé et déclenche le contrat de coordination. Le repository est réactif, et s'active pour mettre à jour le serveur de stockage. Un outil d'audit des modifications illustre un composant passif : il est complètement contrôlé par les contrats de synchronisation.

Cet exemple nous permet de faire une remarque additionnelle sur cette architecture. Manifestement, l'exemple que nous avons présenté est inefficace en termes de performance dû au passage par la couche de coordination et la réplication de l'information. Cet agencement de composants n'a un sens que si le serveur de stockage et le gestionnaire des formulaires ont été conçus de façon indépendante, et nous ne pouvons pas les modifier pour les réutiliser.

Si les composants sont développés spécifiquement pour le domaine, ils peuvent communiquer directement. L'architecture n'interdit pas la communication directe entre composants, elle propose des mécanismes de coordination pour les cas où le concepteur l'estime nécessaire.

Nous pensons que l'architecture de fédération répond bien aux besoins de la couche d'exécution concrète du domaine. Toutefois, il nous reste un point crucial à élucider : dans cette architecture le contrôle global de l'application est déterminé par les composants actifs et les contrats de synchronisation, mais nous voulons que l'exécution des composants concrets soit pilotée par l'exécution du modèle de l'application. Ceci est la responsabilité de la couche de médiation.

3.4.2 La couche de médiation.

L'objectif de la couche de médiation est de préserver la synchronisation entre l'*exécution abstraite* du modèle, assurée par la machine virtuelle du domaine, et l'*exécution concrète*, assurée par la fédération de composants.

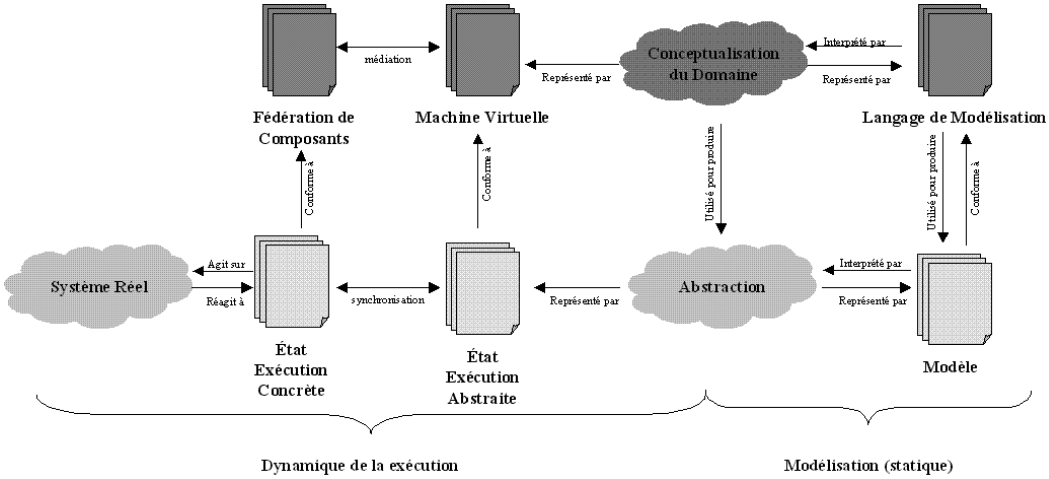

figure 29. Relation conceptualisation – exécution concrète

Un aspect essentiel de la couche de médiation, mis en évidence dans la figure 29, est que conceptuellement elle permet d'établir un lien entre l'abstraction représentée par le modèle et le système réel. Effectivement, nous avons construit la machine virtuelle du domaine de sorte que l'état de l'*exécution abstraite* soit conforme au comportement prescrit par le modèle, voir §3.3.2. Si la couche de médiation, à son tour, maintient cet état synchronisé avec l'état concret de l'exécution de composants, alors le modèle indirectement déterminera le comportement du système réel.

La figure 29 nous donne aussi la clé sur la manière d'entreprendre cette synchronisation. Lors de la description de la machine virtuelle du domaine, nous avons insisté sur le fait que les concepts correspondants à l'état de l'*exécution abstraite* sont matérialisés explicitement, voir §3.3.1. Egalement, dans l'architecture des fédérations de composants, nous avons souligné que l'état interne des composants est synchronisé avec un état partagé matérialisé par l'Univers Commun de coordination. L'idée qui suit naturellement cette observation est de faire coïncider l'état de l'exécution de la machine virtuelle et l'Univers Commun de coordination.

Cette idée a été initialement avancée dans la thèse d'A-T. Le [Le04], qui proposait de changer le regard sur l'Univers Commun d'une fédération : passer d'une vision ascendante dans laquelle il correspond aux concepts partagés par les composants de la fédération, à une vision descendant dans laquelle il correspond à un modèle abstrait du système global représenté par la fédération. Elle propose également la vision d'un domaine comme une abstraction d'une fédération de composants.

La même idée appliquée à notre architecture équivaut à incorporer les responsabilités de la couche de coordination de la fédération dans la machine virtuelle. Notre proposition consiste à considérer les classes dynamiques du méta-modèle comme une abstraction de l'état interne des composants, et de les maintenir synchronisés, tel qu'esquissé dans la figure 30.

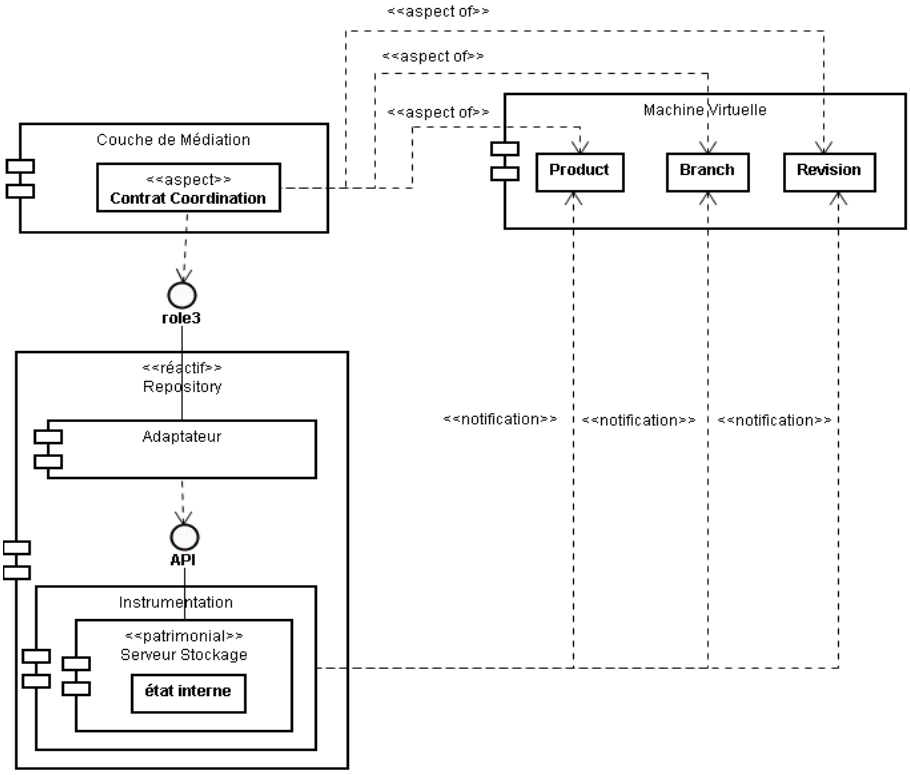

figure 30. Architecture d'exécution du domaine

Lors d'un changement significatif de l'état interne d'un composant, la notification à la couche de coordination se fait par une invocation directe des méthodes du méta-modèle nécessaires pour rendre l'état de l'*exécution abstraite* cohérente avec ce changement. Ceci est illustré par les liens de « notification » dans la figure 30.

Réciproquement, tout changement de l'état de l'*exécution abstraite* peut entraîner éventuellement le déclenchement des contrats de coordination pour synchroniser l'état des composants. Pour invoquer ces contrats, une possibilité est de modifier la machine virtuelle, mais ceci créerait effectivement un couplage très fort entre la machine virtuelle et la couche de composants. Nous voulons éviter ce couplage, car il impliquerait que les évolutions au niveau de la couche de composants auraient un impact sur la machine virtuelle.

Nous faisons donc recours aux techniques de Programmation Orientée Aspect AOP [1][KLM+97] afin d'inverser le sens de cette dépendance : les contrats de coordination font référence aux classes de la machine virtuelle pour spécifier les points de déclenchement, mais la machine virtuelle ignore l'existence des contrats.

Pour programmer les contrats de coordination dans nos expérimentations initiales nous avons utilisé un langage spécialisé développé dans l'équipe par J. Villalobos dans le cadre de ses travaux sur les fédérations de composants [Vill03]. Actuellement, nous utilisons le langage orienté aspects AspectJ [KHH+01], principalement par ses facilités de développement et deboggage.

Pour donner une idée plus concrète de la couche de médiation, nous avons transcrit, dans la figure 31, une partie d'un contrat de coordination du domaine de produit. Il s'agit d'un contrat pour synchroniser un composant `productPool` qui gère un cache mémoire de produits, un composant `productServer` en charge du stockage des produits et la machine virtuelle.

```
import java.util.*;
import fede.product.components.server.*;

public aspect ObjectCache issingleton() {

   List around (ProductType productType):target(productType) &&
        execution(public List ProductType.getProducts())
   {

      List products = productPool.getProducts(productType);
      if (products == null)
      {
         products = productServer.getProducts(productType);
         productPool.associate(productType, products);
      }
      return products;

   }

   private IProductServer productServer = null;
   private ObjectPool productPool = null;

   public ObjectCache() {
      productPool =  (ObjectPool)
         ComponentRegistry.findInstanceByRole("ProductObjectPool");
      productServer = (IProductServer)
         ComponentRegistry.findInstanceByRole("ProductServer");
   }
}
```

figure 31. Contrat de Coordination, en AspectJ

[1] Aspect Oriented Programming

L'événement qui déclenche le contrat est l'appel de la méthode `getProducts` de la classe `ProductType` dans la machine virtuelle du domaine. La logique est directe : s'il trouve la liste de produits dans le cache d'objets il la retourne directement, sinon il délègue l'appel au serveur de produits. Pour simplifier l'exemple, nous avons omis les parties correspondantes aux mécanismes d'invalidation et éviction du cache.

Cet exemple nous permet de faire quelques remarques sur la couche de médiation. En premier lieu, nous pouvons observer que la responsabilité de la médiation est repartie entre le contrat de coordination et les composants eux-mêmes. A partir de notre expérience, nous conseillons de maintenir les contrats de coordination relativement directs : nous les envisageons plutôt comme un mécanisme simple d'orchestration des services de base fournis par les composants.

Un deuxième point à noter est que les interfaces de composants sont définies en termes des classes de la machine virtuelle. Nous aurions pu appeler directement un API de bas niveau à partir du contrat de coordination, mais nous préférons introduire des composants adaptateurs explicitement. Ceci permet de garder inchangés les contrats en cas de substitution d'une implémentation d'un composant par une autre, et contribue à rendre plus explicite le rôle d'un composant en termes d'*exécution abstraite*.

Finalement nous voudrions faire un commentaire sur l'usage des techniques de programmation orientée aspect. Il nous semble que le déclenchement des contrats de coordination est un bon exemple d'un besoin transversal, qui peut impacter plusieurs classes de la machine virtuelle, et dans ce cas l'usage des aspects nous paraît donc approprié. Néanmoins, avec la puissance des mécanismes d'interception d'AspectJ, il est possible de modifier par inadvertance la sémantique de l'exécution de la machine virtuelle. Nous essayons d'utiliser des points de jonction simples pour identifier les événements qui déclenchent les aspects, et de garder une séparation claire des responsabilités pour diminuer ce risque.

En conclusion, la couche de médiation est implémentée par les aspects de coordination et les adaptateurs. Ils s'articulent autour de la machine virtuelle, mais cette dernière reste inaltérée. De cette façon, le méta-modèle du domaine est isolé des évolutions au niveau de la couche de composants (sauf bien sur si celles-ci induisent une modification de la conceptualisation du domaine.)

L'idée d'interpréter un modèle abstrait simultanément avec le système réel a été déjà proposée par Egyed [Egy04], mais dans ses travaux c'est le système réel qui pilote l'interpréteur, afin de pouvoir comparer le comportement effectif du système avec son comportement attendu.

Notre architecture est similaire à l'idée d'*Implémentation Ouverte* [Kic96] [ML+97] : un module logiciel qui expose de façon maîtrisée une partie de son implémentation interne au contrôle externe. Cette approche, appliquée aux langages de programmation, a donné lieu au concept de *Langage à Conception Ouverte* [Ste94] : un système qui propose des mécanismes pour réifier les aspects absorbés par l'implémentation du langage en vue de les modifier de façon contrôlée. Nous considérons donc le langage de modélisation du domaine comme un langage à conception ouverte.

3.5 VARIATIONS DE LA REALISATION.

La section précédente nous laisse entrevoir le fait qu'il existe un énorme espace de variation au niveau des couches de composants et de médiation : nous pouvons vouloir ajouter de nouveaux composants pour tenir compte de nouvelles fonctionnalités, certains composants sont optionnels, d'autres composants peuvent avoir plusieurs implémentations différentes, les contrats de coordination doivent évoluer en fonction des composants présents, etc.

Nous pouvons aussi remarquer que beaucoup de ces variantes sont indépendantes de la conceptualisation du domaine, dans le sens où elles ne modifient pas les modèles existants, ni leur sémantique opérationnelle. Cependant, la maîtrise de cette variabilité est capitale, car elle détermine en définitive l'adéquation du système à l'environnement d'exploitation final. Nous pouvons alors penser que pour une conceptualisation donnée du domaine il peut exister diverses réalisations, qui tiennent compte des propriétés non fonctionnelles de l'environnement d'exploitation.

Dans cette section nous nous intéressons donc aux mécanismes qui vont nous permettre de structurer les artefacts qui constituent la réalisation concrète du domaine. Nous proposons [EV05] d'utiliser deux mécanismes complémentaires : la substitution des composants et les options de réalisation. Nous pressentons également l'outil Mélusine, développé pour aider à la gestion de la variabilité.

3.5.1 Substitution de composants.

L'architecture d'exécution concrète du domaine est une des variantes d'une fédération de composants. Comme dans toute architecture à base de composants, le mécanisme privilégié de gestion de la variabilité est la substitution d'un composant par un autre. Evidement, les composants doivent implémenter la même interface commune pour pouvoir jouer le même rôle dans la composition.

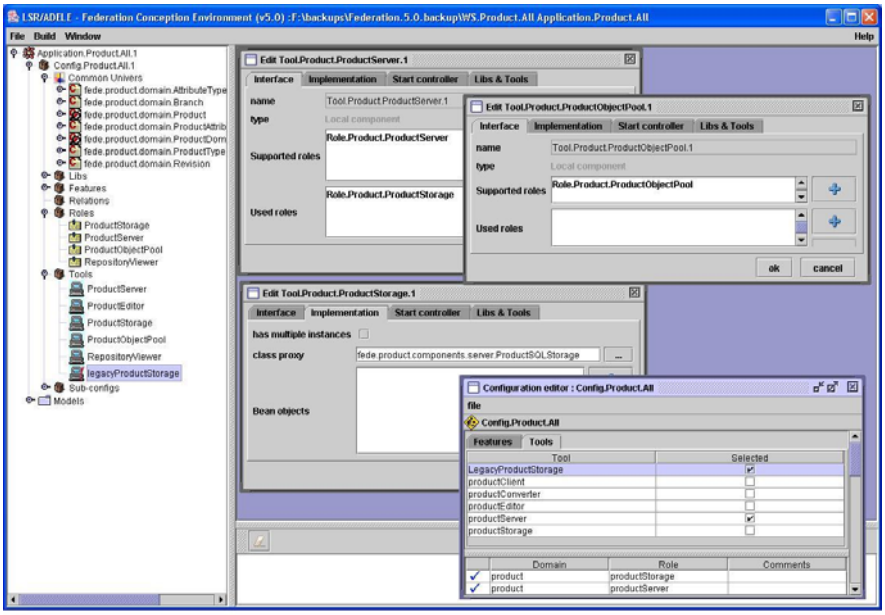

figure 32. Environnement Mélusine : gestion de composants

Nous avons développé dans l'équipe un environnement, Mélusine, pour supporter le concepteur d'un domaine. La figure 32 illustre la partie concernant la définition des composants. On peut observer la définition de plusieurs composants, notamment les composants `ObjectPool`, `ProductServer` et `ProductStorage`. La définition reprend les concepts classiques des modèles à composants : interfaces fournies, interfaces requises et implémentation.

Nous pouvons aussi observer l'outil de configuration qui permet de choisir les composants qui seront déployés : il suffit de sélectionner les composants à déployer parmi ceux disponibles. Evidement, s'il existe des interfaces requises, l'outil demandera de compléter la composition.

Nous ne nous attarderons pas sur cette partie, nous voulons simplement souligner que l'environnement se charge aussi de la gestion du cycle de vie des composants à l'exécution, en particulier des problèmes liés aux composants distribués. Cette partie de l'environnement est une continuation des travaux préalables dans l'équipe sur les fédérations de composants.

3.5.2 Les options de réalisation.

L'analyse du domaine dans les approches de lignes de produits logiciels introduit la notion de feature afin de structurer la variabilité à l'intérieur d'une famille de produits : tout artefact développé doit contribuer à implémenter un des features identifiés.

Dans le cadre global de notre démarche, nous avons préféré une vision de séparation de préoccupations centrée sur le concept du domaine, au lieu d'une vision centrée sur les features. Néanmoins, lors du développement de la couche de médiation d'un domaine particulier, il nous semble que l'utilisation des features optionnels peut être très avantageuse.

Nous proposons donc d'utiliser le feature comme unité modulaire de structuration de la variabilité et de réalisation du domaine : les contrats de coordination seront groupés en fonction du feature auquel ils contribuent, les composants et adaptateurs déployés seront déterminés en fonction des options choisis.

Cependant, il faut insister sur le fait que nous ne cherchons pas à modéliser, avec les options, tous les concepts du domaine, seulement les variantes au niveau des couches d'implémentation. Nous pensons donc que nous n'avons pas besoin de toute la généralité des modèles de features des lignes de produits logiciels [RSP03].

Nous proposons donc d'utiliser un modèle d'options relativement simple : nous ne gérons pas une hiérarchie des options, mais un seul niveau d'options. Les features sont optionnels : chaque feature se comporte comme une unité autonome, qui peut être déployée indépendamment.

La figure 33 montre la partie de l'environnement Mélusine concernant les options. Nous pouvons observer à gauche la définition d'une option comme un ensemble de contrats de coordination. Nous observons aussi l'outil de configuration qui permet la sélection des options qui seront déployés. Quand on choisit une option, l'outil analyse les contrats de coordination et détermine les interfaces requises par le contrat, et

en fonction de cette information il demande de choisir un composant pour jouer le rôle respectif.

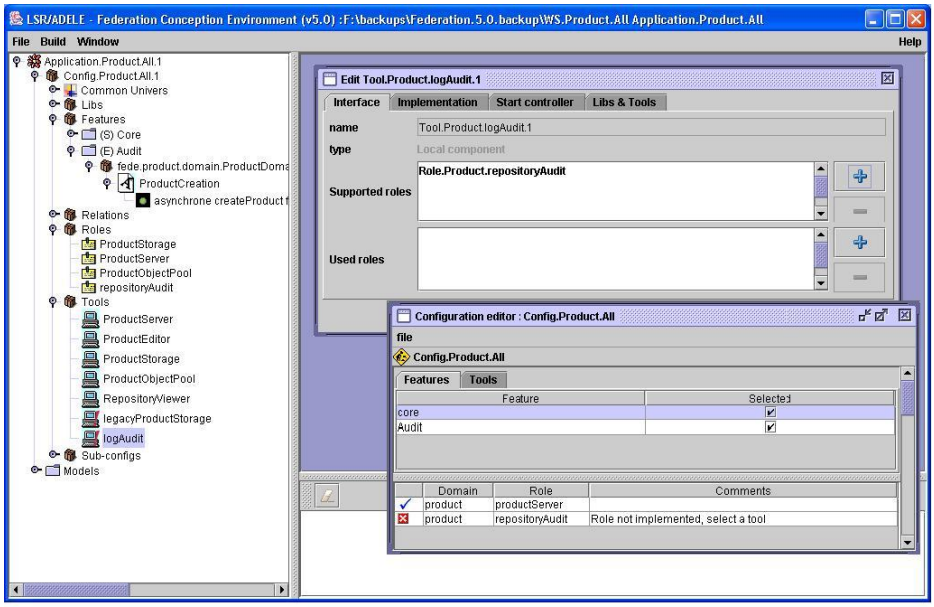

figure 33. Environnement Mélusine : gestion d'options

Pour gérer des interactions entre options, il est possible de définir des relations entre options. Actuellement nous gérons deux types de relation : dépendance et conflit. L'outil de configuration est chargé de valider que l'ensemble de options sélectionné vérifie les contraintes spécifiées.

3.6 LA NOTION D'APPLICATION.

Tout au long de ce chapitre nous nous sommes intéressés aux problèmes liés à l'exécution d'un modèle. Nous les avons abordés selon deux perspectives différentes : le point de vue de la modélisation et le point de vue de la plate-forme d'exécution. Si nous considérons les applications qui font partie du domaine, nous avons donc deux dimensions indépendantes de variabilité: variabilité conceptuelle et variabilité de réalisation.

Quand nous voulons exécuter une application concrète, nous devons fixer un point sur chacune de ces deux dimensions. Dans le cas de la variabilité conceptuelle, ceci correspond à produire un modèle de l'application exprimé dans le langage de modélisation spécifique du domaine. Dans le cas de la variabilité de la réalisation, ceci correspond à définir une configuration cohérente d'options et composants à déployer.

3.7 CONCLUSIONS.

Dans l'introduction de ce mémoire, nous avons défini, très vaguement, un domaine comme un champ d'expertise consacré à un point de vue spécialisé. Notre prémisse est qu'un domaine peut également devenir une unité logiciel de structuration et de réutilisation : il permet à la fois de séparer les diverses préoccupations d'une application, et de fournir un contexte pour développer des composants réutilisables.

Ceci n'est possible qu'à condition de pouvoir formaliser les connaissances et le savoir-faire développé au sein du domaine. Dans une approche centrée sur les modèles, nous pensons que l'élément fondamental de cette formalisation est le méta-modèle d'un langage de modélisation dédié au domaine, un DSL.

Dans ce chapitre nous avons concrétisé cette proposition. Nous avons présenté les diverses étapes dans la formalisation du domaine, et nous avons proposé un nombre d'artefacts concrets à produire, ainsi qu'une architecture pour les articuler.

Le premier artefact produit est le méta-modèle, orienté objet, des concepts statiques du langage. Ce méta-modèle est par la suite étendu pour produire une machine virtuelle du domaine. Celle-ci constitue la couche supérieure de l'architecture d'exécution. La machine virtuelle spécifie la sémantique opérationnelle du langage, et permet l'*exécution abstraite* des modèles.

Une deuxième couche de l'architecture s'intéresse à la plate-forme technologique d'exécution. Elle comporte des composants réutilisables développés pour le domaine, ainsi que l'interfaçage des composants tiers et systèmes patrimoniaux. Cette deuxième couche assure l'*exécution concrète* des modèles.

Ces deux couches sont reliées par une *couche de médiation*, qui assure la synchronisation entre l'exécution de la machine virtuelle et l'exécution des composants du système. Cette couche s'appuie sur des travaux développés précédemment dans notre équipe de recherche autour de la composition et synchronisation de composants sur étagère, à l'aide de techniques de programmation orienté aspects et architectures à services.

Le fils conducteur de notre démarche est le lien étroit entre la conceptualisation du domaine et le méta-modèle du langage. C'est précisément ce fils qui nous permet d'assurer une cohérence entre tous les artefacts développés. Nous pensons aussi qu'un grand nombre des bénéfices imputés aux langages dédiés découlent de l'existence d'un homomorphisme entre la conceptualisation du domaine et la sémantique du langage.

Cette propriété peut être constatée, à posteriori, dans beaucoup de DSLs réussis existants. Néanmoins, elle est rarement considérée comme un critère de conception des langages de modélisation. Or, si nous voulons centrer une démarche de génie logiciel autour des langages dédiés de modélisation, nous devons forcement nous intéresser aux caractéristiques qui détermineront leur succès et leur efficacité. La réflexion que nous avons menée tout au long de ce chapitre, sur les points cruciaux à considérer dans les diverses étapes du développement du langage, est une contribution dans ce sens.

Nous voulons finalement préciser que nous ne prétendons pas proposer dans ce chapitre une méthodologie complète pour le développement d'un DSL ; nous ne pensons pas avoir suffisamment de retour d'expérience pour entreprendre une telle tâche. Nous avons voulu simplement présenter notre proposition d'une façon systématique et, surtout, retracer notre état d'esprit et l'état de nos réflexions face aux multiples questions connexes à la définition d'un langage exécutable de modélisation dédié.

4. COMPOSITION DE DOMAINES.

Notre démarche d'Ingénierie Dirigée par les Modèles s'appui d'une part sur une description précise de différents domaines impliqués dans une application, et d'autre part sur la possibilité de composer ces domaines, afin de les réutiliser pour définir de nouveaux domaines plus complexes.

Dans le chapitre précédent nous avons détaillé la formalisation d'un domaine au moyen d'un langage dédié de modélisation. Le méta-modèle de ce langage nous sert de pivot pour structurer les éléments du domaine en trois niveaux : la machine virtuelle chargée de l'*exécution abstraite* des modèles, une couche de composants pour assurer l'*exécution concrète*, et une *couche de médiation* pour réaliser la synchronisation entre les deux niveaux précédents.

Dans ce chapitre nous nous concentrons sur le problème de la composition de domaines. L'objectif est de définir de nouveaux domaines, plus complexes, à partir de domaines préalablement validés et testés. La composition de domaines permet donc d'améliorer la modularité et d'obtenir de taux de réutilisation plus importants.

4.1 INTEGRATION DES POINTS DE VUE.

Dans notre démarche d'analyse du domaine, voir §3.1.1, nous avons proposé d'élargir la perspective de réutilisation, nous détacher de la vision des lignes de produits logiciels (guidée par les features d'une gamme de produits) et nous orienter vers l'identification d'un ensemble de domaines de base, avec une portée de réutilisation plus étendue.

Chacun de ces domaines correspond à un point de vue particulier qui, dans notre approche, est formalisé par le méta-modèle d'un langage de modélisation dédié au domaine. Toutefois, si nous voulons garder les bénéfices des langages de modélisation dédiés, éprouvés dans les lignes des produits logiciels, ces domaines doivent couvrir une fonctionnalité réduite et bien cernée.

De ce fait, une application sera décrite par un nombre de modèles différents, correspondants aux différentes facettes de l'application, déterminées par les points de vue des domaines impliqués. Clairement ces modèles ne sont pas complètement indépendants : il faut pouvoir spécifier comment les concepts des divers domaines sont

intégrés, dans certains cas il faut introduire de nouveaux concepts pour décrire leur façon d'interagir et éventuellement il faut aussi établir des contraintes de cohérence entre les modèles décrivant l'application.

Conséquemment, l'objectif de la composition des domaines est double : en tout premier lieu, il s'agit de spécifier conceptuellement l'intégration des domaines et, ensuite, de pouvoir regarder les domaines comme des composants réutilisables pour développer des domaines plus complexes.

La proposition que nous avançons est que dans notre approche, grâce à la façon dont nous avons structuré les domaines, il est possible d'atteindre ensemble ces deux objectifs en nous concentrant sur la définition de la composition conceptuelle.

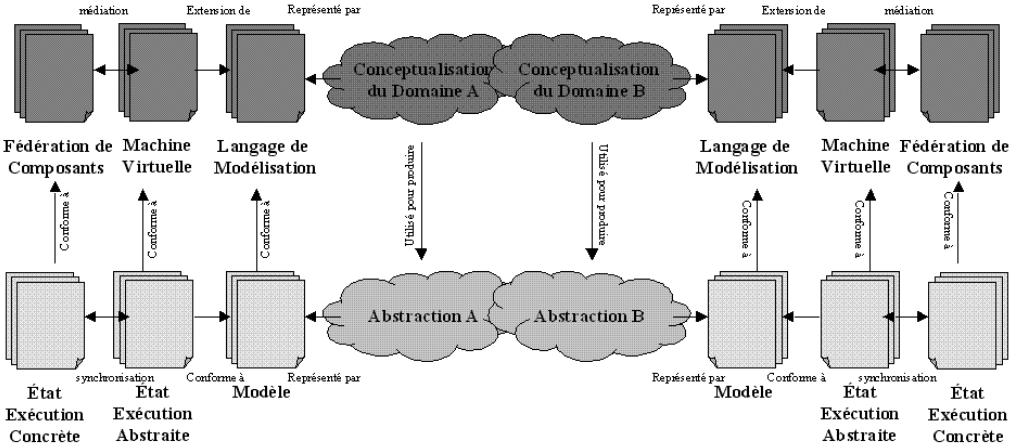

figure 34. Composition conceptuelle

Les idées principales de notre proposition sont mises en évidence quand nous considérons la structure des domaines à composer, tel que l'illustre la figure 34 :

- Notre prémisse est qu'il est plus simple d'exprimer la composition au niveau conceptuel, qu'au niveau de l'infrastructure de composants ;
- Dans notre architecture, ces deux niveaux sont nettement séparés, et la couche de médiation assure leur synchronisation. Ceci nous permet donc de nous affranchir de l'exécution concrète pour nous concentrer, et raisonner en termes de l'exécution abstraite seulement;
- Dans notre approche, il existe une relation étroite entre la conceptualisation du domaine et les concepts matérialisés dans le langage de modélisation et la machine virtuelle. Il est donc possible de raisonner sur la composition en termes de relations entre les concepts réifiés par les méta-modèles des sous domaines ;
- Nous pouvons regarder la composition comme un nouveau domaine, dans lequel la conceptualisation est faite en termes des concepts des domaines de base ;
- Si nous réussissons à spécifier la sémantique de l'exécution du domaine composite en termes de l'exécution des domaines de base, alors indirectement, par l'intermédiaire des couches de médiation, nous allons pouvoir réutiliser et coordonner les composants des domaines impliqués.

Nous proposons donc d'envisager l'intégration des points de vue comme la construction d'un nouveau domaine, développé à partir des concepts matérialisés dans les domaines existants. Afin de concrétiser et illustrer cette proposition nous allons utiliser tout au long du chapitre un scénario de composition entre domaines que nous décrivons ci-dessous.

4.2 UN SCENARIO DE COMPOSITION.

Nous allons reprendre l'exemple de la gestion de configurations introduit dans le chapitre précédent. Le domaine de Produit que nous avons exposé ne s'intéresse qu'à une facette des fonctionnalités globales d'un gestionnaire de configurations : la gestion d'un référentiel des versions. Si nous voulons retrouver la totalité des fonctionnalités envisagées initialement, il nous faut donc composer ce domaine avec d'autres domaines proposant les fonctionnalités manquantes.

Dans notre équipe, nous avons proposé de séparer les fonctionnalités globales des gestionnaires de configuration en plusieurs domaines réutilisables de base : la gestion de produits que nous avons montrée, le stockage de documents, la gestion de workspace et la gestion de procédés [ELV03].

Dans le scénario que nous allons développer, nous nous concentrons sur une des fonctionnalités globales de la composition : l'automatisation de la gestion de versions. En fait, le domaine de produit propose des fonctionnalités pour la création de branches et révisions d'un produit, mais la gestion reste manuelle et à charge de l'utilisateur final. Nous voulons piloter la création des révisions automatiquement par la définition d'un procédé.

Domaine de Gestion de Procédés.

Un procédé peut être défini comme étant une suite d'étapes réalisées dans un but donné. Les procédés ont donc un caractère très général, et ont été mis à profit dans des nombreuses disciplines. En particulier, ils ont été utilisés pour automatiser, partiellement ou totalement, les activités du travail pour améliorer l'efficacité.

La gestion de configurations, se chargeant du contrôle de l'évolution de données complexes, est fortement liée au procédé contrôlant le cycle de vie de ces données au cours de leur création et leur transformation. Un aperçu de cette interaction sera développé dans le scénario de composition utilisé pour illustrer ce chapitre.

La conceptualisation du domaine de gestion de procédés développée dans notre équipe [EVL+03] contient un noyau minimal de concepts, formant les éléments de base pour la définition de tout procédé. Les concepts qui nous intéressent pour notre exposé sont : les activités, les données, les ports et les flots de données.

Une activité est une étape du procédé au cours de laquelle une action est exécutée. Une donnée est produite, transformée et consommée par les activités. Les ports sont les interfaces de l'activité, ils définissent et contrôlent la communication entre l'activité et le monde extérieur. Les flots de données décrivent la façon dont les activités s'enchaînent, et s'échangent des données.

Un modèle de procédé décrit donc explicitement les différentes étapes du cycle de vie des données. Un exemple d'un procédé simple de gestion documentaire va nous

permettre d'illustrer les principaux concepts du domaine. Il est présenté dans la figure 35, ainsi qu'un aperçu de l'environnement de modélisation dédié développé pour le domaine.

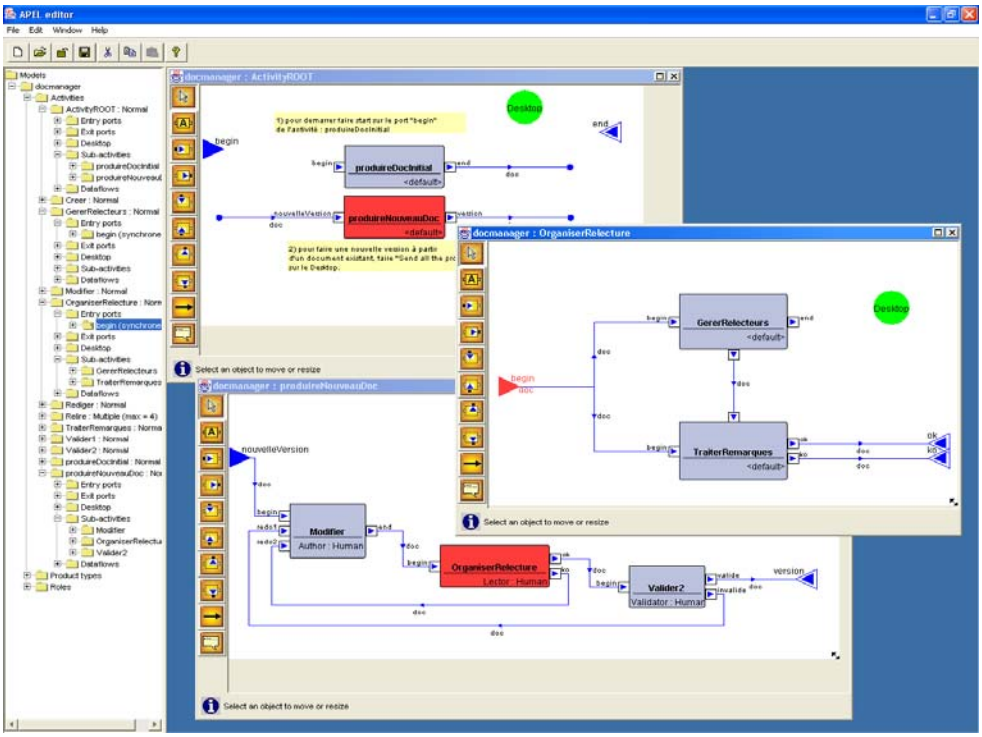

figure 35. Modèle de Procédé

Dans l'exemple nous observons tout d'abord la décomposition en activités : il existe une activité initiale, appelée `ActivityRoot`, qui représente l'ensemble du procédé. Les activités peuvent à leur tour se décomposer en sous-activités. Par exemple, l'activité `OrganiserRelecture` est décomposée en deux sous-activités : `GérerRelecteurs` et `TraiterRemarques`.

Nous observons aussi l'enchaînement entre activités. Certaines activités se suivent séquentiellement, c'est le cas des activités `Modifier` et `Organiser Relecture`. En revanche, certaines activités peuvent se dérouler en parallèle : par exemple, les activités `GérerRelecteurs` et `TraiterRemarques`. Il est aussi possible qu'à la fin d'une activité il soit nécessaire de revenir à une activité précédemment exécutée : par exemple, à la fin de l'activité `Organiser Relecture`, si les remarques n'ont pas pu être conciliées, il faut revenir à l'activité `Modifier`.

L'enchaînement entre activités est défini par l'intermédiaire de flots de données entre les activités. Un flot de données peut relier un port de sortie d'une activité à un port d'entrée d'une autre activité.

Une activité peut comporter plusieurs ports d'entrée et de sortie. Chaque port a un ensemble de données attendues, chaque donnée attendue est représentée par une variable, `doc` dans l'exemple. Pour pouvoir démarrer une activité par un port d'entrée donné, il faut que le port contienne toutes les donnés attendues. Si une activité comporte

plusieurs ports de sortie, il faut choisir un port spécifique pour la terminer. Le port de terminaison doit contenir toutes les données attendues. Un port peut être déclaré *automatique* : dans ce cas, de que le port contiendra toutes les données attendues, et s'il s'agit d'un port d'entrée, l'activité sera démarrée automatiquement; s'il s'agit d'un port de sortie l'activité sera terminée automatiquement.

L'échange de données sur les ports d'entrée/sortie est synchrone : à la fin de l'activité les données qui se trouvent dans le port de terminaison sont transférées vers tous les ports reliés par un flot de données. Si un port de sortie est relié par plusieurs flots de donnés différents, au moment du transfert, une copie des données est faite avant de les envoyer aux ports de destination.

Le domaine permet aussi l'échange asynchrone de données entre activités parallèles, tels que l'illustre le lien vertical entre les activités `GererRelecture` et `TraiterRemarques`. Néanmoins, nous n'allons pas détailler cette fonctionnalité.

Toute activité comporte un port spécial nommé `Desktop`. Il représente les données en cours de modification ou transformation par l'activité. Un `desktop` ne peut être relié par un flot de données qu'aux ports synchrones d'entrée ou sortie de la même activité.

Tous ces concepts ont été matérialisés, et leur sémantique d'exécution spécifiée par le méta-modèle du domaine de gestion de procédés présenté dans la figure 36.

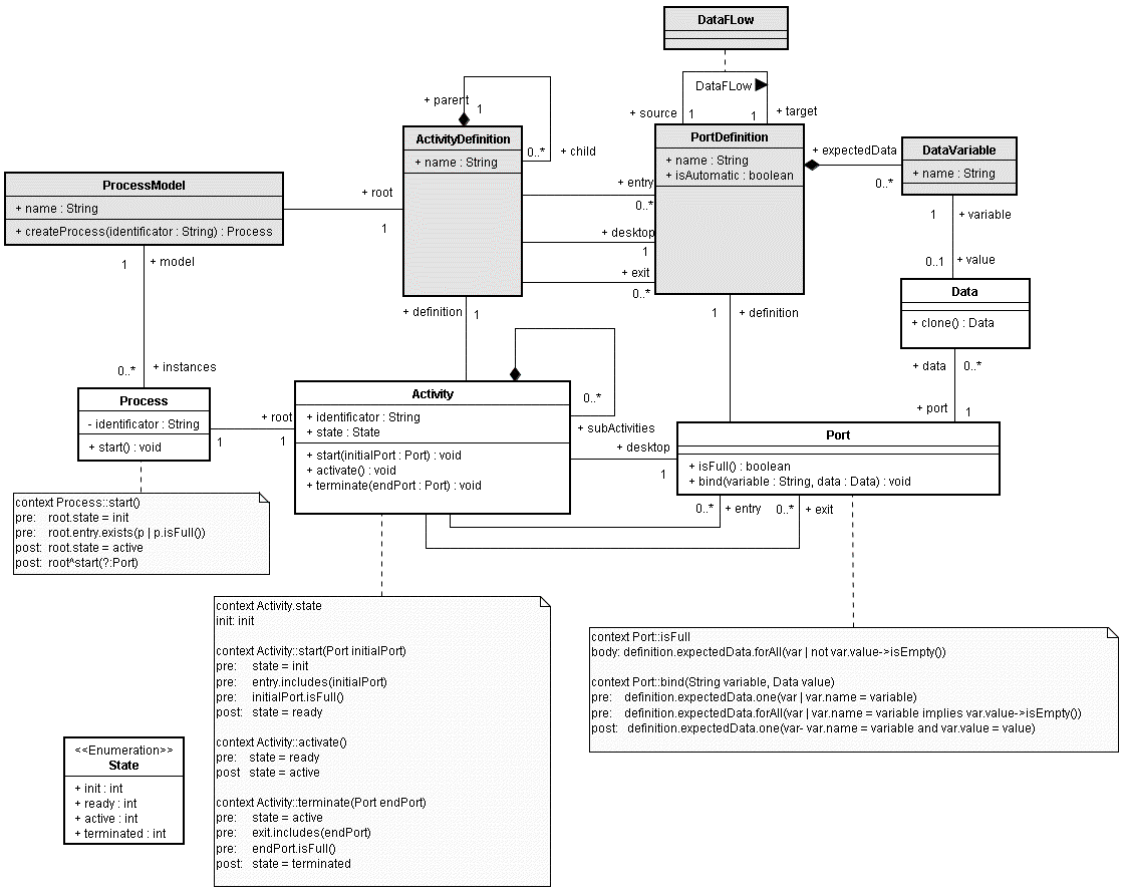

figure 36. Modèle du Domaine de Gestion de Procédé

Nous n'allons pas rentrer dans le détail de ce méta-modèle. Nous pensons que la description que nous avons faite, à l'aide de l'exemple de la figure 35, est suffisante pour comprendre le scénario de composition que nous allons dérouler. Néanmoins, nous voulons faire quelques remarques sur la structure du méta-modèle.

En tout premier lieu nous pouvons observer que ce méta-modèle a la même structure que celui du domaine de Produit présenté dans la figure 25 :

- Les classes qui représentent les concepts utilisés pour exprimer la syntaxe abstraite statique du langage sont séparées des concepts qui capturent l'état dynamique de l'exécution ;
- Des liens de conformité raccordent ces deux parties ;
- Certaines des méthodes de la partie dynamique font référence à la réification du modèle pour assurer la conformité de l'exécution ;
- Finalement, certains concepts du domaine sont absorbés par le langage et ne sont pas visibles lors de la modélisation. C'est le cas par exemple de la notion d'*état d'une activité* qui n'apparaît pas explicitement lors de la modélisation d'un procédé, mais qui est vitale pour comprendre l'exécution du modèle.

Ensuite, nous voulons, à nouveau, faire remarquer la relation étroite et directe entre le méta-modèle et les concepts du domaine. Le méta-modèle nous permet de raisonner au même niveau d'abstraction que la conceptualisation du domaine. Ceci nous paraît une caractéristique essentielle pour réussir la composition.

Fonctionnalités du domaine composite.

Comme nous l'avons déjà mentionné, il existe une forte relation entre la gestion de configurations et la gestion de procédés, et une analyse de toutes leurs interactions possibles demanderait de la part du lecteur une grande expertise dans ces domaines. Nous allons donc développer seulement deux cas d'utilisation simples qui nous permettront d'illustrer notre démarche de composition de domaines.

Le premier cas d'utilisation concerne la notion d'un historique de versions livrables d'un produit. Ce cas fait intervenir d'un côté le domaine de gestion de procédés, car la spécification des activités qui permettent de faire évoluer un produit entre deux versions livrables consécutives peut être capturée aisément par un modèle de procédés. D'un autre côté, il fait intervenir le domaine de Produit pour le stockage et la gestion des versions. Nous voulons donc ajouter dans le domaine composite la fonctionnalité de gestion d'un historique de versions.

Ce cas d'utilisation est intéressant car il illustre l'apparition d'un nouveau concept, « version livrable », dans le domaine composite qui n'existe pas directement dans les domaines que nous voulons réutiliser, même s'il s'agit d'une spécialisation du concept de révision déjà existant dans le domaine de produit.

Le deuxième cas d'utilisation concerne la gestion et stockage de versions intermédiaires d'une donnée lors qu'elle est transformée par les activités d'un procédé. En fait, si l'exécution d'un procédé implique des acteurs humains, il peut s'écouler un temps important entre la fin d'une activité et le démarrage des activités suivantes, et le

système doit assurer la persistance des données dans cet intervalle. Egalement, lorsque différentes copies d'une donnée sont générées pour la gestion des activités parallèles, le système doit garder une trace de la relation entre ces versions.

Ce cas nous semble intéressant car il illustre une situation que nous avons rencontrée fréquemment : un concept abstrait d'un domaine est concrétisé dans le domaine composite, à l'aide des concepts des autres domaines composés.

Cette situation est fréquente parce que, dans l'esprit de la séparation des préoccupations, le concepteur d'un domaine se concentre sur une facette de la fonctionnalité, et il fait abstraction des détails qu'il ne considère pas centraux à cette préoccupation. Mais c'est précisément lors de l'intégration des différents points de vue dans une composition que ces concepts doivent être concrétisés, et les relations entre les diverses abstractions établies.

Dans notre exemple, le domaine de gestion de procédés, tel qu'il a été conçu, se concentre sur les aspects liés aux activités du procédé et leurs interactions. Le concept de « donnée » est donc très abstrait : il se limite à l'idée d'un jeton qui circule entre les diverses activités, et qui peut être dupliqué en cas de besoin. Nous voulons donc concrétiser ce concept abstrait de « donnée » du domaine de Gestion de Procédés, en l'associant dans la composition au concept de révision du domaine de Produit.

4.3 COMPOSITION CONCEPTUELLE.

Le premier pas dans notre démarche est la conceptualisation du domaine composite. Notre proposition consiste à le faire de la même façon que pour le cas d'un domaine simple. C'est à dire, nous allons matérialiser les concepts du domaine par des classes d'un méta-modèle qui décrit un langage de modélisation et la sémantique opérationnelle de son exécution.

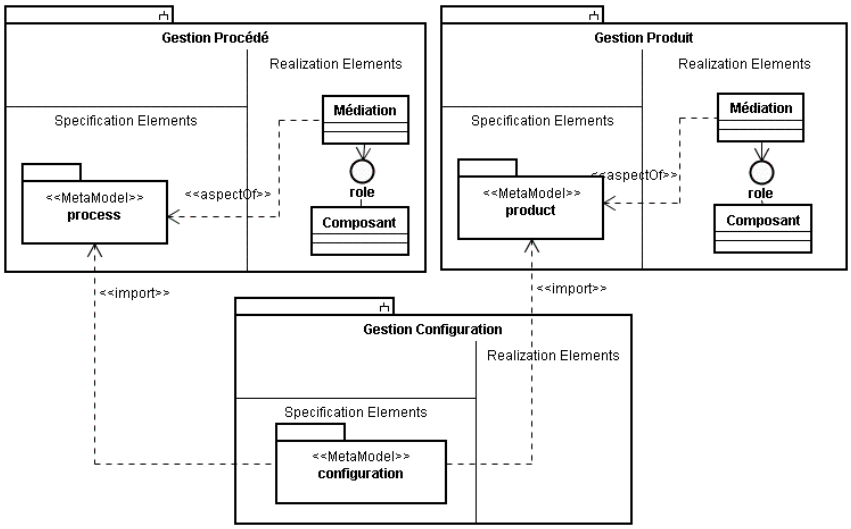

figure 37. Démarche globale de composition

Cependant, nous voulons privilégier la réutilisation des domaines existants. Le méta-modèle du domaine composite est donc développé comme une extension des méta-modèles des domaines de base. L'approche globale est illustrée, pour notre scénario de composition, dans la figure 37.

Avant d'engager la définition du méta-modèle composite, nous avons un certain nombre de choix à faire en vue d'établir le périmètre fonctionnel du domaine composite. Comme c'est le cas pour tout domaine dans notre approche, nous faisons appel à une analyse des caractéristiques communes et variabilités afin de délimiter la portée du domaine, et de décider quels seront les concepts absorbés et visibles dans le langage de modélisation.

Nous n'allons pas présenter l'analyse complète du domaine de gestion de procédés, mais nous énumérons ci-dessous les choix que nous avons faits à l'issue de cette analyse concernant les cas d'utilisation qui nous intéressent:

- Les données d'un procédé sont concrétisées en les associant aux révisions d'un produit du domaine de Produit. Cette association se fera automatiquement, et fera partie des concepts absorbés par le langage ;
- L'historique de versions livrables d'un produit est une séquence temporellement ordonnée de révisions. Les versions historiques seront nommées par un identificateur de la forme « `majeur.mineur` », chaque partie de l'identificateur est une valeur entière. Entre deux versions consécutives de l'historique, le `majeur` ou le `mineur` doit être incrémenté d'une unité. Ce comportement fera partie des concepts absorbés par le langage ;
- Lors de l'association d'une donnée du procédé à une révision de produit, il faut permettre à l'utilisateur final de choisir soit de créer une nouvelle révision avec les valeurs par défaut des attributs, soit de les initialiser à partir des valeurs d'une version historique existante ;
- La création d'une version historique se fera automatiquement à la terminaison d'une activité. Le concepteur de l'application doit pouvoir spécifier les activités qui vont déclencher la création d'une nouvelle version historique. Il doit être aussi possible de spécifier s'il s'agit d'une nouvelle version mineure ou majeure.

Nous pouvons faire plusieurs observations importantes sur cet exemple. En premier lieu, on peut remarquer que la description du domaine composite fait référence aux concepts des domaines existants, et s'appuie sur ces concepts pour pouvoir définir les nouveaux concepts propres à la composition.

Ensuite, nous voulons insister sur le fait que la composition est définie au niveau conceptuel. Nous ne raisonnons pas en termes des composants de la plate-forme de chaque domaine, mais en termes des concepts réifiés dans les différents méta-modèles. Tel que le suggère la figure 37, nous considérons le méta-modèle de chaque domaine comme la spécification du domaine, et les couches de composants et médiation comme sa réalisation. La composition des domaines est complètement définie au niveau des spécifications.

Enfin, on observe que chacun de ces choix a un impact différent sur le domaine composite : certains concernent le langage de modélisation, d'autres la sémantique de l'exécution du domaine composite, et d'autres l'exécution concrète du domaine. Nous

allons les détailler séparément pour pouvoir faire des remarques plus précises dans chaque catégorie.

4.4 COMPOSITION DES META-MODELES.

Notre démarche repose sur l'idée qu'un domaine est à la fois une unité de séparation des préoccupations et une unité de réutilisation, et que ces deux visions se complémentent. C'est le cas aussi pour les domaines composites. Nous voulons donc qu'un domaine composite ne concerne que les aspects introduits par la composition, et qu'il réutilise au maximum les artefacts produits dans les domaines de base.

Comme nous l'avons déjà avancé, nous proposons de réutiliser les méta-modèles de sous domaines pour définir le méta-modèle de la composition. De surcroît, nous voulons aussi garder la séparation des préoccupations et la réutilisation au niveau des modèles.

Plus précisément, nous voulons que le méta-modèle composite se comporte comme un ensemble des points de vue. Le méta-modèle de chaque domaine de base permet au concepteur de décrire, sous forme d'un modèle, une vue particulière d'une application. Le méta-modèle composite, lui, permet de décrire l'interaction entre ces vues. Nous proposons donc d'utiliser le schéma de composition montre dans la figure 38.

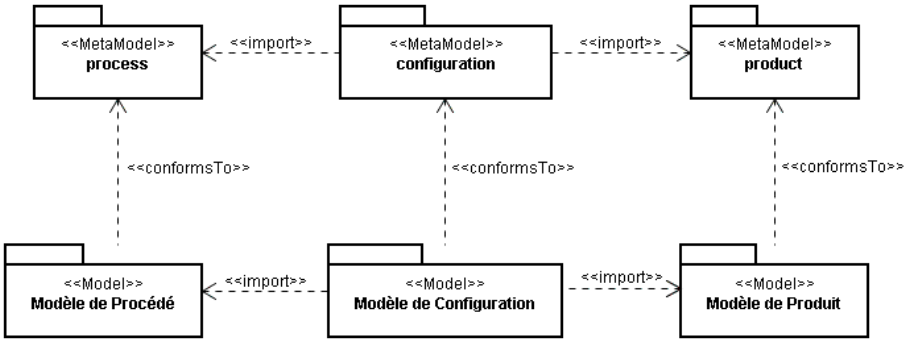

figure 38. Composition de méta-modèles statiques

Sur la figure, le lien de conformité fait référence essentiellement à la conformité syntaxique, car elle détermine en grande mesure la vision du domaine pour le concepteur de modèles. Dans cette section nous nous concentrons donc sur la partie statique du méta-modèle qui spécifie la syntaxe abstraite du langage.

Un point important à souligner est que ce schéma nous permet aussi de réutiliser les environnements de modélisation existants. Le concepteur de l'application utilise l'environnement existant dans chaque domaine pour produire un modèle partiel de l'application, et ensuite il utilise l'environnement du domaine composite pour établir des liens entre ces modèles afin de définir le comportement global.

Le mécanisme principal d'extension dans notre approche est l'établissement de relations entre les concepts matérialisés par les méta-modèles des domaines à composer. Ces nouvelles associations permettent de définir des nouveaux comportements propres au domaine composite, sans modifier les domaines existants.

L'établissement de ces relations n'est pas suffisant, il existe probablement des concepts propres au domaine composite qui ne sont rattachés à aucun concept existant. Ces concepts dits « émergents » sont simplement ajoutés comme des nouvelles classes dans le méta-modèle.

Pour illustrer cette proposition, la figure 39 présente le méta-modèle du domaine composite de notre scénario. Nous avons marqué avec le stéréotype « composition »les classes et les associations ajoutées.

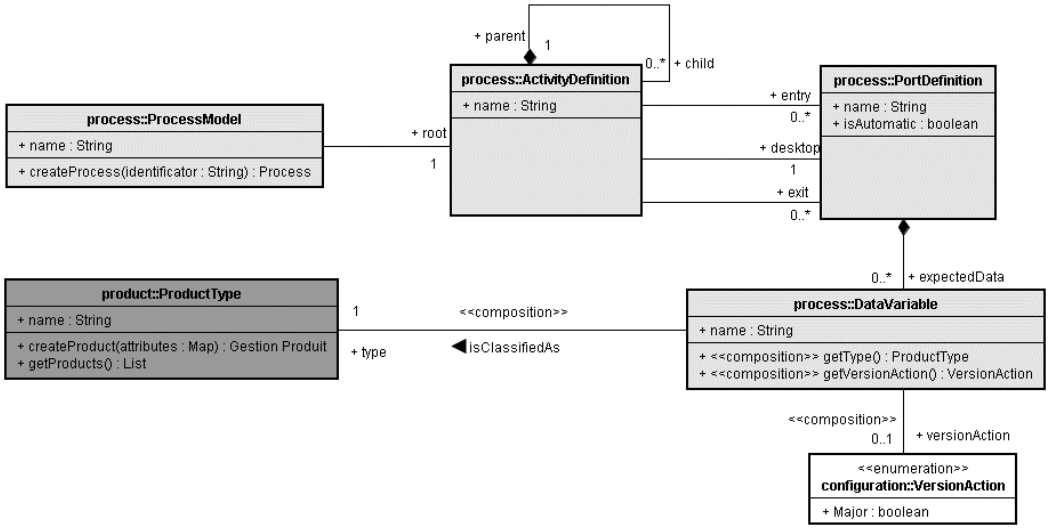

figure 39. Méta-modèle statique de la composition

Nous pouvons observer que pour établir la relation entre un modèle de procédé et un modèle de produit nous avons introduit une association entre les classes `DataVariable` et `ProductType`. De cette façon, il est possible de spécifier dans le modèle composite quel est le type des données manipulées par une activité.

Ceci est un bon exemple de notre idée du domaine comme un point de vue et une unité de séparation des préoccupations : le concepteur du modèle de procédé se concentre sur la définition des activités et leur enchaînement, sans s'occuper de la nature des données échangées ; le concepteur du modèle de produit se concentre sur la définition du modèle de données, sans s'occuper de comment elles vont être manipulées ; et l'intégrateur s'occupe de faire le lien entre ces modèles.

Nous faisons remarquer que bien que les domaines de base soient autonomes et conçus indépendamment, quand ils sont composés dans un nouveau domaine, il existe un recouvrement de concepts similaires. Nous exploitons ces concepts similaires, pour nous servir comme des points d'ancrage pour exprimer les nouvelles relations ajoutées par la composition.

Reprenons maintenant le deuxième cas d'utilisation. Pour gérer la création des versions historiques nous avons introduit une classe dans la composition, `ReleaseVersionning`, qui spécifie le type de version historique qu'il faut créer, majeure ou mineure. Nous avons aussi ajouté une association entre cette classe et la classe `DataVariable`. Cette association spécifie l'action à effectuer lorsque l'activité qui définit cette variable est terminée.

Nous voulons remarquer que le concept de terminaison d'une activité n'existe pas explicitement au niveau du langage du domaine de gestion de procédés, mais il est implicitement associé à la notion de port de sortie. Nous utilisons donc les variables d'un port de sortie pour spécifier indirectement les activités qui vont déclencher la création d'une version historique.

De cette façon, il est possible de spécifier très finement le comportement du versionnement historique : il varie en fonction de l'activité, du port de sortie, voir même de chaque donnée dans le port. Une alternative serait d'associer directement la classe `ReleaseVersionning` à la classe `ActivityDefinition`, mais la spécification du versionnement aurait été moins précise.

Cette observation nous permet d'insister sur le fait que les fonctionnalités ajoutées à la partie statique du méta-modèle sont les seules abstractions visibles aux concepteurs d'applications dans le domaine pour pouvoir spécifier le modèle de la composition. Le choix des abstractions visibles dans le langage est fait lors de l'analyse du domaine, présentée dans la section précédente.

Nous constatons donc qu'il existe toujours dans la conception d'un domaine un compromis entre les concepts visibles dans le langage dédié de modélisation et les concepts absorbés, qui font partie des caractéristiques communes du domaine. Ce compromis détermine le pouvoir d'expression et le niveau de contrôle que peut avoir le concepteur sur l'exécution du modèle.

L'idée de spécifier l'intégration de préoccupations par l'établissant des relations au niveau des méta-modèles a été également avancée dans le cadre des travaux sur le tissage de modèles [BDJ+05] [DBJ+05]. Ces travaux se sont concentrés sur la généralisation de cette idée pour proposer des mécanismes de tissage génériques. Nous pensons que notre démarche peut bénéficier de ces résultats, car ils sont très proches de notre proposition : le modèle de tissage fait référence aux éléments de modèles de base, et ne contient que les liens entre ces éléments.

L'idée d'établir de liens entre modèles existants pour pouvoir les réutiliser, sans le modifier, a été aussi proposée pour la réutilisation de PIMs dans [BG04] [BGB05], mais dans le cadre d'une approche générative.

La composition des méta-modèles a été aussi abordée dans plusieurs travaux dans la communauté de méta-modélisation, car c'est aussi un mécanisme très efficace pour rendre plus modulaire la définition d'un langage. Notre approche est très similaire aux travaux sur les familles des langages [EMS+03] [GS+04], et sur la composition des méta-modèles dans GME [LBM+01] [KML+04].

Ces travaux proposent des mécanismes de composition très sophistiqués pour la définition d'un nouveau méta-modèle à partir des méta-modèles de sous domaines. Il est possible, entre autres, de spécifier : la fusion des classes des sous domaines ; l'héritage des classes de sous domaines ; l'héritage de paquetages de sous domaines ; ou la définition des paquetages paramétrables dans les sous-domaines, qui sont instanciés dans la composition.

Nous avons évalué ces mécanismes, mais ils ne répondent pas à notre besoin de pouvoir réutiliser sans modification les modèles existants, et les environnements de modélisation de sous domaines. Nous ne voyons pas dans la simplicité des mécanismes

de composition que nous utilisons une limitation de notre approche, mais plutôt un compromis : nous privilégions la réutilisation des artefacts de sous domaines sans modification, et ceci nous restreint dans le choix des mécanismes de composition de méta-modèles.

4.5 COMPOSITION DE MODELES.

Le méta-modèle composite devient un nouveau langage de modélisation qui nous permet de spécifier le système sous la nouvelle perspective du domaine composite. Dans l'esprit de notre démarche, les modèles du domaine composite ne comportent que les préoccupations propres à la composition, et qui ne sont pas déjà considérées dans les sous-domaines.

Une application du domaine composite est de la sorte spécifiée par un modèle pour chacun des sous-domaines, plus un modèle composite qui fait référence aux autres modèles et qui établi des liens explicites entre eux. Ces liens sont conformes aux associations définies au niveau du méta-modèle.

Comme dans le cas des domaines de base, l'étape suivante consiste à définir une syntaxe concrète du langage de modélisation, et de fournir aux concepteurs un environnement de modélisation spécifique. La figure 40 présente un aperçu de l'environnement développé pour notre exemple de composition.

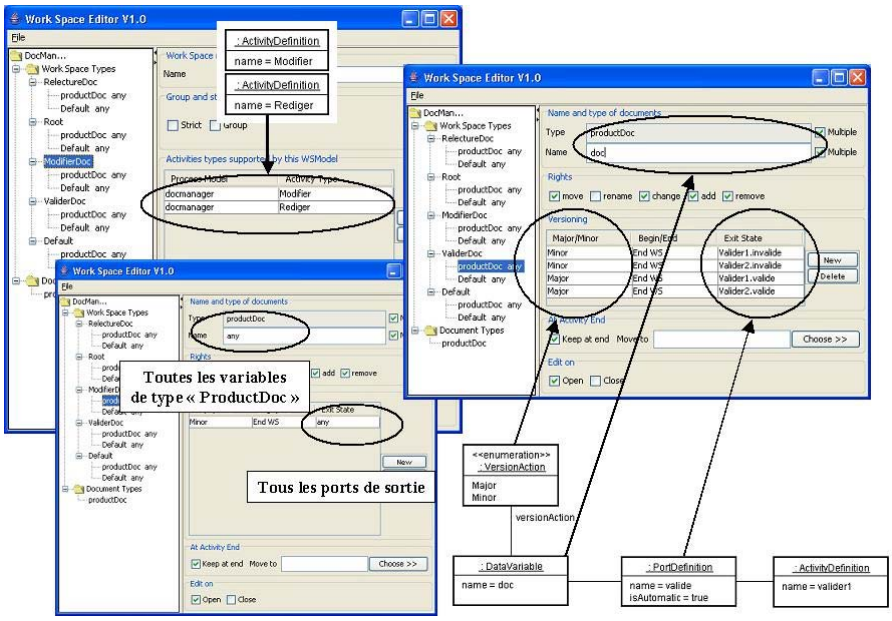

figure 40. Composition de Modèles

Nous pouvons entrevoir, à gauche de la figure, comment l'outil fournit certaines fonctionnalités pour faciliter la création des modèles. Il est par exemple possible de grouper plusieurs activités présentant un comportement similaire, et il est également possible de référencer rapidement toutes les variables ou tous les ports d'une activité.

A droite de la figure, nous pouvons observer comment le modèle composite fait référence aux modèles existants, de procédé et de produit, et permet d'établir les liens pour représenter la composition. Dans l'exemple illustré nous spécifions qu'à la fin de

l'activité `valider1` par le port `valide`, nous allons produire une nouvelle version historique majeure du produit associé à la variable `doc`, de type `ProductDoc`.

A ce moment il faut aussi définir les règles de cohérence qui vont permettre de valider qu'un modèle composite est bien formé. Notons que dans le cas d'un domaine composite nous pouvons imposer de contraintes additionnelles sur les modèles de base.

Par exemple, nous avons associé un type de produit aux variables définies dans les ports d'un procédé, il faut donc faire un contrôle de types pour vérifier qu'il n'existe pas d'erreurs au niveau de la définition des flots de données.

Pour valider ces contraintes et définir la syntaxe concrète du langage de modélisation de la composition, il est aussi possible de faire recours à des frameworks et environnements de méta-modélisation pour développer la syntaxe concrète du langage. Le seul besoin particulier est de pouvoir faire référence dans le modèle composite aux modèles existants.

4.6 COMPOSITION DE MACHINES VIRTUELLES.

Une fois que la composition de la partie statique des méta-modèles est accomplie, la question sur l'exécution des modèles du domaine composite s'impose. Si chaque composition supposait de reprendre, du début, la construction d'une nouvelle machine virtuelle, alors l'intérêt de notre approche serait sérieusement compromis.

Non seulement la réutilisation serait limitée, mais également, par notre façon de composer les modèles, la machine virtuelle du domaine composite devrait forcément ré-implémenter une bonne partie de la sémantique des sous domaines. Ceci conduirait à une duplication importante et, plus gravement, au risque d'avoir des ambiguïtés de la sémantique de sous domaines, en fonction des plusieurs spécifications d'un même domaine dans plusieurs composites.

Nous proposons donc de construire la machine virtuelle du domaine composite en réutilisant les machines virtuelles existantes sans les modifier. L'idée est d'implémenter les nouvelles fonctionnalités par une synchronisation de l'exécution des machines virtuelles des sous-domaines.

4.6.1 Représentation de l'état de l'exécution.

Nous proposons d'utiliser pour la partie dynamique du méta-modèle les mêmes idées que nous avons développées dans les sections précédentes : représenter les concepts émergents par des nouvelles classes ou des nouvelles associations entre les classes existantes des méta-modèles des sous domaines.

Le mécanisme principal de composition est donc, à nouveau, l'établissement de relations entre les concepts matérialisés par les méta-modèles des domaines à composer.

Il peut s'agir des relations de correspondance, qui nous permettent d'identifier des concepts équivalents des sous domaines; dans le sens où ils peuvent être considérés comme de facettes d'un seul concept unifié au niveau de la composition.

Par exemple, nous avons évoqué que le concept abstrait de `Data` du domaine de Gestion de Procédés peut être considéré dans notre scénario de composition comme une

abstraction du concept de `Revision` du domaine de Produit. Nous allons donc matérialiser cette relation de correspondance par une association entre ces deux classes.

Il peut aussi s'agir aussi des nouvelles associations entre concepts existants. Ces nouvelles associations permettent de définir des nouveaux comportements propres au domaine composite.

C'est le cas dans notre scénario de la notion d'historique de versions livrables. Nous pouvons penser qu'il s'agit d'une spécialisation de la notion de branche déjà existante dans le méta-model de Produit : il s'agit d'une séquence de versions, ordonnée temporellement. Au niveau du méta-modèle de la composition, nous allons donc modéliser ce concept par l'ajout d'une nouvelle relation entre les classes `Product` et `Branch`, pour représenter une branche spéciale stockant les versions livrables.

Nous pouvons aussi avoir besoin d'introduire de nouvelles classes pour représenter de nouveaux concepts ou pour spécifier les nouveaux comportements.

Par exemple, au moment d'établir le lien entre une donnée d'un procédé et une révision de produit, nous devons faire une sélection parmi les versions historiques existantes pour initialiser les attributs de la révision. Nous introduisons une classe `ProductSelector` pour assurer cette responsabilité.

Le méta-modèle résultant pour le domaine composite est montré dans la figure 41 ; nous avons marqué avec le stéréotype « composition » les associations et classes ajoutées.

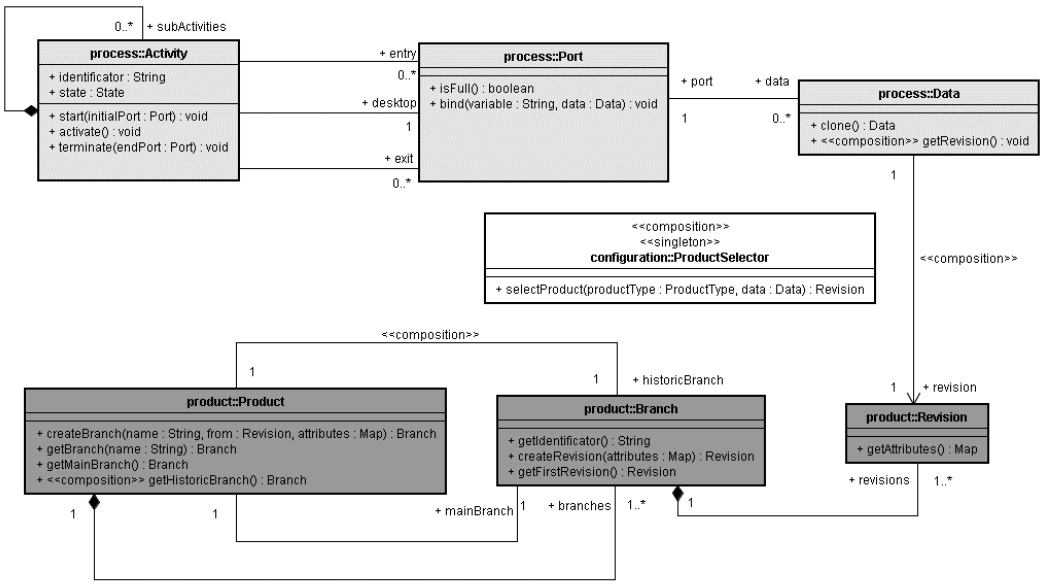

figure 41. Méta-modèle du domaine composite

Ce méta-modèle capture bien la structure de l'information que nous voulons ajouter aux domaines existants. Néanmoins, il nous pose un problème du point de vue de la réutilisation : nous voulons traiter les domaines comme des composants, et pouvoir les étendre et réutiliser sans le modifier.

Nous faisons donc appel à nouveau à des techniques de Programmation Orientée Aspect pour réaliser la composition. En particulier, nous utilisons les facilités du

langage AspectJ pour modifier la structure statique d'un programme et pouvoir ajouter des nouveaux membres aux classes existantes[1].

Nous allons donc construire la machine virtuelle composite comme un ensemble de classes pour représenter les concepts émergents, plus un ensemble d'aspects pour introduire les nouvelles associations aux classes existantes dans les méta-modèles des sous domaines.

La figure 42 présente l'aspect développé pour implémenter les nouvelles associations que nous avons spécifiées dans le méta-modèle du domaine composite. Nous pouvons observer que l'implémentation est très directe, et que nous avons introduit les extensions aux classes statiques et dynamiques du méta-modèle.

```
package configuration;

import product.*;
import process.*;

public aspect Associations {

   ProductType DataVariable.type;
   public ProductType DataVariable.getType() { return type; }

   VersionAction DataVariable.action;
   public VersionAction DataVariable.getVersionAction() {return action;}

   Branch Product.historicBranch;
   public Branch Product.getHistoricBranch() {return historicBranch;}

   Revision Data.revision;
   public Revision Data.getRevision() {return revision;}

}
```

figure 42. Extension de Méta-Modèle, en AspectJ

4.6.2 Dynamique de l'exécution.

Dans la section précédente nous avons spécifié structurellement le méta-modèle du domaine composite, sous la forme d'un ensemble de classes et associations ajoutées aux méta-modèles de sous domaines. Dans cette section nous nous intéressons à la spécification du comportement ajouté par la composition.

La première observation que nous voulons faire est qu'à différence du cas d'un domaine simple la responsabilité du comportement ajouté ne peut pas être affectée à une classe bien précise. Les collaborations que nous voulons spécifier impliquent souvent plusieurs objets des différents domaines, et nécessitent dans certains cas d'introduire des modifications aux comportements existants.

Prenons par exemple le cas de la création d'une nouvelle version historique à la fin d'une activité. Tout d'abord nous déclenchons cette interaction par la méthode `terminate` de la classe `Activity`. Ensuite, nous devons examiner la spécification du modèle composite pour déterminer le comportement à suivre pour chacune de `DataVariables` définies dans le port. S'il faut créer une nouvelle version historique,

[1] Il s'agit du concept de "Inter-Type Declaration" en AspectJ

il nous faut traverser l'association établie entre un `Data` dans le domaine de Procédé et une `Revision` dans le domaine de produit. Une fois le produit identifié il faut créer une nouvelle révision dans la branche correspondant aux versions historiques.

Nous sommes donc, à nouveau, confrontés au problème de la réutilisation de méta-modèles existants sans le modifier. En effet, nous voulons étendre le comportement des domaines existants, tout en gardant la vision du domaine comme un composant fermé non modifiable.

Nous proposons donc d'utiliser la même idée avancée dans la section précédente : utiliser la Programmation Orientée Aspect pour spécifier les interactions ajoutées par la composition. La sémantique opérationnelle du domaine composite est donc spécifiée soit par les méthodes des classes ajoutées par la composition, soit par des aspects qui spécifient des nouvelles interactions entre classes existantes dans le méta-modèles de sous domaines.

Pour illustrer cette proposition nous transcrivons dans la figure 43 une partie de l'aspect correspondant à l'interaction pour la création d'une nouvelle version historique que nous venons de décrire.

```
package configuration;

import java.util.*;

import process.*;
import product.*;

public aspect Historic {
   …
   before(Activity activity, Port exitPort): target(activity) &&
   execution( public void Activity.terminate(Port)) &&
   args(exitPort) {

      for (DataVariable variable: exitPort.getDefinition().getVariables()) {
         if (variable.getVersionAction() != null) {
            Revision source = variable.getValue().getRevision();
            Branch historic = source.getProduct().getHistoricBranch();
            historic.createRevision(source.getAttributes());
         }
      }
   }

}
```

figure 43. Composition de Machines Virtuelles, en AspectJ

Une observation importante dans cet exemple est que nous pouvons identifier dans cet aspect l'implémentation du lien de conformité entre l'état de l'exécution et le modèle : la méthode traverse le lien vers la définition du port, et à partir de là, interroge le modèle composite pour déterminer l'action de versionnement à effectuer pour chacune des variables définies dans le port.

Nous retrouvons donc le même pattern que nous avons identifié dans les machines virtuelles d'un domaine simple, mais cette fois-ci dans les aspects : certaines méthodes de la partie dynamique du méta-modèle font référence au modèle statique pour déterminer leur comportement.

Cet exemple nous permet aussi d'insister sur le point de la correspondance très directe entre le méta-modèle et la conceptualisation du domaine. Nous pouvons observer que ce fait facilite énormément la composition, le volume de code des aspects

est réduit et le niveau d'abstraction nous permet une implémentation très littérale des concepts.

Pour illustrer davantage nos propos, et compléter notre scénario de composition, nous transcrivons dans la figure 44 l'intégralité de l'aspect correspondant au deuxième cas d'utilisation.

```
package configuration;

import java.util.*;

import process.*;
import product.*;


public aspect Concretisation {

   /*
    * Associate Data to Revision
    */

   after(Port port, String variableName, Data data) : target(port) &&
   execution(public void Port.bind(String ,Data)) &&
   args (variableName,data) {

      if (data.getRevision() == null) {
         DataVariable variable =
               port.getDefinition().getVariable(variableName);
         ProductType type     = variable.getType();
         data.revision        = ProductSelector.selectProduct(type,data);
      }
   }


   /*
    * Clone revisions when Data is clones
    */

   after(Data data) returning(Data clone) : target(data) &&
   execution(public Data Data.clone()) {
      Revision source      = data.getRevision();
      Map attributes       = source.getAttributes();
      Revision destination = source.getBranch().createRevision(attributes);
      clone.revision       = destination;
   }
}
```

figure 44. Composition de Machines Virtuelles, en AspectJ

Nous pouvons aussi noter que les méthodes de l'aspect effectivement spécialisent les méthodes existantes dans le domaine de gestion de procédés. Nous observons aussi que, parce que le concept de donnée est très abstrait dans le domaine de gestion de procédés, la spécialisation est relativement simple.

Nous pouvons également constater que dans ce cas une grande partie du comportement est absorbée, et n'est pas contrôlée par le modèle. La seule influence du modèle concerne l'association des types avec les variables dans les ports des activités, ce qui limite le choix des produits manipulés par les activités.

Si nous nous plaçons dans la vision d'un domaine comme un composant réutilisable, l'usage des aspects peut être compris comme une façon d'adapter le composant à une composition. Dans ce sens, notre proposition est similaire à celles des interactions logicielles [BCE+04] et des contrats de coordination [LF05] qui visent l'adaptation dynamique d'un système, en séparant les entités du noyau de système des

règles régissant leur collaboration. Si bien les objectives sont différents, les mécanismes mis en œuvre sont très similaires.

4.7 EXECUTION CONCRETE.

Dans certains cas, l'implémentation des fonctionnalités d'un domaine composite peut nécessiter le développement des composants spécifiques qui doivent être intégrés dans le domaine.

Par exemple, dans notre scénario de composition au moment d'associer une donnée d'un procédé à une révision de produit, nous devons proposer à l'utilisateur la possibilité de choisir parmi les versions historiques existantes. Ceci implique le développement d'un outil interactif, qui peut dépendre de la plate-forme d'exécution, par exemple s'il s'agit d'une application Web ou d'un client lourd.

Evidement, nous ne voulons pas inclure ces fonctionnalités directement dans la machine virtuelle du domaine composite. Nous utilisons donc les mêmes mécanismes proposés dans le chapitre précédent pour un domaine simple. Du point de vue de l'exécution concrète, un domaine composite se comporte comme tous les autres domaines.

Spécification de la configuration.

Au niveau de l'environnement de support Mélusine, il est donc possible de définir des composants, et de spécifier la couche de médiation d'un domaine composite. Nous pouvons aussi utiliser les mécanismes de gestion de la variabilité : la substitution de composants et les options de réalisation. La figure 45 présente un aperçu de la définition des composants et de la configuration du domaine composite.

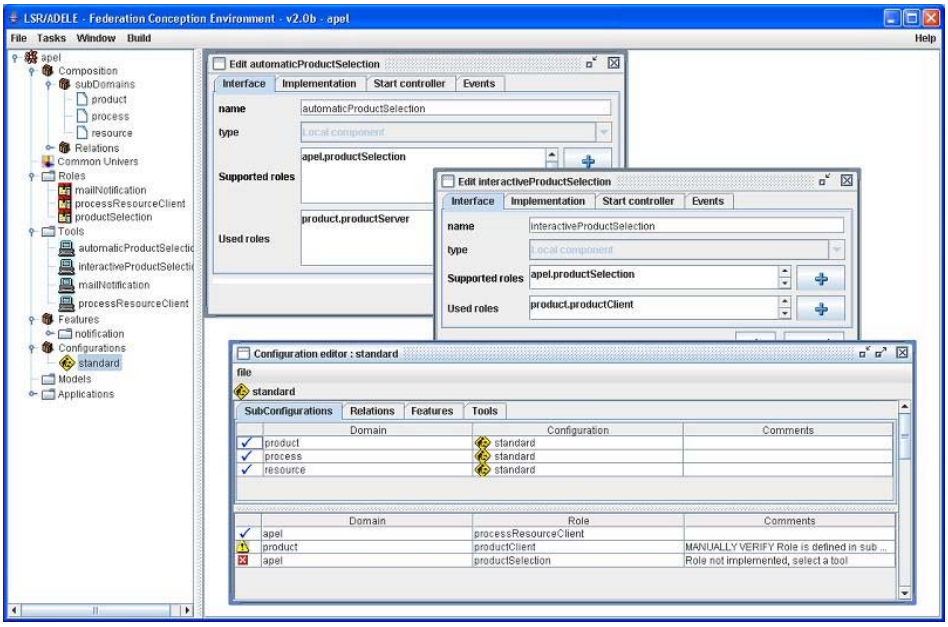

figure 45. Environnement Mélusine : gestion de domaines composites

La seule différence au niveau de la configuration pour un domaine composite est qu'il faut spécifier les configurations à utiliser pour les sous domaines. Notez donc qu'au niveau de la configuration nous essayons aussi de réutiliser les artefacts existants

dans le sous domaines. Une configuration composite fait référence aux configurations de sous domaines.

Exécution de l'application concrète.

Pour exécuter une application concrète, nous devons spécifier un modèle et une configuration du domaine composite. Le modèle de la composition fait référence aux modèles de sous domaines ; et, similairement, la configuration composite fait référence aux configurations de sous domaines. De cette manière, nous fixons donc un point sur chacune des dimensions de l'exécution, pour chacun des domaines impliqués.

Lorsqu'on exécute une application, la machine virtuelle du domaine va s'exécuter simultanément avec les composants concrets. Mais grâce à l'instrumentation et les couches de médiations, leur exécution sera synchronisée.

Dans le cas d'un domaine composite, nous n'avons pas spécifié directement l'interaction entre les composants de sous domaines et ceux du domaine composite. Mais, chaque composant va se synchroniser avec la machine virtuelle de son domaine, et celles si vont être synchronisées entre elles par les aspects de la composition. Le résultat final est que les composants seront coordonnés indirectement.

La figure 46 montre l'exécution concrète d'un modèle de la composition. On peut observer le composant chargé de l'exécution interactive d'un procédé, ainsi que le composant de sélection d'une version historique, qui a été développé pour la composition. L'activation du composant de sélection a été déclenchée par l'affectation d'une donnée à une variable d'un port d'une activité, dans l'occurrence le port `desktop` de l'activité `créer`.

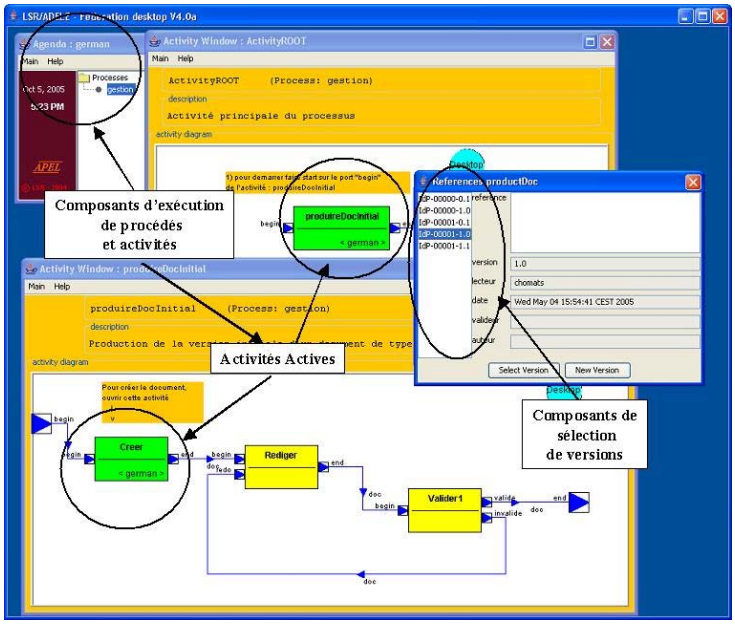

figure 46. Exécution d'une application composite

Nous pouvons observer sur la figure que, bien que nous assurons la synchronisation de l'exécution des composants, du point de vue de l'utilisateur final l'intégration est imparfaite. Il va devoir interagir avec un ensemble disparate de composants faiblement couplés. Dan les cas où les domaines sont construits à partir des

composants développés spécifiquement, il est possible d'envisager une intégration au niveau de l'interface des composants ; cependant, nous n'avons pas exploré cette problématique.

4.8 COMPOSITION MULTI-NIVEAUX.

Tout au long de ce chapitre nous avons proposé de structurer un domaine composite d'une façon analogue aux domaines simples, développés dans le chapitre précédent. La seule différence remarquable est l'usage des techniques de Programmation Orientée Aspect dans l'implémentation du méta-modèle composite.

Notre objectif est de pouvoir recomposer les domaines composites dans des nouvelles compositions plus complexes, et pouvoir ainsi traiter les domaines comme des unités de réutilisation de très gros grain.

Nous avons appliqué ces idées sur la composition des domaines aux différents domaines d'expertise de notre équipe de recherche ([ELV03] [EVL+03] [Vill03] [Le04] [Me04] [Se05]), et nous avons construit l'ensemble des méta-modèles présenté dans la figure 47.

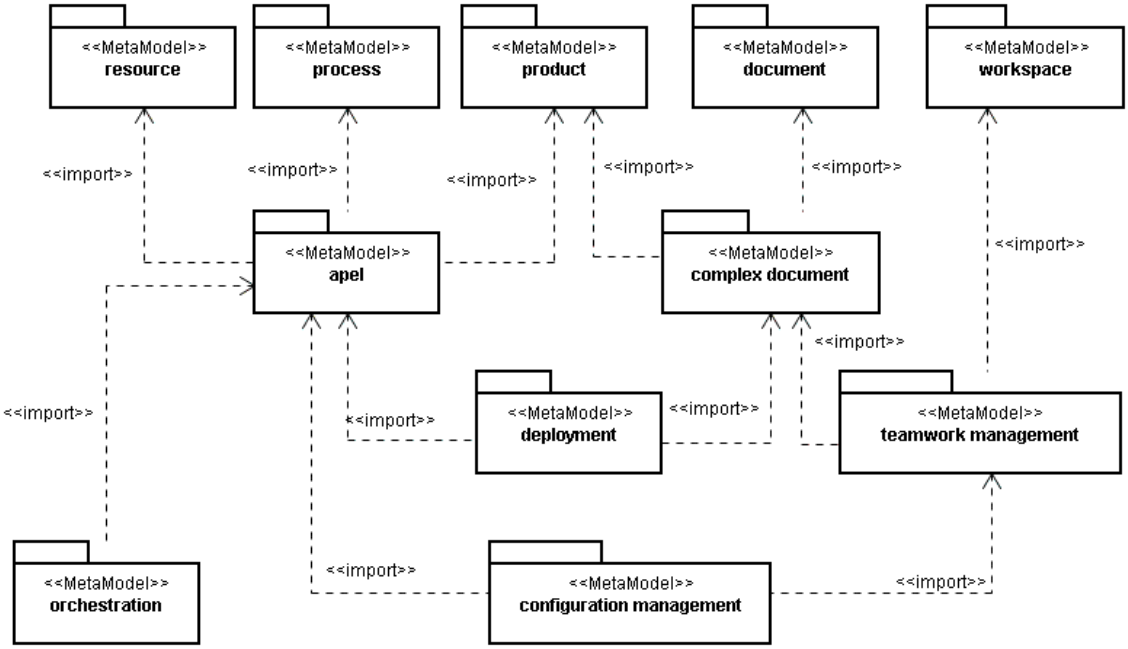

figure 47. Composition multi-niveaux

D'abord, nous faisons noter qu'il est possible d'identifier plusieurs niveaux dans la figure, nous voulons faire quelques observations sur cette structure:

- Il existe un ensemble des domaines très génériques, en haut de la figure, qui implémentent des fonctionnalités transversales. Les critères principaux pour définir ces domaines ont été la réutilisation et l'indépendance. Il s'agit donc des domaines qui fournissent des fonctionnalités de base limitées, et qui ne se recouvrent pas entre eux ;
- Au milieu de la figure, nous trouvons de domaines qui fournissent des fonctionnalités plus spécifiques, pour la gestion des procédés complexes (le

produit Apel) ou le déploiement, par exemple. Notre critère principal dans la conception de ces domaines a été l'autonomie. Nous avons cherché à ce que chacun des ces domaines se comporte comme une ligne de produits spécialisée et autonome ;

- Finalement, les domaines en bas de la figure visent à fournir des fonctionnalités avancées et spécifiques, à partir des domaines existants. C'est à ce niveau que nous avons observé les gains plus importants en termes de réutilisation.

Nous ne pouvons pas généraliser cette structure à tout domaine d'expertise, mais nous pensons qu'elle reflète bien notre proposition d'utiliser les domaines comme unités de réutilisation de très gros grain.

Une deuxième remarque importante sur la composition est qu'elle n'est pas hiérarchique, les domaines sont structurés en graphe. Ceci a permit de structurer les méta-modèles en niveaux, car il est très probable que les domaines intermédiaires partagent un des domaines de base.

La structure en graphe est rendue possible grâce au fait que, dans notre proposition, nous ne modifions pas les méta-modèles lors de la composition, le domaine composite ne peut qu'ajouter des nouvelles classes, associations et interactions. Si on compose deux domaines qui partagent un sous domaine, les fonctionnalités vont se superposer. Clairement, il existe un risque d'interférence entre plusieurs compositions. Nous n'avons pas approfondi sur ce problème, mais c'est un sujet de recherche actif dans la communauté de Programmation Orientée Aspect.

Une dernière remarque est que la composition n'est pas nécessairement entre deux domaines seulement, bien que, dans le scénario présenté, nous ayons choisi une composition binaire pour faciliter la présentation. Par exemple dans le cas du domaine Apel dans la figure 47, il est nécessaire de faire intervenir plusieurs domaines pour pouvoir spécifier complètement les nouvelles fonctionnalités. Dans notre démarche, l'ajout des nouvelles associations ou interactions n'impose aucune contrainte sur le nombre de domaines impliqués.

4.9 CONCLUSION.

La prémisse de notre démarche est que nous pouvons considérer les domaines, à la fois comme un mécanisme de séparation des préoccupations et comme un mécanisme de réutilisation. Nous avons envisagé alors la composition de domaines comme une façon de spécifier conceptuellement l'intégration des différents points de vue d'une application, en s'appuyant sur les concepts existants dans les sous domaines à composer.

Grâce à la structure d'un domaine, le méta-modèle représentant sa conceptualisation, nous nous sommes concentrés sur la composition des méta-modèles, et nous nous sommes affranchis des couches sous-jacentes de composants et médiation.

Nous avons ensuite suivi une démarche complètement analogue à celle proposée pour les domaines de base : d'abord s'intéresser à l'analyse du domaine et à l'établissement de son périmètre fonctionnel ; ensuite, spécifier les concepts visibles du

langage de modélisation, réifiés par la partie statique du méta-modèle ; ensuite proposer une syntaxe concrète et un environnement de modélisation ; et, finalement, spécifier la sémantique de l'exécution au moyen d'une machine virtuelle du domaine.

Dans chacune de ces parties, nous avons proposé des mécanismes pour effectuer la composition. Dans tous les cas, nous avons favorisé les mécanismes nous permettant de réutiliser les artefacts existants dans le sous domaines. Le principal mécanisme que nous avons utilisé est l'ajout de nouvelles relations entre les concepts existantes :

- Cela signifie, au niveau de méta-modèle statique, que le langage de modélisation du domaine ne comporte que les concepts qui n'existent pas dans les sous domaines, et des associations entre les concepts existants ;
- Au niveau des modèles, cela veut dire que le modèle composite permet principalement d'établir des liens entre les éléments des modèles de sous domaines. Dans ce sens, nous pensons que notre proposition est proche de l'idée de tissage de modèles ;
- Au niveau de la machine virtuelle, cela implique l'utilisation des techniques de Programmation Orientée Aspect pour pouvoir ajouter de nouvelles interactions entre les objets des classes des machines virtuelles des sous domaines.

Une caractéristique importante de cette démarche est que le méta-modèle résultant possède la même structure que celui d'un domaine simple. Il est donc possible de composer à nouveau un domaine composite. Ceci nous permet de considérer les domaines comme des unités de réutilisation de gros grains.

A partir de là, il est possible de créer des agencements complexes de domaines qui permettent non seulement de promouvoir la réutilisation, mais également de structurer conceptuellement de vastes espaces de problèmes.

Nous avons mis à profit cette démarche dans notre équipe, et nous avons pu constater ses bénéfices, spécialement quant à la structuration conceptuelle d'un vaste espace métier d'expertise. Nous avons aussi constaté les gains en termes de réutilisation, et surtout la facilité pour introduire de fonctionnalités sophistiquées, tout en raisonnant à haut niveau d'abstraction.

Nous sommes parties d'une idée simple : les langages de modélisation dédiés constituent une approche effective de génie logiciel, mais ils sont limités par sa portée réduite. La composition de méta-modèles est donc un moyen de nous affranchir de cette limitation, sans perdre les points forts de l'approche DSL. La réalisation de cette idée est néanmoins loin d'être triviale, si l'on s'intéresse à tous les aspects d'un langage de modélisation. Nous pensons que notre proposition est une contribution importante pour aborder cette problématique.

5. CONCLUSION ET PERSPECTIVES.

Nous avons proposé une démarche basée sur le concept de domaine. Un domaine étant un champ de connaissance métier bien délimité. Une application est décrite par divers modèles exprimés dans des langages de modélisation propres à chaque domaine. La composition de modèles permet de modéliser des applications complexes couvrant plusieurs domaines simultanément.

L'idée maîtresse de notre proposition est de considérer qu'un domaine est défini par un méta-modèle contenant les concepts essentiels pour spécifier les applications du domaine, et de montrer qu'il est possible de composer ces méta-modèles pour réutiliser les domaines existants et pour définir des domaines plus vastes.

Dans ce chapitre nous faisons une synthèse des principales idées de notre proposition, et nous reprenons certaines de nos réflexions ; dans le but de mettre en avant nos contributions principales, d'identifier les questions ouvertes et les perspectives de ce travail.

5.1 APPLICATIONS A MULTIPLES DOMAINES.

L'élément central de notre proposition est le domaine. Nous nous sommes alors intéressés aux travaux sur l'ingénierie des domaines, développés dans la communauté des lignes de produits logiciels.

Dans les approches de lignes de produits logiciels, le but de l'analyse du domaine est de délimiter son périmètre fonctionnel. Le produit de cette analyse est souvent représenté par un diagramme de features, qui précise les caractéristiques partagées par tous les membres de la ligne de produits, ainsi que leurs caractéristiques distinctives.

Dans cette vision, le domaine est assimilé à une gamme de produits, et devient un contexte qui circonscrit la portée de la réutilisation. Cette spécialisation a démontré de grandes qualités en termes de réutilisation, et une grande facilité pour définir de nouvelles applications à l'intérieur de la ligne de produits.

Cependant, nous pensons, que cette spécificité explique aussi, en partie, les difficultés pour étendre ces bénéfices au-delà de la frontière d'un domaine, et à généraliser l'approche à des domaines plus vastes.

Notre première proposition a été donc d'élargir la perspective de l'analyse du domaine : passer à une vision basée sur un ensemble de domaines. Chaque domaine ayant une fonctionnalité plus réduite que celle d'une ligne de produit, mais un potentiel de réutilisation plus étendu. Ces domaines peuvent ensuite être combinés pour construire des nouveaux domaines plus riches fonctionnellement. C'est cette idée qui nous démarque des approches des lignes de produits, tout en restant dans le cadre général de l'ingénierie des domaines.

Nous avons mené cette idée « jusqu'au bout », et nous avons montré qu'il est possible de composer plusieurs domaines différents pour définir un domaine plus vaste qui à la fois dispose d'une couverture fonctionnelle élargie, qui permet de définir des nouvelles fonctionnalités, mais surtout qui réutilise la totalité des artefacts logiciels développés dans chaque domaine. Nous pensons que c'est une contribution importante aux approches des familles de systèmes.

5.2 L'AMALGAME CONCEPTUALISATION/META-MODELE.

Dans notre vision, nous avons également besoin de délimiter le périmètre du domaine, et de compter avec une représentation abstraite de ses concepts. Nous avons proposé alors de faire une conceptualisation du domaine, et de la matérialiser par un modèle du domaine.

Le modèle du domaine n'est pas simplement la connaissance des experts du domaine, mais une explicitation de cette connaissance, rigoureusement structurée, que nous voulons opérationnelle. Sur ce point, nous avons évoqué le rapport avec les travaux sur les ontologies et la représentation des connaissances, qui restent à explorer.

Cependant, nous ne voulons pas simplement faire une conceptualisation du domaine, nous voulons aussi proposer un langage de modélisation dédié, permettant de spécifier une application à partir des concepts propres au domaine.

Nous avons proposé alors de matérialiser la conceptualisation du domaine par un modèle orientée objet des concepts clés qui permettent de définir les abstractions du domaine. Et nous avons avancé l'idée que ce modèle peut jouer simultanément le rôle de conceptualisation du domaine et de méta-modèle du langage dédié au domaine.

Cette idée nous paraît fondamentale, nous pensons qu'un grand nombre des avantages et des bénéfices imputés aux langages dédiés découlent de l'existence d'un lien étroit entre la conceptualisation du domaine et le méta-modèle du langage. Nous pensons que c'est un sujet qui mérite d'être approfondi, et nous estimons que les réflexions que nous avons menées à ce propos constituent une contribution dans ce sens.

Nous avons insisté à plusieurs reprises sur l'importance, dans notre démarche, du lien entre la conceptualisation et le méta-modèle. En particulier, nous voulons souligner que c'est grâce à ce lien que nous pouvons raisonner sur la composition en termes des concepts des domaines impliqués, pour ensuite pouvoir l'exprimer relativement facilement en termes de nouvelles associations et interactions entres les classes des méta-modèles existants.

5.3 LA MACHINE VIRTUELLE DU DOMAINE.

Une caractéristique distinctive de notre démarche est notre intérêt pour l'exécution des modèles. Nous avons abordé cette problématique sous deux perspectives différentes : celle de la sémantique de l'exécution du langage d'une part, et celle de la plate-forme d'exécution et des bibliothèques de composants du domaine d'autre part.

Nous avons favorisé la vision centrée sur la sémantique opérationnelle du langage, et nous avons proposé la construction d'un interpréteur du langage. Nous avons proposé de réifier les concepts du domaine qui représentent l'état de l'exécution, et de les incorporer au méta-modèle du langage pour construire une machine virtuelle du domaine. Cette machine virtuelle nous permet d'interpréter les modèles du domaine.

Nous avons tenu ici, de nouveau, à maintenir une correspondance directe entre la conceptualisation du domaine et le méta-modèle, et nous avons donc insisté pour que les classes de la machine virtuelle correspondent très directement aux concepts du domaine.

Nous avons aussi tenu à faire une séparation très nette entre les concepts du méta-modèle qui représentent la syntaxe abstraite du langage, que nous avons qualifiés de statiques, et ceux qui représentent l'état de l'exécution, que nous avons qualifiés de dynamiques. Cette séparation des concepts n'est pas indispensable dans notre démarche ; néanmoins, elle nous paraît importante et intéressante par plusieurs raisons :

- Elle permet d'améliorer la modularité du méta-modèle. La définition du langage du domaine implique, outre que le développement de la machine virtuelle, le développement d'une syntaxe concrète et d'un environnement de modélisation. La segmentation du méta-modèle nous permet de développer ces activités en parallèle, tout en gardant les avantages d'un méta-modèle unique ;
- Cette division permet de distinguer très clairement entre les concepts visibles du langage et ceux qui sont absorbés. En faisant cette distinction, le méta-modèle spécifie clairement les caractéristiques communes et les variabilités du domaine, et sépare les variabilités traitées en « temps de modélisation » des celles traitées en « temps d'exécution » ;
- La distinction entre les parties statique et dynamique nous permet de raisonner sur la relation de conformité entre l'état de l'exécution et le modèle ; et, réciproquement, sur la façon dont le modèle prescrit l'évolution de l'état de l'exécution. Dans le cas de langages généraux de modélisation orienté objets, par exemple, les caractéristiques de cette conformité ont fait l'objet de longs débats autour des notions de classification et d'instanciation, et ceci se reflète dans les méta-modèles actuels de ces langages. En revanche, dans le cas des langages dédiés, la nature de cette conformité est souvent confuse, et généralement implicite dans les raisonnements des experts du domaine ; nous pensons que le méta-modèle doit contribuer à clarifier la nature de cette relation de conformité, de la façon la plus explicite possible ;

- Cette séparation permet d'identifier plus clairement les parties de l'interpréteur qui dépendent du modèle en cours d'interprétation. Ceci permet d'envisager la possibilité de générer automatiquement un code exécutable à partir d'un modèle, en faisant appel à des techniques d'évaluation partielle sur l'interpréteur, tel que proposé par [Thi98] ;
- Des patterns similaires ont été utilisés dans les applications qui visent l'adaptabilité. Nous pouvons donc imaginer d'étendre nos travaux à des systèmes dans lesquels le modèle évolue dynamiquement à l'exécution.

Nous croyons que chacun des points mentionnés ci-dessus offre des perspectives pour étendre et généraliser nos idées autour de la notion de modèle exécutable.

Finalement, nous voulons faire remarquer qu'il est possible de réfléchir sur notre démarche comme une transformation progressive de la conceptualisation du domaine en une machine virtuelle capable d'exécuter des modèles, exprimés dans un langage dédié au domaine. Cette optique ouvre la voie à de nombreux travaux futurs sur la possibilité d'assister, voir même automatiser, une partie de cette transformation.

Dans ce mémoire nous avons présenté de façon systématique notre démarche actuelle pour accomplir cette transformation, et nous avons identifié les points que nous considérons importants dans cette transformation. Nous pensons avoir apporté des éléments de réflexion importants pour entreprendre des nouvelles recherches dans cette direction.

5.4 LA COMPOSITION DE META-MODELES.

Le mécanisme que nous avons proposé pour construire un nouveau domaine, à partir de domaines existants, est la composition des méta-modèles. Le méta-modèle du domaine composite résultant comporte de nouvelles classes, de nouvelles associations entre les classes existantes, et de nouvelles interactions entre ces classes.

Notre principale motivation a été la réutilisation de tous les artefacts développés dans les sous domaines. Pour cette raison, nous avons favorisé l'utilisation de technologies de Programmation Orientée Aspect et de tissage de modèles.

Une propriété très importante de ce mécanisme est que le méta-modèle du domaine composite ainsi obtenu garde la même structure que celui d'un domaine simple, et il peut donc être à nouveau composé. Cette façon de composer nous permet d'agencer les domaines dans des structures complexes, afin de pouvoir couvrir un vaste espace fonctionnel. En particulier, il est possible de composer les domaines de façon non hiérarchique pour obtenir une structure en graphe, dans laquelle plusieurs compositions ajoutent des nouvelles fonctionnalités à un domaine de base partagé.

En appliquant notre démarche au domaine d'expertise de notre équipe (gestion de configuration, gestion des procédés), nous avons pu constater les bénéfices de cette approche en termes de réutilisation, et de facilité pour produire des fonctionnalités spécialisées et sophistiquées, tout en gardant un haut niveau d'abstraction lors de la composition. Néanmoins, lors des compositions les plus complexes, certaines des limites de ce mécanisme deviennent perceptibles. La limitation la plus évidente est que le méta-modèle composite est l'union de tous les concepts des méta-modèles des sous

domaines, plus les nouvelles classes et associations ajoutées. Ainsi, après plusieurs niveaux de composition, le méta-modèle devient très complex et difficile à maîtriser.

Cette limitation est en réalité le symptôme que nous nous éloignons du cadre des deux suppositions sous-jacentes à notre approche : (1) chaque domaine correspond à une préoccupation indépendante, et (2) le domaine composite est du même niveau d'abstraction que les sous domaines. Ces suppositions correspondent à notre perception du domaine comme un mécanisme de séparation des préoccupations, et de la composition des domaines comme un mécanisme d'intégration.

Dans nos expérimentations, les domaines de base ont été conçus avec des fonctionnalités bien délimitées et sans recouvrement. Néanmoins, lorsque nous avons composé à de multiples niveaux, nous avons trouvé de nouveaux cas intéressants de composition entre domaines, par exemple :

- Le domaine composite est l'union des méta-modèles, mais le concepteur du domaine composite voudrait masquer certains concepts ou fusionner des concepts redondants des sous domaines. Il s'agit donc d'une forme d'abstraction, dans laquelle le domaine composite n'exhibe pas tous les détails des sous domaines ;
- Le domaine composite possède son propre modèle conceptuel, et l'objectif de la composition est d'implémenter ces concepts à partir des concepts des sous domaines. Dans ce cas, clairement le domaine composite n'a pas le même niveau d'abstraction que les sous domaines. Les concepts du domaine composite doivent être « projetés » sur les concepts des domaines sous-jacents.

Dans ces deux exemples, nous pouvons voir une progression vers un style de composition dans lequel les concepts de sous domaines composés ne sont plus visibles dans le méta-modèle composite. La principale difficulté pour effectuer ce type de composition est de pouvoir réutiliser les artefacts des sous domaines. Par exemple, dans le cas où les concepts du domaine composite seraient d'un niveau d'abstraction plus haut que les sous domaines, nous ne voulons pas exposer le concepteur d'applications dans le domaine composite aux détails des sous domaines. Ceci implique que nous ne pouvons pas réutiliser les environnements de modélisation de ces sous domaines.

Une alternative envisageable dans ce scénario est de générer les modèles des sous domaines à partir du modèle composite. Dans ces cas, nous ne pouvons plus raisonner sur le modèle composite comme le tissage des modèles des sous domaine. Une vision plus proche de l'intention de ce type de composition pourrait être de réfléchir en termes de raffinement de domaines. Dans ce scénario de raffinement de domaines, il faudrait aussi s'intéresser à la notion d'exécution des modèles, et au rapport entre les machines virtuelles de sous domaines et celle du domaine composite. A première vue, il ne s'agit pas de l'établissement de nouvelles interactions entre les concepts existants, mais plutôt d'un empilement de machines virtuelles.

Dans tous les cas, nous pensons qu'il existe un grand nombre de perspectives de recherche prometteuses afin de préciser la nature des différentes relations de composition entre domaines, et de proposer les mécanismes de composition de méta-modèles appropriés.

Dans ce travail, nous avons identifié une relation de composition très générale : l'intégration des domaines autonomes et indépendants. Nous avons aussi proposé un mécanisme de composition adapté, qui possède des propriétés très intéressantes en termes de réutilisation et de facilité de mise en œuvre. Nous estimons donc avoir fait une contribution importante dans cette voie.

6. BIBLIOGRAPHIE.

[AES01] ALVAREZ J., EVANS A., SAMMUT P. – Mapping between Levels in the Metamodel Architecture. - Dans : *Proceedings of the 4th International Conference on the Unified Modeling Language, Modeling Languages, Concepts and Tools <<UML>> 2001*. – Toronto,Canada, 2001. p. 34-46

[AK02] ATKINSON C., KÜHNE T. – Rearchitecting the UML Infrastructure. – *ACM Transactions on Modeling and Computer Simulation.* – 2002, Vol. 12, No. 4, p. 290-321.

[AM04] ANASTASOPOULOS M., MUTHIG D. – An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology. - Dans : *Proceedings of 8th International Conference on Software Reuse: Methods, Techniques and Tools ICSR 2004.* – Madrid, Espagne, 2004. p. 141-156

[Bat05] BATORY D. – Feature Models, Grammars, and Propositional Formulas. - Dans : *Proceedings of the 9th Software Product Line Conference SPLC 2005*. – Rennes, France, 2005.

[BB02] BRETON E., BÉZIVIN J. – Weaving Definition and Execution Aspects of Process Meta-Models. - Dans : *Proceedings of 35th Hawaii International Conference on System Sciences HICSC-35 Track 9.* – Big Island, HI, Etats-Unis, 2002. p. 290b

[BB01] BRETON E., BÉZIVIN J. – Towards an Understanding of Model Executability. - Dans : *Proceedings of the International Conference on Formal Ontology in Information Systems 2001*. – Ogunquit, MA, Etats-Unis, 2001. p. 70-80

[BBB+05] BÉZIVIN J., BLAY M., BOUZEGHOUB M., ESTUBLIER J., FAVRE J-M. *Rapport de Synthèse : Action Spécifique CNRS sur l'Ingénierie Dirigée par les Modèles*, [En Ligne] Centre National de la Recherche Scientifique CNRS, 2005. - Disponible sur Internet : http://www-adele.imag.fr/mda/as/

[BCE+04] BLAY-FORNARINO M., CHARFI A., EMSELLEM D., PINNA-DERY A-M., RIVEIL M. – Software Interactions. – *Journal of Object Technology*. [En ligne] – 2004, Vol. 3 No. 10 p. 161-180. – Disponible sur Internet : http://www.jot.fm/issues/issue_2004_11/article4

[BDJ+05] BÉZIVIN J., DIDONET M., JOUAULT F., VALDURIEZ P. – Combining Preoccupations with Models. - Dans : *Proceedings of the 1st Workshop on Models and Aspects at ECOOP 2005* [En ligne]. – Glasgow, Ecosse, Royaume-Uni, 2005. – Disponible sur Internet : http://www.st.informatik.tu-darmstadt.de:8080/ecoop2005/maw/

[Bez05] BÉZIVIN J. – On the Unification Power of Models. – *Software and System Modeling.* – 2005, Vol. 4, No. 2, p. 171-188

[BFG+01] BOSCH J., FLORIJN G., GREEFHORST D., KUUSELA J., OBBINK H., POHL K. – Variability Issues in Software Product Lines. - Dans : *Proceedings of the 4th International Workshop on Product Family Engineering PFE-4.* – Bilbao, Espagne, 2001.

[BG04] BOUZITOUNA S., GERVAIS M-P. Composition Rules dor PIM Reuse. – Dans : *Proceedings of the 2nd European Workshop on MDA with emphasis in Methodologies and Transformations EWMDA04.* – Canterbury, Royaume-Uni, 2004.

[BGB05] BOUZITOUNA S., GERVAIS M-P., BLANC X. – Model Reuse in MDA. – Dans : *Proceedings of the 2005 International Workshop on Applications of UML/MDA to Software Systems at SERP 2005.* – Las Vegas, NV, Etats-Unis, 2005.

[BHJ+03] BIRK A., HELER G., JOHN I., SCHMID K., VON DER MASSEN T. MÜLLER K. – Product Line Engineering: The State of the Parctice. – *IEEE Software.* – 2003, Vol. 20, No. 6, p. 52-60

[BJM+02] BATORY D., JOHNSON C., MACDONALD B., VON HEEDER D. – Achieving Extensibility Through Product-Line and Domain-Specific Languages: A Case Study.- *ACM Transactions on Software Engineering and Methodology.* – 2002, Vol. 11, No. 2, p. 191-214

[BJV04] BÉZIVIN J., JOUAULT F., VALDURIEZ P. – On the Need for MegaModels. - Dans : *Proceedings of the Workshop on Best Practices for Model Driven Software Development at OOPSLA 2004.* – Vancouver, Canada, 2004.

[BKK+99] BROOKS P., KOCH C., KOVACS Z. LE GOFF J-M, MCCLATCHEY R. – Patterns for Multi-Layered System Architectures. – Dans : *Proceedings of the Metadata and Active Object-Model Pattern Mining Workshop at OOPSLA 1999* [En ligne]. – Denver, CO, Etats-Unis, 1999. – Disponible sur Internet : http://www.joeyoder.com/~yoder/Research/metadata/OOPSLA99/

[Bos02] BOSCH J. – Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization. – Dans : *Proceedings of the 2nd Software Product Line Conference SPLC2.* – San Diego, CA, Etats-Unis, 2002.

[Bos04] BOSCH J. – On the Development of Software Product-Family Components. – Dans : *Proceedings of the 3rd Software Product Line Conference SPLC 2004.* – Boston, MA, Etats-Unis, 2004.

[Bro04] BROWN A. – Model Driven Architecture: Principles and Practice. – *Software and System Modeling.* – 2004, Vol. 3, No. 4, p. 314-327

[BTR05] BENAVIDES D. TRINIDAD P. RUIZ-CORTÉS A. – Automated Reasoning on Feature Models.- Dans : *Proceedings of the 17th Conference on Advanced Information Systems Engineering CAiSE 05*. – Porto, Portugal, 2005.

[CA05] CZARNECKI K., ANTKIEWICZ M. – Mapping Features to Models: A Template Approach Based on Superimposed Variants. – Dans : *Proceedings of the 4th International Conference on Genartive Programming and Component Engineering GPCE 05*. – Tallin, Estonie, 2005.

[CE00] CZARNECKI K., EISENECKER U. – *Generative Programming: Methods, Tools and Applications.* – 1er édition. – Addison Wesley Professional, 2000. – 864 p.

[CES+04] CLARK T., EVANS A., SAMMUT P. , WILLANS J. - *Applied Metamodelling, A Foundation for Language Driven Development*. [En ligne] Xactium, 2004. - Disponible sur Internet : http://albini.xactium.com

[CHE04] CZARNECKI K., HELSEN S., EISENECKER U. – Staged Configuration Using Feature Models. – Dans : *Proceedings of the 3rd Software Product Line Conference SPLC 2004*. – Boston, MA, Etats-Unis, 2004.

[CHE05] CZARNECKI K., HELSEN S., EISENECKER U. – Formalizing Cardinality-Based Feature Models and their Specialization. – *Software Process: Improvement and Practice.* – 2005, Vol. 10, No. 1, p. 7-29

[CHW98] COPLIEN J., HOFFMAN D., WEISS D. – Commonality and Variability in Software engineering. – *IEEE Software.* – 1998, Vol. 15, No. 6, p. 37-45

[CN01] CLEMENTS P., NORTHROP L. – *Software Product Lines: Practices and Patterns.* – 1er édition. – Addison Wesley Professional, 2001. – 608 p.

[Con04] CONSEL CH. – From a Program Family to a Domain-Specific Language. – Dans : *Proceedings of the International Seminar on Domain-Specific Program Generation.* – Dagstuhl Castle, Allemagne, 2004.

[Cop00] COPLIEN J. – *Multi-Paradigm Design*. – Thèse de Doctorat. – Vrije Universiteit, Bruxelles, Belgique, juillet 2000. – 276 p.

[CRB04] COLYER A., RASHID A., BLAIR G. – *On the Separtion of Concerns in Program Families.* [En ligne] – Lancaster University, Computer Departement, 2004. – 11 p. Rapport Technique COMP-001-2004 – Disponible sur Internet : http://www.comp.lancs.ac.uk/computing/aose/Publications.php

[Cza04] CZARNECKI K. – Overview of Generative Programing. – Dans : *Proceedings of the International Workshop on Unconventional Programming Paradigms UPP'04.* – Mont Saint-Michel, France, 2004.

[DBJ+05] DIDONET M., BEZIVIN J., JOUAULT F., BRETON E., GUELTAS G. – AMW: A Generic Model Weaver. – Dans : *Premières Journées sur l'Ingénierie Dirigée par les Modèles* [En ligne]. – Paris, France, 2005. – Disponible sur Internet : http://idm.imag.fr/idm05/

[DSG+03] DEELSTRA S., SINNEMA M., VAN GURP J., BOSCH J. – Model Driven Architecture as Approach to Manage Variability in Software Product Families. - Dans : *Proceedings of the Model Driven Architecture: Foundations and Application MDAFA 2003*. – Enschede, Pays Bas, 2003.

[DK98] VAN DEURSEN., KLINT P. – Little Languages: Little Maintenane? – *Journal of Software Maintenance*. – 1998, Vol. 10 No. 7 p. 75-92.

[EB04] EGYED A., BALZER R. – Integrating COTS software into Systems through Instrumentation and Reasoning. – *Journal on Automated Software Engineering (accepté pour publication, à paraître)*. - [En ligne] disponible sur Internet : http://sunset.usc.edu/~aegyed/publications.html

[Egy04] EGYED A. – Architecture Differencing for Self Management. - Dans : *Proceedings of the 2nd Workshop on Self-Managed Systems at FSE 2004*. – Newport Beach, CA, Etats-Unis, 2004.

[EIV05] ESTUBLIER J., IONITA A., VEGA G. – A Domain Composition Approach. – Dans : *Proceedings of the 2005 International Workshop on Applications of UML/MDA to Software Systems at SERP 2005*. – Las Vegas, NV, Etats-Unis, 2005.

[EJB04] EGYED A., JOHANN S., BALZER R. – Data and State Synchronicity Problems while Integrating COTS Software into Systems. – Dans : *Proceedings of the 4th International Workshop on Adoption-Centric Software Engineering ACSE 2004*. – Edimbourg, Ecosse, Royaume-Uni, 2004.

[ELV03] ESTUBLIER J., LE A-T., VILLALOBOS J. – Using Federations for flexible SCM systems. – Dans : *Proceedings of the 11th Workshop on Software Configuration Management SCM 11*. – Portland, OR, Etats-Unis, 2003.

[EMS+03] EVANS A., MASKERI G., SAMMUT P., WILLANS J. – Building Families of Languages for Model-Driven System Development. – Dans : *Proceedings of the Workshop in Software Model Engineering WiSME at UML 2003* [En ligne]. – San Francisco, CA, Etats-Unis, 2003. – Disponible sur Internet : http://www.metamodel.com/wisme-2003/program.html

[EV05] ESTUBLIER J., VEGA G. – Reuse and Variability in Large Software Applications. – Dans : *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSoft International symposium on Foundations of Software Engineering ESEC/FSE 2005*. – Lisbonne, Portugal, 2005. p. 316-325

[Eva04] EVANS E. – *Domain-Driven Design: Tackling the complexity in the heart of software*. – 1er édition. – Addison Wesley Professional, 2004. – 560 p.

[EVI05] ESTUBLIER J., VEGA G., IONITA A. – Composing Domain-Specific Languages for Wide-Scope Software Engineering Applications. – Dans : *Proceedings of the 8th Conference on Model Driven Engineering Languages and Systems MoDELS 2005*. – Montego Bay, Jamaïque, 2005.

[EVL+03] ESTUBLIER J., VILLALOBOS J., LE A-T., SANLAVILLE S., VEGA G. – An approach and framework for Extensible Process Support. – Dans : *Proceedings of the 9th European Workshop on Software process Technology EWSPT 2003.* – Helsinki, Finland, 2003.

[Fav04] FAVRE J-M. – Foundations of Model (Driven) (Reverse) Engineering: Models. *Proceedings of the International Seminar on Language Engineering for Model-Driven Software Development, Dagstuhl Seminar 04101*. [En ligne] – Dagstuhl Castle, Allemagne, 2004. – Disponible sur Internet : http://drops.dagstuhl.de/portals/04101/

[Fav04b] FAVRE J-M. – Foundations of Meta-Pyramids: Languages vs. Metamodels. *Proceedings of the International Seminar on Language Engineering for Model-Driven Software Development, Dagstuhl Seminar 04101*. [En ligne] – Dagstuhl Castle, Allemagne, 2004. – Disponible sur Internet : http://drops.dagstuhl.de/portals/04101/

[FFB02] FEY D., FAJTA R., BOROS A. – Feature Modeling: A meta-Model to Enhance Usability and Usefulness. - Dans : *Proceedings of the 2nd Software Product Line Conference SPLC 2.* – San Diego, CA, Etats-Unis, 2002.

[FR04] FRITSCH C., RENZ B. – Four Mechanisms for Adaptable Systems, A Meta-level Approach to Building a Software Product Line. – Dans : *Proceedings of the 3rd Software Product Line Conference SPLC 2004.* – Boston, MA, Etats-Unis, 2004.

[GH05] GONZALEZ-PEREZ C., HENDERSON-SELLERS B. – A powertype-based metamodelling framework. – *Software and System Modeling.* [En ligne] – 2005, on-line first issue. – Disponible sur Internet : http://www.sosym.org/

[GPvS05] GUIZZARDI G., FERREIRA L., VAN SINDEREN M. – An Ontology-Based Approach for Evaluating the Domain Appropriateness and Comprehensibility Appropriateness of Modeling Languages. – Dans : *Proceedings of the 8th Conference on Model Driven Engineering Languages and Systems MoDELS 2005*. – Montego Bay, Jamaica, 2005.

[GPvS02] GUIZZARDI G., FERREIRA L., VAN SINDEREN M. – On the role of Domain Ontologies in the design of Domain-Specific Modeling Languages. – Dans : *2nd Workshop on Domain-Specific Visual Languages at OOPSLA 2002* [En ligne]. – Seattle, WA, Etats-Unis, 2002. - Disponible sur Internet : http://www.cis.uab.edu/info/OOPSLA-DSVL2/Papers/

[Gri95] GRISS M. – *Software Reuse: Objects and Frameworks are not Enough.* [En ligne] – Hewlett Packard, Software Technology Laboratory, 1995.- 7p. Rapport Technique HPL-95-03 – Disponible sur Internet : http://www.hpl.hp.com/techreports/95/HPL-95-03.html

[Gri00] GRISS M. – Implementing Product-Line Features By Composing Component Aspects. – Dans : *Proceedings of the 1st Software Product Line Conference SPLC1* . – Denver, CO, Etats-Unis, 2000.

[GS03] GREENFIELD J., SHORT K. – Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools. – Dans : *Companion of the 18th Conference on Object-Oriented Programming, Systems, Languages and Applications OOPSLA 2003.* – Anaheim, CA, Etats-Unis, 2003. – p. 16-27

[GS+04] GREENFIELD J., SHORT K., COOK S., KENT S. – *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* – 1er édition. – Wiley Publishing Inc., 2004. – 666 p.

[GW03] GEPPERT B., WEISS D. – Goal-Oriented Assessment of Product-Line Domains. – Dans : *Proceedings of the 9th Software Metrics Symposium METRICS 03.* – Sidney, Australia, 2003.

[Har02] HARSU M. – *A Survey on Domain Engineering.* [En ligne] – Tampere University of Technology, Institute of Software Systems, 2002. - 26p. Rapport Technique 31 – Disponible sur Internet : http://practise2.cs.tut.fi/pub/

[HG05] HENDERSON-SELLERS B., GONZALEZ-PEREZ C. – Connecting PowerTypes and Stereotypes. – *Journal of Object Technology.* [En ligne] – 2005, Vol. 4 No. 7 p. 83-96. – Disponible sur Internet : http://www.jot.fm/issues/issue_2005_09/article3

[HK03] HAUSSMANN J., KENT S. Visualizing Model Mappings in UML. – Dans : *Proceedings of the 2003 ACM Symposium on Software Visualization.* – San Diego, CA, Etats-Unis, 2003. p. 169-178

[HR04] HAREL D., RUMPE B. – Meaningful Modeling: What's the Semantics of "Semantics". – *Computer.* 2004, Vol. 37, No. 10, p 64-72

[Jack02] JACKSON M. – Some Basic Tenets of Description. – *Software and System Modeling.* – 2002, Vol. 1, No. 1, p. 5-9

[JGM+05] JEZEQUEL J-M., GERARD S., MRAIDHA C., BAUDRY B. – Approche Unificatrice par Les Modèles. – *Rapport Final : Action Spécifique CNRS sur l'Ingénierie Dirigée par les Modèles,* [En ligne] Centre National de la Recherche Scientifique CNRS, 2005. - Disponible sur Internet : http://www-adele.imag.fr/mda/as/

[JSG+04] JANSEN A., SMEDINGA R., VAN GURP J. BOSCH J. – First Class Feature Abstraction for Product Derivation. – *IEE Proceedings Software.* – 2004, Vol. 151, No.4, p. 187-197

[KG04] KOVSE J., GEBAUER CH. – VS-Gen: A Case Study of a Product Line for Versioning Systems. – Dans : *Proceedings of the 3rd Conference on Generative Programming and Component Engineering GPCE 2004.* – Vancouver, Canada, 2004.

[Kic96] KICZALES G. – Beyond the black box: open implementation. - *IEEE Software.* – 1996, Vol. 13, No. 1, p. 8-11

[KHH+01] KICZALES G., HILSDALE E., HUGUNIN J., KERSTEN M., PALM J., GRISWOLD W. – An overview of AspectJ. – Dans : *Proceedings of the 15th European Conference on Object-Oriented Programming ECOOP 2001*. – Budapest, Hongrie, 2001.

[KLM+97] KICZALES G., LAMPING J., MENDHEKAR A., MAEDA C., LOPES C., LOINGTIER J-M., IRWIN J. – Aspect Oriented Programming. – Dans : *Proceedings of the 11th European Conference on Object-Oriented Programming ECOOP 97*. – Jyväskylä, Finlande, 1997.

[KML+04] KARSAI G., MAROTI M., LEDECZI A., GRAY J., SZTIPANOVITS J. – Composition and Cloning in Modeling and Meta-Modeling. – *IEEE Transactions on Control Systems Technology*. – 2004, Vol. 12, No. 2, p. 263-278

[LBC05] LOPEZ-HERREJON R, BATORY D., COOK W. – Evaluating Support for Features in Advanced Modularization Technologies. - Dans : *Proceedings of the 19th European Conference on Object Oriented Programming ECOOP 2005*. – Glasgow, Ecosse, Royaume-Uni, 2005.

[Le04] LE A-T. – *Fédération : une Architecture Logicielle pour la Construction d'Applications Dirigée par les Modèles*. – Thèse de Doctorat. – Université Joseph Fourier de Grenoble, France, janvier 2004. – 200 p.

[Lee00] LEE M. – Model-based Reasoning: a Principled Approach for Software Engineering. – *Software – Concepts and Tools*. – 2000, Vol. 19, No. 4, p. 179-189

[LBM+01] LÉDECZI A., BAKAY A., MAROTI M., VÖLGYESI P., NORDSTROM G., SPRINKLE J., KARSAI G. – Composing Domain-Specific Design Environments. – *Computer*. 2001, Vol. 34, No. 11, p. 44-51

[LF05] LANO K., FIADEIRO J. – Extending UML with Coordination Contracts. - *Software and System Modeling*. [En ligne] – 2005, on-line first issue. – Disponible sur Internet : http://www.sosym.org/

[LKT04] LUOMA J., KELLY S., TOLVANEN J-P. – Defining Domain-Specific Modeling Languages: Collected Experiences. Dans : *Proceedings of the 4th Workshop on Domain Specific Modeling DSM04 at OOPSLA 2004*. – Vancouver, Canada, 2004.

[LO00] VAN DER LINDEN F., OBBINK H. – ESAPS Engineering Software Architectures, Processes, and Platforms for System Families. – Dans : *Proceedings of the 3rd International Workshop on Software Architecture for Product Liness IW-SAPF-3*. – Las Palmas de Gran Canaria, Espagne, 2000. p. 244-252

[Lud03] LUDEWIG J. – Models in Software Engineering – An Introduction. - *Software and System Modeling*. – 2003, Vol. 2, No. 1, p. 5-14

[MA02] MUTHIG D, ATKINSON C. – Model-Driven Product Line Architectures. – Dans : *Proceedings of the 2nd Software Product Line Conference SPLC2*. – San Diego, CA, Etats-Unis, 2002.

[Man02] MANNION M. – Using First-Order Logic for Product Line Model Validation. – Dans : *Proceedings of the 2nd Software Product Line Conference SPLC2*. – San Diego, CA, Etats-Unis, 2002.

[McG04] MCGREGOR J. – Domain * - *Journal of Object Technology*. [En ligne] – 2004, Vol. 3, No. 7, p. 71-81. – Disponible sur Internet : http://www.jot.fm/issues/issue_2004_07/column6

[Me04] MERLE N. – Un méta-modèle pour l'automatisation du déploiement d'applications logicielles. - – Dans : *Première Conférence francophone sur le Déploiement et la (Re)-configuration des logiciels DECOR'04*. – Grenoble, France, 2004

[MFJ05] MULLER P-A., FLEUREY F., JÉZÉQUEL J-M. – Weaving Executability into Object-Oriented Meta-Languages. – Dans : *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems MoDELS/UML 2005*. – Montego Bay, Jamaïque, 2005.

[MHS03] MERNIK M., HEERING J., SLOANE A. – *When and How to Develop Domain-Specific Languages*. [En ligne] – Centrum voor Wiskunde en Informatica, Interactive Software Development and Renovation, 2003. - 33p. Rapport Technique SEN-E0309 – Disponible sur Internet : http://db.cwi.nl/rapporten/

[ML+97] MAEDA C., LEE A., MURPHY G., KICZALES G. – Open Implementation Analysis and Design. – *ACM SIGSoft Software Engineering Notes*. – 1997, Vol. 22, No. 3, p. 44-52

[OB02] VAN OMMERING., BOSCH J. – Widening the Scope of Software Product Lines – From Variation to Composition. - Dans : *Proceedings of the 2nd Software Product Line Conference SPLC2*. – San Diego,CA, Etats-Unis, 2002.

[Omm01] VAN OMMERING R. – Roadmapping a Product Population Architecture. - Dans : *Proceedings of the 4th International Workshop on Product Family Engineering PFE-4*. – Bilbao, Espagne, 2001.

[Par01] PARNAS D. – On the Design and Development of Program Families. – *Software Fundamentals: Collected Papers by David L. Parnas*. Edité par D. Hoffman et D. Weiss. – Boston, MA, Etats-Unis: Addison Wesley Longman Inc. 2001. p. 193-213

[P-D90] PRIETO-DÍAZ R. – Domain Analysis: An Introduction. – *ACM SIGSoft Software Engineering Notes*. - 1990, Vol. 15, No. 2, p. 47-54

[RBY+04] RAZAVI R., BOURAQADI N., YODER J., PEROT J-F., JOHNSON R. – Language support for Adaptive Object-Models using Metaclasses. - Dans : *Proceedings of the 12th European Smalltalk User Group Conference Research Track ESUG 2004*. – Köthen, Allemagne, 2004.

[Rev96] REVAULT N. – *Principes de méta-modélisation pour l'utilisation de canevas d'applications à objets (METAGEN et les frameworks)*. – Thèse de Doctorat – Université Paris 6, France, novembre 1996 – 315 p.

[RFB+01] RIEHLE D., FRALEIGH S., BUCKA-LASSEN D., OMOROGBE N. – The Architecture of A UML Virtual Machine. - Dans : *Proceedings of the 16th Conference on Object-Oriented Programming Systems, Languages, and Applications OOPSLA 2001.* – Tampa, FL, Etats-Unis, 2001. – p. 327-341

[Rie03] RIEBISCH M. – Towards a More Precise Definition of Feature Models. - Dans : *Proceedings of the Workshop on Modelling Variability for Object Oriented Product Lines at ECOOP 2003.* – Darmstadt, Alemagne, 2003.

[RJ96] ROBERTS D., JOHNSON R. – Evolve Frameworks into Domain-Specific Languages. - Dans : *Proceedings of the 3rd Conference on Pattern Languages of Program Design PloP3.* – Monticello, IL, Etats-Unis, 1996.

[RSP03] RIEBISCH M., STREITFERDT D., PASHOV I. – Modeling Variability for Object-Oriented Product Lines– - Dans : *Workshop Reader of 17th European Conference on Object Oriented Programming ECOOP 2003.* – Darmstadt, Allemagne, 2003. - p. 165-178

[RY01] REVAULT N., YODER J. – Adaptive Object-Models and Metamodeling Techniques. - Dans : *Workshop Reader of 15th European Conference on Object Oriented Programming ECOOP 2001.* – Budapest, Hongrie, 2001. - p. 57-71

[Sei03] SEIDEWITS E. – What Models Mean. – *IEEE Sofware.* – 2003, Vol. 20, No. 5, p. 26-32

[SE05] SANLAVILLE S., ESTUBLIER J. – Mélusine: Un environnement de modélisation et de coordination de services. – *Revue d'ingénierie des Systèmes d'Information*. 2005, Vol. 10, No. 3, p. 29-48

[Sim95] SIMOS A. – Organization Domain Modeling (ODM): Formalizing the Core Domain Modeling Life Cycle. – *ACM SIGSoft Software Engineering Notes*. - 1995, Vol. 20, No. Special Issue, p. 196-205

[Sim97] SIMOS A. – Organization Domain Modeling and OO Analysis and Design: Distinctions, Integration, New Directions. – Dans : *Proceedings of the 3rd Conference on Smalltalk and Java in Industry and Education STJA97.* – Erfurt, Alemagne, 1997. – p. 126-132

[Ste94] STEYAERT P., *Open Design of Object–Oriented Languages: A Foundation for Specialisable Reflective Language Frameworks.* – Thèse de Doctorat. – Vrije Universiteit, Bruxelles, Belgique, 1994. – 243 p.

[TH02] THOMPSON J., HEIMDAHL M. – Structuring Product Family Requirements for n-Dimensional and Hierarchical Product Lines. – *Requirements Engineering Journal.* – 2002, Vol. 8, No. 1, p. 42-54

[Thi98] THIBAULT S. – *Langages Dédiés: Conception, Implémentation et Application.*- Thèse de Doctorat - Université de Rennes 1, France, octobre 1998 – 127 p.

[TS05] TOLVANEN J-P., KELLY S. – Defining Domain-Specific Modeling Languages to Automate Product Derivation: Collected Experiences. - Dans : *Proceedings of the 9th Software Product Line Conference SPLC 2005.* – Rennes, France, 2005.

[Vill03] VILLALOBOS J. – *Fédération de Composants : une Architecture Logicielle pour la Composition par Coordination.* – Thèse de Doctorat. – Université Joseph Fourier de Grenoble, France, juillet 2003. – 188 p.

[Wil01] WILE D. – Supporting the DSL Spectrum. – *Journal of Computing and Information Technology CTI.* – 2001, Vol. 9 No. 4, p. 263-287

[Wil03] WILE D. – Lessons Learned from Real DSL Experiments. – Dans : *Proceedings of the 36th Hawaii International Conference on System Sciences Track 9 HICSS 03.* – Hawaii, HI, Etats-Unis, 2003. – p. 325.2

[WL99] WEISS D., LAI C-T-R. – *Software Product-Line Engineering: A family-Based Software Development Process.* – 1er édition. – Addison Wesley Professional, 1999. – 448 p.

[WSG+05] WARBOYS B., SNOWDON B., GREENWOOD M., SEET W., MORRISON R., BALASUBRAMANIAM D., KIRBY G., MICKAN K. – An Active-Architecture Approach to COTS Integration. – *IEEE Software.*- 2005, Vol. 22, No. 4, p. 20-27

[YJ02] YODER J., JOHNSON R. – The Adaptive Object-Model Architectural Style. - Dans : *Proceedings of the 3rd IEEE/IFIP Conference on Software Architectures WICSA 2002.* – Montréal, Canada, 2002. – p. 3-27

[ZHJ03] ZIADI T., HELOUET L., JEZEQUEL J-M. – Modélisation de Lignes de Produits en UML.- Dans : *Proceedings of Langages et Modèles à Objets LMO 2003.* – Vannes, France, 2003.

[ZJ05] ZIADI T., JEZEQUEL J-M. – Manipulation de Lignes de Produits Logiciels : une approche dirigée par les modèles. – Dans : *Premières Journées sur l'Ingénierie Dirigée par les Modèles* [En ligne]. – Paris, France, 2005. – Disponible sur Internet : http://idm.imag.fr/idm05/