

---

# Manipulation de Lignes de Produits Logiciels : une approche dirigée par les modèles

Tewfik Ziadi.<sup>1</sup>, Jean-Marc Jézéquel.<sup>2</sup>

1 tziadi@irisa.fr, IRISA-IUT de Lannion, BP 30219 Lannion Cedex 22300.

2 jezequel@irisa.fr, IRISA-Université de Rennes 1, Campus de Beaulieu 35042, Rennes Cedex.

---

## Résumé

*Le champs du génie logiciel a évolué pour couvrir les nouvelles problématiques introduites par l'extension de la pénétration de l'informatique dans l'industrie. On arrive ainsi à une situation où la problématique n'est plus de développer un seul logiciel à la fois mais plutôt concevoir et développer une ligne (ou une famille) de produits logiciels qui prend en compte des facteurs de variation et permet de minimiser les coûts et les temps de réalisation. Cet article montre comment l'ingénierie dirigée par les modèles est utilisée pour la manipulation de lignes de produits (LdP). Nous étendons en premier lieu UML pour permettre la spécification de la variabilité dans deux types de modèles d'UML : les diagrammes de classes (aspect statique) et les diagrammes de séquence (aspect comportemental). Par la suite, nous formalisons la dérivation de produits en utilisant la transformation de modèles ; nous considérons la dérivation de l'aspect statique et comportemental de produits. Un exemple simple d'une ligne de produit du domaine bancaire est utilisé tout au long de l'article pour illustrer la modélisation et la dérivation.*

## 1. Introduction

Le champs du génie logiciel a évolué pour couvrir les nouvelles problématiques introduites par l'extension de la pénétration de l'informatique dans l'industrie. On arrive ainsi à une situation où la problématique n'est plus de développer un seul logiciel à la fois mais plutôt concevoir et développer une *ligne (ou une famille) de logiciels* qui prend en compte des facteurs de variation et permet de minimiser les coûts et les temps de réalisation. Les facteurs de variation peuvent être techniques (utilisation d'une variété de matériels associés aux logiciels), commerciaux (création de plusieurs versions allant d'une version limitée à une version complète), ou culturels (logiciels destinés à plusieurs pays). Pour illustration, on peut citer les logiciels intégrés aux téléphones mobiles qui doivent supporter plusieurs standards de communication et une variété de langues (dans [10], Maccari *et al.* affirment par exemple que les téléphones Nokia doivent supporter plus de 60 langues).

Cette notion de lignes de produits n'est pas totalement nouvelle car David Parnas dans [13] a déjà étudié les familles de programmes dès 1976. Cependant ce paradigme n'a émergé comme une approche à part du génie logiciel seulement ces dernières années, lorsqu'il a commencé à regrouper une communauté à travers les différents projets européens tels que FAMILIES [1]. Aux États Unis, Le Software Engineering Institute a également créé une action spéciale qui s'intéresse à l'ingénierie des lignes de produits [11]. Dans la littérature, il y a un consensus sur la définition d'une *ligne de produits logiciel* (LdP). Elle est définie comme un ensemble de systèmes partageant un ensemble de propriétés communes et satisfaisant des besoins spécifiques pour un domaine particulier [4]. La notion de *variabilité* est utilisée pour regrouper les caractéristiques qui différencient les produits de la même famille. Les langues supportées sont un

exemple de variabilité dans une LdP du domaine des téléphones mobiles. La gestion de cette variabilité est la première activité pour le développement de lignes de produits. La deuxième activité concerne la construction d'un produit particulier (on parle aussi de *dérivation* de produit) qui consiste en particulier à figer certains choix vis-à-vis de la variabilité définie dans la LdP.

Cet article résume les travaux menés dans l'équipe Triskell de l'IRISA pour la manipulation de lignes de produits par une approche dirigée par les modèles. Nous nous sommes intéressés aux deux dimensions des lignes de produits à savoir la modélisation de la variabilité et la dérivation de produits. Nous étendons en premier lieu UML pour permettre la spécification de la variabilité dans deux types de modèles d'UML : les diagrammes de classes (aspect statique) et les diagrammes de séquence (aspect dynamique). Par la suite, nous formalisons la dérivation de modèles de produits en utilisant *la transformation de modèles*. Nous considérons la dérivation de modèles statiques de produits ainsi que leur comportement.

Le reste de cet article est organisé comme suit. Section 2 introduit un exemple simple d'une ligne de produits du domaine bancaire qui est utilisé pour l'illustrer le processus proposé. Elle présente par la suite notre approche pour la modélisation de l'aspect statique et dynamique des lignes de produits. Section 3 présente une approche, basée sur les transformations de modèles, pour la dérivation automatique de modèles de produits.

## 2. Modélisation des Lignes de Produits

La modélisation de la variabilité est la première activité dans toute approche supportant les LdP. Il existe dans la littérature de très nombreux travaux autour de la manipulation des LdP en UML [15]. La principale hypothèse de la plupart de ces travaux concerne l'utilisation des mécanismes d'extension d'UML pour permettre la spécification de la variabilité dans les modèles UML. En étudiant de plus près ces travaux, des constats d'insuffisances se dégagent et indiquent que la manipulation de LdP en UML manque de maturité. En effet, la majorité des travaux existants s'intéressent seulement à deux aspects de la modélisation UML : aspect fonctionnel (cas d'utilisation) et aspect statique (diagramme de classes) et peu de travaux sur l'aspect dynamique. Dans un précédent travail [17], nous avons formalisé un ensemble d'extensions sous forme d'un *profil UML* pour la modélisation de l'aspect statique mais aussi dynamique de la variabilité dans les LdP. Le reste de cette section présente brièvement ces extensions et les illustre sur un exemple simple d'une ligne de produit dans le domaine bancaire. Le lecteur trouvera toute la formalisation des extensions dans [17].

### 2.1. La ligne de produit banque

Il s'agit d'un ensemble de systèmes produisant les fonctionnalités standards dans le domaine bancaire. Les produits de cette famille sont destinés à plusieurs banques et chaque produit peut offrir les fonctionnalités suivantes :

**Création de comptes (F1)** Les clients sont capables d'ouvrir des comptes mais ils doivent le faire avec une somme minimale de départ. Un compte peut avoir un montant de découvert, et une devise pour sa gestion. Ces deux données sont spécifiées pendant la création du compte.

**Dépôt (F2)** Les clients peuvent déposer de l'argent sur leurs comptes en se présentant au guichet de la banque.

**Retrait (F3)** Les clients peuvent retirer de l'argent sur leurs comptes. Si le compte a un montant de découvert, le client peut retirer une somme jusqu'à cette limite. Sinon, il ne peut pas retirer au delà du solde courant du compte.

**Gestion des devises (F4)** Le système peut offrir une fonctionnalité pour la gestion des devises. Il s'agit en particulier des services de conversion de ou à partir de l'Euro vers les différentes devises : dollar, livre,..etc.

La variabilité dans cette ligne de produits concerne deux aspects : 1) Le support ou non des découverts sur les comptes qui est optionnel, 2) Le support des fonctionnalités de gestion de devise qui est optionnel aussi. La Table 1 montre quatre produits membres de la LdP banque. Le produit BS1 (Bank System 1) par exemple, supporte les découverts sur les comptes et il ne supporte pas la gestion des devises.

TAB. 1 – Les membres de la LdP banque

Produit	Le support des découverts	La gestion des devises
BS1	OUI	NON
BS2	NON	NON
BS3	NON	OUI
BS4	OUI	OUI

## 2.2. Aspect statique

Nous modélisons l’aspect statique de la ligne de produits sous forme d’un diagramme de classes UML étendu par trois stéréotypes pour permettre la spécification de la variabilité [17] : 1) `<<optional>>` pour spécifier l’optionnalité des éléments d’un diagramme de classes. 2) `<<variation>>` et `<<variant>>` pour spécifier l’alternative dans le choix de classes.

La Figure 1 montre le diagramme de classes de la LdP banque avec les extensions introduites. La classe `Convertor` offre deux méthodes permettant la gestion de devises (la conversion de et vers l’Euro). Comme cette fonctionnalité est optionnelle pour les membres de la ligne de produit, la classe `Convertor` est stéréotypée `<<optional>>`. La classe abstraite `Account` est stéréotypée `<<variation>>` et ses deux sous-classes `AccountWithLimit` et `AccountWithoutLimit` sont des classes variantes (elles sont stéréotypées `<<variant>>`). Chaque produit doit choisir une et une seule classe variante. Comme certains produits supportent seulement l’Euro comme monnaie, l’attribut `currency` dans la classe `Account` est optionnel.

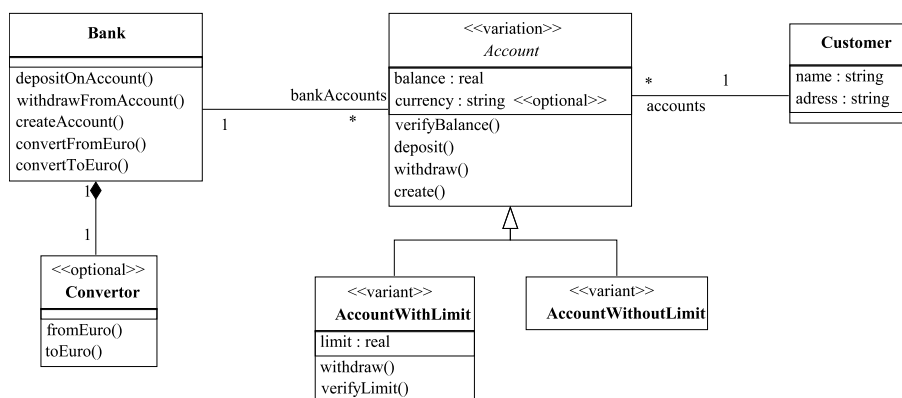


FIG. 1 – Le diagramme de classes de la LdP banque

## 2.3. Aspect dynamique

### 2.3.1 Les diagrammes de séquence d’UML2.0

Pour la modélisation de l’aspect dynamique (ou *comportemental*) de la LdP, nous utilisons la nouvelle version des diagrammes de séquence d’UML2.0. Les diagrammes de séquence d’UML2.0 [12] améliorent les versions précédentes en offrant des mécanismes de composition permettant de définir des interactions de base qui peuvent être par la suite composer dans des interactions combinées en utilisant les

opérateurs d'interaction. Les trois opérateurs fondamentaux sont : **seq** pour la composition séquentielle faible <sup>1</sup>, **alt** pour l'alternative, et **loop** pour l'itération d'une interaction.

La Figure 2 montre les interactions de base de la LdP banque. Le diagramme de séquence `Deposit` par exemple spécifie l'interaction entre l'acteur `Clerk` et les deux objets : `Bank` et `Account` pour déposer de l'argent sur un compte d'un client. La Figure 3 montre une interaction combinée, appelée `BankPL`, associée à la LdP banque. Elle fait référence aux interactions de base de la Figure 2 et les compose en utilisant les opérateurs d'interaction. Selon la notation standard d'UML2.0, les opérateurs d'interaction dans les interactions combinées sont décrits par des cadres rectangulaires avec un coin gauche supérieur labellisé par l'opérateur `seq`, `alt`, ou `loop` (la construction `ref` spécifie la référence aux interaction de base). Les opérands pour les opérateurs de séquence et d'alternative sont séparés par des lignes horizontales discontinues. La composition séquentielle peut être aussi implicitement donnée par l'ordre relatif de référence aux diagrammes de séquence. L'interaction combinée `BankPL` spécifie qu'il y a quatre principales alternatives dans les comportements de la LdP banque qui correspondent aux quatre fonctionnalités : (F1) Création des comptes (F2) Dépôts de l'argent (F3) Retrait d'argent (F4) Gestion des devises.

### 2.3.2 La Variabilité dans les diagrammes de séquence

Pour supporter la variabilité des LdP, nous avons étendu les diagrammes de séquence d'UML2.0 par trois mécanismes de variabilité qui sont formalisés à l'aide des stéréotypes et de tagged values [17] : *optionnalité*, la *variation* et la *virtualité*. L'optionnalité d'une interaction signifie que son comportement est optionnel, c.à.d. tous les messages contenus dans cette interaction peuvent être supprimés dans certains produits. La variation dans les interactions permet à la LdP de définir un ensemble de comportements variants parmi lesquels un produit particulier devra choisir une et une seule variante. La virtualité d'une interaction signifie que son comportement peut être redéfini par une autre interaction de raffinement associée à un produit particulier.

Pour la spécification des diagrammes de séquence des LdP, nous modélisons en premier lieu les interactions de base pour ensuite les composer par une interaction combinée. La variabilité y est alors introduite. L'interaction combinée `BankPL` de la Figure 3 contient de la variabilité et illustre l'optionnalité et la variation :

- Certains produits ne supportent pas les découverts sur les comptes. L'occurrence de l'interaction `SetLimit`, qui permet d'ajouter le montant d'un découvert pendant la création du compte, est définie avec le stéréotype `<<optionalInteraction>>` pour spécifier qu'elle est optionnelle. Le tagged value `optionalPart` prend comme valeur la chaîne de caractère `settingLimit` (voir la Figure 3)
- L'objet `Converter` est stéréotypé `<<optionalLifeline>>` dans l'interaction `BankPL` pour spécifier qu'il est optionnel.
- La gestion de devise est une fonctionnalité optionnelle, donc les occurrences des interactions `SetCurrency`, `ConvertToEuro` et `ConvertFromEuro` sont définies comme optionnelles. Le tagged value `optionalPart` prend comme valeur respectivement `settingCurrency`, `toEuro` et `fromEuro` (voir la Figure 3).
- Il existe deux interactions variantes pour le retrait d'argent à partir des comptes : retrait avec vérification du solde et du montant de découvert, et retrait avec seulement vérification du solde. L'interaction `Withdraw` est stéréotypée `<<variation>>` et les deux occurrences des interactions `WithdrawWithLimit` et `WithdrawWithoutLimit` sont stéréotypées `<<variant>>`.

---

<sup>1</sup>UML2.0 définit deux opérateurs de séquence, "seq" et "strict". La composition séquentielle faible signifie que les événements situés dans la première interaction sur une ligne de vie particulière doivent être exécutés avant les événements de la deuxième interaction situés sur la même ligne de vie. Cependant l'opérateur "strict" spécifie la composition séquentielle forte qui signifie que tous les événements (quelle que soit leur localisation) de la première interaction doivent être exécutés avant les événements de la deuxième interaction.

Le tagged value variationPart prend comme valeur la chaîne withdrawAccount (voir la Figure 3).

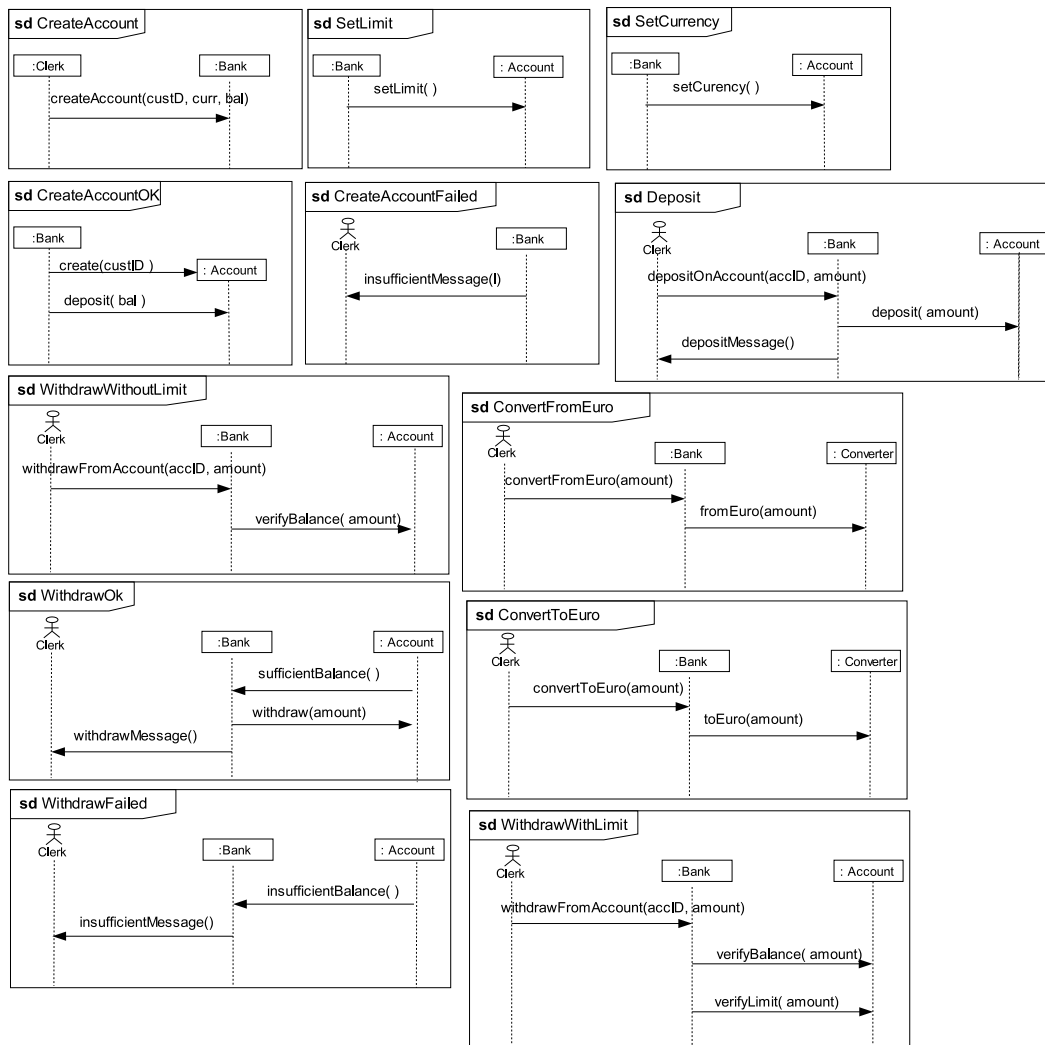


FIG. 2 – Les interactions de base dans la LdP banque

## 2.4 Spécification algébrique des diagrammes de séquence de LdP

Pour permettre une manipulation formelle, nous avons proposé une formalisation algébrique des diagrammes de séquence des LdP sous forme de ce que nous avons appelé les RESD-PL (Reference Expression for Sequence Diagrams of Product Lines) [15]. Une RESD-PL est une expression algébrique dont les termes sont des références vers les interactions de base et les opérateurs sont ceux d'UML2.0 : **seq**, **alt** et **loop** étendus par trois constructions algébriques pour la variabilité : **optional**, **virtual**, et **variation** qui correspondent aux trois mécanismes de variabilité présentés ci-dessus.

**Definition 1** Une Reference Expression for Sequence Diagrams (notée RESD par la suite) est une expression de la forme <sup>2</sup> :

$$\begin{aligned}
 \langle \text{RESD} \rangle & : := \langle \text{PRIMARY} \rangle ( \text{"alt"} \langle \text{RESD} \rangle \mid \text{"seq"} \langle \text{RESD} \rangle ) * \\
 \langle \text{PRIMARY} \rangle & : := E_{\emptyset} \mid \langle \text{IDENTIFIANT} \rangle \mid \text{" ("} \langle \text{RESD} \rangle \text{" )"} \mid \\
 & \quad \text{"loop"} \text{" ("} \langle \text{RESD} \rangle \text{" )"} \\
 \langle \text{IDENTIFIANT} \rangle & : := ( [ "6" , "a" - "z" , "A" - "Z" ] \mid [ "0" - "9" ] ) *
 \end{aligned}$$

<sup>2</sup>Nous utilisons une notation proche de EBNF (Extended Backus Normal Form) pour la définition des expressions de référence.

**Definition 2** Une Reference Expression for Sequence Diagrams of Product Lines (notée *RESD-PL* par la suite) est une expression de la forme :

```

<RESD-PL> : :=<PRIMARY-PL> ("alt" <RESD-PL> | "seq" <RESD-PL>)*
<PRIMARY-PL> : :=E0 | <IDENTIFIER> | "(" <RESD-PL> ")" |
    "loop" "(" <RESD-PL> ")" |
    "optional" <IDENTIFIER> "[" <RESD> "]" |
    "variation" <IDENTIFIER> "[" <RESD> ", " <RESD> (", " <RESD>)* "]" |
    "virtual" <IDENTIFIER> "[" <RESD> "]"

```

Une interaction combinée d'UML2.0 pour les LdP est spécifiée directement sous forme d'une RESD-PL. Toutes les règles de transformation des interactions en RESD-PL sont présentées en détail dans [15]. L'interaction combinée BankPL de la Figure 3 référence quatre interactions optionnelles et une interaction variation. Elle est décrite algébriquement par l'expression de référence suivante :

---

```

EBPL = loop( Deposit alt (CreateAccount seq (CreateAccountOk seq
(optional settingLimit [ SetLimit ] seq optional settingCurrency
[SetCurrency ])) alt CreateAccountFailed) alt (variation
withdrawAccount [ WithdrawWithLimit, WithdrawWithoutLimit ]
seq ( WithdrawOk alt WithdrawFailed)) alt
(optional fromEuro [ ConvertFromEuro ] ) alt
(optional toEuro [ ConvertToEuro ] ))

```

---

### 3 Dérivation automatique de modèles de produits

La section précédente a montré comment l'architecture statique de la LdP et son comportement sont modélisés en utilisant les mécanismes d'extension d'UML. La deuxième difficulté dans la manipulation des LdP concerne la *dérivation de produits*. La dérivation est le processus de construction d'un produit spécifique de la LdP [5]. Utilisant UML, certains travaux existant se contentent seulement de modéliser la variabilité et ne font pas référence à la dérivation de produits [9, 7, 14]. Ceci, à notre avis, restreint l'utilité de ces travaux à un but *descriptif* et les limite à des objectifs de documentation d'architectures à l'aide de modèles UML. Les travaux de [3, 2, 6] font référence à la dérivation de produits mais ne la formalisent pas et ne proposent pas de solutions pour sa mise en oeuvre. Dans les sous-sections suivantes nous proposons d'aller au delà des buts descriptifs et nous proposons d'automatiser le processus de dérivation en utilisant les transformations de modèles. Nous considérons la dérivation de l'aspect statique de produits ainsi que leur comportement.

#### 3.1 Aspect statique

L'approche que nous adoptons pour la dérivation de l'aspect statique de produits est basée sur les *transformations de modèles* UML. Nous formalisons la dérivation comme une transformation de modèles dans laquelle le modèle source est un modèle de LdP (un diagramme de classes) et le modèle cible est un modèle de produit (un diagramme de classes aussi). La dérivation est définie dans un algorithme qui est décrit en détail dans [15] qui comporte trois étapes : Sélection des classes variantes et optionnelles, Spécialisation du modèle, et une étape d'optimisation. La dérivation initialise le modèle de produit par le modèle de la LdP, ensuite des transformations sont appliquées sur ce modèle jusqu'à l'obtention du modèle final du produit :

1. **Sélection des classes variantes et optionnelles** : La première étape consiste à sélectionner les classes variantes et optionnelles qui devront être incluses dans le modèle du produit en se basant sur les choix des utilisateurs responsables de la dérivation.

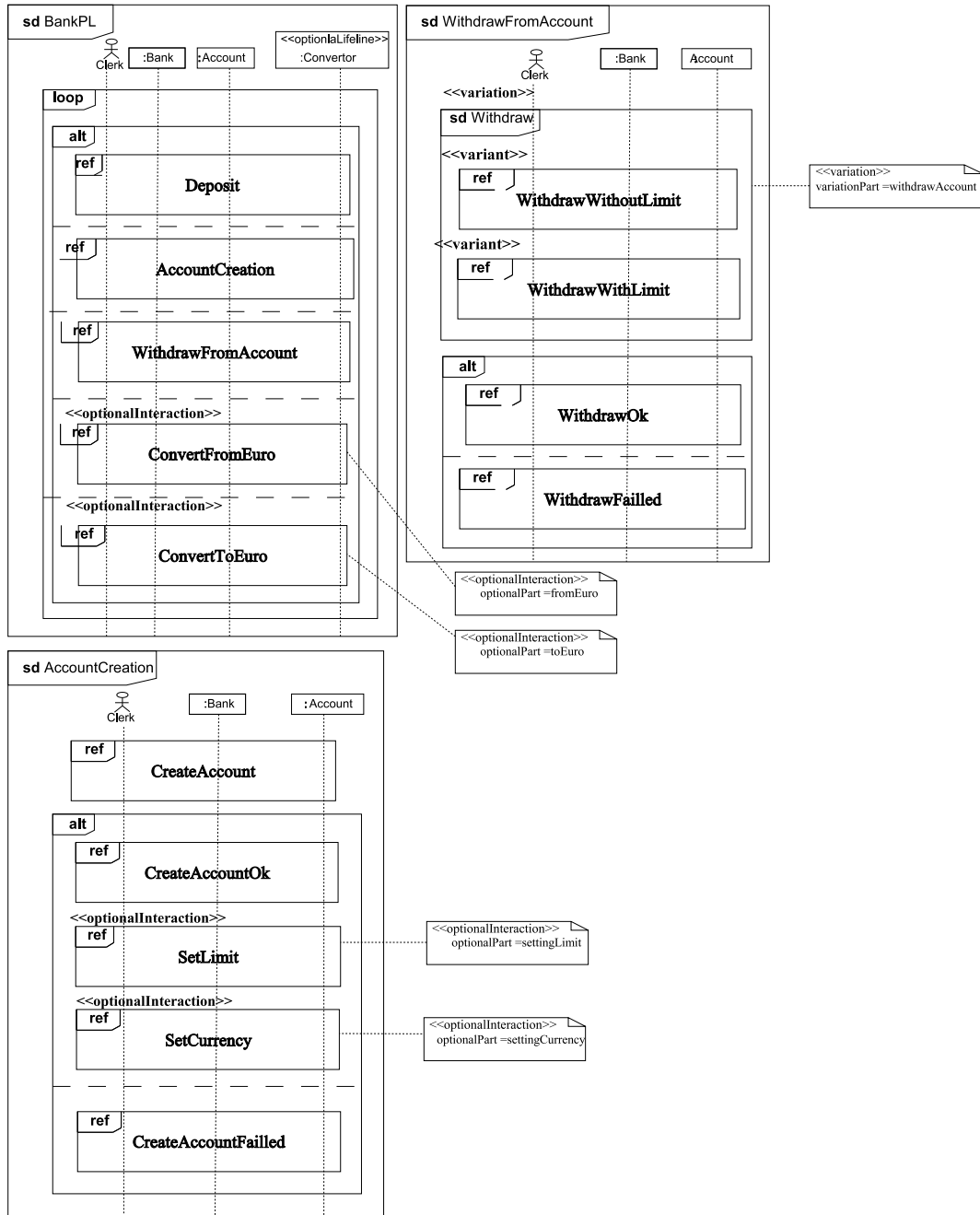


FIG. 3 – L'interaction combinée de la LdP banque avec la variabilité

2. **Spécialisation du modèle** : Dans cette étape toutes les classes variantes et optionnelles qui n'ont pas été sélectionnées dans la première étape sont supprimées du modèle.
3. **Optimisation du modèle** : Dans cette étape, le modèle est optimisé pour obtenir le modèle final du produit.

Pour obtenir le diagramme de classes du produit BS2 par exemple, la classe variante `AccountWithoutLimit` sera la seule classe sélectionnée car ce produit ne supporte ni les limites sur les comptes ni la conversion de devises. La spécialisation du modèle consiste à supprimer les deux classes : `AccountWithLimit` et `Convertor`. L'étape d'optimisation dans ce cas consistera à optimiser l'héritage en supprimant du modèle la classe abstraite `Account` et la remplacer par la classe `AccountWithoutLimit`. Le modèle obtenu est illustré dans la Figure 4. Nous avons réalisé l'algorithme de dérivation de l'aspect statique en utilisant le langage MTL (Model Transformation Language) développé dans l'équipe Triskell. Tout le détail de l'implémentation est disponible sur <http://model-ware.inria.fr/mtl>.

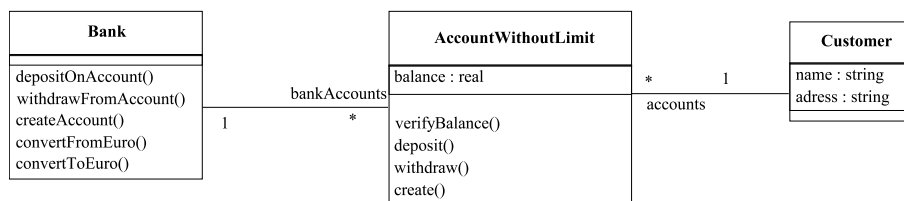


FIG. 4 – Le diagramme de classe pour le produit BS2

## 3.2 Aspect dynamique

Ci-dessus, nous avons proposé de modéliser le comportement de la LdP en utilisant les diagrammes de séquence d'UML2.0. Les diagrammes de séquence ne sont pas le seule formalisme pour la spécification de comportement ; les machines à états représentent un autre formalisme. Elles ont eu leur réussite dans le domaine de la conception rigoureuse des systèmes réactifs, en particulier pour la vérification (model checking), pour la simulation et pour la génération de code [8].

Même si les diagrammes de séquence et les machines à état diffèrent dans la vue qu'ils donnent (les diagrammes de séquence spécifient une *vue inter-objets* du système et les machines spécifient une *vue intra-objet des objets* du système), la relation entre les deux formalismes a attiré beaucoup d'attention et plusieurs techniques ont été proposé pour la génération (appelée aussi *synthèse*) automatique de machines à états à partir de diagrammes de séquence (le lecteur trouvera dans [15] une étude détaillée de ces travaux). La synthèse automatique de machines à états à partir de diagrammes de séquence permet d'assurer une traçabilité entre les exigences (spécifiées par les diagrammes de séquence) et la conception détaillée du système (spécifiée par les machines à états). La synthèse permet aussi d'offrir des possibilités pour la vérification, la simulation, et pour la génération de code à partir de diagrammes de séquence.

Nous proposons donc de formaliser la dérivation comme une synthèse automatique de machines à états spécifiques aux produits à partir de diagrammes de séquence de la LdP. Cette dérivation est réalisée en deux étapes : la dérivation des expressions de produits et ensuite la synthèse de machine à états. Les sous-sections suivantes détaillent ces deux étapes.

### 3.2.1 Des RESD-PL aux expressions de produits

La première étape dans notre approche de dérivation de comportements est de dériver les expressions de produits à partir de l'expression de référence de la LdP (RESD-PL). Les expressions de LdP contiennent des points de variation introduits par les trois constructions de variabilité : *optional*, *variation* et *virtual*. La dérivation des expressions de produits a besoin des décisions associées à ces points de

variation pour obtenir l'expression d'un produit particulier. La notion de *modèle de décision* est utilisée dans les LdP comme un moyen pour enregistrer les décisions nécessaires à la dérivation de produits [2].

Les résolutions de décisions pour un produit particulier sont définies dans ce que nous appelons *une instance du modèle de décision*.

**Definition 3** Une instance du modèle de décision (noté DMI par la suite) associée à un produit  $P$  est un ensemble de couples  $(name_i, Res)$ , où  $name_i$  désigne la nom de l'expression optionnelle, variation ou virtuelle et  $Res$  est la résolution liée au produit  $P$ . Les résolutions sont définies comme suit :

- La résolution d'une expression optionnelle est soit TRUE ou FALSE.
- Pour une expression variation comportant  $E_1, E_2, E_3..E_n$  pour expressions variantes, la résolution est l'entier  $i$  si l'expression  $E_i$  est sélectionnée.
- La résolution d'une expression virtuelle est une expression de raffinement  $E$ .

La Table 2 montre les instances du modèle de décisions associées aux quatre produits de la LdP banque. DM1 par exemple est l'instance du modèle de décision associé au produit BS1 qui supporte les limites sur les comptes et il ne supporte pas les opérations de gestion de devises.

TAB. 2 – Les instances du modèle de décision de la LdP banque

Produit	Instance du modèle de décision (DMI)
BS1	DM1 = {(settingLimit, TRUE), (settingCurrency, FALSE), (withdrawAccount, 1), (fromEuro, FALSE), (toEuro, FALSE)}
BS2	DM2 = {(settingLimit, FALSE), (settingCurrency, FALSE), (withdrawAccount, 2), (fromEuro, FALSE), (toEuro, FALSE)}
BS3	DM3 = {(settingLimit, FALSE), (settingCurrency, FALSE), (withdrawAccount, 2), (fromEuro, TRUE), (toEuro, TRUE)}
BS4	DM4 = {(settingLimit, TRUE), (settingCurrency, TRUE), (withdrawAccount, 1), (fromEuro, TRUE), (toEuro, TRUE)}

Les expressions de produits sont obtenues par une interprétation de l'expression de la LdP dans le contexte d'une instance du modèle de décision. Cette interprétation est basée sur les interprétations associées au trois constructions de variabilité :

**optional** La dérivation d'une expression optionnelle signifie décider de sa présence ou non dans l'expression d'un produit. Si la résolution de l'expression optionnelle est TRUE, cette expression sera présente dans l'expression du produit. Si la résolution est FALSE, l'expression sera supprimée et la dérivation renvoi une expression vide. Ceci est formalisé par l'interprétation suivante :

$$\llbracket \text{optional } name [ E ] \rrbracket_{DMi} = \begin{cases} E & \text{si } (name, TRUE) \in DMi \\ E_{\emptyset} & \text{si } (name, FALSE) \in DMi \end{cases}$$

L'expression vide  $E_{\emptyset}$  est la spécification d'une interaction vide.  $E_{\emptyset}$  est un élément neutre pour la composition séquentielle et l'alternative. Elle est aussi idempotente pour l'opérateur loop, c.à.d. :

- $E \text{ seq } E_0 = E ; E_0 \text{ seq } E = E$
- $E \text{ alt } E_0 = E ; E_0 \text{ alt } E = E$
- $\text{loop } (E_0) = E_0$ .

**variation** La dérivation d'une expression variation signifie le choix d'une expression variante parmi toutes les expressions variantes possibles. Ceci est formalisé comme suit :

$$\llbracket \text{variation name } [ E_1, E_2, \dots ] \rrbracket_{DMi} = E_j \text{ si } (name, j) \in DMi$$

**virtuel** La dérivation d'une expression virtuelle signifie son remplacement par une autre expression de raffinement. Si l'expression de raffinement n'est pas fournie, la dérivation garde par défaut l'expression virtuelle :

$$\llbracket \text{virtual name } [ E ] \rrbracket_{DMi} = E' \text{ si } (name, E') \in DMi, E \text{ sinon}$$

Utilisant les interprétations de base ci-dessus, la dérivation des expressions de produits peut être définie maintenant comme suit :

$\text{Expression-Produit} = \llbracket \text{Expression-LdP} \rrbracket_{DMI}$ , où  $DMI$  est l'instance de modèle de décision associé au produit qu'on veut dériver son expression.

Considérons la LdP banque, l'expression de produit BS2 :  $E_{BS2}$  est obtenue par interprétation de l'expression  $E_{BPL}$  dans le contexte de l'instance du modèle de décision  $DM2$ , c.à.d.  $E_{BS1} = \llbracket E_{BPL} \rrbracket_{DM2}$ , où :

$$DM2 = \{(\text{settingLimit}, \text{FALSE}), (\text{settingCurrency}, \text{FALSE}), (\text{withdrawAccount}, 2), (\text{fromEuro}, \text{FALSE}), (\text{toEuro}, \text{FALSE})\}$$

La dérivation des quatre expressions optionnelles et de l'expression variation dans  $E_{BS2}$  est réalisée comme suit :

$$\llbracket \text{optional settingLimit } [ \text{SetLimit} ] \rrbracket_{DM2} = E_0$$

$$\llbracket \text{optional settingCurrency } [ \text{SetCurrency} ] \rrbracket_{DM2} = E_0$$

$$\llbracket \text{optional toEuro } [ \text{ToEuro} ] \rrbracket_{DM2} = E_0$$

$$\llbracket \text{optional fromEuro } [ \text{FromEuro} ] \rrbracket_{DM2} = E_0$$

$$\llbracket \text{variation withdrawAccount} \\ [ \text{WithdrawWithLimit}, \text{WithdrawWithoutLimit} ] \rrbracket_{DM2} = \\ \text{WithdrawWithoutLimit}$$

L'expression de référence du produit BS2 résultat est l'expression  $E_{BS2}$  ci-dessous. Notons que comme  $E_0$  est un élément neutre pour  $\text{seq}$  et  $\text{alt}$ ,  $E_0$  est supprimé de l'expression du produit.

---

```

EBS2 = loop( Deposit alt (CreateAccount seq
(CreateAccountOk ) alt CreateAccountFailed) alt (
WithdrawWithoutLimit seq ( WithdrawOk alt WithdrawFailed)))

```

---

Appliquons le même processus pour le produit BS4 en dérivant l'expression  $E_{BPL}$  dans le contexte de DM4 nous obtenons l'expression  $E_{BS4}$  :

---

```

EBS4 = loop( Deposit alt (CreateAccount seq (CreateAccountOk
seq SetLimit seq SetCurrency ) alt CreateAccountFailed)
alt ( WithdrawWithLimit seq ( WithdrawOk alt
WithdrawFailed)) alt ( ConvertFromEuro) alt ( ConvertToEuro))

```

---

### 3.2.2 Des expressions de produits aux machines à états

À l'issue de la première étape de dérivation de comportements, nous avons obtenu des diagrammes de séquence spécifiques aux produits (sous forme des expressions de produits). L'étape suivante consiste à générer des machines à états à partir de ces diagrammes de séquence. Dans [16], nous avons proposé une méthode algébrique permettant la synthèse de machines à états à partir des diagrammes de séquence d'UML2.0 (spécifiés sous forme des expressions de référence). Nous utilisons cette méthode dans cette étape pour la synthèse de machines à états à partir des expressions de produits obtenues dans l'étape précédente.

Notre méthode de synthèse proposée dans [16] est basée sur un cadre algébrique pour les machines à états proche de celui de diagrammes de séquence d'UML2.0. Ce cadre définit trois opérateurs de composition de machines à états : "seq<sub>s</sub>" pour la séquence, "alt<sub>s</sub>" pour l'alternative, et "loop<sub>s</sub>" pour l'itération (voir Figure 5).

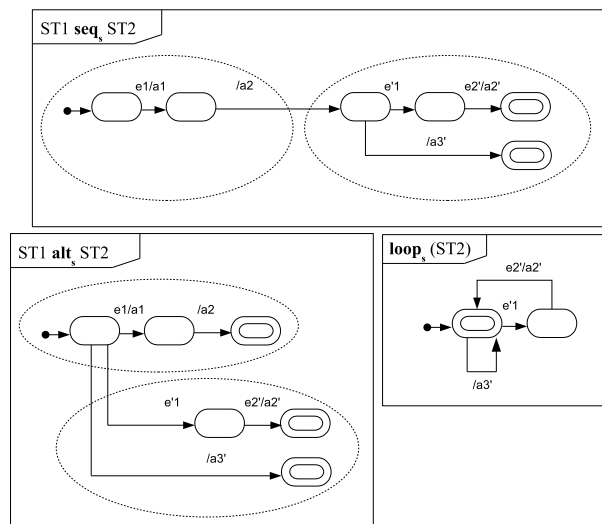


FIG. 5 – Les opérateurs de composition de machines à états

La synthèse utilise un algorithme  $P(DS, O)$  qui synthétise une machine à états pour l'objet  $O$  à partir de diagramme de séquence de base  $DS$  (cet algorithme est présenté en détail dans [16]). La Figure 6 illustre le principe de cet algorithme pour la synthèse d'une machine à état associée à l'objet Bank à partir de diagramme de séquence Deposit. Les événements du diagramme de séquence sont projetés sur la ligne de vie de l'objet et l'algorithme crée une transition dans la machine à états pour chaque événement. Ensuite les expressions de produits sont transformées en expressions de composition de machines à états appelées REST (Reference Expression for StatecharTs). Une REST est définie comme suit :

**Definition 4** Une expression de référence pour les machines à états (noted REST pour Reference Expression for Statecharts, par la suite) est une expression de la forme :

$$\begin{aligned} \langle \text{REST} \rangle &::= \langle \text{PRIMARY} \rangle \ ( \text{"alt}_s\text{"} \ \langle \text{REST} \rangle \ | \ \text{"seq}_s\text{"} \ \langle \text{REST} \rangle ) * \\ \langle \text{PRIMARY} \rangle &::= \text{ST}_\emptyset \ | \ \langle \text{IDENTIFIÉRIER} \rangle \ | \ \text{"} ( \langle \text{REST} \rangle \text{"} ) \ | \\ &\quad \text{"loop}_s\text{"} \ \text{"} ( \langle \text{REST} \rangle \ \text{"} ) \ | \\ \langle \text{IDENTIFIÉRIER} \rangle &::= ( [ "6" , "a" - "z" , "A" - "Z" ] \ | \ [ "0" - "9" ] ) * \end{aligned}$$

La construction des REST à partir des expressions de produits est basée sur une correspondance entre les opérateurs d'interaction et les opérateurs de machines à états. Les références aux interactions de base sont remplacées par des références aux machines à états de base obtenues par l'algorithme  $P(DS, O)$  et les opérateurs d'interaction sont remplacés par les opérateurs de machines à états. Par exemple, l'expression de référence REST pour l'objet Bank obtenue à partir de l'expression  $E_{BS2}$  ci-dessus est  $\text{REST}_{(BS2, Bank)}$  :

---


$$\begin{aligned} \text{REST}_{(BS2, Bank)} &= \text{loop}_s ( P(\text{Deposit}, Bank) \ \text{alt}_s ( P(\text{CreateAccount}, Bank) \\ &\text{seq}_s ( P(\text{CreateAccountOk}, Bank) ) \ \text{alt}_s ( P(\text{CreateAccountFailed}, Bank) ) \\ &\text{alt}_s ( ( P(\text{WithdrawWithLimit}, Bank) \ \text{seq}_s ( P(\text{WithdrawOk}, Bank) \ \text{alt}_s \\ &P(\text{WithdrawFailed}, Bank) ) ) ) ) \end{aligned}$$


---

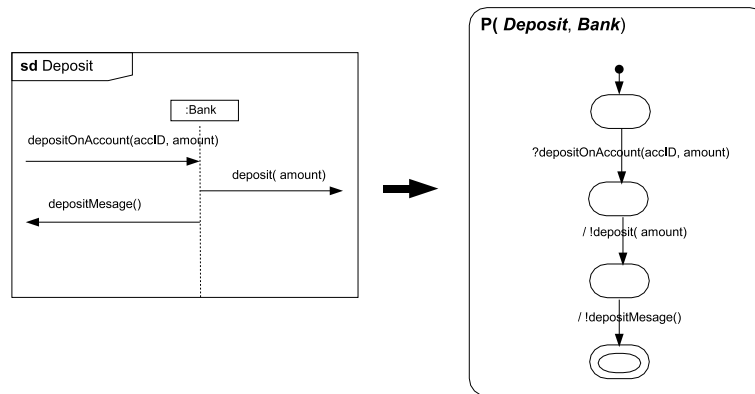


FIG. 6 – Exemple de la synthèse d'une machine à états à partir d'une interaction de base

Une machine à états est obtenue à partir de REST par composition de machine à états de base. La Figure 7 montre la machine à états obtenue à partir de  $\text{REST}_{(BS2, Bank)}$ .

Pour le produit BS4, une expression  $\text{REST}_{(BS4, Bank)}$  est obtenue (voir ci-dessous). La Figure 8 montre la machine à états de l'objet Bank obtenue à partir de  $\text{REST}_{(BS4, Bank)}$ . Notons que comme les deux produits BS2 et BS4 diffèrent par la présence ou non du montant de découvert et le support des opérations de conversion, les machines à états générées (La Figure 7 et la Figure 8) diffèrent par les transitions qui concernent l'ajout et la vérification du montant de découvert et celles qui concernent la conversion des devises. La différence entre les deux machines à états est illustrée par des zones grises dans la Figure 8.

---


$$\begin{aligned} \text{REST}_{(BS4, Bank)} &= \text{loop}_s ( P(\text{Deposit}, Bank) \ \text{alt}_s ( P(\text{CreateAccount}, Bank) \\ &\text{seq}_s ( P(\text{CreateAccountOk}, Bank) \ \text{seq}_s ( P(\text{SetLimit}, Bank) \ \text{seq}_s \\ &P(\text{SetCurrency}, Bank) ) ) \ \text{alt}_s ( P(\text{CreateAccountFailed}, Bank) ) \ \text{alt}_s \\ &( P(\text{WithdrawWithLimit}, Bank) \ \text{seq}_s ( P(\text{WithdrawOk}, Bank) \ \text{alt}_s \\ &P(\text{WithdrawFailed}, Bank) ) ) \ \text{alt}_s ( P(\text{ConvertFromEuro}, Bank) ) \ \text{alt}_s \\ &( P(\text{ConvertToEuro}, Bank) ) ) \end{aligned}$$


---



**Remerciements** This work has been partially supported by the ITEA project ip00009, FAMILIES in the Eureka  $\Sigma$ ! 2023 Programme.

## Références

- [1] FAMILIES project, <http://www.esi.es/en/projects/families/>, 2003.
- [2] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. *Component-based Product Line Engineering with UML*. Component Software Series. 2001.
- [3] C. Atkinson, J. Bayer, and D. Muthig. Component-Based Product Line Development : The Kobra Approach. In P. Donohoe, editor, *First Software Product Line Conference (SPLC1)*, pages 289–309. Kluwer Academic Publishers, August 2000.
- [4] P. Clements and L. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wisley, Boston, 2001.
- [5] Sybren Deelstra, Marco Sinnema, and Jan Bosh. Product derivation in software product families : a case study. *The journal of Systems and Software*, 74(2) :173–194, January 2004.
- [6] O. Flege. System Family Architecture Description Using the UML. Technical Report IESE-Report No. 092.00/E, IESE, December 2000.
- [7] H. Gomaa. Object Oriented Analysis and Modeling for Families of Systems with UML. In W. B. Frakes, editor, *IEEE International Conference for Software Reuse (ICSR6)*, pages 89–99, June 2000.
- [8] D. Harel and E. Gery. Executable object modeling with statecharts. In *Proceeding of International Conference on Software Engineering (ICSE 1996)*, 1996.
- [9] W. El Kaim. Managing variability in the lcat split/daisy. In *Proceedings of Product Line Architecture Workshop. The First Software Product Line Conference (SPLC1)*, pages 21–32, 2000.
- [10] A. Maccari and A. Heie. Managing infinite variability. In Jilles van Gorp and Jan Bosch, editors, *Software Variability Management Workchop*, number IWI preprint 2003-7-01, pages 28–34. Research institute of Mathematics and Computing Science of the university of Groningen, 2003.
- [11] L.M. Northrop. A framework for software product line practice -version 3.0. Web <http://www.sei.cmu.edu/plp/framework.html>, Software Engineering Institute, 2002.
- [12] Object Management Group OMG. Unified modeling language specification version 2.0 : Superstructure. Technical Report pct/03-08-02, OMG, 2003.
- [13] D.L Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE2(1) :1–9, March 1976.
- [14] S. Robak, R. Franczyk, and K. Politowicz. Extending the uml for modeling variability for system families. *International Journal of Applied Mathematics Computer Sciences*, 12(2) :285–298, 2002.
- [15] T. Ziadi. *Manipulation de Lignes de Produits en UML*. PhD thesis, University of Rennes 1, Dec 2004.
- [16] T. Ziadi, L. Hérouët, and J-M. Jézéquel. Revisiting statecharts synthesis with an algebraic approach. In *International Conference on Software Engineering, ICSE'26, Edinburgh, Scotland, United Kingdom*, May 2004.
- [17] T. Ziadi, L. Hérouët, and J-M. Jézéquel. Towards a uml profile for software product lines. In *Proceedings of the Fifth International Workshop on Product Family Engineering (PFE-5)*, volume 3014 of LNCS, pages 129–139. Springer Verlag, 2003.