



Modélisation de l'Aspect Dynamique de la Variabilité dans les Lignes de Produits

Rapport de Stage

David SUAREZ

Juin 2011

Master 2 Recherche en Informatique

Encadrants : Joël CHAMPEAU, Stephen CREFF

Equipe : LISyC équipe IDM, ENSTA Bretagne

Résumé

Pour les développeurs des systèmes informatiques d'aujourd'hui la complexité est une problématique importante à gérer et ce, pendant tout le cycle de développement logiciel. Des approches comme l'ingénierie dirigée par les modèles et les lignes de produits logiciels proposent des solutions aux problèmes de gestion de la complexité, diminution de coûts et amélioration de la qualité du logiciel final. Dans ce document nous présentons une fusion de ces approches, comme résultat d'une étude de la bibliographie existante ainsi que de nouvelles propositions dans le domaine de la modélisation du comportement des lignes de produits logiciels.

La composition de modèles comportementaux est une tâche complexe liée à la construction d'une ligne de produits logiciels et également à la dérivation de ses produits finaux. Ayant comme objectif d'automatiser cette tâche, un Framework de composition de diagrammes de séquence UML 2.0 est présenté comme résultat des travaux réalisés pendant le stage de master recherche. Les motivations pour développer cet outil, liées à l'ingénierie du domaine, l'ingénierie d'application et l'évolutivité des lignes de produits, sont également exposées. Des exemples et des explications détaillées des algorithmes de composition ainsi que des transformations de modèles, sont proposées au lecteur pour lui permettre de mieux comprendre le processus de composition.

Sommaire

1. Introduction 4
2. IDM 6
   a. La (Méta) Modélisation 6
   b. La transformation de modèles 7
   c. Les Profils UML 8
3. Lignes de produits 9
   a. Variabilité 10
   b. Modélisation de la variabilité 11
      i. UML et des Profils 12
      ii. Méta-modèles 16
      iii. Diagrammes de caractéristiques 20
4. Composition comportementale 23
   a. Transformation « Diagrammes de Séquence -> Machines à état » 24
   b. Composition de Machines à Etat 27
      i. Composition Séquentielle 28
      ii. Composition Alternative 29
   c. Transformation « Machines à état -> Diagrammes de Séquence » 30
5. Diagrammes de caractéristiques et Ordre total 34
6. Outil Implémenté (Plugin RSA) 38
7. Travaux à venir 41
   a. Méta modèle Kermeta 41
   b. Composition de comportements en parallèle 42
8. Conclusion 43

Bibliographie 44

1. Introduction

Les systèmes informatiques complexes font partie de la vie quotidienne de la société moderne. Que ce soit pour retirer de l'argent dans un distributeur ou tout simplement pour téléphoner depuis notre mobile, nous interagissons avec des infrastructures complexes de jour en jour. Ces systèmes, composés à la fois d'autres sous systèmes, doivent être modélisés et analysés pour comprendre leur dynamique interne et externe, c'est-à-dire, les relations entre les parties qui les constituent et également la manière dont ces systèmes interagissent avec le monde extérieur.

L'ingénierie dirigée par les modèles (IDM) est une approche qui cherche à gérer cette complexité en ayant comme objectif de fournir aux architectes de ces infrastructures complexes, des moyens pour représenter, analyser et transformer des abstractions de ces systèmes et que l'on appelle «modèles».

De façon parallèle à cette nécessité de compréhension accrue lors de la conception des systèmes informatiques, il a toujours existé un intérêt lié à sa productivité économique. De nombreuses études ont été faites depuis le début même de la programmation(1), pour trouver des méthodes de développement qui permettent de réduire les coûts et le temps de développement d'un projet logiciel. Ce processus fait apparaitre l'approche « Ligne de produits logiciels », une alternative efficace pour gérer le développement basé réutilisation d'un ensemble de logiciels avec des propriétés, fonctionnalités ou composants en commun.

Nous pouvons apprécier donc, deux disciplines différentes qui travaillent ensemble aujourd'hui dans le domaine de la Modélisation des Lignes de Produits Logiciels. Dans la littérature, plusieurs langages, outils et méthodologies on été proposés pour représenter de manière abstraite ces ensembles de logiciels qui partagent des caractéristiques communes, en gardant une différentiation entre l'aspect statique (structurel) et l'aspect dynamique (comportemental). Ce dernier, étant au cœur du stage de recherche réalisé.

Le centre d'intérêt des travaux réalisés tout au long du stage est la construction incrémentale et les aspects évolutifs de l'aspect dynamique de la variabilité dans les Lignes de Produits Logiciels, c'est-à-dire, par exemple, la modification d'un modèle comportementale de base pour répondre à des changements au niveau des exigences. Pour ce faire, le concept de la composition comportementale, abordé par quelques articles scientifiques, a été étudié. L'objectif de ce document est de faire un parcours théorique de tous les concepts clés traités lors de cette introduction et de montrer le processus suivi tout au long du stage pour atteindre la proposition d'un Framework de composition comportementale.

Le rapport traite des points suivants : Dans le deuxième paragraphe, les bases du paradigme IDM ; troisième paragraphe, un parcours des concepts basiques des Lignes de Produits Logiciels ainsi que les différentes propositions pour les modéliser; quatrième paragraphe, la composition comportementale : transformation de modèles et algorithmes ; ensuite, dans le cinquième paragraphe le langage des diagrammes de caractéristiques agrémenté de la notion d'ordre totale ; sixième paragraphe, la définition d'un méta modèle qui lie un diagramme de caractéristiques à un ensemble de modèles de comportement (diagrammes

de séquence) et qui permet de faire de la composition comportementale ; ensuite dans le septième paragraphe, la description d'un plugin RSA qui implémente les sujets traités dans les paragraphes 3,4,5 et 6 ; finalement, dans le paragraphe huit, un résumé des travaux à venir et dans le paragraphe neuf, quelques conclusions du travail réalisé pendant ce stage de master recherche.

2. IDM

L'ingénierie dirigée par les modèles est une approche qui utilise les modèles comme base du cycle de développement logiciel. La tendance croissante de l'utilisation de ce paradigme aujourd'hui, peut être comparée à celle vécue par l'ingénierie du logiciel dans les années 80 comme décrit dans (2), lors du passage du paradigme de programmation structurée vers la programmation orientée objet. Depuis longtemps, les modèles ont commencé à être utilisés par les équipes de développement logiciel pour la représentation des solutions proposées à un problème donné. Son étude, le développement d'outils et de langages permettant de concevoir, représenter et transformer ces modèles, ont eu une croissante utilisation dans le monde académique et également dans la conception d'applications industrielles. Ci-après sont présentés quelques concepts de l'ingénierie dirigée par les modèles.

a. La (Méta) Modélisation

Même si les réponses à « qu'est-ce qu'un modèle ? » et de « qu´est-ce que modéliser ? » sont loin d'être unanimes, comme l´a exposé Jochen Ludewig dans(3), dans cette section, l'objectif est de familiariser le lecteur avec les concepts fondamentaux du paradigme IDM. Selon (4), un modèle est « une simplification d'un système, construite avec un objectif spécifique et qui doit permettre la résolution de problèmes de la même façon que le système d'origine l'aurait fait ». Cette abstraction du système consiste en une extraction des concepts les plus importants, une identification de ses propriétés principales et des relations entre ces concepts(4). Pour ce faire, la définition d'un langage de modélisation est nécessaire. Ce processus fait partie de la méta-modélisation et est un concept fondamental de l'ingénierie dirigée par les modèles(5). La méta-modélisation permet aux équipes de développement de définir des langages de modélisation spécifiques au domaine étudié, et de cette manière, d'augmenter l'expressivité et la puissance d'abstraction des modèles construits à partir de ces langages (Méta-modèles). Par la suite, un exemple de cas de modélisation et méta-modélisation est fourni pour permettre au lecteur de mieux appréhender la suite du document.

Dans notre exemple, le système à modéliser est un téléphone portable : Pour commencer, nous allons identifier certaines informations importantes sur un **téléphone** : le *modèle* (3GS), la *marque* (Apple), la *couleur* (noir), la *taille*, etc. Nous pouvons dire également qu'un téléphone est composé de plusieurs éléments comme : un **écran**, une **antenne**, un **appareil photo**, et pour chacune de ces sous-parties nous pourrions identifier, encore une fois, des informations importantes qui vont nous aider à comprendre le système une fois modélisé. Nous avons identifié quelques concepts clés (en gras), quelques attributs ou informations concernant ces concepts (en italique) et une relation entre certains de ces concepts (souligné). Maintenant, nous avons tout ce qu'il nous faut pour construire un modèle, très simple, de notre système (Figure 1).

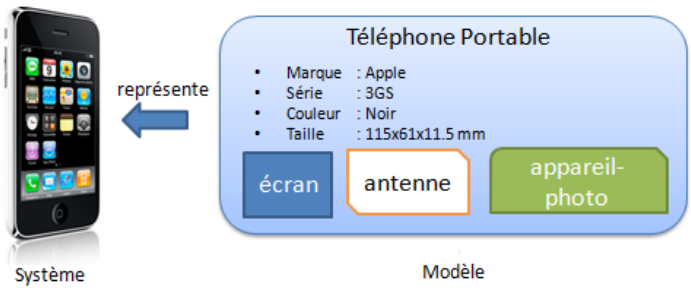


Figure 1. Modélisation simple d'un système

Maintenant nous pouvons dire que notre modèle représente notre système. Ensuite, considérons que l'intention est de trouver les concepts de base qui ne vont pas nous permettre de modéliser seulement des téléphones portables mais, de manière plus générale, des systèmes. Ici nous allons expliciter les concepts qui sont utilisés pour décrire notre modèle et donc créer un modèle du langage de modélisation qui sera utilisé pour le représenter. Nous pouvons donc dire, que notre modèle **représente** notre système et qu'il est **conforme à** un méta-modèle (Figure 2).

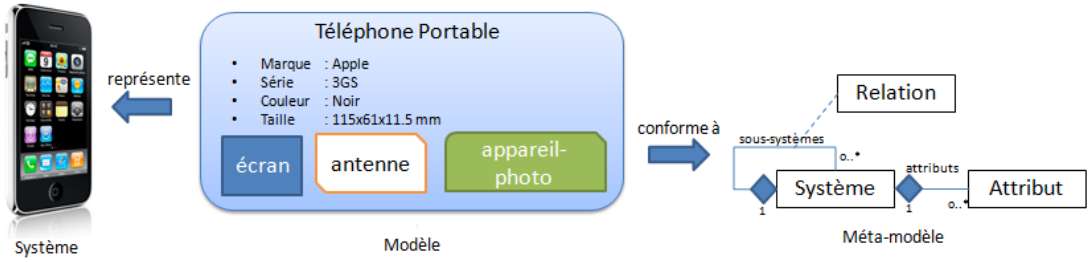


Figure 2. Système, Modèle et Méta-modèle

b. La transformation de modèles

Maintenant que les concepts de modèle et méta-modèle sont clarifiés, nous allons présenter un autre concept clé de l'IDM : la transformation de modèles. Selon (2) transformer un modèle, c'est le convertir en un autre. Une transformation est une séquence de règles bien définies qui prennent un modèle en entré (*source model*) et, en utilisant une logique de transformation, génèrent un nouveau modèle (*target model*) (6). Le processus de transformation peut conserver ou pas le même méta-modèle entre le modèle d'origine et le modèle transformé. Dans le premier cas la transformation est dite endogène. Dans le deuxième cas, la transformation est de type exogène. Certaines transformations sont de type M2M (modèle vers modèle), d'autres sont de type M2T (modèle vers texte) comme la génération de code. Pour une classification plus complète des différents types de transformations voir (6). Pour mieux comprendre le concept de transformation de modèles, voyons un exemple concret de cette procédure : Notre modèle d'entrée est le modèle du téléphone portable présenté dans la Figure 1 et la sortie attendue est une description textuelle de notre système. Notre transformation contient trois règles :

- *void Transformation(Système s)* : Prend le modèle d'entrée, initialise la sortie, appelle rootTransform() et génère le fichier txt de sortie.
- *string rootTransform(Système s)* : Transforme un système en une chaine de caractères.

- *string getAttributes(Système s)* : Transforme la liste des attributs d'un système en chaîne de caractères.

Selon la définition de ces trois règles, notre transformation (en pseudo code) et son résultat sont:

```
void Transformation(Système s){
  string str = "Système : "+s.name;
  str = rootTransform(s);
  imprimerFichier(str);
}

string rootTransform(Système s){
  string str = "Système : "+s.name;
  str += getAttributes(s);
  foreach(ss in sous-systemes){
     str += rootTransform(ss)
  }
}
```

```
…
Output:
```

str.txt

```
Système: Téléphone portable
Attributs: Apple 3GS Noir
115x61x11.5mm
Composé par :
        Système : écran
        Système : appareilPhoto
        Système : antenne
```

Nous supposons que l'implémentation des méthodes manquantes récupère les différentes valeurs comme chaînes de caractères et réalise le formatage pour obtenir le fichier de sortie montré.

c. Les Profils UML

Aujourd'hui, nous disposons de plusieurs langages de modélisation et d´outils qui implémentent ces langages. L'un des plus connus est l'UML (*Unified Modeling Language*). UML est un langage de modélisation défini et adopté par l'OMG (*Object Management Group*) comme un « standard pour représenter des spécifications structurelles, comportementales, architecturales et des structures de données et de business »[1]. L'extension du langage standard UML, est réalisée par le concept de Profil UML. Un profil UML est selon (7), «un ensemble de stéréotypes, *tagged values*[2] et contraintes, qui peuvent être utilisés pour étendre le méta modèle UML ». De cette manière nous pouvons modéliser un domaine spécifique grâce à l'utilisation d'un langage de modélisation généraliste avec une extension propre au domaine traité. Un exemple de profil UML est le suivant :

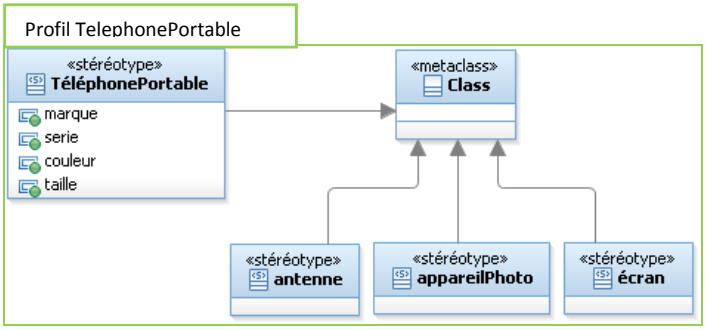


Figure 3. Profil UML téléphone portable

Grâce à la définition de ce profil UML, l'utilisateur peut annoter un diagramme de classes, dans cet exemple, avec les notions de *« TelephonePortable », « antenne », « appareilPhoto »*

[1] Définition disponible sur http://www.uml.org/

[2] Ce concept est valide dans la spécification UML 1.4 et il est remplacé par le concept de propriété depuis la version 2.0.

et *« écran ».* Les stéréotypes définis pourront être appliqués à toutes les classes d'un modèle UML qui importe ce profil, puisque ces stéréotypes étendent la méta-classe *Class.*

L'un des enjeux principaux de ce stage a été de trouver un langage de modélisation qui permette d'établir des modèles clairs et compréhensibles du comportement d'une ligne de produits logiciels, en nous permettant de raisonner sans ambiguïtés. Dans la section suivante, ce concept de Ligne de produits est exposé.

3. Lignes de produits

Suite aux premiers travaux sur la réutilisation proposés par McIlroy dans(8), David Parnas introduit le concept de « *Program Families* » dans (9), où il définit une ligne de produits (LdP) ou « *Program Family* », comme « un ensemble de programmes dont les propriétés communes sont si nombreuses que c'est avantageux d'étudier d'abord ses propriétés communes avant d'analyser les membres individuels de l'ensemble ». Ce concept, complété par des études postérieures sur : l'ingénierie du domaine (10), (11), les composants et la réutilisation avec UML (12), (13), les *features* (14), devient la référence pour le développement de composants réutilisables et également pour le développement logiciel basé-réutilisation. Dans la suite du document, les concepts « ligne de produits » et « famille de produits » seront traités de manière équivalente. Egalement, le professionnel des TICs qui travaille dans le domaine de la gestion des lignes de produits, est nommé Ingénieur LdP pour « *Ingénieur des Lignes de Produits* »

L'un des intérêts de cette approche est de réduire le coût et le temps de développement d'un ensemble de logiciels qui partagent des caractéristiques communes. Comme cela a été exposé dans (15) et (16), cette réduction du coût et du temps de développement a plusieurs origines. Premièrement, la réutilisation de modules existants réduit la durée de la phase de développement logiciel et d'autre part, la réutilisation de parties de l'application qui ont déjà été testées et vérifiées, réduit le risque de trouver des anomalies de fonctionnement et donc, réduit également le coût de maintenance du logiciel. Egalement, dans (15) et (16), d'autres motivations pour utiliser les Lignes de Produits Logiciels sont également exposées, comme l'amélioration de qualité du logiciel, de sa fiabilité et de sa sûreté.

Comme décrit dans(17), le processus de l'Ingénierie des Lignes de Produits comporte deux activités entrecroisées qui se complémentent et qui interagissent depuis la construction de la ligne de produits jusqu'à la construction des produits finaux. Ils sont brièvement décrits ci-dessous :

- **Ingénierie du domaine**(18) (*Domain Engineering*), appelé aussi « développement par réutilisation » : Les informations d'expériences précédentes concernant la construction de systèmes sont récoltées, stockées et organisées comme un ensemble d'artefacts réutilisables.
- **Ingénierie d'applications**(18) (*Application Engineering*), appelé aussi « développement par la réutilisation » : Est le processus complet de la construction des produits finaux en utilisant l'ensemble d'artefacts identifiés lors de la phase de l'ingénierie du domaine. Ce processus peut aussi être appelé « dérivation » des produits finaux (7).

Il y a différents modèles d'adoption et d'évolution d'une ligne de produits logiciels, comme cela a été exposé dans (10). Nous parlerons notamment de l'approche réactive et de l'approche proactive. Dans l'approche réactive, la construction de la ligne de produits se fait de manière incrémentale, c'est-à-dire, qu'il y a une configuration de base du processus d'implémentation et à partir de cette base, et de l'identification des points de variation existants, les membres de la famille de produits sont obtenus. L'approche proactive, part d'un ensemble de produits différents et à partir de cet ensemble la famille de produits est construite, grâce à un processus de généralisation de propriétés, caractéristiques et composants permettant d'identifier les éléments communs, ainsi que les points de variation de la ligne de produits. Ces deux approches ont été utilisées avec succès dans l'industrie : *Nokia Networks* pour l'approche *réactive* et *Philips Medical Systems* dans le cas de la stratégie *proactive*. Les expériences de ces deux projets, la manière dont les approches ont été implémentées ainsi que les résultats obtenus sont disponibles dans (15).

Dans les deux approches, l'identification des points communs ainsi que des différences entre les produits fait partie de la gestion de la variabilité. Pour mieux comprendre ce concept clé, la section suivante en donne une définition.

a. Variabilité

La variabilité dans les lignes de produits a été définie dans(19) comme le «regroupement des caractéristiques qui différencient les produits de la même famille ». Ce concept est lié à celui de «point de variation ». Un point de variation identifie la partie du système où une variation va être introduite dans la famille de produits, donnant naissance à un nouveau membre ou produit dans la ligne. Nous pouvons également adopter la définition donnée dans (15) d'un point de variation comme « la description de l'ensemble des différences entre les systèmes finaux d'une famille de produits ».

L'activité d'identifier, représenter, exploiter, implémenter et faire évoluer la variabilité dans une ligne de produits, est appelée la « gestion de la variabilité ». Puisque toute l'architecture de la ligne de produits et ses propriétés dépendent de la gestion de la variabilité, cette activité constitue l'enjeu principal lors de la conception d'une famille de produits (20).



Figure 4. Exemple de Ligne de Produits

Pour mieux comprendre la notion de ligne de produits, point de variation et variants, un exemple de ces concepts est donné dans la Figure 4. Cette ligne de produits contient un nombre de composants et de fonctionnalités de base, communes à tous les téléphones de la

famille, qui sont représentées par le téléphone simple qui se trouve à gauche de la figure. Ces fonctionnalités de base peuvent être p. ex. : L'envoi de SMS, le module d'appels, la gestion des contacts, etc. Ensuite, il y a trois décisions de conception ou configurations différentes, qui mélangent les options disponibles en termes de type d'écran et type de clavier. Finalement, dans la partie droite de la figure, nous pouvons observer trois produits dérivés à partir de notre ligne de produits. Dans cet exemple nous pouvons identifier deux points de variation, le clavier et l'écran, avec ses variantes, {azerty, pas de clavier, standard} et {inclinable, tactile, HD}, respectivement :

Pour compléter le concept de variabilité, une classification des différents types de variabilité dans les lignes de produits a été présentée dans (15):

- Similitude (*Commonality*) : C'est une caractéristique qui est commune à tous les produits de la famille. Cette caractéristique est implémentée comme partie de la plateforme ou noyau commun de la famille.
- Variabilité (*Variability*) : C'est une caractéristique qui peut être commune à quelques produits, mais pas à tout les membres de la famille de produits. Cette caractéristique doit être modélisée comme un point de variation et implémentée de manière à ce qu'elle soit présente seulement dans les produits sélectionnés.
- Produit-Spécifique (*Product-specific*) : C'est une caractéristique qui est incluse seulement dans un produit de la famille. Elle est généralement la réponse à une demande spécifique du client pour le produit concerné. Cette caractéristique ne sera pas intégrée dans la plateforme de la ligne de produits, mais doit être compatible avec celle-ci.

Par la suite, quelques propositions de modélisation des lignes de produits et de leurs variabilités sont exposées.

b. Modélisation de la variabilité

Après avoir défini le concept de variabilité, nous avons besoin de trouver un moyen de l'exprimer de manière claire, c'est-à-dire, de trouver un langage de modélisation qui nous permette d'inclure les concepts liés aux lignes de produits logiciel, pour ainsi représenter notre système.

La différentiation faite dans la suite du document, entre l'aspect statique et dynamique de la variabilité correspond à celle qui figure dans (7). Lorsque nous parlons d'aspect statique, l'objectif est de modéliser la structure de la ligne de produits, l'organisation de ses différents composants, points de variation et variantes, sans se focaliser sur le comportement et l'interaction de ces différents composants. De manière analogique, l'aspect dynamique fait référence au comportement du système, à la manière dont ses composants interagissent entre eux, en prenant en compte les différents points de variation et les différentes configurations disponibles des variantes de la famille de produits.

Dans le domaine de la modélisation de l'aspect statique, il y a de nombreux travaux realisés et en cours, p.ex. L'utilisation du langage UML et ses extensions (profiles), la proposition de méta-modèles et aussi l'utilisation des diagrammes de caractéristiques (*feature diagrams*). Dans la section suivante, ces différentes propositions sont exposées et, dans les cas des Profiles UML et les méta-modèles structuraux des lignes de produits des nouvelles

propositions, faites pendant ce stage, sont également présentées avec une justification de la nécessité d'améliorer les propositions existantes.

Egalement, du côté dynamique, le langage des diagrammes de séquence UML 2.4 et, encore une fois, l'utilisation des profiles UML comme moyen d'expression de la variabilité, seront présentés comme l'alternative courante pour modéliser l'aspect comportemental dans les lignes de produit logiciel. Comme l'objectif du stage se centre sur la construction incrémentale des lignes de produits logiciels de petite taille, d'autres propositions de modélisation orientées vers le domaine de la vérification formelle et les problèmes d'explosion combinatoire dans les familles de produits, notamment ceux présentées dans (21) et (22), ne sont pas étudiées dans ce document puisqu'elles ne sont pas dans la portée des travaux réalisés.

i. UML et des Profils

Dans cette section du document une approche UML + Profile UML est exposée. Selon la classification présentée dans (23) cette approche peut être considérée de type A1 : « *Annotation d'un modèle de base en utilisant des extensions* ». La première proposition de ce type a été faite par Clauss dans (24), (25), où il propose l'application de profiles UML pour étendre le méta-modèle du diagramme de classes UML. En utilisant les stéréotypes « *variationPoint* » et « *variant* » l'auteur inclut la notion de variabilité dans les diagrammes de classes. L'optionalité d'un élément du modèle peut être exprimée grâce au stéréotype « *optional* » et l'auteur propose également des *tagged values* pour spécifier le moment dans lequel la variabilité ou l'optionalité seront inclues dans la conception de la ligne de produits. Les méta-classes étendues par ces stéréotypes sont : *Class, Component, Package, Collaboration* et *Association*. La Figure 5 montre un exemple d'un diagramme de classes étendu avec le profil proposé. Ce travail constitue une approche purement statique et n'inclut pas des propositions de modélisation comportementale.

Postérieurement, l'équipe *Triskell* de Rennes dans (7) et (26), présente une approche similaire mais qui inclut en plus une extension du méta modèle des diagrammes de séquence. Concernant l'aspect statique, les stéréotypes qui ont été proposés sont « *optional* » : pour spécifier les éléments du modèle qui peuvent être omis dans certains produits de la famille, « *variation* » : pour identifier un point de variation et « *variant* » : pour marquer les alternatives ou choix résultants

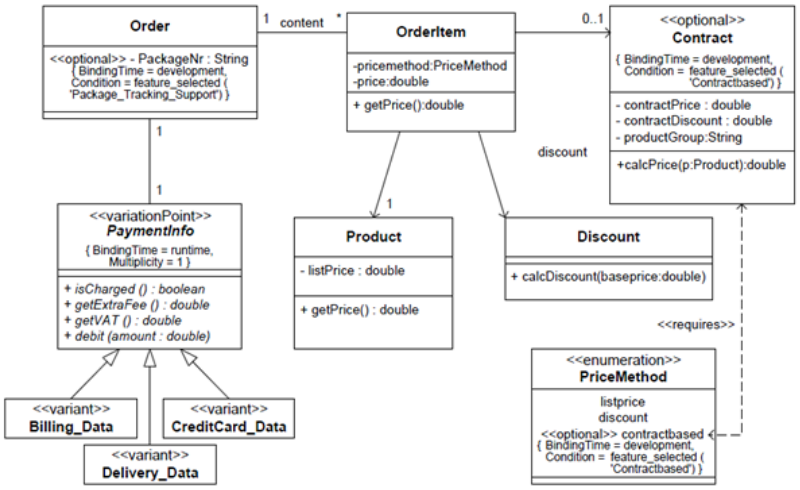


Figure 5. Diagramme de classes étendu avec le profil UML proposé par Clauss

de ce point de variation. Un point de variation est donc, représenté par une classe abstraite stéréotypé comme « *variation* » et par un ensemble de sous classes stéréotypés comme « *variant* ». Les méta-classes étendues par ce profil UML sont : *Package, Feature* et *Classifier*. Dans la Figure 6, nous pouvons observer un exemple de modélisation statique en utilisant ce profil UML.

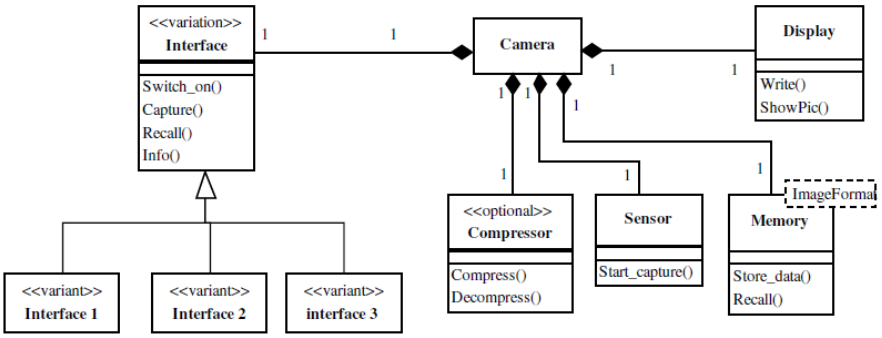


Figure 6. Diagramme de classes étendu avec le profil UML proposé par Ziadi et al.

Du côté de la modélisation dynamique, Ziadi et al. proposent également dans (7) une extension des diagrammes de séquence UML à l'aide des stéréotypes suivants : « *optionalLifeline* » et « *optionalInteraction* » : pour exprimer l'optionalité des éléments d'un diagramme de séquence, « *variation* » et « *variant* » : pour exprimer les points de variation et les variantes comme dans la partie statique, et finalement le stéréotype « *virtual* » : qui inclut la possibilité de redéfinir un comportement de plusieurs manières différentes dans les produits finaux de la famille. La Figure 7, est un exemple de ce que nous pouvons faire en termes de modélisation dynamique en utilisant le profil exposé. Ces travaux ont été implémentés initialement en utilisant la spécification UML 1.4 dans (7) et étendus grâce aux nouveaux operateurs des diagrammes de séquence inclus depuis UML 2.0 (*loop, alt, ref, etc.*) dans (27):

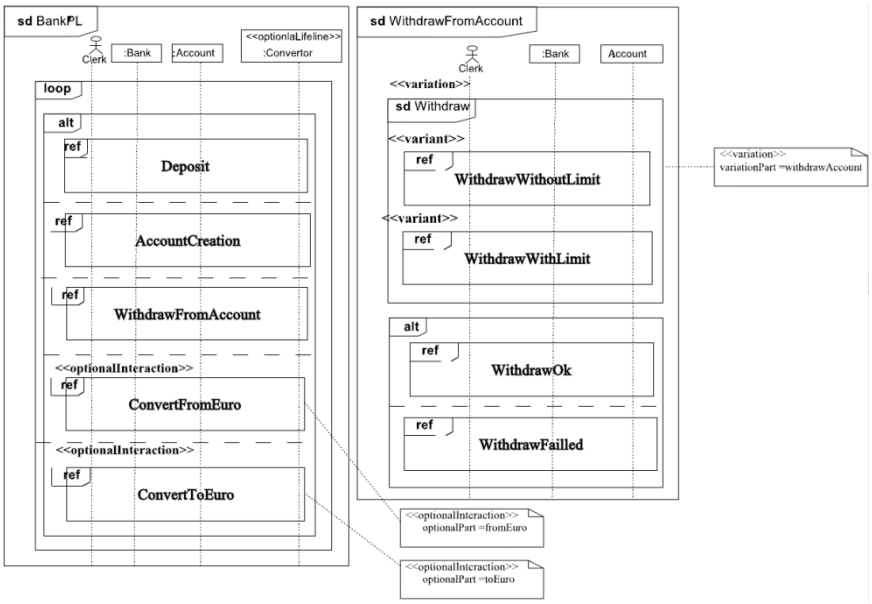


Figure 7. Diagramme de séquence étendu avec le profil UML proposé par Ziadi et al.

Ensuite, la Figure 8 donne un aperçu global du profil présenté :

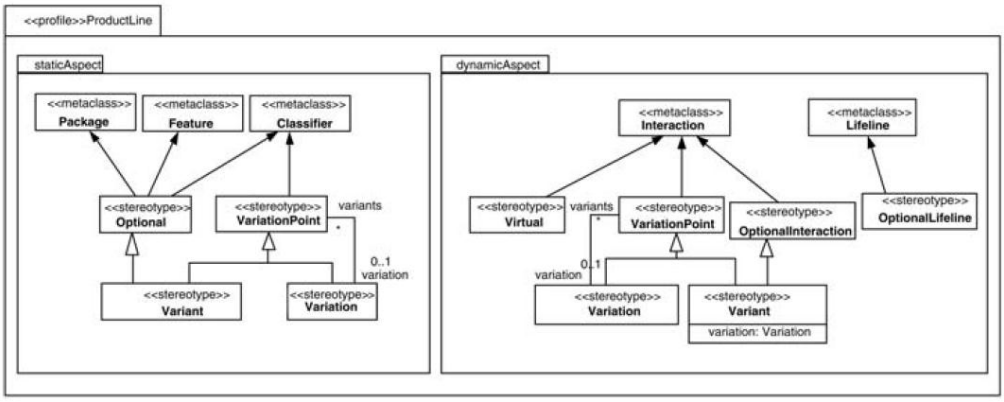


Figure 8. Aperçu global du profil proposé dans (7) et (26)

Des exemples d'utilisation d'un ensemble de stéréotypes, proposés par Gomaa, sont disponibles dans (13). Ce travail ajoute plusieurs alternatives de modélisation des lignes de produits logiciel depuis différents points de vue. Du côté fonctionnel, l'auteur décrit la possibilité de marquer des cas d'utilisation avec les stéréotypes « *kernell* », « *optional* » et « *variant* » et des relations entre eux marquées avec des stéréotypes standard d'UML 2.4 : « *extend* » et « *include* ». Pour les diagrammes de classes, la variabilité est exprimée grâce à l'utilisation de l'héritage et des classes abstraites, accompagnées d'une collection de stéréotypes un peu plus large qui inclut ceux présentés auparavant ; y sont présents en plus les notions de « *control* », « *algorithm* », « *interface* » et « *entity* ». Pour une explication plus détaillée de la sémantique de chacun de ces stéréotypes voir (13). Le travail de Gomma inclut aussi des propositions pour étendre le méta modèle des diagrammes de machine à état UML en utilisant la notion d'héritage et des machines à état paramétrées. Comme ces travaux s'éloignent de notre centre d'intérêt ils ne sont pas davantage présentés dans ce document.

Dans (28), Sun et al. suivent une méthodologie similaire à celle de Gomaa et abordent la modélisation de la variabilité à partir de différents points de vue. La proposition des auteurs dans ce travail, couvre quatre des treize diagrammes disponibles en UML 2.4 : diagrammes de classes, d'activité, de séquence et de déploiement. Malgré cette diversité de propositions, cet article ne dispose pas d'un diagramme de profil UML qui permette au lecteur d'identifier de manière claire et précise, les méta-classes étendues et les relations entre les différents stéréotypes présents dans le profil utilisé.

En ayant comme objectif de réaliser une synthèse des travaux présentés lors de cette section, et en général dans la littérature disponible, une des premières tâches réalisées lors de ce stage a été la proposition d'un nouveau profil UML. Ce profil apparaît, non comme une addition à ceux qui existent déjà, mais plutôt comme une compilation de l'état de l'art en matière de modélisation statique et dynamique des lignes de produit logiciel.

L'avantage de ce profil par rapport à l'existant réside en ce qu'il regroupe les concepts communs aux travaux réalisés précédemment et propose un mécanisme d'extension complet mais en même temps simple et facile à comprendre. Ci-dessous, dans la Figure 9, nous présentons un aperçu du profil proposé. Ensuite, dans la Figure 10, le lecteur peut trouver une description plus détaillée des méta-classes *Classifier* et *InteractionFragment.*

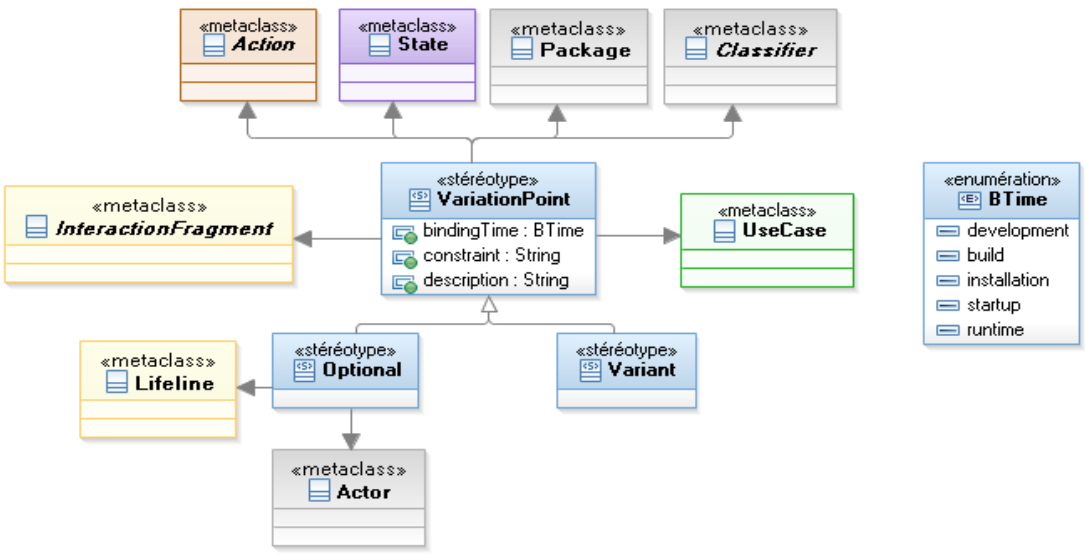


Figure 9. Profil UML proposé

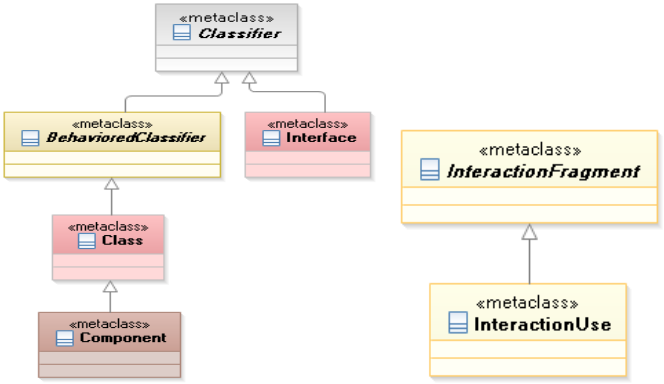


Figure 10. Méta-classes *Classifier* et *InteractionFragment*

Les diagrammes UML 2.4 sur lesquels ce profil peut être appliqué sont :

- Diagramme de Classes (méta-classes en rouge)
- Diagramme de Séquence (méta-classes en jaune)
- Diagramme d'Activités (méta-classes en marron claire)
- Diagramme de Composants (méta-classes en marron foncé)
- Diagramme de Cas d'Utilisation (méta-classes en vert)
- Diagramme de Machines à Etat (méta-classes en mauve)
- Diagrammes de Package

Les méta-classes en couleur argentée sont celles qui sont utilisées dans plusieurs diagrammes différents en même temps. Par exemple, la méta-classe *Actor* est utilisée dans les diagrammes de séquence mais aussi dans les diagrammes de cas d'utilisation.

ii. Méta-modèles

Dans cette section, quelques propositions de différents auteurs, concernant les méta-modèles de variabilité dans les lignes de produits, sont présentées.

Dans (29), Tessier et al. proposent un méta-modèle pour modéliser la variabilité. Ce méta-modèle, représenté par la Figure 11, inclut plusieurs concepts importants qui sont expliqués ci-dessous. Un « élément variable » est un élément qui a été spécifie comme variable directement par l'utilisateur. En revanche, un « élément variable propagé » est un élément qui acquière la qualité de variable grâce à ses relations avec d'autres éléments du modèle. La notion de « groupe de variation » regroupe un ensemble d'éléments variables et définit un type de regroupement entre eux. Dans l'énumération *VariationGroupKind* nous pouvons observer les différents types de regroupement identifiés par les auteurs. Par exemple, le type *équivalence* signifie que tous les éléments qui font partie du groupe doivent être soit présents soit absents dans le modèle à un moment donné. Pour une explication détaillée des autres types de regroupement voir (29). Ce méta-modèle n'inclut pas le concept d'optionalité et le concept de « relation » est implicite dans les éléments qui font partie d'un « groupe de variation ». Cela veut dire que pour exprimer une relation simple de dépendance entre deux éléments du modèle, il faut forcement créer un

nouveau group de variation. Un inconvénient de cette solution est le cas pour lequel nous avons un élément qui appartient à plusieurs groupes en même temps : la notation graphique pourrait devenir incompréhensible très rapidement, suivant si plusieurs éléments sont dans des groups avec des *VariationGroupKind* différents.

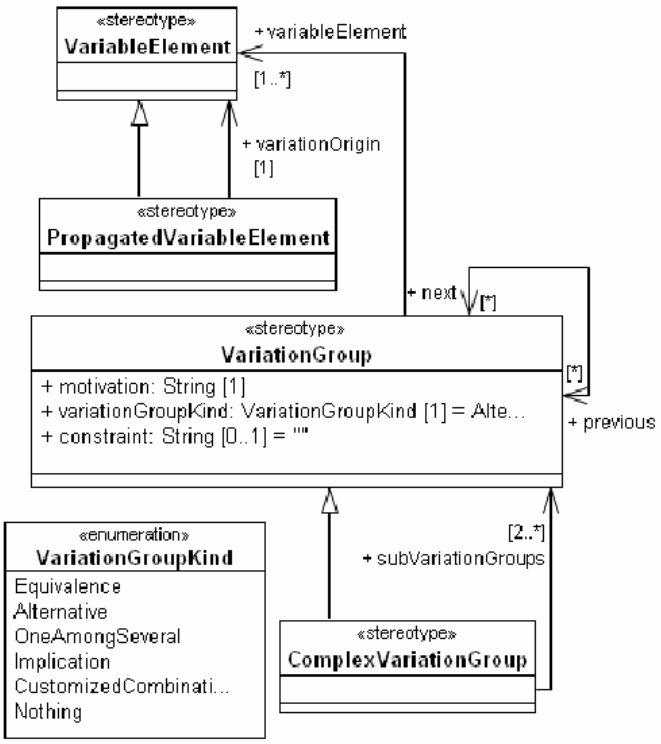


Figure 11. Méta modèle pour modéliser la variabilité (Tessier et al.)

Une proposition différente est faite par Morin dans (30) et est présentée dans la Figure 12. Cette proposition, à la différence de la précédente, inclut la notion d'optionalité. Les auteurs présentent tous les concepts importants d'une ligne de produits comme : point de variation, les éléments variables, les contraintes et les relations *and, or, xor*, etc. Ce méta-modèle présente aussi les operateurs « homogène » et « hétérogène ». Malgré la justification donnée par les auteurs pour inclure cette différentiation dans l'article, elle ajoute une complexité pas toujours nécessaire lors de la description d'une famille de logiciels qui augmente également la complexité des contrôles de cohérence et les contraintes à définir sur le méta-modèle. Pour nos besoins cette notion n'est pas nécessaire.

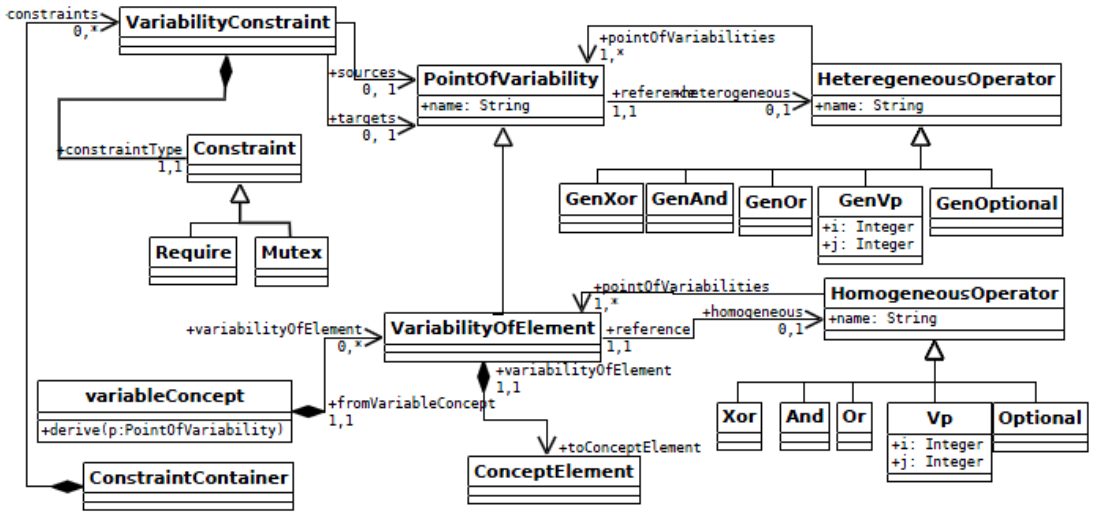


Figure 12. Méta-modèle variabilité Morin et al. (30)

Un méta-modèle des Lignes de produits logiciels, qui inclut également la notion de variabilité, a été proposé pendant ce stage de recherche. Ce méta-modèle n'a pas la prétention d'être utilisé à une échelle industrielle mais plutôt de couvrir quelques cas d'étude. A suivre, nous présenterons les concepts principaux ainsi que les relations identifiées dans ce méta-modèle. Un aperçu du méta modèle est disponible dans la Figure 13.

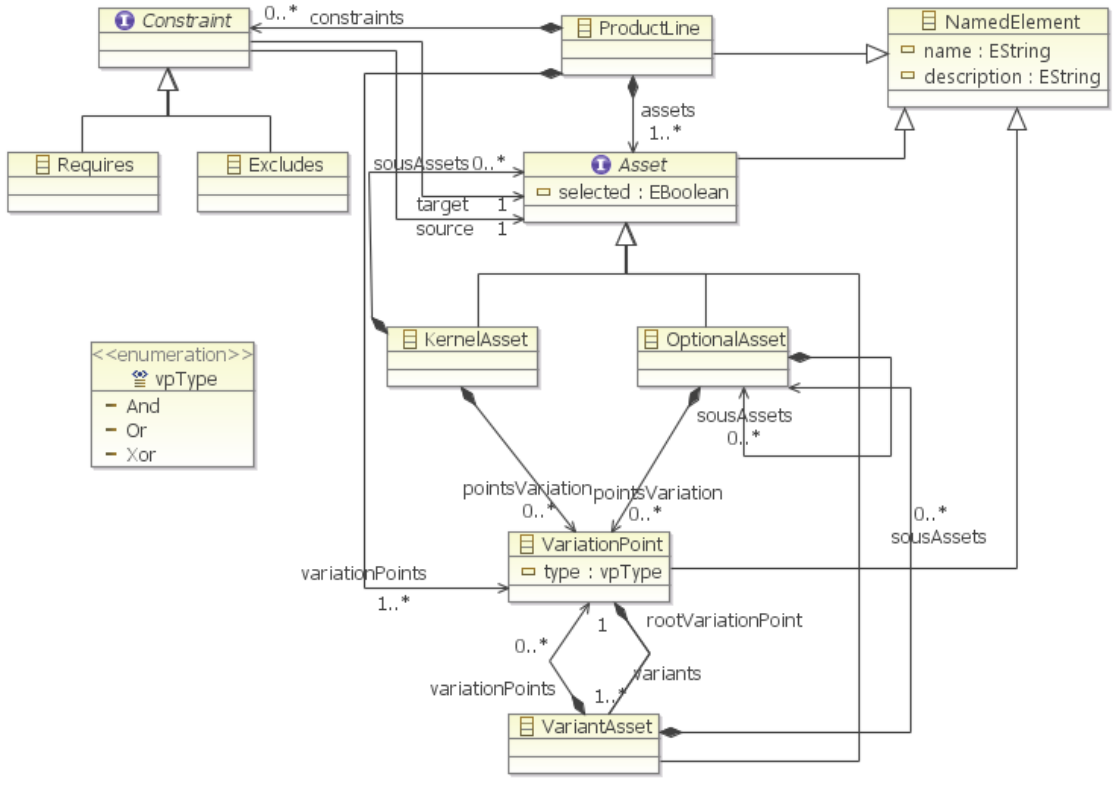


Figure 13. Méta modèle LdP proposé

Le point de départ est la définition du concept de « Ligne de Produit ». Une ligne de produits est composée par un ensemble de « *assets* ». Un *asset* est un artefact, un module ou une partie de la famille. Un *asset* peut être : commun à tous les membres de la famille (*KernelAsset*), présent seulement dans quelques uns d'entre eux (*OptionalAsset*).et aussi il y a des *assets* qui représentent les différentes variantes d'un point de variation (*VariantAsset*). Une ligne de produits inclut également plusieurs points de variation « *VariationPoint* » et ceux-ci sont liés aux *VariantAsset* qui implémentent les différentes alternatives produites lors de l'inclusion du point de variation dans la ligne de produits. Un point de variation peut être de différents types : *And*, tous les variantes de ce point de variation doivent être présentes dans les produits de la ligne ; *Or*, l'utilisateur a le choix entre les variantes et finalement *Xor*, qui implique qu'il existe une exclusion mutuelle entre les variantes, c'est-à-dire, que si y en a une qui est sélectionnée, les autres ne doivent pas l'être de manière simultanée. Finalement, nous trouvons les contraintes du méta-modèle qui sont des relations d'exclusion mutuelle et de dépendance entre les *assets*. Les méta-classes « *ProductLine* », « *Asset* » et « *VariationPoint* » héritent les attributs *nom* et *description* de la méta-classe *NamedElement.*

Il y a un comportement particulier quand nous utilisons les *assets* de type *Kernel* ou *Optional* et l'opérateur And. Ce comportement est illustré dans la Figure 14 et est décrit par la suite. Si en dessous d'un *asset* optionnel il y a un point de variation de type And, les variantes de ce point de variation doivent être considérées comme optionnelles également. Le même comportement se reproduit avec les *assets* de type Kernell.

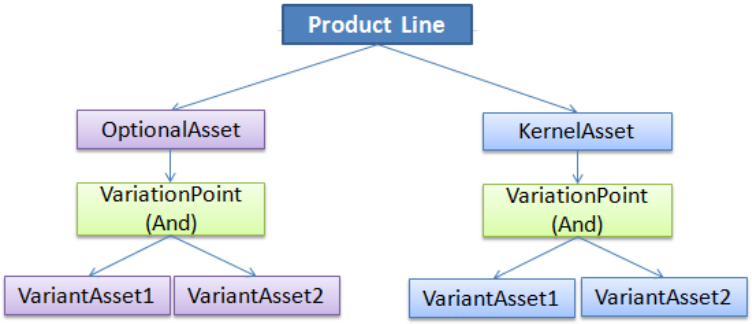


Figure 14. Exemple Ligne de Produits

D'autres contraintes plus complexes, utilisées pour valider la cohérence de la ligne de produits, seront présentées ci-après en utilisant le langage OCL[3].

Quelques exemples des contraintes proposées pour ce méta modèle sont :

- Dans la ligne de produits, tous les *assets* de type *kernel* sont sélectionnés :

```
Context : KernelAsset
self.selected
```

[3] Object Constraint Language. Voir http://www.omg.org/spec/OCL/

- Dans une relation d'exclusion, si la source est sélectionnée la destination ne doit pas l'être et vice-versa.

```
Context : Excludes
not ((self.source.selected and self.target.selected) or
(self.target.selected and self.source.selected))
```

- Si un *asset* a besoin d'un autre et si la source de cette relation est sélectionnée, la destination doit l'être également.

```
Context : Requires
if self.source.selected then
        self.target.selected
else
        false
endif
```

- Pour un point de variation : s'il est de type *And* toutes ses variantes sont sélectionnées ; Or, il y a au moins une variante sélectionnée; *Xor*, il y a seulement une variante sélectionnée.

```
Context : VariationPoint

if self.type = vpType::And then
     if self.variants->forAll(v:VariantAsset | v.selected) then
        true
     else
        self.variants->forAll(v:VariantAsset |not v.selected)
     endif
else
     if self.type = vpType::Or then
        self.variants->select(v:VariantAsset | v.selected)->size()>=1
     else
        if self.type = vpType::Xor then
           self.variants->select(v:VariantAsset | v.selected)->size()=1
        else
           false
        endif
     endif
endif
```

iii. Diagrammes de caractéristiques

Une troisième alternative de modélisation de la variabilité lignes de produits est présentée dans cette section. Il s'agit de l'utilisation des diagrammes de caractéristiques ou *feature diagrams*, qui permet à l'utilisateur d'exprimer de manière arborescente la structure d'une famille de produits, ses différents points de variation et variantes. Egalement, l'objectif de l'utilisation des diagrammes de caractéristiques est d'exprimer les relations existantes entre les différentes fonctionnalités ou *features* du système. Cette approche a été largement utilisée et plusieurs travaux ont été proposés pour compléter la notation de base presentée par Kang et al. dans (31). La Figure 15, montre un résumé de ces variantes de notation.

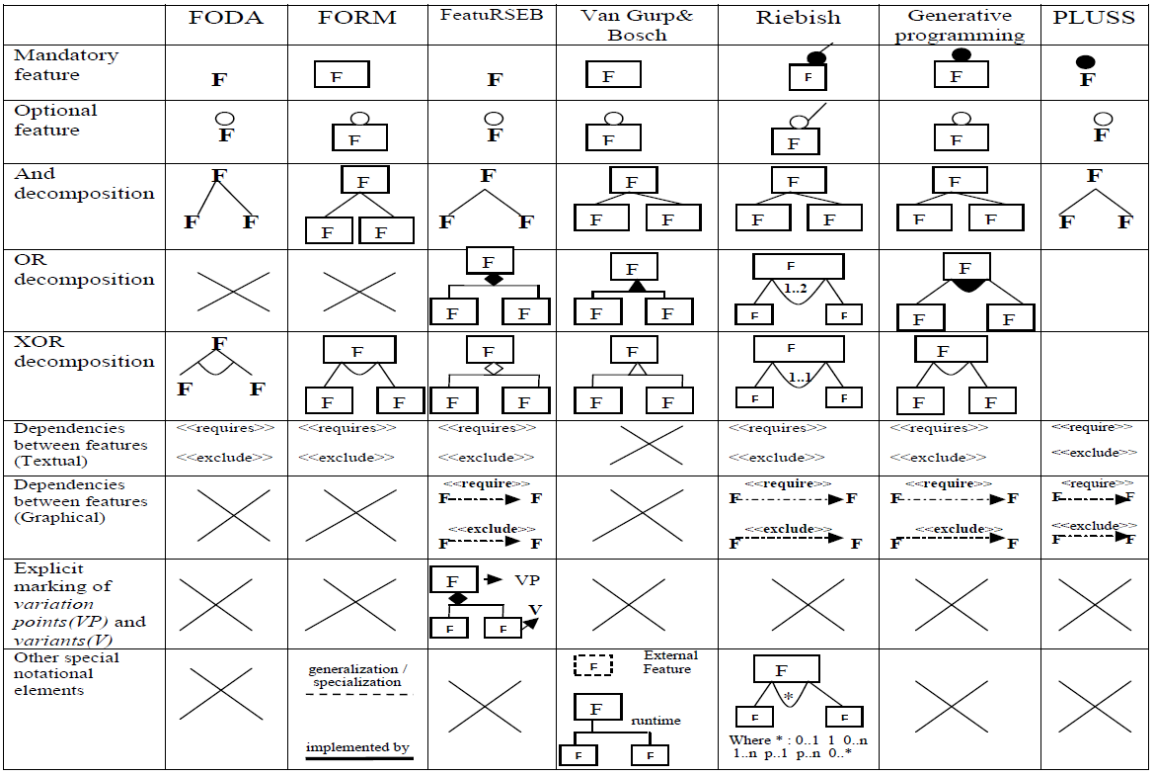


Figure 15. Résumé des propositions de notation pour les diagrammes de caractéristiques (23).

Par la suite, un exemple de l'utilisation de cette notation graphique est présenté pour illustrer la façon dont les lignes de produits sont représentées en utilisant un langage similaire à celui proposé par Kang et al. dans (32) :

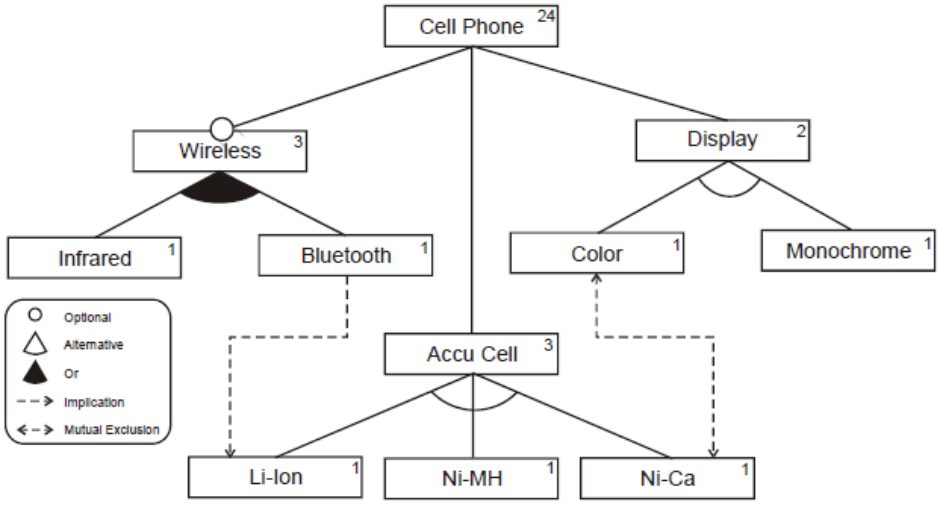


Figure 16. Exemple d'un diagramme de caractéristiques(33)

Dans cet exemple, nous pouvons voir une version un peu plus élaborée de notre téléphone portable. Les *features* marquées avec un cercle blanc par-dessus, sont considérées comme optionnelles, c'est-à-dire qu'elles peuvent être inclues ou pas dans certains produits. Le reste des features, qui ne sont pas marquées comme optionnelles, sont considérées comme obligatoires et seront présentes dans tous les produits de la ligne. Dans le nœud « *Accu Cell* » qui correspond au type de

batterie incluse dans le téléphone, nous pouvons observer une relation de type alternative, c'est-à-dire, qu'un produit de la ligne ne peut inclure qu'un type de batterie parmi les trois proposés. Dans le cas des options de communication *Wireless*, nous retrouvons une relation de type « Or », c'est-à-dire qu'un téléphone de notre famille peut inclure soit la fonctionnalité de communication par infrarouge, soit par Bluetooth, soit les deux. Le fait que cette fonctionnalité soit optionnelle veut dire aussi que dans notre famille nous pouvons trouver des téléphones qui ne possèdent aucun des moyens de communication *Wireless*.

Dans la section Diagrammes de caractéristiques et Ordre total, un méta modèle des diagrammes de caractéristiques agrémenté de la notion d'ordre total est proposé pour guider la tâche de composition comportementale décrite dans la section suivante.

4. Composition comportementale

Nous allons définir la composition comportementale comme « *le processus par lequel nous obtenons un comportement complexe comme résultat de la composition d'autres comportements simples* ». Le fait d'envisager la description du comportement d'un système comme la composition des comportements de ses sous systèmes, suscite l'intérêt de traiter ce sujet lors du stage de recherche. Des motivations additionnelles sont la dérivation de produits finaux ou ingénierie d'applications (expliquée précédemment) et le caractère évolutif que nous devons trouver dans les systèmes d'aujourd'hui. Cette adaptabilité permet aux systèmes, et donc à ses modèles, de changer en même temps que son environnement et ainsi répondre de manière efficace aux changements quotidiens présents dans un projet de développement logiciel. Nous allons traiter deux exemples qui vont mettre en évidence la nécessité de cette procédure de composition. Nous allons continuer à travailler avec l'exemple du téléphone portable traité depuis le début de ce document.

Puisqu'il est plus simple de définir le comportement d'une petite partie d'un système que de le décrire en entier, l'ingénieur LdP chargé de modéliser le système fait un diagramme de séquence pour chaque composant « simple » du téléphone : antenne, appareil-photo, écran, etc. Ensuite il voudrait, grâce à un modèle de variabilité de sa ligne de produits (un *feature diagram* par exemple) générer automatiquement un diagramme de séquence incluant tous les comportements simples décrits précédemment et donc, construire une description comportementale de la famille de produits de manière incrémentale. Celui-ci constitue le premier cas où nous avons identifié la nécessité de faire de la composition de comportements.

Ensuite, supposons que le comportement de la ligne de produits ait déjà été modélisé et décrit à l'aide d'un diagramme de séquence UML. La conséquence d'un changement dans les exigences du cahier de charges est l'ajout de nouvelles fonctionnalités dans le comportement de la ligne de produits. Comment faire pour reporter ces changements de manière simple sur les différents artefacts du modèle comportemental sans avoir à les modifier un par un? Comment contribuer à l'évolutivité du système ? Il s'agit là d'un deuxième cas où l'utilisation des techniques de composition comportementale pourrait faciliter et optimiser une tâche de l'ingénieur LdP.

Plusieurs problèmes dans le domaine de la composition comportementale ont été identifiés par la pratique tout au long du stage et finalement également observés dans les travaux récents d'Istoan et al. dans (17). La définition d'un ordre lors de la dérivation de produits dans une famille de produits (ingénierie de l'application) ainsi que les opérateurs de composition ont été identifiés comme des points clés à traiter dans les travaux à venir dans l'Ingénierie de l'Application.

Ayant pour objectif de réaliser un Framework de composition de diagrammes de séquence UML avec pour motivation de base de proposer une solution concrète aux problèmes identifiés précédemment, plusieurs travaux existants ont été pris comme base de départ. Ces travaux comportent notamment des études de la composition de machines à état et la transformation de diagrammes de séquence en diagrammes de machines à état. Plusieurs modifications et évolutions ont été faites sur ces travaux initiaux et également un nouvel

algorithme de génération de diagrammes de séquence à partir de diagrammes de machine à état a été proposé.

Le processus envisagé et développé pour composer des diagrammes de séquence UML est celui décrit dans la Figure 17.

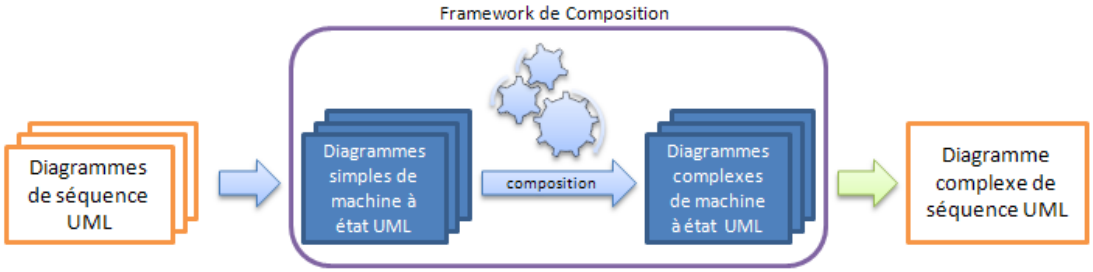


Figure 17. Framework de composition

Premièrement il y a une transformation qui génère des diagrammes de machine à état simples à partir des diagrammes de séquence de base. Ensuite, une composition algébrique de ces machines à état simples nous permet d'obtenir des diagrammes de machine à état complexes. Finalement, une transformation nous permet de passer de ces machines à état complexes vers un diagramme de séquence qui est le résultat de la composition des diagrammes de base. Cette orchestration de transformations déjà existantes ainsi que la définition de la dernière transformation (flèche verte) sont, jusqu'à maintenant, les résultats d'implémentation technique les plus importants de ce stage.

Dans les sections suivantes, une description de chacune de ces étapes est fournie pour éclaircir le processus de composition comportementale.

a. Transformation « Diagrammes de Séquence -> Machines à état »

Ce processus de transformation a été étudié et décrit dans (27) par Ziadi et al. Il est basé sur la notion de projection et les définitions de diagramme de séquence et de machines à état données dans cet article. Nous allons faire un parcours rapide de ces concepts pour permettre au lecteur de comprendre les algorithmes décrits dans ce chapitre du document. Pour plus d'informations le lecteur est invité à se référer à l'article d'origine.

Définition 1: *Un diagramme de séquence basique SD est un 6-uplet* $(E, \leq, \alpha, \phi, A, I)$*, où : E est un ensemble d'événements,* $\leq$ *est un ordre partiel imposé par les lignes de vie et les messages, A est un ensemble d'actions (émission et réception de messages), I est un ensemble d'objets qui participent à l'interaction et* α *et* ϕ *sont des fonctions qui associent respectivement un nom d'action et une localisation (i.e un objet affecté par l'événement) à un événement.*

Définition 2: *Une machine à état simple ST est un 6-uplet* $(S, s_0, E, A, \delta, J)$*, où : S est un ensemble d'états,* s_0 *est l'état initial, E est un ensemble d'événements, A est un ensemble d'actions,* $\delta \subseteq S \times E \times A \times S$ *est la relation de transition et* $J \subseteq S$ *un ensemble de Junction States. Ces Junction States sont des états similaires aux états finaux dans une machine à état standard mais ici seront utilisés comme états de fusion dans quelques opérations de composition.*

Définition 3: *Une projection $\pi_o(s)$ d'un diagramme de séquence SD sur un objet O, est la restriction d'ordre ≤ des événements situés sur la ligne de vie de O. Comme cette restriction est un ordre total, nous allons considérer la projection comme l'ensemble $\pi_o = e_1.e_2 \dots e_n$, tel que $\{e_1.e_2 \dots e_n\} = \phi^{-1}(O)$ et $e_1 < e_2 < \cdots e_n$. Où ϕ^{-1} est l'inverse de la fonction ϕ de la Définition 1.*

Le résultat de la transformation d'un diagramme de séquence est un ensemble de machines à état, une pour chaque ligne de vie (chaque objet), avec la projection des événements d'envoi et de réception de messages. La proposition faite dans l'article traite seulement les diagrames de sequence très simples et qui n'incluent pas les *Fragments Combinés (CombinedFragment)* et des *Operandes d'Interaction (InteractionOperand)*, concepts inclus dans la specification UML 2.4[4].

Dans la Figure 18, l'algorithme de base de cette transformation est présenté:

```
Algorithme: P(S,O)
Source: Un SD simple et un objet O
Target: Une machine à état ST_o = (S,s_o,E,A,δ,J)
-----------------------------------------------------
Créer l'état initial s_o
étatCourant := s_o
E:=∅; A:=∅;  S:={s_o}; J:=∅; δ:=∅
EvénementsProjectés :=π_o(s)
if EvénementsProjectés is empty then
        retourner (ST_o)
else
        for i=1 to |EvénementsProjectés| do
                e_i = EvénementsProjectés[i]
                Créer un nouveau état s; S= S ∪ (s)
                if e_i est un événement de réception then
                        E:= E ∪ (e_i)
                        Tr:=( étatCourant, ∅, e_i, s)
                        δ:=δ ∪ (Tr)
                else
                        if e_i est un événement d'énvoi then
                                A:= A ∪ (e_i)
                                Tr:=( étatCourant, e_i, ∅, s)
                                δ:=δ ∪ (Tr)
                        end if
                end if
                étatCourant:= s
        end for
        J := étatCourant
        retourner (ST_o)
end if
```

Figure 18. Algorithme de transformation SD->ST

Cet algorithme est exécuté pour toutes les lignes de vie du diagramme de séquence et produit une machine à état séquentielle avec des transitions d'envoi, notés avec le nom du message envoyé précédé du symbole « **!** », et des transitions de réception, notés avec le nom du message reçu précédé du symbole « **?** ». Un exemple de la transformation obtenue est montré dans la Figure 19. Ici nous pouvons observer un utilisateur qui allume son téléphone portable et ensuite le téléphone envoie une notification pour confirmer qu'il est allumé.

[4] Ces concepts ont été inclus dans le méta modèle des diagrammes de séquence UML depuis la version 2 .0. Pour une description détaillée voir (36).

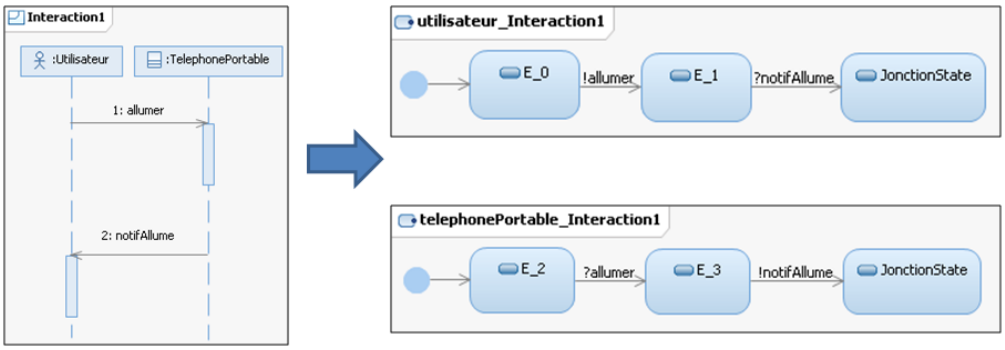


Figure 19. Transformation simple SD->ST

Cet algorithme avait déjà été implémenté par Ziadi et al. sous la forme de plugin RSA[5] comme résultat des travaux faits dans (34). Ce plugin a été utilisé comme base pour ensuite, ajouter la possibilité de transformer des diagrammes de séquence ayant des *Fragments Combinés*. Ceci permet le traitement de diagrammes plus complexes.

Comme exemple de résultat de cette amélioration, l'outil est en capacité de traiter des cas comme celui exposé dans la Figure 20.

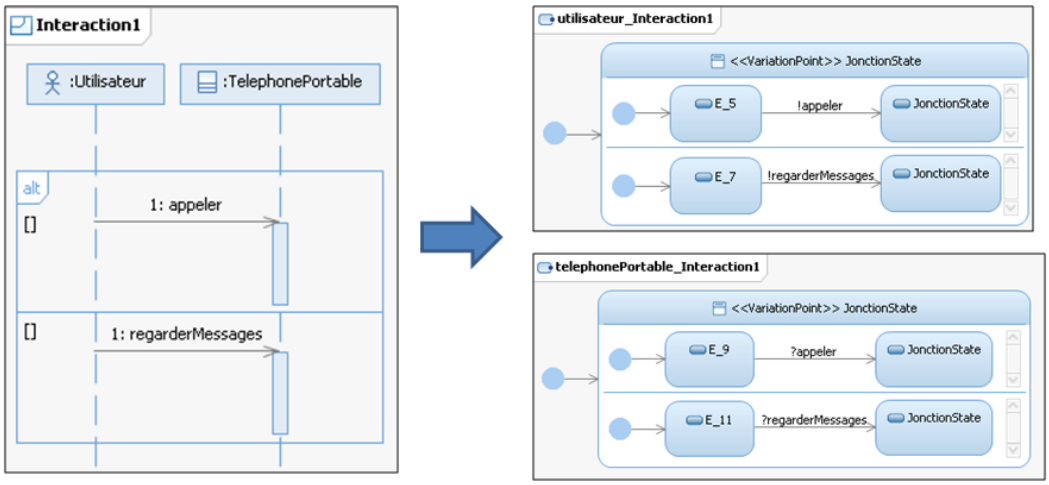


Figure 20. Transformation complexe SD->ST

La transformation d'un *CombinedFragment* donne comme résultat un état orthogonal stéréotypé comme « *VariationPoint* ». Ici nous utilisons le profil UML défini dans la section 3.b.i puisque JonctionState hérite de State, qui est une méta-classe étendue par le stéréotype « *VariationPoint* ». Ce stéréotypage est fait pour indiquer à l'utilisateur que la sémantique de l'état orthogonale à été changée, puisque normalement un état de ce type est sensé représenter plusieurs comportements qui s'exécutent en parallèle et dans ce cas nous voulons juste exprimer la possibilité de choisir entre les différents opérandes du *CombinedFragment.* Cette modification à été introduite parce qu'elle simplifie beaucoup l'implémentation de la transformation complexe des diagrammes de séquence.

L'extension faite à l'algorithme consiste à regarder d'abord si l'élément e_i, provenant de la projection $\pi_o(s)$, est vraiment un événement. Dans le cas des diagrammes de séquence

[5] IBM ®Rational Software Architect.

simples, cette validation est toujours vraie mais dans le cas des diagrammes de séquence complexes, le résultat de la projection peut être un fragment combiné. Ensuite, un traitement est fait sur e_i selon son type : s'il est un événement, nous procédons comme décrit auparavant ; sinon, s'il est un fragment combiné, l'algorithme décrit dans la Figure 21 montre la procédure exécutée. L'algorithme parcourt tous les opérandes du fragment combiné, pour chaque fragment il crée une nouvelle région dans l'état orthogonal. Ensuite, une projection du fragment est faite dans la région crée comme le fait apparaître la Figure 18. Dans cet algorithme, lorsque nous écrivons S_{op} p.e. nous faisons référence aux états de l'opérande *op* qui est en cours de traitement.

```
Algorithme: Génération d'un état orthogonal à partir d'un fragment combiné.
Source: Un fragment combiné fc
Target: Un état orthogonal O =(RO)
--------------------------------------------------------------------------
Créer l'état O
Regions de O, RO = ∅
for each Operand : op in fc.operands
        Créer une région r = (E_op, A_op, S_op, J_op, δ_op)
        RO := RO ∪ (r)
        Créer l'état initial dans r, s_ri
        étatCourant := s_ri
        E_op:= ∅; A_op:=∅; S_op:={s_ri}; J_op:=∅; δ_op:=∅
        EvénementsProjectés :=π_e(op)
        if EvénementsProjectés is empty then
                retourner (O)
        else
                for i=1 to |EvénementsProjectés| do
                        e_i = EvénementsProjectés[i]
                        Créer un nouvel état s dans la region r; S_op= S_op ∪ (s)
                        if e_i est un événement de réception then
                                E_op:= E_op ∪ (e_i)
                                Tr:=(étatCourant, ∅, e_i, s)
                                δ_op:=δ_op ∪ (Tr)
                        else
                                if e_i est un événement d'envoi then
                                        A:= A ∪ (e_i)
                                        Tr:=( étatCourant, e_i, ∅, s)
                                        δ_op:= δ_op ∪ (Tr)
                                end if
                        end if
                        étatCourant:= s
                end for
                J := étatCourant
        end if
end for
retourner(O)
```

Figure 21. Transformation de fragments combinés

Dans la section suivante, l'objectif est de montrer les operateurs de composition et leurs algorithmes utilisés dans notre Framework de composition des diagrammes de séquence.

b. Composition de Machines à Etat

Après avoir présenté la génération des machines à état dans la section précédente, nous allons continuer avec la description de la composition algébrique de ces machines à état. Nous allons traiter deux des operateurs définis par Ziadi et al. dans (27) et (34), notamment les operateurs de composition séquentielle et alternative.

i. Composition Séquentielle

Comme décrit dans (27), le résultat de la composition séquentielle de deux machines à état est une machine à état qui décrit le comportement de la première machine à état *suivi* par le comportement de la deuxième. Dorénavant cet opérateur sera noté : *seq*. Dans cette partie, nous allons utiliser les définitions fournies au début du chapitre.

La composition séquentielle de deux machines à état *ST1* et *ST2* = $(S, s_0, E, A, \delta, J)$, où :

- L'état initial de *ST1 seq ST2* est l'état initial de la première machine à état, si celle-ci n'est pas vide, ou le premier état de la deuxième dans le cas contraire.

$$s_0 = \begin{cases} s_0^1 \; si \; ST1 \neq \emptyset \\ s_0^2 \; sinon \end{cases}$$

- $$S = \begin{cases} S^1 \cup S^2 - \{s_0^2\} \; si \; (s_0^2 \notin J^2 \; \vee ST2 = \emptyset) \\ S^2 \; si \; ST1 = \emptyset \\ S^1 \cup S^2 \; autrement \end{cases}$$

- Les événements et les actions de *ST1 seq ST2* sont l'union de ceux existants dans chacun des opérandes.

$$E = E^1 \cup E^2$$

$$A = A^1 \cup A^2$$

- La composition séquentielle préserve toutes les transitions de ses opérandes, excepté les transitions sortantes de l'état initial de ST2 quand ST2 n'est pas un cycle. Pour faire la concaténation des deux machines à état, des transitions sont créées à partir de chaque JonctionState de la première machine à état vers tous les successeurs de l'état initial de la deuxième machine à état. Ceci peut être exprimé comme :

$$\delta = \delta^1 \cup (\delta^2 \cap S \times E \times A \times S) \cup \{(j, e, a, s) \in J^1 \times E^2 \times A^2 \times S^2 | (s_0^2, e, a, s)\}$$

Cette nécessité d'ordre, implicite dans l'operateur, a été identifiée comme une problématique lors de la composition de modèles comportementaux dans (17) puisqu'elle rend impossible l'obtention du même résultat en changeant l'ordre des opérandes. Dans cet article, la volonté est de trouver des operateurs de composition commutatifs pour automatiser au maximum la tâche de composition. Pour notre part nous considérons que, en sachant que l'origine de nos machines à état est un modèle séquentiel (diagrammes de séquence) où l'ordre est une des notions basiques, l'intervention de l'utilisateur ou d'un autre système pour définir l'ordre des operateurs est nécessaire. De plus, en utilisant une approche de construction incrémentale de la famille de produits l'ordre de la composition sera introduit progressivement et sera toujours connu au moment de la composition.

ii. Composition Alternative

La description de cet opérateur à été donnée également dans (27). Malgré sa simplicité conceptuelle, le résultat d'une composition alternative en utilisant l'algorithme proposé par les auteurs de cet article était difficilement utilisable pour la génération de diagrammes de séquence à partir de machines à état. Dans un article plus récent (34), les auteurs proposent un nouvel opérateur dit de « *composition parallèle* », qui cherche à exprimer la simultanéité d'exécution des machines à état données en entrée. Après quelques analyses concernant les possibles implémentations de la transformation de machines à état dans des diagrammes de séquence, nous avons décidé d'utiliser l'algorithme de composition parallèle et de modifier la sémantique du résultat, comme nous l'avons fait pour les états orthogonaux précédemment. Une explication détaillée est fournie par la suite :

Normalement une alternative était exprimée dans (27) comme montré ci-dessous :

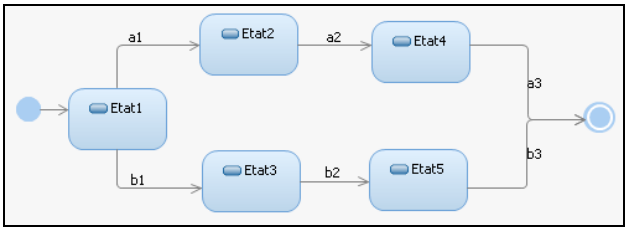


Figure 22. Alternative standard dans une machine à état

Etant dans l'état 1 la machine à état change son état courant à l'état 2 quand le message a1 est envoyé, p.ex. La modification faite consiste à représenter une alternative comme ceci :

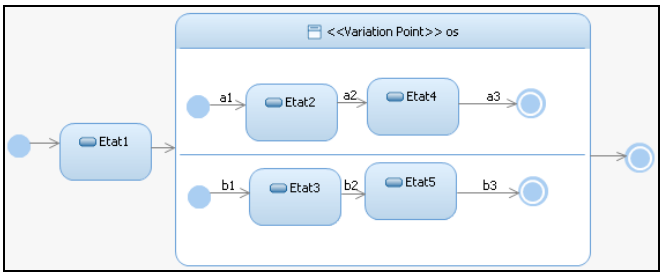


Figure 23. Alternative dans une machine à état en utilisant des états orthogonaux

Etant dans l'état 1, si le message envoyé est a1, la machine à état exécute le comportement spécifié dans la première région de l'état orthogonale *os* stéréotypé comme *« VariationPoint »*. Nous avons choisi cette représentation équivalente qui permet de faire des transformations et des compositions itératives et de cette façon rendre plus utilisable et extensible le Framework de composition proposé.

Pour comprendre l'algorithme de cet opérateur de composition, nous devons d'abord étendre la notion de machine à état simple donnée dans la Définition 2, pour inclure la notion d'états orthogonaux.

Définition 4 : *Une machine à état est une 11-uplet* $(SS, COS, s_0, E, A, J, \delta, \Re, \phi, subregions, \omega)$ *où : SS est un ensemble d'états simples, COS est un ensemble d'états orthogonaux,* s_0 *est un ensemble d'états initiaux,* $\Re$ *est un ensemble de régions, E est un ensemble d'événements, A est un ensemble d'actions, J est un ensemble de Junction States,* $\delta \subseteq SS \times E \times A \times SS$ *est une relation de transition,* $\phi \subseteq SS \cup \delta \rightarrow \Re$ *est une relation qui lie chaque élément à son conteneur,* $subregions \subseteq COS \times \Re$ *est une relation qui lie chaque état orthogonal aux les régions qu'il contient et* $\omega \subseteq SS \cup COS \cup \delta \rightarrow \Re$ *est une relation qui lie chaque élément avec son conteneur.*

Comme décrit dans (34), la composition alternative de deux machines à état SM1 et SM2 donne comme résultat une nouvelle machine à état avec un état orthogonal. Cet état contient deux sous-régions et dans chaque région nous mettons tous les éléments correspondants à chaque machine à état d'entrée. Dorénavant, cet opérateur sera appelé *alt*. SM1 alt SM2 = $(SS, COS, s_0, E, A, J, \delta, \Re, \phi, subregions, \omega)$ *où:*

- $s_0 = s_0^1 \cup s_0^2$ l'état initial de SM1 *alt* SM2 est l'union de ceux présents dans les deux opérandes.
- $SS = SS_1 \cup SS_2; A = A_1 \cup A_2; E = E_1 \cup E_2$, l'ensemble des états simples, des actions et des événements est l'union de ceux présents dans les deux opérandes.
- $\delta \subseteq S \times E \times A \times S = \delta_1 \cup \delta_2$, l'ensemble des transitions de SM1 alt SM2 est l'union de celles des deux opérandes.
- La machine à état résultante contient un nouvel état orthogonal *os (marqué avec le stéréotype « VariationPoint »*[6]*),* deux régions *r1* et *r2*. $COS = COS_1 \cup COS_2 \cup \{os\}; \Re = \Re_1 \cup \Re_2 \cup \{r1, r2\}$ et $subregions = subregions_1 \cup subregions_2 \cup \{(os, r1), (os, r2)\}$.
- La nouvelle région r1 contient tous les éléments de SM1 et r2 contient tous les éléments de SM2. C'est-à-dire, $\omega = \omega^1 \cup \omega^2 \cup \{(e1, r1) \in SS_1 \cup COS_1 \cup \delta_1 \times \{r1\}\} \cup \{(e2, r2) \in SS_2 \cup COS_2 \cup \delta_2 \times \{r2\}\}$.

Cette solution à été proposée comme une réponse à l'absence d'opérateurs qui permettent de faire de la composition des features alternatives, identifié dans (17) comme un problème courant dans le domaine de la composition comportementale des lignes de produits logiciel.

c. Transformation « Machines à état -> Diagrammes de Séquence »

Jusqu'à maintenant, nous savons générer des machines à état à partir des diagrammes de séquence UML et aussi composer ces machines à état pour décrire le comportement de notre famille de produit en utilisant un operateur séquentiel et un operateur alternatif. Dans cette section, nous allons expliquer la transformation proposée pour reconstruire un diagramme de séquence à partir d'un ensemble de machines à état. Cette transformation peut être vue comme l'inverse de la transformation montrée dans le chapitre 4.a.

[6] Modification par rapport à la version originale de l'algorithme pour changer la sémantique du résultat.

```
Algorithme: Génération d'un diagramme de séquence à partir d'un ensemble de machines à
état.
Source: Un package pkg contenant l'ensemble des machines à état
Target: Une interaction interact
-------------------------------------------------------------------------------------------
Variables :
interact  :=Ø, Interaction contenant les lignes de vie et les messages
lifelines :=Ø, Liste des lignes de vie de l'interaction finale
smQueues  :=Ø, Liste des transitions des machines à état en forme de Queues ordonnées
currentTr :=Ø, Transition courante
result    :=Ø, Stack contenant soit des transitions d'envoi suivies par sa transition de
               réception correspondante soit des états orthogonaux

Première partie : Exploration et stockage des transitions des machines à état

for each (Machine à état : st in pkg) do
       lifelines:=lifelines ∪ creerLifeLine(st.getName())
       smQueues:=smQueues ∪ construireQueue(st)
end for

Transition tr:=récupérer la première transition d'envoi dans smQueues
if (tr != null) then
       currentTr:=tr
       result.push(tr) //ajouter la transition au résultat
       process(result,smQueues) //appeler la fonction récursive process
else
       OrthogonalState os:= getOrthogonalState(smQueues)
       if (os != null) then
              result.push(os) //ajouter l'état orthogonal au résultat
              process(result,smQueues) //appeler la fonction récursive process
       end if
end if

Deuxième partie : Génération de l'interaction à partir des transitions récupérées

result := swap(result)
while (result is not empty) do
       if (result.peek() instance of Transition) then //traitement pour une transition
              String msg := result.peek().getName()
              Lifeline l1 := result.pop().getLifeline()
              Lifeline l2 := result.pop().getLifeline()
              interact.createMessage(msg,l1,l2)
       else
              if (result.peek() instance of OrthogonalState) then //traitement pour un
                     OrthogonalState os := result.poll()          //état orthogonal
                     interact.createCombinedFragment()
                     for each (Region : r in os) do
                            Queue q := r.getQueue()
                            while (q is not empty) do
                                   String msg := q.peek().getName()
                                   Lifeline l1 := q.poll().getLifeline()
                                   Lifeline l2 := q.poll().getLifeline()
                                   interact.createMessage(msg,l1,l2)
                            end while
                     end for
       end if
end while
```

Figure 24. Algorithme de génération d'une Interaction à partir de Machines à Etat UML

Pour comprendre l'implémentation de cet algorithme, montré en pseudo-code dans la Figure 24, nous allons le diviser en deux parties. La première consiste en la récupération de toutes les transitions des machines à état présentes dans le package d'entrée et de les stocker dans des structures de données de manière ordonnée. Ensuite ces structures, crées et peuplées dans la première partie, sont parcourues et, à partir des informations trouvées, une Interaction UML est générée. Plus de détails sont fournis par la suite, grâce à un exemple de génération d'une interaction à partir des machines à état suivantes :

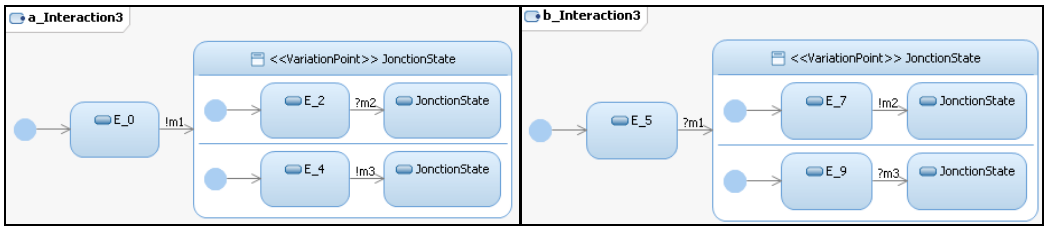


Figure 25. Machines à Etat d'exemple

La première structure de données utilisée pour stocker les transitions des machines à état est une liste de queues d'objets appelée *smQueues*. Ces objets peuvent être soit des Transitions soit des listes de queues de transitions (dans le cas des états orthogonaux). Pour les machines à état d'exemple la structure *smQueues* est :

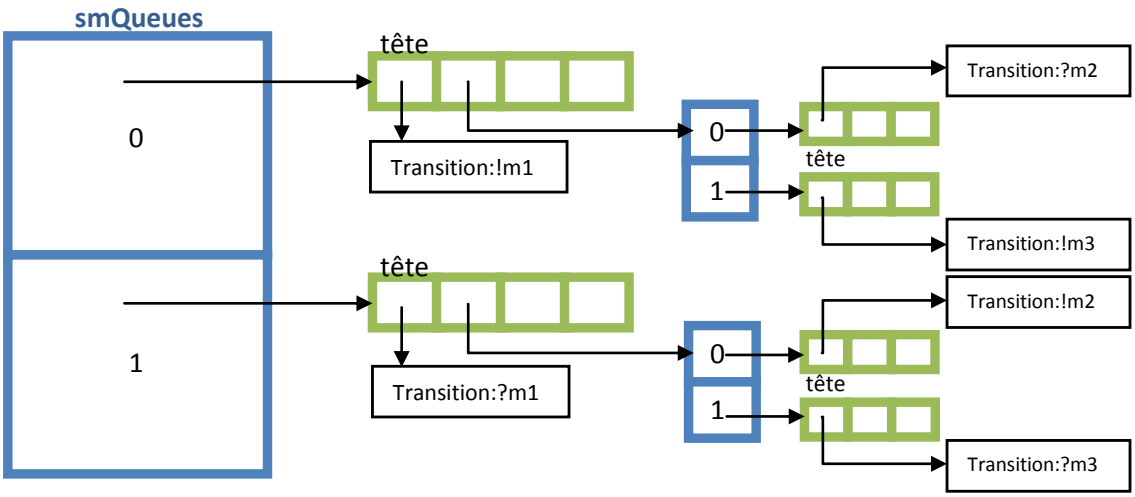


Figure 26. Structure de données *smQueues*

Dans cette figure (23) les structures verticales et dessinées en bleu sont des Listes et les structures horizontales et dessinées en vert sont des Queues.

Ensuite, la méthode récursive *process(result, smQueues)* est chargée de lire toutes les informations enregistrées dans *smQueues* et les ordonner dans la pile *result*. Pour notre exemple, la pile *result* après l'exécution et sortie de la méthode *process* (et après l'inversion de son ordre grâce à l'utilisation de la méthode *swap(result)*) est :

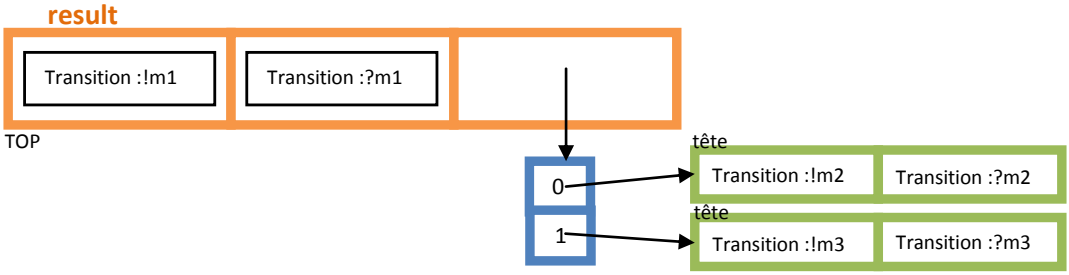


Figure 27. Structure de données *result*

Après avoir organisé les transitions comme montré dans la Figure 27, il ne reste qu'à parcourir cette structure et générer le diagramme de séquence en faisant la transformation

inverse de celle montrée dans la Figure 19 pour les diagrammes simples et de celle montrée dans la Figure 20 pour les diagrammes complexes. Ceci nous permet d'obtenir, pour notre exemple, le diagramme de séquence montré par la Figure 28 :

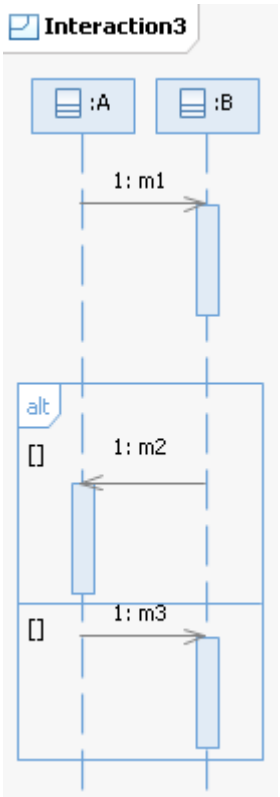


Figure 28. Résultat de la transformation ST->SD

5. Diagrammes de caractéristiques et Ordre total

Comme cela a été exposé dans (17), le fait de ne pas connaître l'ordre suivant lequel la composition comportementale doit être faite est un problème qui ne permet pas d'automatiser à 100% cette activité. Dans le cadre de la composition incrémentale des lignes de produits logiciels de petite taille, cet ordre est implicitement défini par le processus de construction de la ligne de produits. Dans le cadre d'une approche dite *A3* selon (23)*: « Connexion des diagrammes de caractéristiques avec des fragments du modèle »*, une solution proposée pendant ce stage est compatible avec une des solutions proposées dans (17). Cette solution consiste à fournir des informations additionnelles dans le diagramme de caractéristiques utilisé pour modéliser la variabilité (section 3.b.iii). Nous avons appelé cette solution « les diagrammes de caractéristiques agrémentés de la notion d'ordre total ». Dans cette section, l'objectif est de montrer comment cette approche peut résoudre le problème de l'ordonnancement des tâches de composition.

Inclure une notion d'ordre totale dans le diagramme de caractéristiques d'une ligne de produits de grande taille est une tâche complexe ; une solution rejetée dans (17). Cependant, puisque nous travaillons avec la notion de construction incrémentale, l'ingénieur LdP doit ajouter une par une les fonctionnalités ou les nouvelles variantes de la famille de produits.

Depuis le début de la construction de la famille de produits, l'ingénieur LdP a une idée de l'ordre dans lequel les différents composants doivent être exécutés. Cette notion d'ordre par rapport aux éléments déjà existants dans la famille est présente tout au long de la construction de la ligne de produits et nous permet donc de décrire un ordre total entre ses composants.

Définition 5 : *Soit (C, ≤) un ensemble ordonné de caractéristiques qui appartiennent à un diagramme D. La relation d'ordre total ≤ pour deux caractéristiques* $x, y \in C \mid x \neq y$*, est définie grâce à la relation finished-to-start (f2s) comme:*

$$x \leq y \rightarrow x\, f2s\, y$$

Cette relation finished-to-start, permet d'établir une dépendance d'exécution entre deux caractéristiques de l'ensemble. Explicitement, si une caractéristique x est en relation f2s avec une caractéristique y, cela signifie que pour s'exécuter, x doit attendre que y soit finie.

Selon cette définition, pour tout couple de caractéristiques différentes nous pouvons connaître l'ordre d'exécution grâce à la fonction binaire *f2s*.

Nous avons repris un méta-modèle des diagrammes de caractéristiques de base. Par la suite, le méta-modèle ainsi que las modifications incluses pour établir cette relation d'ordre total sont expliqués. Dans la Figure 29, un aperçu du méta-modèle proposé pour créer des diagrammes de caractéristiques agrémentés est présenté :

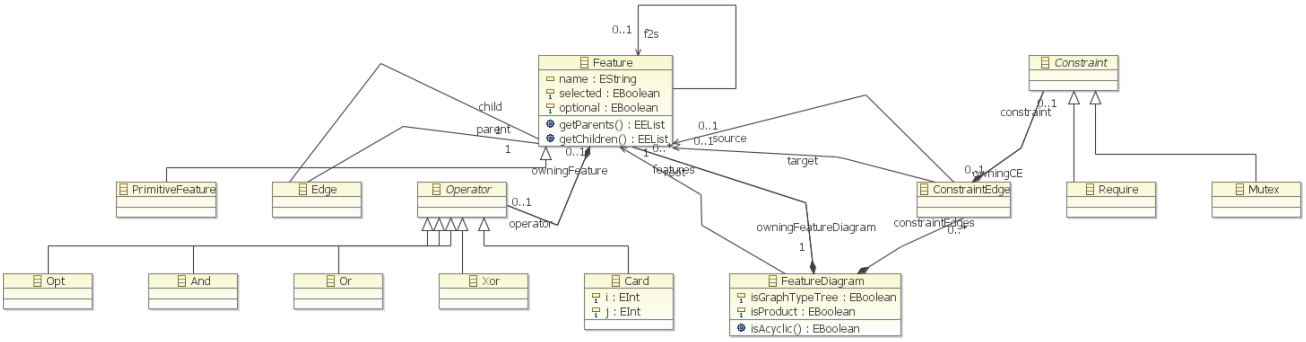


Figure 29. Méta-modèle des diagrammes de caractéristiques avec la notion d'ordre totale

Dans ce méta-modèle l'élément racine est la méta-classe *FeatureDiagram*. Un *FeatureDiagram* est composé par un ensemble de caractéristiques (dont une caractéristique *root*, c'est-à-dire la racine du diagramme). Nous retrouvons également une hiérarchie implicite dénotée par les relations *child* et *parent* entre la méta-classe *Feature* et la méta-classe *Edge.* Une caractéristique peut contenir un opérateur et cet opérateur peut être de type : *Opt, And, Or, Xor* ou *Card*. Ces opérateurs ont une sémantique similaire à ceux définis dans (30). Dans un diagramme de caractéristiques nous retrouvons également des contraintes, exprimées à la méta-classe *ConstraintEdge* et qui peuvent être de type *Require* ou *Mutex*. Dans le processus de dérivation de produits finaux, seulement les caractéristiques où l'attribut *selected* est vrai seront sélectionnées pour faire partie du produit en cours de dérivation. Comme une relation d'appartenance de chaque caractéristique vers un ou plusieurs modèles est possible (c'est la base de la réutilisation), un méta-modèle additionnel pour faire ce *mapping* est nécessaire.

Selon (16), une feature de type *Primitive* est celle qui ont un intérêt parce qu'elles ont une influence dans le produit final résultant. Une feature de type *Compound* est celle qui sert seulement comme un nœud intermédiaire, pour exprimer la décomposition. Pour nous, une feature *Compound* n'a pas un diagramme de séquence associé tandis qu'une *Primitive* oui.

La relation *f2s* de la Définition 5, est représentée dans le méta-modèle par la référence *f2s* de la méta-classe *Feature* vers elle même. Le fait d'inclure cette relation d'ordre entre les caractéristiques, implique la définition de nouvelles contraintes sur le méta-modèle et nous présentons ci-dessous la description textuelle de quelques unes déjà identifiées. Comme cela est expliqué dans la section 7.a, un méta-modèle additionnel est en cours de développement et sera utilisé pour lier les diagrammes de caractéristiques et les diagrammes de séquence UML, et le fait de ne pas encore connaître ce méta-modèle nous empêche de décrire ces nouvelles contraintes à ajouter directement en langage OCL.

Les contraintes sont :

- Toutes les features du diagramme doivent contenir une référence ***f2s*** vers une autre, sauf pour la feature root.
- Dans le cas d'une feature X qui contient un opérateur *Xor* : Toutes les features filles de X ont une référence ***f2s*** vers X, puisqu'il y en a qu'une qui sera sélectionnée parmi les autres.

- Dans le cas d'une feature X qui contient un opérateur O*r* ou *And* : La première feature fille de X, a une référence ***f2s*** vers sa feature parent X. Ensuite, chaque nouvelle feature fille aura une référence ***f2s*** vers la feature fille ajoutée juste avant elle.
- Deux features X1 et X2 qui référencent une troisième feature X par des relations *f2s,* c'est-à-dire *X1* ***f2s*** *X* et *X2* ***f2s*** *X,* sont considérées comme features filles de X.

Le méta-modèle qui sera utilisé pour lire les diagrammes de caractéristiques doit prendre en compte les remarques suivantes :

- Dans le cas d'une feature X qui contient un opérateur *Opt* : Supposons que cette feature référence une feature Y par une relation ***f2s***, c'est-à-dire X ***f2s*** Y. Si la feature X n'est pas sélectionnée (attribut *selected* = false), et qu'elle était référencée par une relation ***f2s*** provenant d'une feature Z, le méta-modèle doit ignorer les relations X***f2s***Y et Z***f2s***X et établir une nouvelle relation Z***f2s***Y.
- La différence entre l'opérateur *And* et l'opérateur *Or* c'est que pour l'opérateur *And* toutes les features filles seront sélectionnées (attribut *selected* = true) et donc, le méta-modèle ne doit pas ignorer les références ***f2s*** vers des features non sélectionnés, comme cela a été expliqué pour l'opérateur *Opt*.
- Si le méta-modèle de composition réalise des tâches de dérivation de produits, les features de type compound doivent être traitées comme des features qui contiennent l'opérateur Opt.

L'objectif de cette section est de donner un exemple pour mettre au claire les principes de cette approche et les résultats de son utilisation dans le domaine de la composition de comportements. Comme la section 4 l'a montré, le travail développé permet de composer directement des diagrammes de séquence UML 2.0. Ces modèles comportementaux sont connectés à un diagramme de caractéristiques avec un ordre d'exécution défini, comme élément guide du processus de composition. Par la suite, un exemple de ce type de procédure est présenté.

Le diagramme de caractéristiques d'exemple montré dans la Figure 30, modélise une famille simple de téléphones portables. Nous pouvons observer les relations f2s entre les différentes caractéristiques différentes et établir à partir de ces dernières un ordre d'exécution d'exemple. Dans ce cas, l'utilisateur appelle d'abord et ensuite il envoie soit un message de texte simple (SMS), soit un message multimédia(MMS).

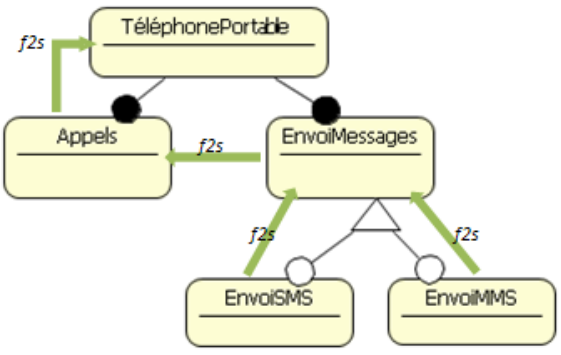


Figure 30. Diagramme de caractéristiques agrémenté

Selon les informations que nous pouvons observer dans le diagramme de caractéristiques, le comportement qui s'exécute en premier est celui d'appeler un correspondant. L'utilisateur appelle, le téléphone le met en communication avec son correspondant avec qui l'utilisateur parle et quand la conversation est finie l'utilisateur raccroche. Cette partie du comportement est présente dans tous les produits de la ligne parce qu'elle est obligatoire (noté avec le cercle noir dans le diagramme présenté). Ensuite, nous retrouvons les caractéristiques *EnvoiSMS* et *EnvoiMMS* qui sont deux variantes alternatives du point de variation *EnvoiMessages*, c'est-à-dire, que dans le processus de la dérivation d'un produit, l'ingénieur LdP doit faire un choix entre une caractéristique ou l'autre.

Finalement, le diagramme de séquence généré, grâce à l'information de l'ordre d'exécution des différentes caractéristiques est celui montré dans la Figure 31.

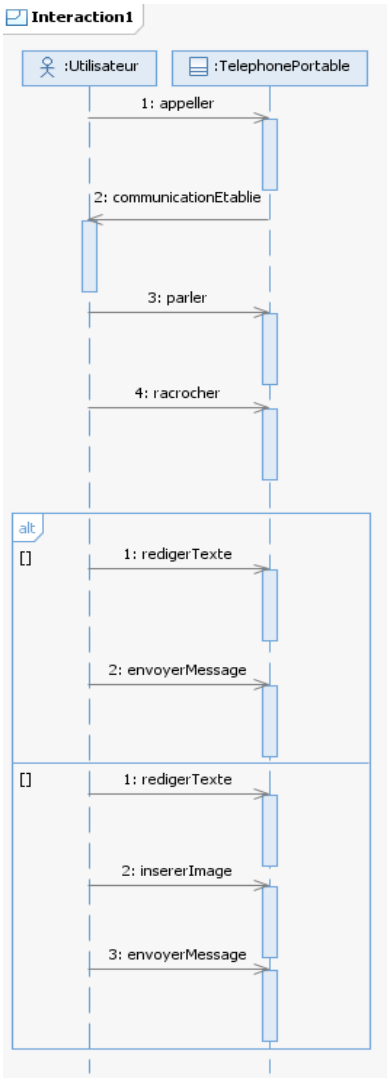


Figure 31. Diagramme de séquence résultant de la composition

6. Outil Implémenté (Plugin RSA)

L'outil est implémenté dans un plug-in pour l'environnement de développement Rational® Software Architect 8.0 d'IBM. Cet environnement est basé sur la plateforme *eclipse*[7] et orienté vers le développement logiciel dirigé par les modèles. Ses fonctionnalités avancées dans le domaine de la transformation de modèles et la modélisation UML, font de RSA un outil largement utilisé de manière industriel pour la conception de systèmes et de logiciels.

Le plugin développé ajoute un nouveau fournisseur de transformations (*TransformationProvider)* à la plateforme RSA. Ce nouveau fournisseur propose un nouveau groupe de trois transformations comme on peut le voir dans la Figure 32. Les deux premières transformations : *StateMachine to SequenceDiagram* et *SequenceDiagram to StateMachine,* correspondent aux sections 4.c et 4.a, respectivement.

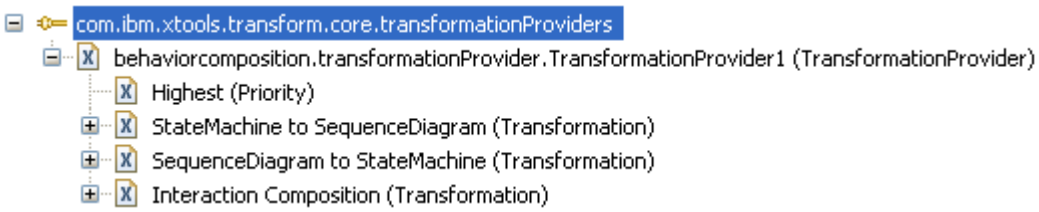


Figure 32. Transformation Provider[8]

La transformation *Interaction Composition* implémente la totalité de notre Framework de composition et inclut les algorithmes de composition présentés dans 4.b. Cette transformation enchaîne les autres pour composer directement des diagrammes de séquence UML. Toutes les transformations ont été codées en Java en utilisant l'API UML 2.0.

Dans la Figure 33, un diagramme de packages basique est présenté pour mieux comprendre la structure du plugin :

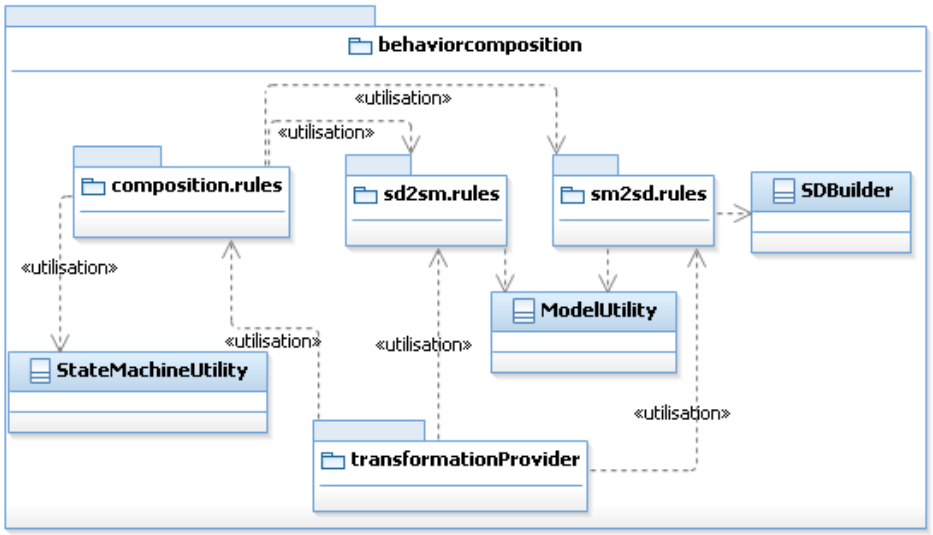


Figure 33. Diagramme de packages *behaviorcomposition*

[7] http://www.eclipse.org/

[8] L'accès à cet explorateur arborescent des extensions apportées par le plug-in est accessible depuis l'onglet Extensions du fichier plugin.xml.

StatemachineUtility est la classe qui contient les algorithmes de composition. Pour cela, elle est utilisée par le package *composition.rules* qui contient les règles de la transformation *Interaction Composition*.

SDBuilder est la classe qui contient les méthodes qui permettent de construire une interaction, créer des lignes de vie, envoyer des messages et d'autres opérations liées aux diagrammes de séquence UML. Elle est utilisée dans le processus de génération des interactions UML c'est-à-dire par les règles de la transformation *StateMachine to SequenceDiagram* du package *sm2sd.rules*.

ModelUtility est une classe multi propos utilisé pour charger des packages UML, des modèles, des attributs, etc. depuis le workspace à partir de son nom ou de son URI[9]. Pour cette raison elle est utilisée par les règles des packages *sm2sd.rules* et *sd2sm.rules.*

Le package *transformationProvider* contient les classes utilisés par RSA pour lier une transformation avec sa représentation graphique dans l'environnement de développement. Nous trouvons également des classes pour lier chaque transformation avec l'ensemble de ses règles. Un aperçu du module de création des différentes configurations de transformation à exécuter est montré dans la Figure 34 :

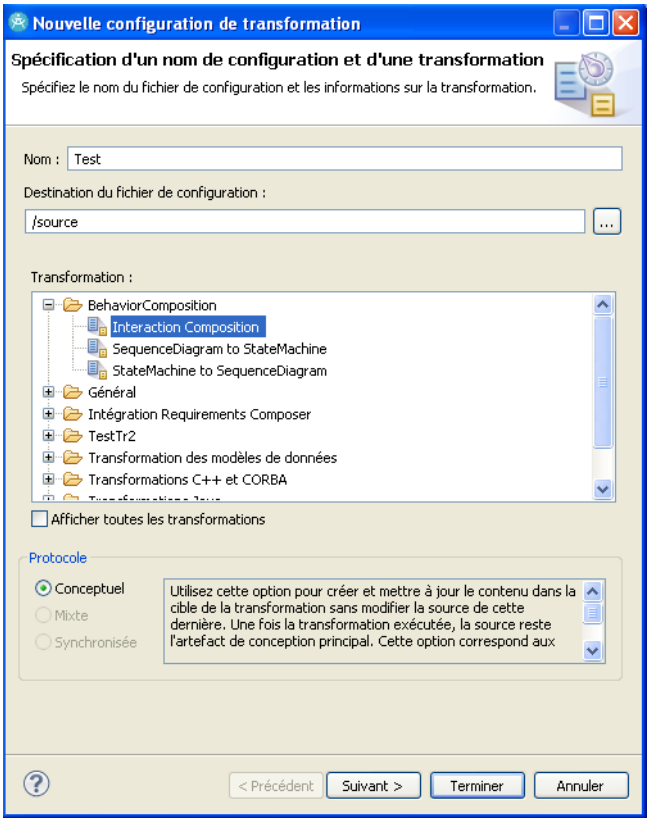


Figure 34. Nouvelle configuration de transformation

Pour y accéder, une fois le plug-in déployé sur RSA, il faut aller sur le menu Modélisation -> Transformer -> Nouvelle configuration…

[9] Uniform Resource Identifier

Un fichier d'extension .tc (*transformation configuration*) est créé dans le dossier de destination sélectionné et il faut ensuite l'éditer pour indiquer la source et la destination de la transformation dans l'onglet « Source et Cible » de l'éditeur de configurations de transformation, comme montré dans la Figure 35 :

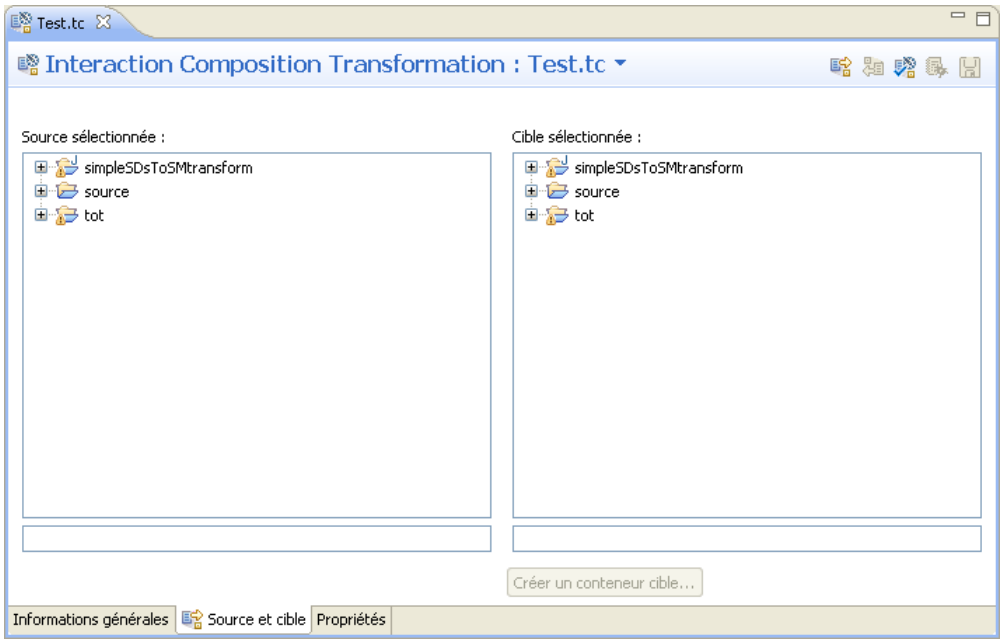


Figure 35. Editeur des configurations de transformation

Pour cette étape il est nécessaire de faire quelques précisions pour chacune des transformations disponibles:

Transformation	Entrée	Sortie
Interaction Composition	L'entrée de cette transformation doit être un package UML contenant les deux interactions à composer.	Une interaction UML 2.0* créée dans le dossier spécifié comme destination de la transformation.
StateMachine to SequenceDiagram	L'entrée de cette transformation doit être un package UML contenant les machines à état à transformer.	Une interaction UML 2.0* créée dans le dossier spécifié comme destination de la transformation.
SequenceDiagram to StateMachine	L'entrée de cette transformation doit être une interaction UML 2.0.	Un ensemble de machines à état créé dans le dossier spécifié comme destination de la transformation.

*Pour afficher le diagramme de séquence correspondant il suffit de faire un click droit et sélectionner l'option Ajouter un nouveau diagramme -> Diagramme de séquence.

7. Travaux à venir

Dans cette section nous présentons quelques travaux qui sont actuellement en cours de développement et que nous envisageons de finir d'ici la fin du stage pour étendre le Framework proposé et augmenter le niveau d'automatisation du processus de composition.

a. Méta modèle Kermeta

Les tâches de composition et dérivation de comportements dans les lignes de produits logiciels ont une complexité très importante qui empêche de les automatiser à 100%. Ce problème a été identifié dans (17) et selon les auteurs il n'y a pas encore de solution qui permette de composer des modèles comportementaux sans l'intervention de l'utilisateur.

Comme réponse partielle aux problèmes exposés dans cet article, nous présentons ici, les diagrammes de caractéristiques, agrémentés de la notion d'ordre totale et liés à des diagrammes de séquence. Même si cette approche a besoin d'une intervention de l'utilisateur pour définir l'ordre d'exécution des différents composants de la ligne de produits, elle est valide et utilisable dans un processus de construction incrémentale comme cela a été montré précédemment.

Cependant, nous avons besoin d'une technique qui permette à l'ingénieur LdP de (après avoir créé son diagramme de caractéristiques ordonné) générer automatiquement la description comportementale d'un des produits ou de la totalité de la famille de produits. Une proposition sur laquelle des travaux à venir sont envisageables est l'utilisation de Kermeta[10], un langage permettant de décrire et exécuter une sémantique opérationnelle au niveau méta-modèle, pour décrire un méta-modèle permettant de faire le lien entre les diagrammes de séquence et les diagrammes de caractéristiques agrémentés.

Ce méta-modèle Kermeta aura d'un côté la tâche de la lecture du diagramme de caractéristiques (en respectant l'ordre d'exécution défini par l'ingénieur LdP) et d'un autre l'exécution du Framework de composition de diagrammes de séquence, selon les informations récupérées dans l'étape précédente. Un schéma explicatif du travail attendu du méta-modèle est montré dans la Figure 36.

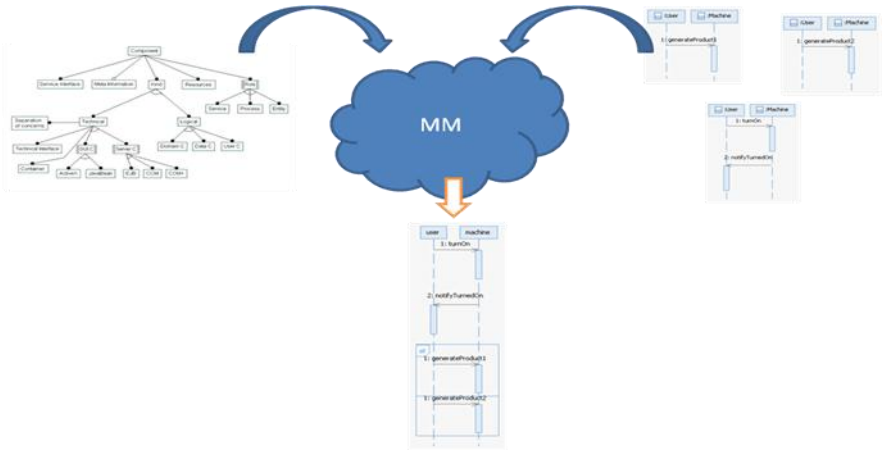


Figure 36. Méta-modèle Kermeta

[10] http://www.kermeta.org/

b. Composition de comportements en parallèle

Dans les systèmes complexes d'aujourd'hui l'exécution de tâches et de comportements en parallèle est une nécessité pour des raisons de performance, de réactivité et de consommation de ressources. Pour modéliser une ligne de produits, nous disposons de moyens d'exprimer cette simultanéité d'exécution comme les états orthogonaux dans les Machines à état UML 2.0 montrés dans ce document. Pour des contraintes de temps, l'implémentation d'opérateurs de composition parallèle de comportements, compatibles avec le Framework proposé, n'a pas été faite.

Cependant, il est possible d'implémenter un opérateur *par* qui fasse le même travail que l'opérateur de composition alternative mais sans stéréotyper l'état orthogonale produit. De cette façon l'opérateur gardera la sémantique de parallélisme de base des états orthogonaux UML 2.0. Ensuite, l'implémentation de techniques de synchronisation de ressources et d'envoi et réception de messages est également envisagée.

8. Conclusion

Ce document est le produit et la synthèse d'environ cinq mois de travail dans la modélisation de l'aspect dynamique des lignes de produits logiciels. Pendant ce temps, une étude bibliographique importante a été faite pour identifier les problématiques existantes dans ce domaine, en faisant un focus sur les travaux concernant la construction incrémentale des lignes de produits logiciels et la composition de comportements.

Dans un premier temps, les motivations pour travailler avec le paradigme IDM et l'approche Ligne de Produits ont été exposées. Ceci nous a permis de présenter le contexte du stage et d'identifier l'importance de traiter ce sujet dans un master recherche en informatique. Après, un parcours des propositions existantes, concernant la modélisation des aspects statique et dynamique des Lignes de Produits Logiciels, a été fait en identifiant les points clés de chaque approche. Dans le cas des profils UML et des méta-modèles de variabilité deux nouvelles alternatives ont été proposées avec l'objectif de synthétiser les travaux existants et traiter les faiblesses rencontrées lors de cette étude.

Egalement l'approche de la composition comportementale a été identifiée comme une solution aux problèmes liés à l'ingénierie des applications et à la gestion de l'évolutivité des modèles comportementaux dans les lignes de produits logiciels. Par conséquent, une proposition d'implémentation d'un Framework de composition comportementale a été exposée étape par étape en expliquant les différentes transformations et algorithmes utilisés lors du processus de composition. Plusieurs travaux existants ainsi que des nouvelles propositions on été intégrés pour concevoir un outil permettant à l'ingénieur LdP de composer directement des diagrammes de séquence UML 2.0. Pour chaque étape de ce processus, des exemples simples ont été présentés, ceci en ayant comme objectif d'aider le lecteur à mieux comprendre la proposition faite.

Ensuite le langage des diagrammes de caractéristiques, largement utilisé pour modéliser des lignes de produits logiciels, a été étendu avec la notion d'ordre total. Ceci nous a permis de spécifier l'ordre dans lequel les caractéristiques du diagramme seront exécutées et de cette façon guider le processus de composition comportementale, tout en utilisant l'approche de la construction incrémentale des lignes de produits logiciel.

Enfin, quelques travaux à venir ont été également identifiés. Pour automatiser encore plus le processus de composition, l'implémentation d'un méta-modèle Kermeta est envisagée. Grâce à un diagramme de caractéristiques agrémenté existant, ce méta-modèle sera en capacité d'accomplir la tâche de composition des diagrammes de séquence liés à chacune de ces caractéristiques. L'ajout d'un opérateur additionnel pour compléter le Framework proposé avec la notion de parallélisme, fait également partie des travaux en cours.

Finalement, ce stage m'a permis de découvrir le monde de la recherche et le dynamisme du travail dans une équipe experte dans le domaine de la modélisation. La méthodologie de travail autonome, utilisé au sein du laboratoire de recherche, m'a également aidé à renforcer mes capacités de proposition de solutions à des problèmes complexes. Les différents défis techniques rencontrés lors du développement des solutions proposées, ont été surmontés grâce à la formation obtenue pendant mes cours du Master Recherche et l'aide constante de mes collègues de l'équipe IDM de l'ENSTA Bretagne/LISyC.

Bibliographie

1. **Kang, Sugumaran et Park.** *Applied Software Product-Line engineering.* Boca Raton, FL : Auerbach Publications, 2009.

2. **Combemale, Benoît.** Ingénierie Dirigée par les Modèles (IDM). [En ligne] 2008. [Citation : 4 Mai 2011.] http://hal.archives-ouvertes.fr/docs/00/37/15/65/PDF/mde-stateoftheart.pdf.

3. *Models in software engineering—an introduction.* **Ludewig, J.** s.l. : SoSyM 2, 2003, Vol. (3), pp. 5–14.

4. *Towards a Precise Definition of the OMG/MDA Framework.* **Bézivin, Jean et Gerbé, Olivier.** San Diego, USA : ASE'01, Automated Software Engineering, 2001.

5. *The Role of Metamodeling in MDA.* **Atkinson, Colin et Kühne, Thomas.** 2002, Proc. UML 2002 Workshop Software Model Eng., pp. 67–70.

6. *Classification of Model Transformation Approaches.* **Czarnecki, Krzysztof et Helsen, Simon.** 2003, OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture.

7. *Towards a UML Profile for Software Product Lines.* **Ziadi, Tewfik, Hélouët, Loïc et Jézéquel, Jean-Marc.** 2004, PFE 2003, LNCS 3014, pp. 129-139.

8. **McIlroy, M. D.** *Mass produced software components.* Garmisch, Germany : s.n., 1968. pp. 138-155.

9. *On the Design and Develpement of Program Families.* **Parnas, David.** 1, 1979, IEE Transactions on Software Engineering, Vol. SE 2, pp. 1-9.

10. *Easing the Transition to Software Mass Customization.* **Krueger, Charles W.** PFE 2001, Bilbao, Spain : Springer-Verlag, 2001, Vol. 4th international workshop.

11. *DARE: Domain analysis and reuse environment.* **Frakes, W., Prieto-Diaz, R. et Fox, C.** 1998, Annals of Software Engineering, Vol. 5, pp. 125-141.

12. **Atkinson, C.J., et al.** *Component-based product line engineering with UML.* Edinburgh : Addison-Wesley, 2002.

13. **Gomaa, Hassan.** Designing Software Product Lines with UML 2.0: From Use Cases to Pattern-Based Software Architectures. *Reuse of Off-the-Shelf Components.* s.l. : Springer Berlin / Heidelberg, 2006.

14. *Feature-oriented product line engineering. .* **Kang, K. C., J., Lee et Donohoe, P.** 2002, IEEE Software, pp. 58-65.

15. **Linden, Frank, Schmid, Klauss et Rommes, Eelco.** *Software Product Lines in Action.* Berlin : Springer-Verlag, 2007. pp. 3-5.

16. **Pohl, Klaus, et al.** *Software Product Line Enginnering.* Berlin Heidelberg : Springer-Verlag, 2005.

17. *Issues in model-driven behavioural product derivation.* **Istoan, Paul, Biri, Nicolas et Klein, Jacques.** New York, USA : ACM, 2011.

18. *Vers des lignes de produits flexibles : Apports de l'ingénierie dirigée par les modèles à la dérivation de produits.* **Jézéquel, Jean-Marc et Perrouin, Gilles.** 3, Paris, France : Lavoisier, 2008, Vol. 14. 1262-1137 .

19. **Ziadi, Tewfik et Jézéquel, Jean-Marc.** *Manipulation de Lignes de Produits Logiciels une approche dirigée par les modèles.* Paris : s.n., 2005. pp. 1-14.

20. **Millymaki, Tommi.** *Variability Management in Software Product Lines.* Tampere, Finland : Tampere University of Technology, 2001.

21. **Classen, Andreas, et al.** Symbolic Model Checking of Software Product Lines. *ICSE '11.* Honolulu, Hawaii, USA : s.n., 2011.

22. **Classen, Andreas, et al.** Model Checking Lots of Systems. *ICSE '10.* Cape Town, South Africa : s.n., 2010.

23. *Survey and Classification of Software Product Line Variability Modeling Techniques.* **Istoan, Paul, Klein, Jacques et Jézéquel, Jean-Marc.** 2010, IEEE Transactions on Software Engineering, pp. 1-16.

24. *Generic modeling using UML extensions for variability.* **Clauss, M.** 2001, Vol. OOPSLA.

25. *Modeling Variability with UML.* **Clauss, M. et Jena, I.** 2001, Vol. GCSE Young Researchers Workshop.

26. *Software Product Line Engineering with the UML: Deriving Products.* **Ziadi, Tewfik et Jézéquel, Jean-Marc.** Rennes, France : s.n., 2006.

27. **Ziadi, Tewfic, Hélouet, Loïc et Jézéquel, Jean-Marc.** Revisiting Statechart Synthesis with an Algebraic Approach. *Proceedings of the 26th International Conference on Software Engineering.* Washington, DC, USA : IEEE Computer Society, 2004.

28. *Modeling and managing the variability of Web service-based systems.* **Sun, Chang-a, et al.** 83, s.l. : Elsevier Inc., 2010, Vol. The Journal of Systems and Software, pp. 502–516.

29. *Variability Management on Behavioral Models.* **Tessier, Patrick, Servat, David et Gérard, Sébastien.** Essen, Germany : VaMoS 2008, 2008.

30. **Morin, Brice, et al.** Weaving Variability into Domain Metamodels. *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems.* Berlin, Heidelberg : Springer-Verlag, 2009.

31. **Kang, K. C., et al.** *Feature-oriented domain analysis (foda) feasibility study.* s.l. : Carnegie-Mellon University Software Engineering Institute, 1990.

32. *Form: A feature-oriented reuse method with domain-specific reference architectures.* **Kang, K., et al.** 1998, Vol. 5, pp. 143-168.

33. **von der Maßen, Thomas et Lichter, Horst.** Determining the Variation Degree of Feature Models. [auteur du livre] H. Obbink et K. Pohl. *SPLC 2005.* Berlin Heidelberg : Springer-Verlag, 2005.

34. *From Requirements to Code Revisited.* **Ziadi, Tewfik, Blanc, Xavier et Raji, Amine.** 2009, Vol. IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, pp. 228-235.

35. **Group, Object Management.** *OMG Unified Modeling LanguageTM (OMG UML) Superstructure .* [http://www.omg.org/spec/UML/2.4/Superstructure/Beta2/PDF] Janvier 2011.

36. *Draco: A Method for Engineering Reusable Software Systems.* **Neighbors, J.M.** s.l. : ACM Press Frontier Series, 1989, pp. 295-319.