Une ligne de produits corrects par construction

Thi-Kim-Dung Pham[1], Catherine Dubois[2], Nicole Levy[1]
1. Conservatoire National des Arts et Métiers, lab. Cedric, Paris
2. ENSIIE, lab. Samovar, Évry

Résumé

L'ingénierie des lignes de produits logiciels met l'accent sur la gestion de la variabilité et la réutilisation. Dans cet article, nous complétons cette approche avec la recherche de garanties de correction sur les produits issus d'une ligne de produits. Nous proposons une méthode permettant de produire des produits corrects par construction à partir d'une ligne de produits. Cette méthode s'appuie sur un langage, FFML, inspiré de FoCaLiZe et incorporant des mécanismes pour exprimer la variabilité, et deux outils : un compilateur vers FoCaLiZe et un composeur. Le composeur permet de générer automatiquement des produits corrects par construction à partir d'une configuration valide. Les outils de FoCaLiZe permettent de vérifier les preuves de correction et de générer du code exécutable OCaml. La méthode est illustrée dans cet article sur un exemple simple de ligne de produits.

1 Introduction

D'après Clements et Northrop dans [3], une ligne de produits décrit un ensemble (fini) de systèmes qui partagent un ensemble commun de caractéristiques (*features* en anglais) répondant à des besoins spécifiques, par exemple ceux d'un segment de marché ou d'un domaine. Ces systèmes sont développés par une composition précise d'un ensemble commun d'artefacts de base (*assets* en anglais).

Ainsi, l'utilisation d'une ligne de produits a principalement deux objectifs : l'expression de la variabilité et la réutilisation. Une ligne de produits est spécifique à un domaine et exprime la variabilité du domaine et des besoins du problème décrit. Elle identifie systématiquement les points communs et les points variables. Ces points sont les caractéristiques mentionnées dans la définition de Clements et Northrop. Pour les visualiser, il est fréquent d'utiliser un diagramme de caractéristiques (*Feature diagram*) qui permet de structurer les décisions sous forme d'un arbre *FODA* (pour *Feature Oriented Domain Analysis*) [5] avec des relations entre nœuds père et fils qui peuvent être Obligatoires ou Optionnelles, ou décrire des choix exclusifs ou multiples.

Le deuxième objectif essentiel des lignes de produit est la réutilisation. Pour cela, à chaque caractéristique sont associés des artefacts à réutiliser. Ainsi, configurer un système c'est choisir des caractéristiques et réutiliser les artefacts associés.

Dans les différentes définitions des lignes de produits, la notion d'artefact varie et peut recouvrir différents sens. Il peut s'agir d'une spécification en langue naturelle ou en langage formel ou même d'un morceau de code. L'idée est que l'artefact représente la décision associée à une caractéristique et le problème est que pour le réutiliser, il faut pouvoir le composer avec d'autres artefacts.

Notre objectif est la définition d'une ligne de produits dans laquelle un artefact décrit formellement est associé à chaque caractéristique avec, pour objectif, de pouvoir configurer des systèmes corrects par construction, c'est-à-dire prouvés corrects par rapport à leur spécification. Remarquons, que si de nombreux travaux portent sur la validation d'une ligne de produits, peu portent sur la validation des produits issus de lignes de produits. Dans [1], les auteurs proposent de trouver des défauts dans les produits générés en C et Java en suivant une démarche basée sur le model-checking. Notre approche est proche de celle de Thüm [8] qui utilise la notion de contrat et les vérifie de manière déductive. Nous nous distinguons principalement par le fait que nous réutilisons et composons les preuves automatiquement.

Afin de pouvoir déployer une ligne de produits corrects par construction, nous proposons un langage adapté à la description des lignes de produits, orienté variabilité, FFML, ainsi qu'un mécanisme de composition qui permet de construire des produits corrects par construction. Le langage FFML est inspiré de FoCaLiZe, langage qui permet de spécifier un logiciel, de l'implémenter dans un langage fonctionnel et de prouver la correction du programme à l'aide du prouveur Zenon [4]. FFML, comme nous l'avons montré dans [7], lui apporte une orientation Ligne de Produits en introduisant des modules spécifiques et des constructions décrivant les décisions de développement prises. Deux outils accompagnent cette proposition : un compilateur de FFML vers FoCaLiZe ainsi qu'un outil de génération automatique de produits par composition d'artefacts définis à l'aide du langage FFML. Les deux outils sont présentés succinctement ici, ils sont formalisés dans [6].

La ligne de produits utilisée comme exemple dans cet article, inspirée de [8], décrit la gestion d'un compte bancaire. La caractéristique racine, notée BA pour BankAccount, décrit la gestion de base d'un compte : mémorisation du solde courant et transactions (ajout ou retrait d'argent) sur un compte. Un client peut retirer plus d'argent du compte que disponible dans une limite fixée. Comme le montre le diagramme de caractéristiques de la figure 1, la caractéristique BA dispose de trois caractéristiques filles optionnelles DailyLimit (DL), LowLimit (LL) et Currency (CU). DL permet à la banque de limiter le montant retiré chaque jour. LL autorise un client à débiter son compte uniquement d'un montant supérieur à une limite basse. CU permet de gérer la devise. Enfin, la caractéristique CE pour CurrencyExchange, fille optionnelle de CU,

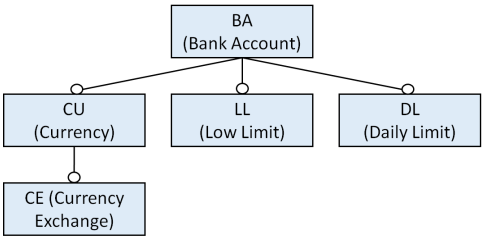

FIGURE 1 – Ligne de produits des comptes bancaires

réalise la conversion de devises. Un produit est configuré par une sélection de ces caractéristiques. Par exemple, un utilisateur qui désire un système de gestion bancaire avec une limite fixée pour les retraits journaliers et à chaque retrait, choisira les caractéristiques BA, LL et DL.

L'article est structuré de la manière suivante : le langage FFML est présenté dans la section suivante et illustré sur l'exemple de la ligne de produits des comptes bancaires. En section 3, nous montrons comment est généré un produit correct par construction à partir d'une configuration. Nous concluons en section 4.

2 Développement de la ligne de produits

2.1 Artefacts en FFML

A chaque nœud du modèle de caractéristiques est associé un artefact écrit dans un module du langage FFML. Ce dernier contient 3 parties : la spécification, l'implantation et les preuves de correction. La spécification contient les signatures des fonctions développées dans le module et les propriétés attendues sous la forme de formules du premier ordre, l'implantation des fonctions est décrite dans un style fonctionnel et les preuves de correction des fonctions sont faites par rapport aux propriétés énoncées dans la spécification. Le langage FFML est inspiré de FoCaLiZe, ainsi les preuves sont écrites dans le langage de preuve de FoCaLiZe et apparaissent dans FFML sous la forme de commentaires spécialisés. La particularité de FFML est qu'il demande de ne décrire, pour une caractéristique donnée, que ce qui change par rapport à la caractéristique dont elle est issue (ou caractéristique parent) : nouvelles fonctions, propriétés nouvelles ou modifiées, nouvelles preuves. La syntaxe concrète et la sémantique de FFML sont décrites dans [6].

Les modules associés aux nœuds BA, DL et LL du modèle de la figure 1 sont reproduits (partiellement) à la figure 2, ces modules sont répartis dans des fichiers différents et portent le nom de la caractéristique associée.

Le listing 1 montre le code associé à la caractéristique racine BA. Deux fonctions $update$ et get_bal ainsi qu'une constante $over$ sont déclarées et définies. Le type BA désigne le type des comptes bancaires, il est associé au type concret int à la ligne 11. Un compte est donc représenté par son solde. La spécification de la fonction $update$ (i.e. son contrat) est ici la propriété bal_succ_BA. Elle indique que le retrait ou le dépôt est possible si, après cette opération, une somme suffisante reste sur le compte (solde supérieur à $over$). Une propriété dite invariante, $ba\_bal\_gr\_over$, spécifie la fonction get_bal. Cette propriété devra être satisfaite par toutes les caractéristiques filles et donc tous les produits issus de la ligne de produits. Le module se termine par les scripts de preuve des propriétés précédentes : la preuve de la première propriété se fait en utilisant la définition des fonctions $update$ et get_bal, la seconde propriété est admise, elle ne peut être prouvée directement. Elle est en fait induite par les preuves que la propriété est préservée par les opérations qui ont un résultat de type BA, par exemple ici la preuve de la propriété bal_succ_BA.

```
fmodule BA
  signature update: BA -> int -> BA;
  signature get_bal: BA -> int;
  signature over: int;
  contract get_bal :: invariant property
        ba_bal_gr_over:
   all x : BA, get_bal(x) >= over;
  contract update :: property bal_succ_BA: all
        x : BA, all a : int,
   (get_bal(x) + a) >= over -> get_bal (update
        (x,a)) = get_bal(x) + a;
  representation = int;
  let get_bal(x) = x;
  let over = 0;
  let update (x, a) =
     if ((get_bal(x) + a) >= over) then
          get_bal(x) + a
     else get_bal(x);
  proof of ba_bal_gr_over = assumed;
  proof of bal_succ_BA = by definition of
        update, get_bal;
```

Listing 1 – BA

```
fmodule LL from BA
  signature limit_low: int;
  contract update :: property bal_succ_LL
   refines BA!bal_succ_BA
   extends premise ((a >= 0) || (a <=
        limit_low));
  representation = BA;
  let limit_low = -10;
  let update (x, a) =
    if ((a >= 0) || (a <= limit_low))
    then BA!update (x, a)
    else x;
  proof of bal_succ_LL = by definition of
        update, get_bal, over
     property BA!bal_succ_BA, int_ge_le_eq;
```

Listing 2 – LL

```
fmodule DL from BA
  signature limit_with: int ;
  signature get_with: DL -> int ;
  signature next_day : DL -> DL;
  contract update :: property bal_succ_DL
   refines BA!bal_succ_BA
   extends premise (a <= 0) /\ (get_with(x) +
        a >= limit_with);
  contract update :: property dl_upd_limit:
        all x : DL, all a : int,
   ((a<=0)/\(get_with(x)+a>=limit_with)) ->
     get_with(update(x,a) = get_with(x) + a;
  representation extends BA with int;
  let limit_with = -70;
  let get_with (x) = second(x);
  let next_day (x, a) = (x, limit_with);
  let update (x, a) =
     if (a <= 0) then
          if (get_with(x)+a>=limit_with) then
            (BA!update(x,a), get_with(x)+a)
          else x
     else (BA!update(x,a), get_with(x));
  proof of bal_succ_DL =
    <1>1 assume x: DL, a : int,
     hypothesis h1: (a <= 0) /\ (get_with(x) +
          a >= limit_with),
     prove (get_bal(x) + a) >= over -> (
          get_bal (update(x,a))) = get_bal(x)
          + a
     <2>1 prove first(update(x,a)) = BA!update
          (first(x),a)
          by definition of first, make,
               update hypothesis h1
     <2>e qed by step <2>1 definition of over,
          get_bal property BA!bal_succ_BA
    <1>e conclude ;
  proof of dl_upd_limit = by definition of
       update, get_with;
```

Listing 3 – DL

FIGURE 2 – Quelques artefacts de la ligne de produits des comptes bancaires

Le listing 3 définit DL à partir de BA. Le module introduit deux nouvelles fonctions get_with et $next\_day$ (les propriétés et preuves relatives à $next\_day$ ne sont pas développées ici) et une nouvelle constante $limit\_with$. Toutes les fonctions définies dans BA sont présentes dans DL mais $update$ a un comportement légèrement modifié : on ne doit pas dépasser le montant journalier de retrait autorisé. La spécification de $update$ est donc un raffinement de la spécification de $update$ dans BA (cette construction a été décortiquée dans [7]). On remarquera que le script de preuve de cette propriété réclame la preuve d'un lemme intermédiaire. Une nouvelle propriété de $update$, dl_upd_limit, est ajoutée et prouvée. Le type concret est étendu : on adjoint à la représentation du compte bancaire dans BA, un autre entier qui est la somme retirée dans le jour. La représentation devient donc un couple d'entiers. La fonction $update$ est redéfinie et ré-utilise la fonction $update$ de BA (notée $BA!update$). Le module LL (listing 3) suit un schéma similaire.

2.2 Vérification des preuves de correction

La compilation en FoCaLiZe, à ce stade, permet de faire les preuves des propriétés et de les vérifier. La démarche du développeur est la suivante : pour une caractéristique donnée, il écrit le code FFML et le compile à l'aide du compilateur FFML ; il obtient un premier code FoCaLiZe dont il complète les scripts de preuve ; les preuves sont faites par Zenon et le tout est vérifié - par Coq - lorsque la compilation du fichier FoCaLiZe est lancée. Une fois les scripts de preuve complétés et vérifiés, ils sont copiés-collés aux bons endroits dans le fichier FFML, en tant que commentaires.

3 Génération de produits corrects par construction

Une fois la ligne de produits décrite en FFML, un utilisateur peut obtenir différents produits de manière automatique (ou quasi-automatique). Pour cela, il doit fournir une configuration valide, c'est-à-dire un ensemble de caractéristiques qui respectent les contraintes exprimées dans le modèle. Par exemple {BA, DL, LL} est une configuration valide mais {BA, DL, CE} ne l'est pas car CE requiert la présence de CU. La vérification de la validité d'une configuration est orthogonale à notre travail et de nombreux outils rendant ce service existent (par exemple [2]). L'outil présenté ici se veut complémentaire des outils existants et considère les configurations étudiées comme ayant déjà été validées. Nous supposons également que les contraintes du modèle de caractéristiques font foi. Ainsi nous ne vérifions pas que les caractéristiques puissent engendrer des interactions. Si deux caractéristiques ne sont pas compatibles, une contrainte d'exclusion est attendue.

A partir d'une configuration valide et des fichiers FFML associés aux caractéristiques choisies, le composeur FFML produit un fichier FFML qui correspond au produit demandé. A l'aide du compilateur FFML vers FoCaLiZe, il est traduit en un fichier FoCaLiZe qui, une fois compilé produira un exécutable en OCaml.

Le résultat de la composition pour la configuration {BA, LL, DL} est illustré ci-dessous (listing 4).

```
1  fmodule DLLL from LL
2    signature limit_with : int;
3    signature get_with : DLLL -> int;
4    signature next_day : DLLL -> DLLL;
5    contract update ::  property bal_succ_DL
6     refines LL!bal_succ_LL
7     extends premise (a <= 0) /\ (get_with(x) + a >= limit_with);
8    representation extends LL with int;
9    ...
10   let update (x,a) =  if (a <= 0) then
11    if (get_with(x) + a >= limit_with)
12    then (LL!update(x,a),get_with(x) + a)
13    else x
14    else (LL!update(x,a),get_with(x));
15   proof of bal_succ_DL =  ... ;
```

Listing 4 – Le module FFML correspondant à la configuration {BA, LL, DL}

On peut le voir comme une extension/modification des fonctionnalités associées à LL. Le composeur agit en appliquant une opération de composition binaire,

capable de combiner les propriétés, les définitions de fonction et les scripts de preuve [6]. Cette opération génère automatiquement de nouvelles propriétés, implantations et scripts de preuve. Les modules FFML associées aux caractéristiques de la configuration sont composés deux par deux en suivant un parcours *Gauche-Droite-Racine* du modèle de caractéristiques. Notons que l'arbre FODA considéré est ordonné et que la composition suit cet ordre, donnant, par exemple pour {BA, DL, LL}, un module qui dérive de LL plutôt que de BA ou DL.

La table 1 présente un bilan quantitatif de la génération de produits pour l'exemple des comptes bancaires. Le code FFML compte 14 propriétés en FFML et 18 preuves faites par Zenon. Il y a plus de preuves en FoCaLiZe qu'en FFML car la compilation vers FoCaLiZe génère des preuves, toutes faites automatiquement [7]. A partir de cette ligne de produits, nous pouvons construire 12 produits. La colonne cP (resp. cPf) dénombre les propriétés (resp. scripts de preuve) générées par la composition. La colonne auto indique le nombre de preuves faites par Zenon à partir des scripts générés par la composition et manu le nombre de preuves dont le script a été donné manuellement [1]. La colonne Ze-Pf donne le nombre total de preuves à faire. Les autres configurations ne figurent pas dans la table car les produits correspondants sont les modules FFML associés aux nœuds du modèle de caractéristiques. Ainsi, le produit associé à {BA, CU, CE} est le module CE.

Configuration	cP	cPf	Ze-Pf	auto	manu
{BA,DL,LL}	5	6	7	7	0
{BA,DL,CU}	4	5	7	6	1
{BA,LL,CU}	3	3	6	6	0
{BA,DL,LL,CU}	5	6	8	7	1
{BA,LL,CU,CE}	3	3	8	7	1
{BA,DL,CU,CE}	3	3	9	7	2
{BA,DL,LL,CU,CE}	5	6	10	8	2

TABLE 1 – Bilan quantitatif

4 Conclusion

Dans cet article, nous avons illustré la méthode que nous proposons pour produire des produits corrects par construction à partir d'une ligne de produits. Cette méthode s'appuie sur un langage, FFML, inspiré de FoCaLiZe et incorporant des mécanismes pour exprimer la variabilité, et deux outils : un compilateur et un composeur. Le composeur permet de générer automatiquement des produits corrects par construction à partir d'une configuration valide et le compilateur permet de traduire tout module FFML en un programme FoCaLiZe. Cette compilation permet non seulement d'obtenir du code exécutable écrit en OCaml mais aussi de vérifier les preuves. Actuellement, certains scripts de preuve doivent être produits manuel-

1. Tous les scripts manuels sont dus à la même erreur de Zenon qui est en cours de correction.

lement, des travaux complémentaires sont en cours pour automatiser ces preuves. Enfin, la méthode est expérimentée actuellement sur un exemple plus conséquent : une ligne de produits concernant le poker et ses différentes règles applicables. Cette ligne de produits permet de générer automatiquement 16 produits et leurs propriétés, en combinant ses 12 caractéristiques. Les preuves à réaliser sont simplifiées et sont en partie obtenues automatiquement.

Références

[1] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for product-line verification : case studies and experiments. In *ICSE'15, San Francisco, CA, USA, 2013*, pages 482–491, 2013.

[2] D. Benavides, S. Segura, P. Trinidad, and A. R. Cortés. FAMA : tooling a framework for the automated analysis of feature models. In *VaMoS 2007, Limerick, Ireland, 2007*, pages 129–134, 2007.

[3] P. Clements and L. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, 2001.

[4] FoCaLiZe. http://focalize.inria.fr/.

[5] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, 1990.

[6] T.-K.-D. Pham. *Development of Correct-by-Construction Software using Product Lines*. Draft, thèse de doctorat, CNAM, soutenance prévue en 2017.

[7] T.-K.-D. Pham, C. Dubois, and N. Levy. Vers un développement formel non incrémental. In *AFADL'2016*, pages 106–113, Besançon, France, 2016.

[8] T. Thüm, I. Schaefer, M. Kuhlemann, and S. Apel. Proof composition for deductive verification of software product lines. In *VAST'11*, pages 270–277. IEEE Computer Society, 2011.