Development of Correct-by-Construction Software using Product Lines

Thi-Kim-Dung Pham

le cnam

École Doctorale Informatique, Télécommunications et Électronique

Centre d'étude et de recherche en informatique et communications

THÈSE *présentée par* :

Thi-Kim-Dung PHAM

soutenue le : **16 Novembre 2017**

pour obtenir le grade de : **Docteur du Conservatoire National des Arts et Métiers**

Discipline : **Sciences de l'information et de la communication**

Spécialité : **Informatique**

Development of Correct-by-Construction Software using Product Lines

THÈSE dirigée par :

Mme DUBOIS Catherine	*Professeure, ENSIIE (Directrice de thèse)*
Mme LEVY Nicole	*Professeure, CNAM (CoDirectrice de thèse)*

RAPPORTEURS :

M MOREAU Pierre-Etienne	*Professeur, École des Mines de Nancy*
M SALINESI Camille	*Professeur, Université Paris 1 Panthéon - Sorbonne*

JURY :

Mme METAIS Elisabeth	*Professeure, CNAM (Président du jury)*
Mme DUCHIEN Laurence	*Professeure, Université Lille - Sciences et Technologies*
M GIBSON J.Paul	*Maître de conférences, Telecom SudParis*
Mme URTADO Christelle	*Maître de conférences, IMT Mines d'Alès*

Acknowledgments

I gratefully acknowledge my supervisors, Catherine Dubois and Nicole Levy, who trusted my capabilities. They have given me the possibility chose my research topic. They have always taken the time to answer my questions and to find solutions to scientific as well as non-scientific problems. Their critical comments helped me to find a challenging research topic and to improve my writing. They have been really supportive of my career goals, to provide me with the protected academic time to pursue those goals.

I would like to thank Pierre-Etienne Moreau and Camille Salinesi who have reviewed this thesis. Thank you for spending the time to read this document in details and provide valuable feedback and constructive suggestions. I would also like to thank them as well as Laurence Duchien, Paul Gibson, Elisabeth Metais and Christelle Urtado, for accepting to be part of the defense committee.

Besides my supervisors, I would like to thank other professors who supported me for working on new languages and motivated me to keep going, whenever needed. François Pessaux helped me to develop my tools at the beginning. Elisabeth Metais encouraged me to overcome difficulties in research

A special thanks to Le Tuan Anh, who as a good friend, was always willing to help and gave his best suggestions. It would have been a lonely lab without him. Many thanks to Tran Quang Hoa, Trang Long, Tran Khang for encouraging me in bad time. My research would not have been possible without their helps.

Nobody has been more important to me in the pursuit of this project than the members of my family. I would like to thank my parents, whose love and guidance are with me in whatever I pursue. They are the ultimate role models. Most importantly, I wish to thank my loving and supportive husband, Vu Thanh Son, and my wonderful daughter, Vu Lumia, who provide unending inspiration.

Abstract

Nowadays diversity of software raises difficulties for many companies and organizations. While software product line engineering is considered as a solution and used in many domains for decades, research about the development of correct-by-construction software product lines is still up-to-date and necessary. We begin this thesis with an overview of how existing techniques were applied to develop and guarantee the correctness of software product lines. We propose a solution based on the design and implementation of a language FFML (Formal Feature Module Language) inspired by the FoCaLiZe language and providing mechanisms for expressing reuse and variability. This language allows to specify, implement a feature and prove correctness by giving hints to the automatic theorem prover Zenon. We develop a compiler for FFML to FoCaLiZe. We also provide a composition mechanism which applied to a valid user configuration automatically computes a final product correct-by-construction, meaning that the code of the product is correct with respect to its specifications. The specifications of the final product are obtained by composing the specifications of the features involved in the user configuration, the code is obtained by composing the code of the features and the proofs are also produced by composition. Finally, we evaluate our methodology by building a poker software product line.

Keywords : Correct-by-Construction Development, Software Product Lines, variability, formal specification, Formal proof, FoCaLiZe, Zenon

Résumé

Aujourd'hui, la diversité des logiciels pose des difficultés à de nombreuses entreprises et organisations. Alors que l'ingénierie des lignes de produits logiciels est considérée comme une solution possible et utilisée dans de nombreux domaines depuis des décennies, la problématique du développement de lignes de produits corrects par construction est toujours d'actualité. Cette thèse commence par une présentation de quelques techniques existantes appliquées pour développer et garantir la correction des lignes de produits logiciels. Nous proposons une solution basée sur la conception et la mise en œuvre d'un langage FFML (Formal Feature Module Language) inspiré du langage FoCaLiZe et fournissant des mécanismes pour exprimer la réutilisation et la variabilité. Ce langage permet de spécifier, implanter une fonctionnalité et prouver sa correction en donnant des indications au prouveur automatique de théorèmes Zenon. Nous développons un compilateur de FFML en FoCaLiZe. Nous fournissons également un mécanisme de composition qui, appliqué à une configuration valide fournie par l'utilisateur, produit automatiquement un produit final correct-par-construction, ce qui signifie que le code produit est correct par rapport à ses spécifications, elles-aussi obtenues par composition des spécifications des caractéristiques impliquées dans la configuration de l'utilisateur. Enfin, nous évaluons notre méthodologie en construisant une ligne de produits logiciels pour le poker.

Mots-clefs: Développement correct-par-construction, ligne de produits logiciels, variablilité, spécification formelle, preuve formelle, FoCaLiZe, Zenon

Résumé de la thèse

Instruction

Dans SPLE, un actif est un artefact employé dans le développement de logiciels. Les actifs doivent être alloués et gérés pour créer différents produits. Ainsi, les points communs et la variabilité entre les logiciels doivent être identifiés et les produits d'un SPL - appelés membres de la gamme de produits ou variantes de produits - sont caractérisés par leurs fonctionnalités communes et aussi par leurs différences [Pohl et al. 2005]. Pohl et al. a également introduit un cadre de ligne de produits logiciels qui est considéré comme une base de nombreuses technologies pour le développement de lignes de produits logiciels (SPL). Le principe, centré sur ce qui pourrait être réutilisé par d'autres produits, devrait être réalisé en premier, puis sur ce qui peut être ajouté afin de répondre à diverses exigences. Récemment, de nombreuses approches ont été proposées pour analyser les points communs et la variabilité de la modélisation en se concentrant sur ce principe. De nombreux types de modèles ont été proposés pour l'analyse des SPL, tels que le modèle de caractéristiques [Kang et al. 1990]; modèles de variabilité orthogonale [Pohl et al. 2005], ou modèle de décision [Schmid et al. 2011], etc. Cependant, l'approche appelée modélisation de caractéristiques, qui utilise des modèles de caractéristiques pour représenter les relations graphiques entre les caractéristiques est la plus populaire. Ces modèles offrent une flexibilité dans l'ajout de nouvelles fonctionnalités pour créer différents produits de SPL, en s'adaptant aux clients individuels.

La dérivation de produit est le processus de création de variantes de produit à partir d'un SPL. Ce processus définit la manière dont les actifs sont sélectionnés en fonction d'une configuration de fonction donnée et spécifie comment ces éléments sont assemblés afin de

générer le produit souhaité. L'approche de la programmation générative est une solution qui a d'abord été proposée et réalisée par Czarnecki [Czarnecki and Eisenecker 1999] dans le but de générer automatiquement les produits via un générateur. Récemment, de nombreuses approches se sont concentrées sur la programmation orientée fonction (FOP), une technique de mise en œuvre qui s'est avérée efficace pour générer des produits [Prehofer 1997; Apel et al. 2013a; Thüm et al. 2014b]. Une autre technique de mise en œuvre, la programmation orientée Delta [Schaefer et al. 2010], prend également en charge le processus de génération de produits, mais plus complexe et flexible que FOP.

L'exactitude-par-construction (CbyC) est l'une des approches efficaces principalement pour le développement de systèmes automatisés de sécurité et de sécurité en construisant des unités démontrables [Hall and Chapman 2002; Kourie and Watson 2012; Watson et al. 2015]. En utilisant un paradigme de raffinement, ces unités sont construites à chaque étape d'affinage, et ainsi les systèmes sont gérés et vérifiés plus facilement. Notre principale préoccupation de cette thèse est de développer des SPL et d'appliquer cette approche dans le but de générer automatiquement des produits corrects par construction. En raison des avantages de cette approche [Batory 2015], nous sommes convaincus que notre direction de recherche est prometteuse, surtout après avoir détecté de nombreuses techniques et méthodes existantes qui peuvent aider à élaborer cette direction.

Objectifs de la thèse

Notre objectif principal est de développer des SPL corrects par construction.En examinant la littérature de recherche sur l'application de l'approche CbyC aux SPL, nous surmontons les lacunes des recherches précédentes et identifions les principales propositions pour notre direction. Premièrement, nous nous concentrons sur les travaux publiés sur les techniques d'analyse et de mise en œuvre des SPL qui permettent de créer ensemble des modules de spécifications, de code d'implémentation et d'exactitude dans les mêmes unités afin de les réutiliser plus efficacement. Deuxièmement, pour recevoir les produits finaux corrects en plus de poursuivre une technique de mise en œuvre existante, nous avons besoin d'une méthodologie dans laquelle les unités sont composées pour créer les produits finaux dont la correction est maintenue. Troisièmement, afin de réaliser la méthodologie,

nous avons réalisé que le développement de SPL correct par construction en utilisant des langages existants se heurte à de nombreux problèmes et limitations. Par conséquent, nous avons créé un nouveau langage qui aide les développeurs à écrire les modules plus facilement et plus efficacement. Quatrièmement, pour adapter les demandes de la méthodologie, nous avons défini des mécanismes de composition indépendants à appliquer à ces modules. Sur la base de nos connaissances acquises grâce aux techniques existantes, nous construisons un outil de génération automatisé pour élaborer ces mécanismes. Enfin, pour évaluer notre approche, nous l'avons appliquée à un cas pratique: un Poker SPL.

Contribution de thèse

- Nous avons développé un langage, appelé FFML (Formal Feature Module Language), qui permet d'exprimer la variabilité des artefacts. Ce nouveau langage se compose de syntaxe et de sémantique et soutient le développement de SPL suivant l'approche CbyC plus facilement. Un SPL est analysé dans les modules qui sont écrits en FFML. Un outil appelé outil FFML Compiler a été construit pour compiler ces modules.

- Les mécanismes de composition ont été définis pour composer des modules FFML de SPL. Nous avons développé un outil générateur, appelé FFML Product Generator, qui implémente ces mécanismes de composition pour générer automatiquement les produits finaux corrects.

Chapitre 1: État de l'art

Dans ce chapitre, nous visons à introduire les connaissances de base sur lesquelles la thèse est basée.

Lignes de produits logiciels

Le terme ligne de produits logiciels a d'abord été mentionné par Bass et al. dans [Bass et al. 1998] tout en proposant une architecture pour cela. Au cours de la dernière décennie, ce terme a été utilisé pour remplacer le terme famille de programmes. L'ingénierie de SPL a été enrichie par Clements et Northrop où les actifs de base sont déterminés et

gérés [Clements and Northrop 2001]. En plus de proposer les mécanismes de partage d'un ensemble commun d'actifs de base, les auteurs ont décrit comment appliquer ces mécanismes dans la pratique. La variabilité a été mentionnée comme une clé pour ces mécanismes. Pohl, Böckle et Linden ont défini le cycle de développement complet d'un SPL [Pohl et al. 2005], dans lequel il y a deux processus principaux: l'ingénierie de domaine et l'ingénierie d'application. L'intention du processus d'ingénierie de domaine est d'établir une plate-forme réutilisable pour une ligne de produits qui contient tous les types d'artefacts. Les artefacts associés sont liés les uns aux autres. La communauté et la variabilité sont définies dans la plateforme. En revanche, l'ingénierie d'application est le processus par lequel les produits finaux sont dérivés de la plate-forme établie dans le processus d'ingénierie de domaine et des demandes des clients. Séparer le développement de la gamme de produits en ces deux processus garantit l'expression de la variabilité et offre la flexibilité nécessaire pour choisir les artefacts pour les différents produits, selon les besoins.

Modélisation de fonctionnalités

La gestion de la variabilité est une tâche principale liée au succès des SPL. Dans SPLE, une caractéristique est un comportement caractéristique spécifié comme une unité d'exigences, des fonctions techniques ou des caractéristiques non fonctionnelles [Kang et al. 1990], qui est associé à ses actifs dans un SPL. Dans la modélisation de caractéristiques, un produit d'un SPL diffère des autres par ses caractéristiques impliquées. Un modèle de caractéristiques est défini comme une représentation de "l'information de tous les produits possibles d'une gamme de produits logiciels en termes de caractéristiques et de relations entre eux" dans [Benavides et al. 2010]. Un modèle de fonctionnalité d'un SPL est un ensemble hiérarchique des fonctionnalités du SPL. Ces caractéristiques sont communes ou variantes à différents niveaux d'abstraction et sont liées par des relations et liées par des contraintes.

Un modèle de caractéristique est souvent représenté graphiquement sous la forme d'un arbre, également appelé diagramme de caractéristiques (ou FODA) [Kang et al. 1990]. Une fonctionnalité parent se rapporte à ses entités enfant par des relations. Les relations de base

sont éventuellement facultatives (les fonctionnalités enfants sont facultatives), obligatoires (les fonctionnalités enfants sont obligatoires) ou (une ou plusieurs fonctionnalités enfants peuvent être sélectionnées), alternative (une seule fonction enfant peut être sélectionnée) ou et (toutes les fonctions enfant doivent être sélectionnées).

Configuration

La modélisation d'entités permet la sélection d'entités flexibles à partir d'un modèle de fonctionnalité. Un utilisateur peut sélectionner une collection cohérente ou valide de fonctionnalités, appelée configuration, pour un produit attendu.

Artifacts

Dans la thèse, nous considérons un SPL qui est conçu dans les caractéristiques d'un modèle de fonctionnalité. Les actifs de ce SPL sont divisés en trois types d'artefacts: la spécification, le code et l'exactitude. Les artefacts de spécification sont associés aux fonctionnalités des entités. Ces fonctionnalités sont implémentées par les artefacts de code. Les preuves d'exactitude sont des artefacts utilisés pour prouver que les implémentations de ces fonctionnalités répondent à leurs spécifications. Afin de construire tout produit final de la SPL, nous avons besoin de mécanismes de composition définis pour tous ces artefacts. Sur la base d'une configuration sélectionnée par l'utilisateur, les artefacts, associés aux fonctionnalités impliquées dans cette configuration, sont composés afin de générer le produit souhaité.

Implementation Technologies

La technologie de mise en œuvre est une technologie utilisée pour générer les produits finaux des SPL. Le principe de base de la technologie est que, en fonction d'une configuration, les modules liés aux fonctionnalités impliquées dans la configuration sont composés (automatiquement ou non automatiquement) pour établir le produit final. Il existe des technologies d'implémentation populaires telles que FOP et Delta-oriented programming (DOP) [Schaefer et al. 2010].

Développement de logiciels corrects par construction

CbyC est un style de développement qui permet à l'application d'évoluer progressivement par le biais de petites étapes. La clé de l'approche est la garantie d'exactitude que l'application se comporte comme spécifié à chaque étape. La notion de raffinement a été introduite par [Dijkstra 1976], puis Back [Back 1978] a proposé une technique de raffinement par étapes pour construire des programmes corrects. Le calcul de raffinement est introduit pour raisonner sur des programmes, dans lesquels l'exactitude d'un programme est préservée pendant que le programme est affiné ou amélioré en utilisant la technique de raffinement par étapes [Morgan 1993; Back and Wright 1998].

FoCaLiZe

Le langage FoCaLiZe a une saveur orientée objet permettant l'héritage, la liaison tardive et la redéfinition [Prevosto and Doligez 2002]. Ces caractéristiques sont très utiles pour réutiliser les spécifications, les implémentations et les preuves. Nous avons travaillé avec l'environnement FoCaLiZe7 qui fournit un ensemble d'outils pour spécifier et implémenter des fonctions et des instructions logiques avec leurs preuves. Le développement de FoCaLiZe est facile à réaliser (en programmation fonctionnelle) car il permet au développeur d'écrire du code d'implémentation dont le style est fermé au langage fonctionnel, OCaml. Un programme source FoCaLiZe est analysé et traduit en sources OCaml pour l'exécution et sources Coq pour une vérification formelle.

Chapitre 2: Méthodologie pour générer des produits finis automatisés

Assurant l'exactitude et la fiabilité de tous les produits des SPL, les techniques traditionnelles, telles que la vérification des types, la vérification des modèles et la démonstration des théorèmes, doivent faire face à de nombreux défis. Dans la vérification de la gamme de produits, nous analysons si tous les produits de la gamme répondent à leurs spécifications. Récemment, certaines approches ont été proposées pour vérifier les LSP, mais elles sont souvent simplement utilisées comme preuve de concept pour les techniques de

vérification et ne sont pas justifiées empiriquement. Au meilleur de notre connaissance, ce chapitre présente la méthodologie comme la première discussion systématique sur la façon de générer des produits automatisés de lignes de produits en utilisant une approche CbyC.

In this chapter, we aim at presenting our the whole approach that is to facilitate the development of correct-by-construction software using product lines. The approach allows features together with their proofs to be reused. A mechanism to compose features with their proofs is proposed. The variability that is all the possible solutions of a problem, is expressed. We introduce the way we develop a SPL and generate its correct-by-construction final products.

Pour avoir une vue complète de la méthodologie, nous considérons à la fois le développeur du SPL et l'utilisateur de la SPL intéressé par un ou plusieurs produits.

Méthodologie de View Developer

Le développeur de logiciels cherche toujours à faciliter la réutilisation des artefacts. En appliquant le principe de la technique FOP, un développeur FFML travaille sur le modèle de fonctionnalité d'un SPL et développe les modules associés aux fonctionnalités.

- À partir du modèle de fonctionnalité, en utilisant le langage FFML, le développeur écrit pour chaque fonction un module FFML qui reflète la fonctionnalité. Le module (un fichier .fm) inclut les propriétés, en spécifiant les fonctionnalités associées à la fonctionnalité, et le code implémentant ces fonctionnalités.

- Les modules écrits sont envoyés au compilateur FFML pour vérifier leur syntaxe et leur sémantique, puis traduits dans les fichiers FoCaLiZe correspondants.

- Ces fichiers traduits sont envoyés au compilateur FoCaLiZe. Si le compilateur découvre des erreurs, elles sont signalées au développeur FFML.

- FoCaLiZe développeur remplit les indices de preuve dans les preuves (qui ont été supposés précédemment) de chaque module. Sur la base de ces indices de preuve, le Zenon Prover appelé par FoCaLiZe, trouve automatiquement les preuves.

- Les astuces de preuve sont copiées dans les preuves des modules FFML sous forme de commentaires du développeur FFML.

Le langage FFML est créé pour l'écriture de modules FFML. En fait, le langage prend en charge la technique FOP permettant de modulariser les artefacts en modules. La technique FOP ne contient que les principes de base, c'est-à-dire que chaque caractéristique d'un SPL est réfléchie par un module FFML. Ainsi, pour générer automatiquement les produits d'un SPL, nous devons définir les mécanismes de composition de ces modules. En outre, trois types d'artefacts (spécifications, code d'implémentation et preuves d'exactitude) contenus dans les modules rendent le processus de composition plus complexe.

Méthodologie de la vue utilisateur

L'utilisateur sélectionne certaines fonctionnalités qui sont censées être essentielles pour un produit attendu. La configuration sélectionnée, si elle est valide, est envoyée à l'outil FFML Product Generator. Le générateur récupère tous les modules impliqués dans la configuration à partir de la base de données d'actifs. Les modules concernés sont composés et intégrés dans un produit final décrit comme un ensemble de modules composites FFML, appelé produit FFML.

- L'utilisateur sélectionne certaines fonctionnalités qui sont censées être essentielles pour un produit attendu.
- La configuration sélectionnée par l'utilisateur, si elle est valide, est envoyée à l'outil FFML Product Generator. Le générateur récupère tous les modules impliqués dans la configuration à partir de la base de données d'actifs.
- Les modules concernés sont composés et intégrés dans un produit final décrit comme un ensemble de modules composites FFML, appelé produit FFML. Le générateur incorpore un ensemble de règles de composition qui sont définies pour composer toutes sortes d'artefacts (spécifications, preuves de code et de correction).

Le résultat de FFML Product Generator est le produit final contenant les artefacts composites obtenus à partir de tous les modules impliqués. Les spécifications composites

sont garanties par l'implémentation composite. Le processus de génération du produit est automatique, mais certaines preuves peuvent être effectuées manuellement (en raison de l'état actuel du FFML Product Generator).

Product Generation Process

Les produits FFML, construits à partir d'un modèle de fonctionnalité, sont les résultats attendus générés par l'outil FFML Product Generator. Pour une configuration valide, les modules FFML correspondant aux entités concernées sont collectés dans un diagramme de module. Le diagramme du module a la même hiérarchie que le modèle d'entités mais est limité aux entités impliquées dans la configuration. Les nœuds du diagramme sont les modules liés les uns aux autres via des relations. Basé sur le diagramme du module et la mise en œuvre de l'opération de composition, FFML Product Generator compose tous les artefacts des modules pour construire les produits FFML.

Chapitre 3: Langage FFML

Dans ce chapitre, nous expliquons pourquoi nous avons décidé de créer le langage FFML, la définition d'un module FFML, la grammaire et la sémantique de FFML. Compilateur FFML, un outil de traduction et l'exactitude du compilateur est également discuté.

Vers un langage formel pour des lignes de produits logiciels correctes par construction

FoCaLiZe est un langage efficace pour spécifier, implémenter et prouver des logiciels. Cela permet le développement de programmes corrects par construction. Dans nos premiers travaux, nous avons commencé avec l'hypothèse que le développement de SPL correct-par-construction peut être réalisé avec le support de ce langage mais nous avons rapidement réalisé qu'il n'est pas facile de les développer en FoCaLiZe.

Comme indiqué dans l '«Introduction», nous avons présenté les raisons pour lesquelles nous choisissons le FOP comme méthode de conception et de gestion de la variabilité des SPL. Le principe de base de la technique FOP est que chaque caractéristique du modèle de fonctionnalité d'un SPL est mappée à un module séparé qui implémente la fonctionnalité.

Ainsi, chaque module peut contenir son code d'implémentation incluant ses preuves de spécification et d'exactitude.

Inspiré par la technique FOP et FoCaLiZe, nous avons décidé de créer FFML (pour Formal Feature Module Language) qui est proche de FoCaLiZe mais apporte de nouveaux mécanismes et réduit les limitations pour atteindre nos objectifs principaux pour développer des SPL corrects par construction.

La méthode utilisée en FFML pour représenter la variabilité (réutilisation et modification de la spécification) des SPL en termes de syntaxe est inspirée de la programmation orientée delta (DOP) proposée par Schaefer et al. [Schaefer et al. 2010].

En tant que principes de base de la technique FOP, chaque caractéristique d'un modèle de fonctionnalité est implémentée par un module correspondant. Après avoir été écrits en FFML, ces modules sont intégrés dans des fichiers séparés.

Définition du module

Avec le désir de développer des SPL corrects par construction au moyen de la technique FOP, chaque nœud (caractéristique) du modèle de caractéristique d'un SPL est associé à un module. Chacun de ces modules nécessite la spécification des comportements attendus en tant que collection de propriétés. Un module contient également le code l'implémentant. Cette implémentation est prouvée plus tard pour répondre à sa spécification par une collection de preuves d'exactitude. Cela nous motive à définir un module FFML qui contient trois types d'artefacts: la spécification, le code et la preuve d'exactitude, comme les actifs d'une structure d'unité.

Module FFML

Comme présenté dans la section précédente, un module FFML contient trois types d'artefacts dans un seul paramètre. La définition du module permet la description de la communalité et de la variabilité par rapport aux artefacts du module parent. Un module est une modification de son module parent. Par conséquent, nous définissons un module FM comme une structure d'unité (comme une classe en Java ou une espèce en FoCaLiZe). Un module FFML FM est un tuple contenant les éléments: nom, nom du parent, décla-

ration de la fonction, propriété, type de représentation, définition de la fonction et preuve d'exactitude.

FFML Grammar

Notre objectif principal dans la définition de FFML est de proposer un langage proche de FoCaLiZe qui permet déjà au développeur d'écrire les modules de fonctionnalités à partir d'un modèle de fonctionnalité d'un SPL. Tous les artefacts sont placés dans un seul paramètre. FFML est inspiré de FoCaLiZe, en particulier en ce qui concerne les styles d'écriture des spécifications, les preuves de code et de correction. FFML et FoCaLiZe diffèrent principalement dans la manière de structurer et d'organiser l'information. Cependant, comme nous le verrons dans cette section, FFML permet au développeur de se concentrer sur l'expression de la réutilisation et la modification des artefacts de module. La syntaxe de FFML est définie et décrite dans cette section.

Syntaxe

La grammaire concrète de FFML est présentée dans cette section. Un module est introduit par un mot-clé f_module et son nom. Le mot-clé **from** exprime qu'un module ayant un identifiant de nom est étendu à partir de son module parent nommé parent, qui suit le principe de FOP.

Classification of Properties

Nous définissons une propriété suivant le principe de conception par contrat comme invariant, nouveau ou raffinant.

"From" - Réutilisation et mécanismes de modification

Le mot clé de est très important pour FFML en apportant des mécanismes de réutilisation et de modification. Un module courant peut être construit en modifiant à partir de son module parent en utilisant le mot-clé de. Ce mot-clé implique plusieurs mécanismes, tels que l'héritage, la modification et l'importation.

Sémantique

Dans cette section, nous présentons la sémantique de FFML étroitement liée aux significations du mot clé **from**. Nous discutons en détail comment les significations du module et de ses éléments sont calculées de manière séquentielle. Nous discutons également de la relation entre une propriété d'affinage et sa correction d'exactitude.

Compilateur FFML dans FoCaLiZe

Dans cette section, nous commençons par présenter nos notations pour représenter les règles de traduction de FFML. Nous présentons les grammaires abstraites de FFML et FoCaLiZe par types. En utilisant ces grammaires abstraites, nous expliquons nos fonctions de traduction implémentées dans le compilateur FFML. Une fonction de traduction de module traduisant un module FFML en FoCaLiZe est introduite, puis nous décrivons comment le compilateur FFML traduit les signatures, les propriétés, le type de représentation, les (re) définitions de fonctions et les preuves d'exactitude.

Résumé

FFML se concentre sur la construction d'un environnement convivial pour le développeur et n'augmente pas l'effort d'encodage. Ceci est considéré comme un avantage de FFML et explique pourquoi un module FFML est implémenté dans FFML plus facilement et plus rapidement que dans FoCaLiZe.

Chapitre 4: Génération de produits FFML

Cette section tente de répondre aux questions sur la façon de rendre le processus de génération automatisé et correct.

Les exigences de base du processus de génération automatisé sont discutées. Nous définissons une opération de composition binaire pour les modules FFML et analysons comment l'opération se comporte sur chaque type d'artefact impliqué dans les modules. Nous décrivons les règles de composition pour la mise en œuvre de cette opération en détail et expliquons le processus de génération des produits finaux. À l'aide de l'outil

Product Generator, la génération automatique des produits finaux corrects est illustrée sur le compte SPL du compte bancaire.

Exigences de base de la génération automatisée de produits

Poursuivant l'objectif principal d'un outil de génération automatique, à partir des méthodes existantes, nous donnons les exigences de base qui sont essentielles pour générer des variantes de produit correctes dans cette section.

Tout d'abord, nous avons seulement besoin d'une opération de composition binaire qui est appliquée pour une paire de modules, puisque la composition de plusieurs modules peut être effectuée par paire selon le diagramme du module. Deuxièmement, l'outil FFML Product Generator doit contenir des fonctions automatisées, mettant en œuvre des règles de composition pour composer deux modules. Troisièmement, il est nécessaire d'avoir une technique de mise en œuvre pour établir des produits

Module Composition Operation

Dans cette section, nous présentons notre opération de composition binaire définie pour deux modules. L'opération est appliquée pour le calcul de la composition de tous les types d'artefacts (spécification, code et exactitude) contenus dans ces deux modules. Les artefacts sont composés par l'opération qui est spécifique à ce type d'artefact.

Notation 1 (Binary Composition Operation) *Given two modules FM_2 and FM_1, the composition of module FM_2 with module FM_1, represented by $FM_2' = FM_2 \bullet FM_1$, is the binary composition operation that forms a composite module FM_2' from module FM_2 and refers to module FM_1 as the parent of the composite module via a* `from` *relation.*

Analyse de composition

Dans cette section, nous nous concentrons sur l'analyse de la façon dont l'opération de composition binaire, réalisée sur des types d'artefacts spécifiques. Nous analysons également comment l'opération de composition binaire est déployée sur toutes sortes d'artefacts: signature, propriété, type de représentation, définition de fonction et preuve d'exactitude.

omposition de la propriété

Notre méthodologie applique les principes de la conception par contrat pour spécifier la fonctionnalité de SPL. Autrement dit, pour chaque fonction, le développeur peut écrire un ensemble de propriétés spécifiant ses comportements. Nous supposons que l'opération de composition ne concerne que les propriétés liées à une fonction.

Règles de composition

Dans cette section, nous nous concentrons sur la définition de l'opération de composition binaire pour les modules en exprimant les fonctions implémentant les règles de composition pour tous les artefacts impliqués dans deux modules: signature, propriété, type de représentation, définition de fonction et preuve d'exactitude. Les règles de composition de deux modules sont définies par des fonctions qui sont utilisées pour les éléments séparés des modules.

Règles de composition des propriétés

Nous discutons les règles de composition pour deux ensembles de propriétés, qui sont spécifiés après l'approche de conception par contrat. Selon notre classification des propriétés, nous donnons les formules de composition correspondantes.

Preuve de correction

Une preuve d'exactitude est écrite pour prouver qu'une implémentation satisfait une certaine propriété. Par conséquent, avant de composer les preuves d'exactitude, nous devons considérer les propriétés liées à ces preuves. Nous commençons par considérer la façon dont ces propriétés sont composées, d'où la carte aux cas de composition des preuves.

Propriétés de la composition binaire

Dans cette section, nous décrivons les propriétés de base de notre opération de composition binaire, à savoir l'associativité et l'identité

Génération de produits finaux

L'établissement de la variante de produit à partir d'un modèle de fonction et d'une configuration nécessite la composition de tous les modules associés du diagramme de module associé à la configuration. Cependant, pour composer tous les modules connexes, nous avons maintenant besoin de règles de construction basées sur le diagramme des fonctionnalités (telles que l'ordre des modules et la structure de la structure). Nous appelons ce processus la composition basée sur un diagramme de module. Dans cette section, nous expliquons comment les produits finaux sont générés en fonction de l'opération binaire et du processus de composition basé sur un diagramme de module.

Composition basée sur un diagramme de module. La composition basée sur un diagramme de module (MDC pour abréger) est un processus récursif qui utilise la composition binaire précédemment définie pour composer des modules et des modules composites.

Application sur le compte bancaire SPL

L'outil FFML Product Generator est écrit en OCaml avec environ un millier de lignes de code. L'outil permet à un utilisateur de sélectionner une configuration en entrée et renvoie le produit final correspondant. Grâce à notre outil générateur, les douze produits du compte bancaire SPL ont été générés automatiquement. Le compte bancaire SPL a été analysé avec cinq caractéristiques et développé dans les cinq modules correspondants d'environ 400 LOC en FFML.

Chapitre 5: Évaluation

In this chapter we deal with a bigger and more complex example, Poker product line, that is developed using our methodology.

Étude de cas: Gamme de logiciels de poker

Pour évaluer notre méthodologie, nous développons un exemple qui est plus complexe que la gamme de produits Bank Account avec plus de fonctionnalités et de produits finaux. Nous choisissons Poker SPL. Cependant, au lieu des spécifications, leurs définitions de

fonctions sont simples et presque vides. En raison de l'absence d'informations nécessaires, dans cette section, nous décidons de créer notre propre SPL Poker en collectant les variantes du jeu de poker, en résumant les règles qui s'y rapportent, puis en concevant un modèle de fonctionnalités.

Exemple: Ligne de produits de poker

Dans cette section, nous avons collecté les variantes du jeu de poker, résumé leurs règles associées et ensuite conçu un modèle de fonctionnalité.

Analyse et développement de ligne de produits de poker

Sur la base des règles de poker collectées et du modèle de fonctionnalité conçu dans la section précédente, nous analysons et développons les fonctionnalités de Poker SPL en modules FFML dans cette section.

BasicPoker. Nous spécifions les règles simples applicables à tous les types de jeu de poker dans la fonction BasicPoker. Le jeu de poker de base est joué avec un paquet standard de 52 cartes.

BasicMPoker. La fonctionnalité BasicMPoker est construite en tant qu'enfant de BasicPoker. Les joueurs qui rejoignent le jeu peuvent utiliser de l'argent (jetons) pour jouer. Un tour d'enchères a lieu à chaque fois avant ou après une transaction dans laquelle les joueurs ont l'opportunité de parier sur leurs mains.

Basic36Poker. Basic36Poker est construit comme un enfant de BasicPoker, il est joué avec un jeu de cartes de 36 cartes dans lequel les cartes de 2 à 5 sont supprimées.

BasicWPoker. BasicWPoker est construit en tant qu'enfant de BasicPoker. Après le quatrième tour de cartes, une carte est mise sur la table et cette carte est vue par tous les joueurs. Les trois autres cartes ayant le même rang que la carte sont sauvages. Le dernier tour final de cartes est distribué pour donner la cinquième carte aux joueurs.

MFormPoker. MFormPoker est construit en tant qu'enfant de BasicMPoker. Dans le jeu, après avoir distribué des cartes à chaque joueur, certaines cartes (1, 2 ou 3) continuent d'être retirées du paquet et tournées vers le haut sur la table. Ces cartes sont appelées

"cartes communautaires".

DrawMPoker. DrawPoker est un enfant de MFormPoker dans lequel la liste de cartes de communauté est vide. Les joueurs peuvent rejoindre un tour d'enchères avant le premier deal. Le premier pari est déterminé par un nombre, appelé "ante" et décidé par tous les joueurs.

Évaluation

Dans cette section, nous nous concentrons sur l'analyse des résultats obtenus en développant Poker SPL. La validité et la limitation de notre méthodologie sont également discutées.

Le Poker SPL a été analysé avec douze caractéristiques, dont sept ont été développées dans les sept modules correspondants: BasicPoker (BP), BasicMPoker (BMP), Basic36Poker (B36P), BasicWPoker (BWP), MFormPoker (MFP), DrawMPoker (DMP), TexasHold'emMPoker (THP) d'environ 800 LOC en FFML. Ces sept modules ont été traduits en FoCaLiZe par FFML Compiler.

Validité de la méthodologie

D'après les résultats obtenus à partir des tables statistiques du compte bancaire SPL et du Poker SPL, nous pouvons voir que ces deux SPL ont été développés avec succès. Les produits générés sont corrects par construction. Les artefacts, c'est-à-dire les propriétés et les preuves, sont composés automatiquement par nos outils. Bien que certaines preuves manquent de preuves, la plupart des preuves sont faites automatiquement.

Le développement des modules de SPL utilisant FFML est plus facile que dans FoCaLiZe. Le développeur écrira moins de LOC. En réduisant la complexité et en économisant les efforts de la génération automatique des produits corrects tout en développant les deux SPL, notre méthodologie s'est avérée efficace et fiable.

Conclusion

Dans ce chapitre, nous résumons brièvement notre travail sur la thèse et donnons les contributions. Ensuite, nous discutons de futurs travaux potentiels sur le développement

de lignes de produits correctes par construction.

Contributions

- La contribution principale de cette thèse se réfère à une méthodologie efficace pour développer des SPL et générer automatiquement des produits finis corrects en utilisant une approche proche de l'approche CbyC.

- La méthodologie a été mise en œuvre en utilisant les techniques FoCaLiZe et FOP. Les résultats obtenus en utilisant nos outils pour générer les produits finaux corrects du compte bancaire SPL et du Poker SPL démontrent l'applicabilité de notre méthodologie.

- Les mécanismes de modification de tous les types d'artefacts (spécification, code et exactitude) ont été définis. En particulier, une propriété peut être affinée à partir d'une propriété existante, réutilisant ainsi la preuve correspondante. La preuve était en réalité plus facile à réaliser. Ces mécanismes aident à réutiliser les artefacts et à réduire les efforts pour l'écriture d'artefacts.

- Une opération de composition pour toutes sortes d'artefacts au niveau du module a été définie et implémentée dans notre outil. Cet outil peut générer automatiquement des produits sans intervention de l'utilisateur. L'effort de vérification est considérablement réduit grâce à la réutilisation des épreuves.

- Une chaîne d'outils avec FFML Compiler et FFML Product Generator a été développée. Il prend en charge à la fois le développeur et l'utilisateur lors du développement de SPL et de la génération de produits corrects par construction.

Travail futur

- Terminez le développement du Poker SPL.

- Analyser les propriétés dont les preuves sont effectuées manuellement, donc définir de nouvelles règles de composition d'épreuve afin de supporter ces propriétés.

- Impliquez FFML afin de prendre en compte de nouvelles façons de réutiliser, telles que l'introduction de nouveaux paramètres dans une propriété d'affinage.
- Développer une interface utilisateur graphique (GUI) pour notre chaîne d'outils et y intégrer un outil permettant de vérifier la validité des configurations.
- Adaptez notre méthodologie à d'autres langages, tels que B ou Java au lieu de FoCaLiZe.

Contents

List of Tables

List of Figures

Introduction

Nowadays together with diversity of customers' requirements, many companies are confronted with a rising demand for individualized products. For instance, the diversity of devices forces developers to build many variants of a software. Then, a solution for developing similar products to respond to the numerous customers' needs becomes necessary. The problematic is not new: in 1976 Parnas [Parnas 1976] named this kind of products "*a program family*" which is defined as "a set of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members". Later, a program family is called *Software Product Line* (SPL) which is introduced by Bass et al. [Bass et al. 1998] as "a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way". The same design of software is placed into core assets, then shared and reused across multiple products [Bass et al. 1998; Clements and Northrop 2001]. Numerous research publications indicate the achievements and benefits gained by applied Software Product Line Engineering (SPLE) [Pohl et al. 2005; Apel et al. 2013a; Thüm et al. 2014a]. Because of the benefits gained from SPLE, such as achieving large-scale software, reducing costs, improving time-to-market, and bringing higher quality, this technique is considered as a methodology for developing the diversity of software. Consequently, SPLE has been used widely in many domains and organizations in the IT industry [Linden et al. 2007; Benduhn et al. 2015].

Context of Thesis

In SPLE, an asset is any artifact employed in the development of software. The assets must be allocated and managed to create different products. Thus, the *commonality* and *variability* across software must be identified and products of a SPL - called product line members or *product variants* - are characterized by their common functionalities and also by their differences [Pohl et al. 2005]. Pohl et al. also introduced a *software product line framework* that is considered as a base of many technologies for developing Software Product Lines (SPLs). The principle, focusing on what might be reused by other products, should be performed first, and then on what is able to be added in order to satisfy various requirements. Recently, many approaches for analyzing commonalities and modeling variability have been proposed focusing on this principle. Many kinds of models have been proposed for analyzing SPLs, such as *feature model* [Kang et al. 1990], orthogonal variability models [Pohl et al. 2005], or decision model [Schmid et al. 2011], etc. However, the approach called *feature modeling*, which uses feature models for representing graphical relations between features is the most popular. These models provide flexibility in adding new functionality to create various products of SPLs, adapting to individual customers.

Product derivation is the process of building product variants from a SPL. This process defines how assets are selected according to a given feature configuration, and it specifies how those assets are assembled in order to build the desired product. Besides resolving existing problems in software engineering when developing a single application, SPLE also has to face some practical specific problems, such as, ensuring the automation of product derivation or the establishment of independent platforms, etc. The approach of *generative programming* is a solution which was first proposed and realized by Czarnecki [Czarnecki and Eisenecker 1999] with the purpose of generating automatically the products via a *generator*. This generating process is called *product generation* and the product produced from this process is called a *final product*. The approach was enriched by a step-wise refinement method [Batory et al. 2003] that allows adding features for developing more complex software. Batory [Batory 2005b] also proposed the ability for developing a collection of generator tools, each of which is specific for a kind of artifact, since it adapts itself to the diversity of software. Recently, many approaches focus on *Feature-Oriented Program-*

ming (FOP), an implementation technique, that has proven to be effective in generating products [Prehofer 1997; Apel et al. 2013b; Thüm et al. 2014b]. Another implementation technique, Delta-oriented programming [Schaefer et al. 2010], also supports product generation process but more complex and flexible than FOP.

It is the fact that the *security and safety-critical systems* use increasingly SPLE, such as medical, aircraft flight control, automotive systems, etc. [Braga et al. 2012]. The request for guaranteeing the correctness of product variants derived from generators becomes very important in the context of critical systems. This requires the use of specification and verification methods and techniques satisfying SPLE principles. Namely, when developing a SPL, numerous products are produced automatically by generators, these methods and techniques have to adapt to not only guarantee the correctness but also to save time and effort. They should take advantage of commonalities and differences of product variants, and thus try to reuse specifications and verification results.

Early, *modular verification methodologies* have been proposed, first for type checking [Thompson 1991; Aversano et al. 2002], for model checking [Fisler and Krishnamurthi 2001] and then for theorem proving [Poppleton 2007; Thüm et al. 2011]. Recent research has shown that the strategies in analyzing and verifying SPLs, such as *family-based*, *product-based* and *feature-based*, also bring efficiency and advantages [Thüm et al. 2014a; Apel et al. 2013d]. The techniques mentioned above can help to automatically generate correct final products. Some writers also offered solutions for developing SPLs efficiently and mostly automatically, such as in [Delaware et al. 2011, 2013; Thüm 2015]. However, the automated generation requires more efforts, more efficient methods and strategies for verifying and generating all final products, especially in ensuring correctness.

The *Correctness-by-construction* (CbyC) is one of the effective approaches mainly for developing automated security- and safety-critical systems by building demonstrable units [Hall and Chapman 2002; Kourie and Watson 2012; Watson et al. 2015]. Using a refinement paradigm, these units are built in each refining step, and thus systems are managed and verified easier. Our main concern of this thesis is developing SPLs and applying this approach with an objective to generate automatically correct-by-construction products. Because of the benefits of this approach [Batory 2015], we are confident that

our research direction is promising, especially after detecting many existing techniques and methods which can support to elaborate this direction.

Objectives of Thesis

Our main goal is to develop correct-by-construction SPLs. Surveying research literature about applying the CbyC approach on SPLs, we overcome shortcomings of previous research and identify the major proposals for our direction. First, we focus on published works on analysis and implementation techniques for SPLs which allow creating modules of specifications, implementation code and correctness proofs together into the same units in order to reuse them together and more effectively. Second, to receive the correct final products besides pursuing an existing implementation technique we need a methodology in which the units are composed to create the final products whose correctness is remained. Third, in order to achieve the methodology, we realized that developing correct-by-construction SPLs using existing languages faces many problems and limitations. Hence, we created a new language that supports developers to write the modules easier and more effectively. Fourth, to adapt the requests of the methodology, we defined an independent composition mechanisms to be applied to these modules. Based on our knowledge gained from the existing techniques, we build an automated generation tool to elaborate these mechanisms. Finally, to evaluate our approach, we applied it to a practical case: a Poker SPL.

Contribution of Thesis

The objective of this dissertation is to propose a methodology for developing SPLs. The methodology follows the software life cycle of a SPL and generates the correct final products from feature selections. With the proposed methodology, we have accomplished the following contributions.

- We have developed a language, called **Formal Feature Module Language (FFML)**, that allows expressing the variability of artifacts. This new language consists of syntax and semantics, and supports to develop SPLs following the CbyC approach more easily. A SPL is analyzed into the modules which are written in FFML. A tool, called **FFML Compiler tool**, has been built for compiling these modules.

- The composition mechanisms have been defined for composing FFML modules of SPLs. We have developed a generator tool, called **FFML Product Generator**, that implements these composition mechanisms to generate automatically the correct final products.

According to the first proposal described previously, we begin by finding and selecting a relevant implementation technique for expressing the variability of all artifacts types (specification, code implementation and correctness proof) of SPLs. Although a SPL may be defined at different levels of abstraction, we adopted *Feature-Oriented Programming* (FOP) technique [Prehofer, 1997]. There are many reasons why we choose this technique. First, it is a technique whose principles allow any SPL to be analyzed at feature level and modularized all artifacts employed in each feature into a *feature module* that implements the feature. Second, these principles are a core which may be transferred into different systems, since if our methodology developed on it can be independently applied to different languages. Third, based on the technique we can focus on the definition of a feature module that contains three kinds of artifacts and also leverages the variability across a family of products, since a maximum of the artifacts can be reused. Consequently, it is possible to conclude that FOP technique is a rational choice for modeling the commonality and variability of SPLs.

As indicated above, the benefits gained from FOP lead us to choose it as a core implementation technique deployed into our methodology. The methodology, proposed in the thesis, have overcome the existing limitations, such as, many efforts and much time for ensuring the correctness and the automated generation of final products. The methodology saves verification time and is friendly to developers.

Initially, FoCaLiZe is a formal language that we choose to implement the FOP technique. The language allows writing correct-by-construction software efficiently [Prevosto and Doligez, 2002]. FoCaLiZe also contains a tool chain supporting the developer to develop and execute the software. FoCaLiZe proofs are sent to the Zenon prover (embedded in FoCaLiZe) which produces proofs that can be verified by Coq for more confidence [Bonichon et al., 2007]. Zenon automatically proves that software meets its specifications using

the proof hints given in the correctness proofs. We rely on FoCaLiZe in developing correct-by-construction software and on Zenon in automated proving its correctness. However, FoCaLiZe does not have a SPL flavor while our main purpose aims at developing SPLs. From the limitation and the advantage of the language, we decide to develop the **FFML** language based on FoCaLiZe. This new language and its compiler tool (**FFML Compiler**) support for developing SPLs following the CbyC approach but more easily.

In the initial stage of our methodology, a SPL is analyzed into the features using FOP technique. The corresponding feature modules are written into FFML by the developer. Each FFML module contains both the specification and the implementation code. These modules are compiled by the **FFML Compiler** tool into FoCaLiZe files. The specification is verified using Zenon, a theorem prover for FoCaLiZe. We consider a collection of all the feature modules as a resource for developing the products of the SPL. In the second step of the methodology, the user selects the features for his product. The configuration is determined based on these selected features. Using a filter operation, it is possible to check the configuration validity, and if the configuration is valid, a generator receives it as an input. In the third stage, based on this valid configuration, the generator will collect the relevant modules from the resource. These collected modules are composed automatically through the composition rules that are defined and embedded into the generator (**FFML Product Generator**). The result of the composing process is a FFML product. In the final stage, using the **FFML Compiler** tool, the product is translated into FoCaLiZe program which is compiled using FoCaLiZe Compiler and verified using Zenon.

To ensure the realization of this methodology for automatically generating correct products, we investigate existing techniques and tools supporting automated composition of specifications and correctness proofs (such as FEATUREHOUSE [Apel et al. 2009], FEATUREIDE [Thüm et al. 2014b], etc.). We recognize that there is no automated tool for composing correctness proofs. Hence, in the thesis we target a more powerful tool than the existing ones that allows expressing the variability of three kinds of artifacts and also composing all of them automatically. Consequently, we have defined the composition rules for FFML modules and developed the **FFML Product Generator** tool. The final products of a SPL can be generated automatically using this tool.

Road Map of Thesis

As our main goal is to develop correct-by-construction SPLs, we demonstrate the contributions of this dissertation by the following chapters:

- We begin by survey literature on SPLE and CbyC approaches in the State of the Art. Based on the overview and the insights obtained, we analyze the existing problems and suggest ways to solve them for our main goal before entering other main chapters.
- In Chapter 2, we explain our methodology for developing SPLs and generating automatically correct final products. Different views offer a global perspective on the methodology. We also discuss languages, mechanisms and tools required for developing correct-by-construction SPLs.
- In Chapter 3, we propose the FFML language for writing SPLs. We start justifying the definition of the language. Based on three kinds of classified artifacts (specification, implementation code and correctness proof), we define the syntax and semantics of the language. We also discuss how FFML supports variability of these artifacts along with the FOP technique selected. We present the description of our tool, **FFML Compiler**, with the translation rules from FFML into FoCaLiZe. The correctness of the FFML translation is discussed, and we conclude that FFML is a language to develop SPLs efficiently following CbyC approach.
- In Chapter 4, we focus FFML product generation. The requirements for automatic generation of the correct products from feature selections, are discussed. We introduce a binary composition operation for composing FFML modules, and then analyze the composition cases related to this operation. The operation is defined by means of composition rules that are applied for each kind of artifacts. The relations of specification, code and correctness proofs are clarified, and thus together with the compositions of specification and implementation code, correctness proofs are also composed. The process of generating final products and a quick view of the **FFML Product Generator** tool are described in the end of the chapter.
- In Chapter 5, to evaluate our methodology, besides the Bank Account SPL deployed

in the thesis as a running example, we develop one more complex SPL, a Poker SPL. The final products generated from our tools (FFML Compiler and FFML Product Generator tool) are verified, tested and validated. Consequently, we give our evaluation of the methodology.

After describing the main contributions mentioned above, we conclude our thesis and discuss future work in the last chapter.

Chapter 1

State of the Art

In this chapter we aim at introducing the basic knowledge which the thesis is based on. We also give a critical review of the context of the research question stated in the chapter Introduction and related issues. The chapter is organized as follows. The main concepts of SPLs are introduced in Section 1.1. We discuss the feature modeling principles, the implementation technologies, the product line analysis strategies and verification. The initial requirements while developing correct-by-construction software is discussed in Section 1.2. The main principles of the correctness by construction approach and the FoCaLiZe language which follows this approach are presented in this section. The research works relating to your main thesis goal are also analyzed. We finish the chapter with a summary of the approaches presented in Section 1.3.

1.1 Software Product Lines

In the beginning, *program family* was early defined in [Douglas 1968] as a set of similar programs which are developed together. The idea of this approach is that making the families of software components reduces encoding efforts. *Stepwise refinement* method has first introduced in [Dahl et al. 1972] and formalized for producing program families in Back [1978]. Similarly to this method, Parnas proposed another one, called sequential development [Parnas 1976], in which the variability is defined as the common properties of programs should be carried out before analyzing others of individual members. These authors also suggested to use generator(s) for program generation instead of implementing

variability at runtime.

The term *software product line* was first mentioned by Bass et al. in [Bass et al. 1998] while proposing an architecture for it. During the last decade, this term has been used to replace the term program family. The engineering of SPL was enriched by Clements and Northrop where the core assets are determined and managed [Clements and Northrop 2001]. Besides proposing the mechanisms for sharing a common set of core assets, the authors described how to apply these mechanisms in practice. The variability was mentioned as a key for these mechanisms. Pohl, Böckle and Linden defined the complete development life cycle of a SPL [Pohl et al. 2005], in which there are two main processes: domain engineering and application engineering. The intention of the domain engineering process is for establishing a reusable platform for a product line that contains all types of artifacts. The related artifacts are linked to each other. The commonality and the variability are defined in the platform. By contrast, application engineering is the process where the final products are derived from the established platform in the domain engineering process and from customers' requests. Separating the product line development into these two processes ensures to express variability and brings the flexibility in choosing the artifacts for the different products as required.

In the next sections, we focus on describing in detail the techniques in developing SPLs. In Section, 1.1.1, we explain feature modeling and how it supports to manage the variability. Bank Account product line is analyzed as a running example. We also present the definition of configuration in feature modeling. In Section 1.1.3, we describe implementation technologies that support the generation of final products. The strategies for analyzing SPLs are discussed in Section 1.1.4, and thus indicate the verification approaches in which these strategies appear. Finally, we summarize and relate the techniques to our work.

1.1.1 Feature Modeling

Variability management is a main task related to success of SPLs. In SPLE, a *feature* is a characteristic behavior specified as a unit of requirements, technical functions or non functional characteristics [Kang et al. 1990], that is associated to its assets in a SPL. Variability can be understood as the allocation of the features that makes one product

different from others in the same product family [Pohl et al. 2005]. In *feature modeling*, a product of a SPL differs from others by their involved features. A *feature model* is defined as a representation of "the information of all possible products of a software product line in terms of features and relationships among them" in [Benavides et al. 2010]. A feature model of a SPL is a hierarchical set of the features of the SPL. These features are either common or variant at different levels of abstraction and they are related by relationships and bound by constraints.

A feature model is often graphically depicted as a tree, also called a *feature diagram* (or feature oriented domain analysis (FODA) [Kang et al. 1990]). A parent feature relates to its child features by relationships. The basic relationships are possibly *optional* (child features are optional), *mandatory* (child features are required), *or* (one or more child features can be selected), *alternative* (only one child feature can be selected) or *and* (all child features must be selected). Notice that the root feature always appears in all products. And if a child is selected for a product, its parent must be part of the product.

Besides these relationships, the constraints among features can be represented by inclusion and exclusion statements, such as, *requires*, *excludes* and other more complex ones [Batory 2005a]. A *requires* constraint means that if a feature B is required by a feature A, the presence of A in a product implies the presence of B. An *excludes* constraint means that if a feature A excludes a feature B, it is impossible to have both A and B in the same product. We show the example of a feature model in Figure 1.1.

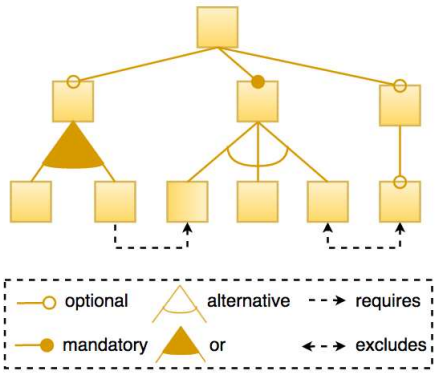

Figure 1.1: A sample feature model

Cardinality-based feature model [Czarnecki et al. 2005] is extended from feature model with two new relationships, namely feature cardinality and group cardinality. A feature cardinality is a sequence of intervals denoted $[n..m]$, in which n is the lower bound and m is the upper bound. The number of instances of the feature that can be involved in a product is determined by these intervals. A group cardinality is an interval denoted $\langle n :: m \rangle$, where

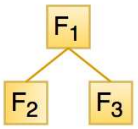

Figure 1.2: Tree A

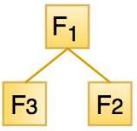

Figure 1.3: Tree B

n is the lower bound and m is the upper bound. When a feature is selected for a product, this interval limits the number of its child features that can be part of the product. In order to include more information about features, the extended feature model [Benavides et al. 2005] is also defined to extend feature model with more complex constraints among attributes and features.

Besides feature model [Kang et al. 1990], there are many other kinds of models that are also used for analyzing SPLs with more sophisticated constraints, such as, the *orthogonal variability model* [Pohl et al. 2005], the *decision models* [Schmid et al. 2011], etc. However, it appears that using feature models in analyzing SPLs brings enough flexibility while adding new functionalities or selecting combinations of features. Hence, the basic feature model is considered as a standard description used to manage the variability of SPLs [Benavides et al. 2010]. In this thesis, we will use this model. But for us, the nodes are ordered and the tree A (Figure 1.2) and B (Figure 1.3) are not the same.

1.1.1.1 Bank Account Software Product Line

As a running example, we consider the bank account product line, described in [Thüm et al. 2012a], that is analyzed using feature modeling. Its feature diagram is shown in Figure 1.4. It illustrates a family of products allowing the management of bank accounts. The root feature *BankAccount* (BA for short) provides the basic management of an account. It allows the bank to store the current balance and the amount of money added into or withdrawn from an account. A customer can withdraw more money from the account than available if it is within an over limit. The feature BA has three optional child features *DailyLimit* (DL for short), *LowLimit* (LL for short) and *Currency* (CU for short). The feature CU has one optional child feature *CurrencyExchange* (CE for short).

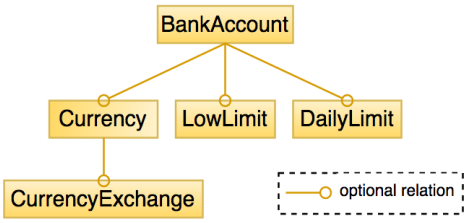

Figure 1.4: Feature diagram of bank account product line

The feature DL allows the bank to limit the amount of money withdrawn in a day. The feature LL indicates that the bank authorizes a customer to withdraw money from the account only if the amount is greater than a low limit. The feature CU accommodates the management of currency. Finally, an optional feature CE is established as a child of CU to enable the calculation of currency exchange. A product is collected by a selection of these features. For example, the user wants to build a bank account management system when a limit is put on withdrawals in a day and on each withdrawal. In this case, features DL, LL and BA will be collected to establish this bank account product.

1.1.1.2 Configuration

Feature modeling allows flexible features selection from a feature model. A user can select a coherent or valid collection of features, called *configuration*, for an expected product. Benavides et al. [Benavides et al. 2010] introduce the definitions associated to the configuration as follows:

- *Configuration.* Given a feature model with a set of features FE, a configuration is a 2-tuple of the form (SE, RE) such that both $SE \subseteq FE$ and $RE \subseteq FE$, in which SE is the set of features to be selected, RE is the set of features to be removed and $SE \cap RE = \emptyset$.
- *Full configuration.* If $SE \cup RE = FE$ the configuration is called full configuration.
- *Partial configuration.* If $SE \cup RE \subset FE$ the configuration is called partial configuration.
- *Valid configuration.* A valid configuration is a configuration when the constraints expressed in the feature model are satisfied.

A full configuration contains all features necessary for deriving a final product. A

partial configuration (PC) and a full (FC) configuration possibly formed from the feature model such as described in our bank account example in Figure 1.4 are represented as follows:

$$PC = (\{LowLimit, DailyLimit\}, \{Currency\})$$

$$FC = (\{BankAccount, LowLimit, DailyLimit\}, \{Currency, CurrencyExchange\})$$

There is a fact that not all configurations are valid. For example, as defined PC is not valid configuration. The validity of configurations is necessary and it is clear that from only the valid ones will be used to combine their features to establish the corresponding products. However, a partial configuration can also be interesting if there is no contradiction between its included features.

Over the last decade, the analysis of feature models has been achieved automatically: giving features selection as input the user can receive some results, such as, the validity of configuration, the number of products, finding full configurations, etc. The automation of feature model analysis is due to analysis operations [Benavides et al. 2010]. An operation takes a configuration as input and returns a result as output. For example, the validity of a configuration can be checked by a *valid configuration* operation. Another *filter* operation takes a (partial) configuration and a feature model and returns all of the possible full configurations that can be derived from the model containing the selected features from the configuration. These operations help the user in the process of selecting relevant features and configurations. There are many tools supporting the operations, such as a prototype extending **fmp** (an Eclipse plug-in) [Czarnecki et al. 2005], *pure::variants*[1], **guidsl** [Batory 2005a], etc.

To understand deeply the mentioned operations, we look into the specification methods for feature models. By modeling the feature models into formulas (according to the logic paradigm or other methods), we can use the operations embedded into automated tools. In fact, there are many ways to specify a feature model. The first paradigm which is used popularly, is the *propositional logic*. The developer expresses his feature model using propositional formulas. Initially, Batory [Batory and O'Malley 1992] used grammars

[1] https://www.pure-systems.com

to specify feature models and then the idea of making the connection between feature models with propositional formulas is proposed in [Mannion 2002]. There are two primary propositional logic based tools which are widely used: SAT solver [Berre and Parrain 2010] used in [Batory 2005a; Benavides et al. 2007; Mendonça et al. 2009] and binary decision diagram (BDD) solver [Whaley 2016] used in [Benavides et al. 2007; Mendonça et al. 2008]. Other ways to specify feature models use *constraint programming*. A feature model is mapped to a constraint satisfaction problem (CSP) [Tsang 1993]. Using a off-the-shelf solver, the CSP can be analyzed automatically. The usage of constraint programming is first proposed in [Benavides et al. 2005] and widely used in [Djebbi et al. 2007; White et al. 2008]. The popular tools based on constraint programming are CHOCO [2], GNU prolog [3], etc. Based on the specifications of feature models, the operations are realized using analysis tools in order to calculate the desired results. For example, to check the validity of a configuration, a valid configuration operation will calculate the satisfiability of the formula (in propositional logic or constraint programming, etc.) related to the configuration.

In this thesis, we will rely on the possibility of using any of these tools to verify the validity of a configuration and will not propose a new solution. When talking about valid configurations, we will assume that the user will uses such tools to verify it.

1.1.2 Artifacts

In *Software Product Line Engineering*, the same design of software is placed into core assets, then shared and reused across multiple products [Bass et al. 1998; Clements and Northrop 2001; Pohl et al. 2005]. An asset is any artifact employed in the development of the software. All the assets must be allocated and managed to create different products. They are associated to the features of the feature model that contains the functionalities of the software.

In the thesis, we consider a SPL that is designed into the features of a feature model. The assets of this SPL are divided into three kinds of artifacts: specification, code and

[2] http://choco-solver.org/
[3] http://www.gprolog.org/

correctness proof. The specification artifacts are associated to the functionalities of the features. These features are implemented by the code artifacts. The correctness proofs are artifacts used to prove that the implementations of these features meet their specifications. In order to build any final product of the SPL, we need composition mechanisms defined for all of these artifacts. Based on a configurations selected by user, the artifacts, associated to the features involved in this configuration, are composed in order to generate the desired product.

1.1.3 Implementation Technologies

Implementation technology is a technology that is used to generate the final products of SPLs. The basic principle of the technology is that given a configuration, the modules related to the features involved in the configuration are composed (automatically or not automatically) to establish the final product. In this section, we explore several well-known implementation techniques for SPLs that have been evaluated to bring benefits in terms of modularization, asset re-usability and asset composition.

The early implementation technique uses *mixins* in which a mixin is a "subclass definition that may be applied to different superclasses to create a related family of modified classes" [Bracha and Cook, 1990]. In particular, a mixin can add fields and methods to an existing class and override existing methods [Flatt et al., 1998]. The ideas of modeling the variability of code artifact and encapsulating all variants into a meta-product are initiated in [Post and Sinz, 2008]. Batory et al. proposed AHEAD tool suite that provides a concept of several tools, each of which relates to a specific artifact type [Batory et al., 2003]. A keyword `super` is used to express the relation between a class to its superclass.

Feature-Oriented Programming technique was proposed first in [Prehofer, 1997] which is an extension of object-oriented programming. Its principles for (composing objects) generating automatically products are proposed in [Prehofer, 2001] based on a feature module and method refinement. A keyword `original` is used to override the original method and refers to the original method body. Apel et al. [Apel et al., 2009] improved AHEAD and proposed a tool chain, called FEATUREHOUSE, for generating automated software composition. The composition operation for feature modules of a given configuration is

presented in [Apel et al. 2010] and implemented in [Apel et al. 2013b] with superimposition. Apel et al. continued proposing a feature-based specification and verification which support writing feature specification in separate and composable units [Apel et al. 2013c]. A tool chain, called SPLVERIFIER, implemented the method for specification and correctness proof [Apel et al. 2013d]. However, the method was only realized for a model checking approach. In addition, this tool chain are is only available for modules written in Java and C.

Adhere the theorem proving approach and FOP technique, Thüm et al. [Thüm et al. 2014b] have developed the tool FEATUREIDE, improved from FEATUREHOUSE [Apel et al. 2009], that implements the mechanisms for automated generation of products including both their specifications and code artifacts. But proof artifacts are not included. The tool is now only available for modules written in Java.

Besides FOP, Delta-oriented programming (DOP) is a different approach for designing and implementing SPLs [Schaefer et al. 2010] which is based on the concept of program deltas. Features are implemented into corresponding delta modules that are supported to add, modify, remove artifacts [Hähnle et al. 2013]. The composition of artifacts can be implemented by means of uninterpreted assertions [Hähnle and Schaefer 2012] which are generalized with keyword `original`. Let us note that this approach is really interesting but only focuses on single composition mechanism for specification while what we need is mechanisms for not only specification, but also for other artifacts, e.g. correctness proofs.

1.1.4 Product-line Analysis Strategies and Verification

In this section, we focus on describing techniques for analyzing SPLs, and thus indicate their appearance in the verification approaches.

Product-line analysis is necessary to be able to respond to a large number of products. The key idea of product-line analysis is to exploit analysis strategies before developing a product line, hence reduces analysis effort and improve the quality. Recently, research about product-line analysis is interested and classified based on different strategies [Kolesnikov et al. 2013; Apel et al. 2013d; Thüm et al. 2014a]. These strategies are classified into three basic ones based on what the analysis is applied to, i.e, features,

products or the whole product line.

An analysis strategy of features is called *feature-based analysis*. That is, a feature is implemented with all artifacts that are analyzed in isolation without considering other features or the feature model. Although the strategy can reduce the analysis effort, other characteristics, such as feature interactions and the validity of feature combinations are not considered.

Another analysis strategy is *product-based analysis* that generates and analyzes all the products of a product line individually using a kind of model (e.g, feature model). The basic idea of this strategy is that the developer sample a smaller number of products, usually based on some coverage criteria, such that still reasonable statements on the correctness or other properties of the entire product line are possible to analyze.

The products of a product line have similar assets shared between these products [Czarnecki 2002; Apel et al. 2013a]. Checking products individually leads to redundant computations. Another analysis strategy, bringing more efficiency, is called *family-based analysis*. The analysis operates on domain artifacts as feature modules instead of products. The valid combinations of features, specified by a feature model, are also considered.

In the publication [Thüm et al. 2014a], Thüm studied conceptually the strengths and weaknesses of each three mentioned product-line analysis strategies. The main difference indicated between them is the ability to avoid extent redundant computations. A SPL analyzed with the product-based strategy has redundant computations because of the similarities between its products. To avoid redundancies, we can analyze a SPL with the feature-based strategy. However, this strategy allows considering domain artifacts in isolation, it does not support noncompositional properties and we can only analyze compositional properties. The family-based strategy supports both compositional and noncompositional properties, hence avoids redundancies. The author has also proposed the combinations of these three strategies to minimize redundant efforts.

The three analysis strategies (feature-based, product-based and family-based) are found when surveying on the the publications relating to our domain. These strategies are used for model checking and theorem proving.

Product-line Model Checking

Pursuing product-based analysis strategy, Plath and Ryan have introduced the first approach for model checking applied to SPLs [Plath and Ryan 2001]. Since then, many publications have proposed different approaches for model checking, most of which has followed family-based strategy [Apel et al. 2011, 2013d; Classen et al. 2014; Thüm et al. 2014c] and the remaining of which has pursued product-based and feature-based strategies [Fisler and Krishnamurthi 2001; Liu et al. 2011; Plath and Ryan 2001; Apel et al. 2013d].

Product-line Theorem proving

Based on the survey publications about the analysis strategies used for theorem proving, we recognize that there are fewer publications for theorem proving than for model checking. Most of the publications, such as [Harhurin and Hartmann 2008; Delaware et al. 2011; Hähnle and Schaefer 2012; Damiani et al. 2012] have applied product-based and feature-based analysis strategies. Only several research works, such as [Thüm et al. 2014c] and [Pham et al. 2015], have applied family-based strategy for theorem proving. It means that this research field should be still explored.

The strategies for analyzing SPLs bring efficiency and advantages [Thüm et al. 2014a; Apel et al. 2013d]. Selecting a relevant analysis strategy can help to reduce redundant efforts and generate automatically a large number of products. However, besides the strategies, product generation process still requires more efforts and more efficient methods in ensuring the correctness of products.

1.1.5 Summary of Software Product Line

We have presented the main principles of the SPLE. We have discussed the key concept in SPLs that is variability. Managing variability among a set of products is one of the big challenges for the success of any SPL. The tools for feature modeling focus on the operations for selecting and managing configurations. We discussed the technologies and tools that are necessary to automatically generate final products from different configurations. We

presented the product-line analysis strategies which are considered as a necessary step for analyzing and verifying complex SPLs.

In our approach, we do not specifically deal with the problem of selecting and verifying a configuration and product-line analysis strategies, but rather with the product generation process. We focus on the implementation technique that is able to support all kinds of artifacts and enable the automated generation for a product line. In order to follow such an implementation technique, we consider that a variability model has to be defined, so that, developers can create multiple product configurations that are used as input to the product generation. This model can be analyzed automatically by different tools, to take advantage of the configuration management capabilities of the tools.

1.2 Development of Correct-by-Construction Software

Correct Programming is a perpetual requirement when implementing software. A program is written from a particular specification implementing it. The implementation is run on the assumption that it meets its specification. In most cases the program does not perform exactly as it should. How should this problem be tackled? Testing cannot ensure the complete absence of errors; only a formal proof of correctness can guarantee that a program meets its specification [Thompson 1991].

To prove that programs are correct, correctness proofs can contribute practically to software correctness. However, proofs can rapidly become too complex if there is no efficient method for incrementally producing them. A method which can help solving this issue, called Correctness by Construction (CbyC), is a combination of formal methods and incremental developments [Chapman 2006]. It is proved to be a promising method thanks to the benefits obtained while developing a software, such as a decrease of ambiguity (the major cause of bugs), an avoidance of repetition, management, etc. The method has been applied in industry, demonstrating its effectiveness for reducing defects and increasing productivity, especially in order to develop security and safety-critical applications [Hall and Chapman 2002; Barnes et al. 2006] and B [Abrial 2005].

We introduce CbyC, together with the techniques and languages supporting it, in Sec-

tion 1.2.1. In Section 1.2.2, we describe in more detail FoCaLiZe, a language pursuing this approach. Finally, we summarize and relate CbyC to our work.

1.2.1 The correctness by Construction Approach

CbyC is a development style that allows application to progressively evolve by refinement via small steps. The key of the approach is the correctness warranty that the application behaves as specified at each step. The notion of *refinement* was first introduced by [Dijkstra 1976] and then Back [Back 1978] has proposed *stepwise refinement* technique for constructing correct programs. The *refinement calculus* is introduced for reasoning about programs, in which the correctness of a program is preserved while the program is refined or improved using the stepwise refinement technique [Morgan 1993; Back and Wright 1998].

Because of the gains, CbyC approach is also applied to many research applications. SPARK is one of the early languages which applies the approach, it is used for developing high-quality software [Hall and Chapman 2002]. The developer implements the software in SPARK, using the Toolsets (including static verification and design method). Guarded Command Language (GCL) is another language supporting the definition of pre- and post-conditions. An GCL program is engineered using CbyC method an translated into a common language, such as Java, C++, etc. for execution [Kourie and Watson 2012].

Besides the languages mentioned above, there is a language which is very popular in specification and verification domain, also pursues the CbyC approach: B. The method B [Abrial 2005] is a method to specify, design and build sequential software. In B, a specification is stepwise refined into an executable code. At each step, it must be formally proven that the previous steps properties are still satisfied. An evolution of B, Event-B has the purpose of modeling event based systems [Abrial 2010]. To the present time, there are several industrial projects written in B and used in practice, such as, transportation systems, banking, etc. Rodin toolset[4], Atelier B[5], and B toolkit[6] are tools supporting the B method and have proved to be powerful and easy in managing correctness proofs.

[4] http://www.event-b.org/
[5] www.atelierb.eu
[6] https://github.com/edwardcrichton/BToolkit

FoCaLiZe is a language built for developing correct-by-construction programs [Prévosto 2003] .The language contains the formal method design for the modularity of specification and verification, and other mechanisms for managing all the development life cycle of correct-by-construction software [Ayrault et al. 2009]. In the next section, we will present what the language is and discuss how it supports effectively software development.

1.2.2 FoCaLiZe

In this section, we briefly present FoCaLiZe. The FoCaLiZe language has an object oriented flavor allowing inheritance, late binding and redefinition [Prevosto and Doligez 2002]. These characteristics are very helpful to reuse specifications, implementations and proofs. We worked with the FoCaLiZe[7] environment that provides a set of tools to specify and implement functions and logical statements together with their proofs. The development in FoCaLiZe is easily proceeded (in functional programming setting) because of allowing the developer to write implementation code whose style is closed to the functional language, OCaml. A FoCaLiZe source program is analyzed and translated into OCaml sources for execution and Coq sources for formal verification.

A FoCaLiZe specification can be seen as a set of algebraic properties describing relations between input and output of the functions implemented in a FoCaLiZe program. These properties are written in a very common precise language, the first-order logic. For writing code, FoCaLiZe offers a pure functional programming style close to ML, featuring strong typing, recursive functions, data types and pattern-matching. Proofs written using the FoCaLiZe proof language are sent to the Zenon prover which produces proofs that can be verified by Coq for more confidence [Bonichon et al. 2007]. The FoCaLiZe proof language is a declarative language in which the programmer states a property and gives hints to achieve his proofs which are performed by Zenon.

FoCaLiZe units are called *collections*. They contain entities in a model akin to classes and objects or types and values. A collection is called a complete unit when it implements a species whose all properties are proved and all functions are defined. A collection has functions and properties which can be called using the "**!**" notation. It is derived from

[7] http://focalize.inria.fr

other units called *species* which can specify and implement functions and also contain proofs.

A species defines a set of entities together with functions and properties characterizing them. At the beginning of a development, the representation of these entities is usually abstract, it is defined later during the development. The type of these entities is referred as `Self` in any species. Species may contain specifications, functions and proofs. More precisely species may specify a function or a property (with respectively `signature`, `property` keywords) or implement them (`let` keyword when a function is defined, `proof of` keyword to introduce a proof of a property). A `let` defined function must match its signature and similarly a proof introduced by `proof of` should prove the statement given by the `property` keyword. Statements belong to first order typed logic.

As said previously, FoCaLiZe has an object-oriented favor and integrates inheritance, late binding and redefinition to ease reuse and modularity. Inheritance allows the definition of a new species from one or several other species. The new species inherits all the functions, properties and proofs from its parents. Some syntactical mechanisms are provided to prevent ambiguities. A species may provide a definition for a function that is only specified in its parents. It may also redefine a function when this one is already defined in a parent but in that case the signature is maintained (no overloading). Multiple inheritance comes with a late binding mechanism close to the one found in object oriented languages. FoCaLiZe and its tool set, based on an unique framework, brings the benefits, such as, easy reuse and modularity, ambiguity prevention, automated proving, etc. These characteristics are helpful to write and reuse specifications, code and correctness proofs.

1.2.3 Correct-by-Construction Software Product Line

Because of the benefits gained from CbyC, we consider adopting this development style for SPLs.

According to the introduction of the thesis about the first experiment in developing SPL using FoCaLiZe, we faced several issues. Even if the characteristics of FoCaLiZe are helpful to reuse specifications, code and correctness proofs, FoCaLiZe does not have SPL flavor. Namely, FoCaLiZe does not contain any technique supporting the basic princi-

ples of product line engineering. For example, to be able to manipulate feature-oriented programming technique, we have to split the artifacts, belonging to a feature, into small related species, hence reuse or compose to other artifacts. In addition, the relations between species are not easy to understand even by an FoCaLiZe developer. Especially, when a SPL is modeled into several features, the number of species and the relations become higher and the specifications more complex. Furthermore, FoCaLiZe does not supports the refinement or the modification of a property. In order to reuse, a trick was made by copying the statement of the property. All of these manipulations, while developing SPLs using FoCaLiZe mentioned above, have not specialized that make FoCaLiZe difficult to be used by non non FoCaLiZe experts.

Let us consider the research work [Apel et al. 2013d] using model checking approach. The authors proposed a tool chain, called SPLVERIFIER, implementing a method which is activated on both specification and correctness proof. This tool chain is now only available for modules in Java and C. Although model checking can be applied fully automatically, it may be very hard to prove strong properties about SPLs. By contrast, using theorem proving facilitates proving these properties.

Let us consider another research work [Thüm et al. 2011] on the development of SPLs, that adheres to the theorem proving approach. It is closely related to our main goal but does not pursue CbyC. Besides expressing the variability of code artifact using FOP, Thüm et al. applied design-by-contract for expressing variability of specification artifacts, i.e., contracts, into the same unit together with code artifact [Thüm et al. 2012a]. Specification artifacts are expressed using the Java Modeling Language (JML) [Burdy et al. 2005]. The correctness proof artifact was consider as a part by itself and placed outside but related to the unit. They use Why and Krakatoa[8] to generate proof obligations for the proof assistant Coq. When generating a final product, its corresponding specification and code artifacts are composed via the mechanisms proposed in [Thüm 2015]. This mechanism of proof composition was proposed in [Thüm et al. 2011], in which partial proofs related to units were also composed. The tool FEATUREIDE, improved from FEATUREHOUSE [Apel et al. 2009], implemented all of these mechanisms. These tools are now only available for

[8] http://why.lri.fr/

modules in Java.

A limitation of this research is that the relation between specification and correctness proofs are not formally defined, in other words, they are activated independently to each others. Moreover, while generating a final product, the related units are composed without taking into account the proofs while we consider that both specification and proofs should be composed together. In addition, they did not define formal composition rules for proof artifact, which is very important for automated generation of correct products. In fact, they only analyze and demonstrate in text how proofs written in Coq can be composed. This means that there is still no automated generator for all products.

Delaware et al. [Delaware et al. 2011] have proposed a method to prove development correctness aiming at the reuse of proof artifacts. The authors have showed how to develop SPLs with theorems and proofs built from modules. Similar to the partial proofs in [Thüm et al. 2011], proof fragments are written in Coq and located into each module, and thus enable their reuse. These fragments are composed to build a complete correctness proof for each product. However, the method have been only implemented on a limited language domain, namely on a SPL of programming languages: Featherweight Java (FJ) and Featherweight Generic Java (FGJ). Let us not that there is no module-level composition operation that eases the composition of new modules. In other words, without this operation, the method can not be automated composition of proof fragments to build new languages.

1.2.4 Summary of Development of Correct-by-Construction Software

In this last section, we have presented the CbyC approach and the FoCaLiZe language that supports the development of correct-by-construction programs. We have also indicated the limitations found when we adopted CbyC for SPLs. In our approach we rely on the principles of stepwise development and take advantage of the strong points of the FoCaLiZe language. In order to pursue our main goal, we also look for solutions that can overcome existing limitations, such as defining a composition operation at module level for all kinds of artifacts including proofs. These solutions would guarantee to be able to reuse encodings, and the correctness of products generated from automatic product generator.

1.3 Summary

In this chapter we have briefly introduced the principles and basic concepts of two main approaches in software engineering: Software Product Lines and Correctness-by-Construction approach. We consider that these approaches can be combined for developing correct-by-construction SPLs. We have reviewed research directions to develop SPLs employing the approaches presented in this chapter.

The next chapters describe the contributions of this dissertation. We illustrate our methodology that allows us to develop correct-by-construction SPLs in Chapter 2. Later, we concentrate on the details of the FFML language introduced to give a SPL favor to FoCaLiZe and the FFML Product Generator tool. The language is defined for writing SPLs in Chapter 3. The tool is developed for automated generation of final products in Chapter 4. An evaluation of our methodology is presented in Chapter 5 by mean of the development of an example : a Poker SPL game.

Chapter 2

Methodology for Generating Automated Final Products

Ensuring correctness and reliability of all products of SPLs, the traditional techniques, such as type checking, model checking, and theorem proving, have to face many challenges. In product-line verification, we analyze whether all products of the product line meet their specifications. Recently, some approaches have been proposed to verify SPLs, however, they are often just used as proof-of-concept for verification techniques and not justified empirically. To the best of our knowledge, this chapter presents the methodology as the first systematic discussion of how to generate automated products of product lines using a CbyC approach.

According to the expectation presented in the introduction, we propose a methodology, developed along the SPL life cycle, generating the correct final product from the feature selection of a feature model. The methodology responses to the following requirements: the existing limitations of verification technique are overcome; the product generation is automated; and the products of SPLs generated by the methodology are correct;

In this chapter, we aim at presenting our the whole approach that is to facilitate the development of correct-by-construction software using product lines. The approach allows features together with their proofs to be reused. A mechanism to compose features with their proofs is proposed. The variability that is all the possible solutions of a problem, is expressed. We introduce here the way we develop a SPL and generate its correct-by-construction final products.

- We use theorem proving techniques and pursuing CbyC approach,
- We choose the FOP technique as the core implementation technique deployed into the methodology,
- We define a module containing all kinds of artifacts (specifications, implementation code and correctness proofs). We propose to build a new language, called **FFML**, to write such modules. Using the language, a developer can easily implement the features of a SPL.
- We develop a tool, called **FFML Compiler**, translating FFML modules into FoCaLiZe code.
- We define formally a composition operation at module level applied for kind of artifacts. The relationships between specification and correctness proof artifacts are explained clearly.
- We develop a tool, called **FFML Product Generator**, collecting all involved FFML modules of a feature selection and implementing the defined composition operation. The tool allows the user to choose configurations from a feature model, and then receive the corresponding final products.

To have a comprehensive view of the methodology, we consider it from both the developer of the SPL and the user of the SPL interested in one or more products. In Section 2.1, we explain our methodology from the developer side. A description of how the methodology supports the user to select and build his own products, is given in Section 2.2.

2.1 Methodology from Developer View

Software developer always looks for facilitation of reusing artifacts. In order to explain why our methodology can support the reuse of artifacts, we begin by describing the methodology from the developer view in Figure 2.1. Applying the principle of FOP technique, an FFML developer works on the feature model of a SPL and develops the modules associated to the features. The process is conducted as follows:

1. From the feature model, using FFML language the developer writes for each feature an FFML module that reflects the feature. The module (a file *.fm*) includes the properties, specifying the functionalities associated to the feature, and the code implementing these functionalities. The correctness proofs corresponding to the properties are assumed at this very first step.

2. The written modules are sent to FFML Compiler to check their syntax and semantics, and then translated into the corresponding FoCaLiZe files *.fcl*.

3. These translated files are sent to the FoCaLiZe Compiler. If the compiler finds out errors, they are reported to the FFML developer.

4. FoCaLiZe developer fills proof hints into the proofs (which were assumed previously) of each module. Based on these given proof hints, the Zenon Prover called by FoCaLiZe, automatically finds the proofs.

5. The proof hints are copied back into the proofs of FFML modules as comments by the FFML developer.

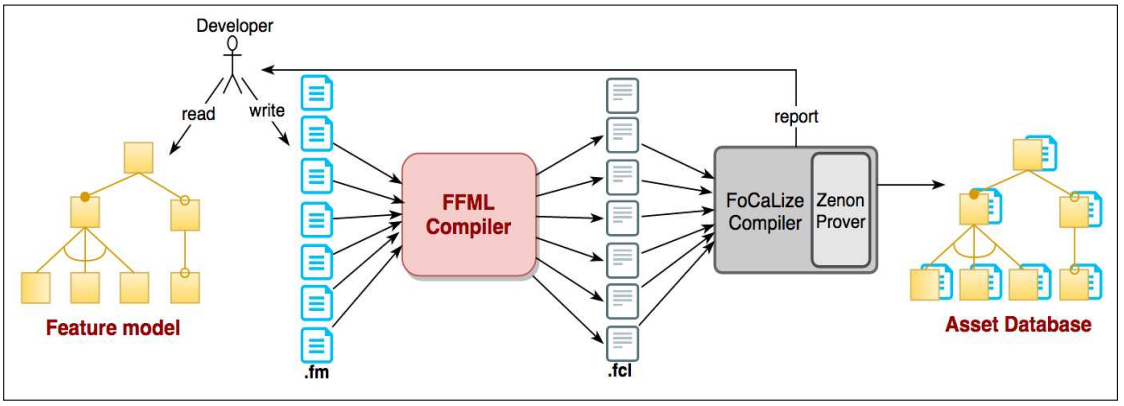

Figure 2.1: The methodology from the developer view

The result achieved after completing the compilation with no errors, is the FFML modules associated to the feature model (the right model in Figure 2.1). We put these modules and the feature model into a database (called the asset database) of the SPL which will be used to generate the final products. Notice that, the FFML developer has to know about FoCaLiZe and Zenon to produce the correctness proofs and understand the errors messages.

As indicated in the beginning of this chapter, FFML language is created for writing FFML modules. In fact, the language supports the FOP technique allowing artifacts to be modularized into modules. The FOP technique only contains the basic principles, i.e., each feature of a SPL is reflected by an FFML module. Hence, to generate automatically products of a SPL we have to define the mechanisms for composing these modules. Furthermore, three kinds of artifacts (specifications, implementation code and correctness proofs) contained in the modules make the composition process more complex.

Example. For Bank Account product line presented in Section 1.1.1.1, we write five modules in FFML associated to the five features of the feature diagram (Figure 1.4): module BA associated to feature BankAcount, module DL associated to feature DailyLimit, module LL associated to feature LowLimit, module CU associated to feature Currency and module CE associated to feature CurrencyExchange. The feature diagram and the modules are saved into the system as asset database. The user can read this database to configure his own product with the expected functionalities.

In the next section, we will consider the automated generation of products from the user view.

2.2 Methodology from User View

The activity of generating an FFML product is illustrated in Figure 2.2. We show another view of our methodology that allows the user to choose a configuration as input and receives an FFML product as output. The process is as follows:

1. From the asset database of a SPL (the left model in Figure 2.1) the user selects some features which are supposed to be essential for an expected product. To decide which features must be selected in the product, the user has to look at the feature model and the specifications in the FFML modules.

2. The configuration selected by the user, if it is valid, is sent to the FFML Product Generator tool. The generator retrieves all the modules involved in the configuration from the asset database.

3. The involved modules are composed and built into a final product that is described as a set of FFML composite modules, called the FFML product. The generator embeds a set of composition rules that are defined for composing all kinds of artifacts (specifications, code and correctness proofs). It can report to the user several warnings that can appear while composing the artifacts, such as a lack of proof hints.

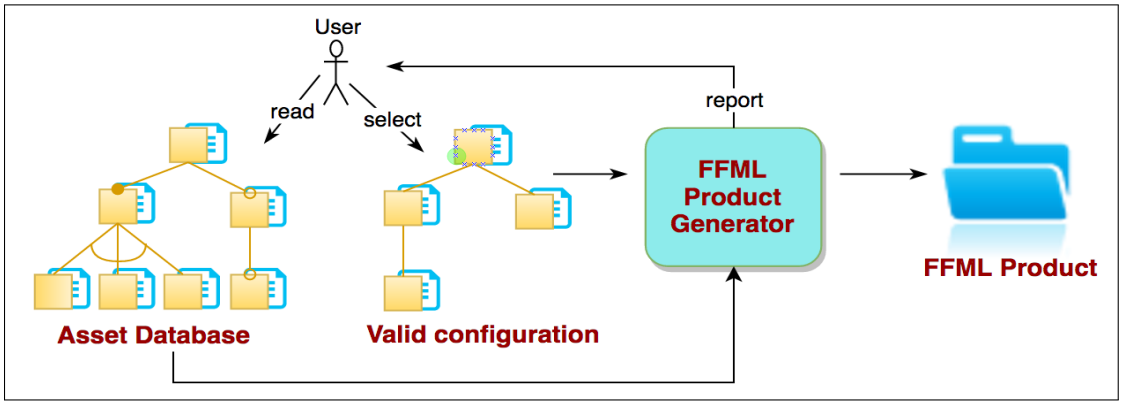

Figure 2.2: The methodology from the user view

The result from FFML Product Generator is the final product containing the composite artifacts obtained from all involved modules. The composite specifications are guaranteed to be satisfied by the composite implementation. The process of the product generation is done automatically but some proofs may be done manually (because of the current status of the FFML Product Generator).

A configuration selected by the user is a potentially partial one, and may be valid or not. In our work, we require a valid full configuration, however, there are many tools checking the validity of configurations, such as a prototype extending **fmp** (an Eclipse plug-in) [Czarnecki et al. 2005], *pure::variants*[1], **guidsl** [Batory 2005a], etc. In the scope of this thesis, we do not focus on these operations but use the result from them. Furthermore, we assume that a configuration is not a set of features but a sequence of features imposing a certain order of composition. We will also consider only valid configurations.

[1] https://www.pure-systems.com

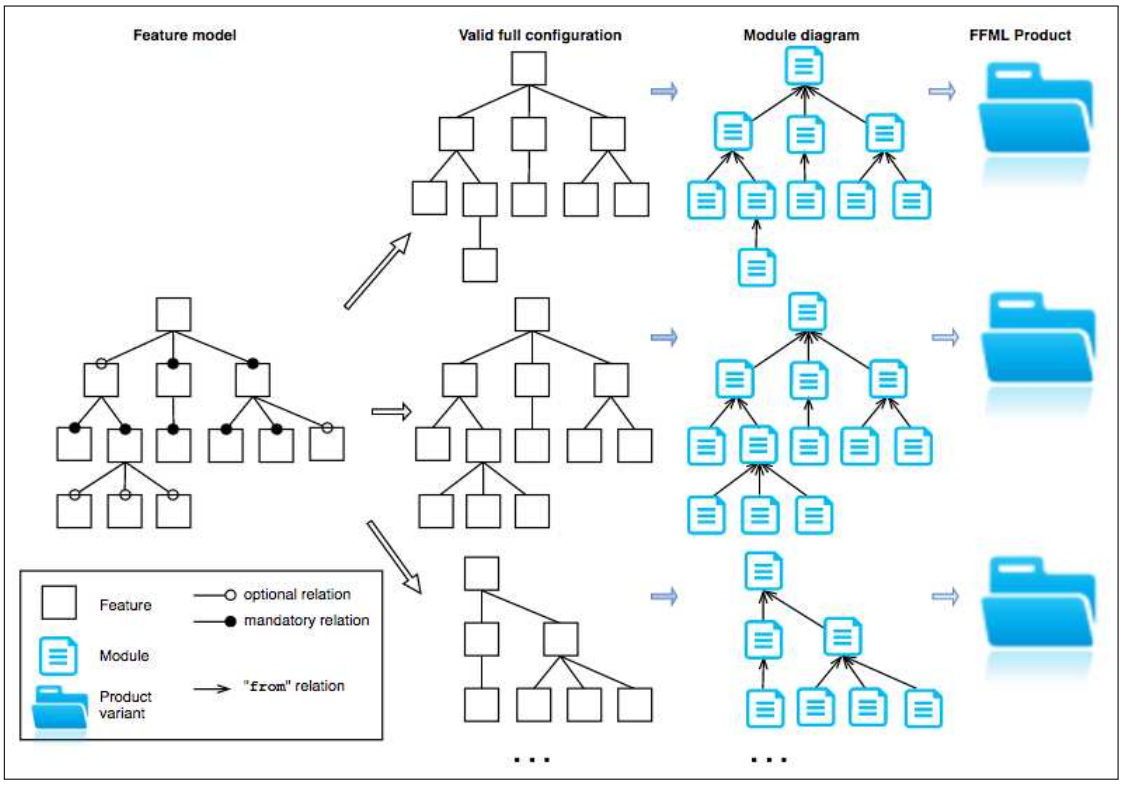

Figure 2.3: FFML products

Product Generation Process

Assuming that the composition mechanisms for the modules are reliable, an FFML product, generated by the FFML Product Generator, contains the composite artifact established by these mechanisms. If the composite correctness proofs prove that the product satisfies the composite specifications, we can conclude that the product is correct.

A global picture of the product generation process is illustrated in Figure 2.3. The FFML products, built from a feature model, are the expected results generated by the FFML Product Generator tool. For a valid configuration, the FFML modules corresponding to the involved features are collected in a module diagram. The module diagram has the same hierarchy as the feature model but is restricted to the features involved in the configuration. The nodes of the diagram are the modules that are related to each other via relationships. Based on the module diagram and the implementation of the composition operation, FFML Product Generator composes all the artifacts of the modules to build the FFML products.

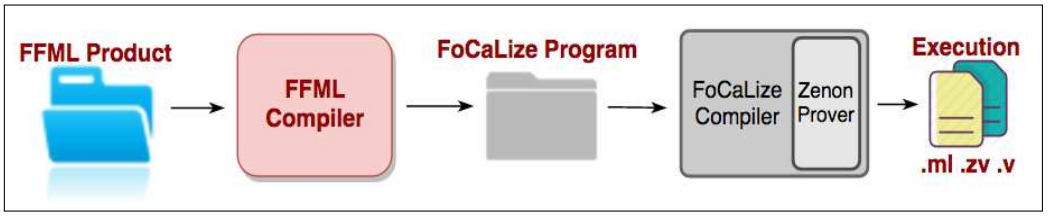

Figure 2.4: Translation of an FFML Product into FoCaLiZe

Product Execution

The result, established by the methodology after composing all involved modules of a configuration, is an FFML product that is also translated into a FoCaLiZe program for verification, and thus this program is compiled into an OCaml executable file. This process is illustrated in Figure 2.4. For each module of the product, the composite implementation code is proven to meet the composite specifications by the composite correctness proofs established by the composition process. FoCaLiZe calls Zenon Prover to prove automatically the satisfaction of the composite properties.

Products of Bank Account Product Line

We consider the example of Bank Account product line mentioned in Section 2.1. From the asset database, the system allows a user to select one of the twelve valid following configurations:

$$C_1 = (\{BankAccount\})$$

$$C_2 = (\{BankAccount, DailyLimit\})$$

$$C_3 = (\{BankAccount, LowLimit\})$$

$$C_4 = (\{BankAccount, Currency\})$$

$$C_5 = (\{BankAccount, Currency, CurrencyExchange\})$$

$$C_6 = (\{BankAccount, DailyLimit, LowLimit\})$$

$$C_7 = (\{BankAccount, DailyLimit, Currency\})$$

$$C_8 = (\{BankAccount, LowLimit, Currency\})$$

$$C_9 = (\{BankAccount, DailyLimit, LowLimit, Currency\})$$

$$C_{10} = (\{BankAccount, DailyLimit, Currency, CurrencyExchange\})$$

$C_{11} = (\{BankAccount, LowLimit, Currency, CurrencyExchange\})$

$C_{12} = (\{BankAccount, DailyLimit, LowLimit, Currency, CurrencyExchange\})$

Assume that the configuration C_6 is selected with the expectation of receiving a product with the functionalities obtained from the features BankAccount, DailyLimit and LowLimit. Besides the basic management from BankAccount, the product also contains the functionalities, allowing the bank to limit the amount of money withdrawn in a day and the amount of each withdrawal, obtained from both DailyLimit and LowLimit. The system will take the corresponding modules (i.e., module BA, DL and LL) as input, compose these three modules, and produce the FFML final product. This product is translated into an FoCaLiZe program that is correct-by-construction and can be executed.

2.3 Summary

In this chapter we describe our methodology for generating correct products of SPLs from both the developer view and the user view. Besides proposing an automated generation, we also expect that our methodology to be able to apply to other target languages (which are required to be able to express specifications, implementation code as well as correctness proofs, such as B or Java). The requirements and also the solutions for implementing the methodology are presented. A tool chain realizes the methodology in order to allow the user to choose a collection of features and receives corresponding correct final product. In the next chapters, we present how to adapt our methodology to FoCaLiZe by building a new language FFML and how to generate the correct products of a SPL.

Chapter 3

FFML Language

This chapter is a part of work [Pham et al. 2015] which presented FFML language at FMSPLE'15. Furthermore, the chapter shares the material with [Dubois et al. 2016] that described the advantages of FFML in developing SPLs.

We begin by explaining why we decided to create the FFML language in Section 3.1. The definition of a FFML module is described in Section 3.2. In Section 3.3.1 we show the grammar of FFML and then its semantics is explained in Section 3.4. FFML Compiler, a translation tool is presented in Section 3.5 and the correctness of the compiler is discussed in Section 3.6. The last section is a summary of this chapter.

3.1 Towards a Formal Language for Correct-by-construction Software Product Lines

As indicated in Section 1.2.2, FoCaLiZe is an efficient language for specifying, implementing and proving software. It allows the development of correct-by-construction programs. In our early work, we started with the hypothesis that the development of correct-by-construction SPLs can be achieved with the support of this language but we rapidly realized it is not easy to develop them in FoCaLiZe.

The different limitations we found while developing SPLs in FoCaLiZe are the reasons why we decide to build another language. We analyzed these limitations in Section 1.2.3. The main limitation is that FoCaLiZe does not have SPL favor, namely it does not contain any automated implementation technique for SPLs. To develop SPLs in FoCaLiZe, the

developer has to conduct a lot of manipulations that are sometimes difficult to understand.

With our main purpose of developing correct-by-construction SPLs, we begin by finding a relevant implementation technique applicable to our work. We investigate several implementation techniques to assist us in achieving the purpose. An obvious fact that over the last decade, FOP become a well-known technique for reusing assets in SPLE [Apel et al. 2013a]. As indicated in the "Introduction", we presented the reasons why we choose FOP as a method for designing and managing the variability of the SPLs. The basic principle of FOP technique is each feature of a SPL's feature model is mapped to a separate module that implements the feature. Hence, each module can contain its implementation code including its specification and correctness proofs. We can notice that the modularity of correctness proofs was proven effective by many previous works, such as [Delaware et al. 2011; Thüm et al. 2011; Pham et al. 2015; Batory 2015]. Once the proofs can be modularized and built as other artifacts, such as specification and implementation code, the proof artifacts related to a module can be written into a unit and evolved together with its specification and code in a common language. We can also build mechanisms for reusing both specification and correctness proof in the language. This will make the formal development of SPLs become more understandable and reduce the coding efforts.

Inspired by FOP technique and FoCaLiZe, we decided to create **FFML** (for Formal Feature Module Language) which is close to FoCaLiZe but brings new mechanisms and reduces the limitations to reach our main purposes for developing correct-by-construction SPLs [Pham et al. 2015]. A compiler of the language can specialize the complex developing manipulations in FoCaLiZe. The language supports modularity, variability management and also composition of artifacts. Thanks to FOP's basic principles, the organization of the artifacts is proceeded in independent modules. These modules will be translated into FoCaLiZe to check their correctness. Zenon, the prover embedded in FoCaLiZe, automatically proves that the implementations meet their specifications using the proof hints given in the correctness proofs. We rely on FoCaLiZe in developing correct-by-construction software and on Zenon in automated proving its correctness.

In FFML, the properties of SPLs are specified in the same way as in FoCaLiZe but they are managed in a different way. The method we apply to the specification of SPLs

is inspired by design by contract [Meyer 1992]. The technique has appeared for more than two decades, but recently it is used widely to specify formally the behaviors of SPLs [Thüm et al. 2012b]. Meyer and Thüm et al. defined the techniques in object-oriented programming setting. However, we put ourselves in functional programming setting and consider properties as the first order formulas specifying the desired behaviors of functions. Moreover, following the key idea of design by contract [Meyer 1992], for each function there are a collection of properties which specify its desired behaviors. In fact, we apply this basic idea to manage the writing of the properties. The properties in FFML is linked to a function and they are calculated together when the function is (re)defined. As a result, it is easier to organize and manage the properties following the functions which they are related to.

The method used in FFML for representing the variability (the reuse and modification of the specification) of SPLs in terms of syntax is inspired by delta-oriented programming (DOP) which was proposed by Schaefer et al. [Schaefer et al. 2010]. As indicated in the chapter Background (Section 1.1.3), DOP is extended from FOP with a purpose for implementing SPLs more flexibly. Its principles allow the developer to add, modify and even remove implementation code, such as, classes, functions and interfaces in *delta modules*. We apply similar mechanisms for FFML while refining and modifying artifacts. In addition, we are also motivated by the generation principle of DOP that from a *core module* (which is necessary for all product variants) and the delta modules, the final products can be established. These principles are also applied in the translation of FFML into FoCaLiZe.

As the basic principles of FOP technique, each feature of a feature model is implemented by a corresponding module. After written in FFML, these modules are embedded into separate files (*.fm*). Although FFML is inspired by FoCaLiZe, FFML's syntax is designed so that it is suitable for writing, reusing and modifying artifacts easily. FFML allows the user to write only what differs from one module to another one, hence reduces inessential code. Finally, these modules are translated into files (*.fcl*) in FoCaLiZe by FFML Compiler.

3.2 Module definition

In this section, we begin by giving a classification of module artifacts in Section 3.2.1 then the definition of an FFML module in Section 3.2.2 which contains these classified module artifacts.

3.2.1 Classification of Module Artifacts

With the desire to develop correct-by-construction SPLs by means of FOP technique, each node (feature) of the feature model of a SPL is associated to a module. Each of these modules requires the specification of the expected behaviors as a collection of properties. A module also contains the code implementing it. This implementation is later proved to meet its specification by a collection of correctness proofs. This motivates us to define an FFML module which contains three kinds of *artifacts*: *specification, code* and *correctness proof*, as the assets of a unit structure. We use the term artifact from now to call all of these assets.

In our context, each module includes its artifacts: specification, code and correctness proofs. Specification is given here as a set of function declarations (signatures) and expected properties or requirements. Technically these properties are logical formulas relating together some functions described only by their signatures. Thus, in our setting a specification is close to an algebraic specification of an abstract data type [Goguen et al. 1976]. Code artifact has to be understood as the implementation of the functions declared in the specification. In our work, we place ourselves in a functional programming setting. The proof artifacts concern the correctness of the code with respect to the specification. The three kinds of artifacts are defined as follows:

- **Specification artifact** includes function declarations and the properties associated to the functions. A function declaration or signature only describes the name of the function, and the types of its arguments and result. A property is expressed by a first order formula. For instance, a square function, named *pow*, is declared with input and output types as *double*. A property *not_negative* of *pow* is written as "***property*** *not_negative:* ***all*** *x :* ***double****, pow(x) >= 0;*", meaning that the returned value of the

function is always positive. In FFML context, a property is either a new one which is expressed by a new logical formula or refined/modified from an existing property of a *parent* module. FFML will support the developer to write these properties.

- **Code artifact** consists of the definition of the representation type and the function definition/re-definitions. The representation type of a module is the concrete type associated to the abstract data-type specified in the specification artifact. It will be a concrete type (*à la ML*) or a Cartesian product of concrete types (complex). In an FFML module, the representation type is unique for each module and can be also established from the representation type of a *parent* module. For instance, the representation type of a module B, is denoted by $A * int$, in which A is the parent of B. $A * int$ means that the representation in B is the one in A extended by an integer. If type A is string then type B is the type of tuples made of a string and an integer. A function may be (re)defined after the representation type has been defined. The implementation of a function can refer to the existing one of a *parent* module.

- **Proof artifact** contains correctness proofs. These proofs appear as comments in FFML. The proofs are done in FoCaLiZe, that is, they are done on the translated code. Zenon Prover will automatically prove and the developer will copy them back to FFML. In a module, the correctness proofs are written corresponding to the specified properties. While writing the correctness proof of a property which refines/modifies the same property of a parent module, the developer can mention the parent's as a property proof hint. This proof hint allows the reuse of the corresponding correctness proof from the parent module.

3.2.2 FFML Module

As presented in the previous section, an FFML module contains three kinds of artifacts in a single setting. The module definition enables the description of the commonality and the variability with respect to the artifacts of the parent module. A module is a modification from its parent module. Hence, we define a module FM as a unit structure (like *class* in Java or *species* in FoCaLiZe). An FFML module FM is a tuple containing

the following elements:

$$FM := (MN, PN, S, P, R, D, Pf) \tag{3.1}$$

Equation 3.1 represents the module named MN which is a child of module named PN. In a special case if FM is the root module (i.e, the FFML module associated to the root of the feature model), there is no PN element. The module only contains the artifacts which are extended/modified from its *parent* (PN). The specification of module FM consists of a set of function declarations S (including constants) and a set of properties P that specify the desired behaviors of these functions. The declared functions S are (re)defined later and the definitions are collected into D. The properties P are proved by correctness proofs Pf written by a FoCaLiZe developer. The representation type R implements the abstract data type as a concrete type. The elements of the representation types are always extended from that of the parent PN. The representation type R and the function (re)definitions D are the code artifact of the module.

The FFML syntax is created with the purpose of writing the modules represented in Equation 3.1. Each of these modules are put into a separate file *.fm*. FFML grammar provides keywords in order to represent all included artifacts. However, the last element Pf being proofs, written and completed in FoCaLiZe, are copied back into the file $.fm$.

We define a *complete module* in Definition 1 as a module in which all of its declared functions are defined and all of its properties are proved. Moreover, as shown in the beginning of this chapter, FFML is inspired by FoCaLiZe, so the definition of the complete module is also related to the completeness of species which is a condition to execute the species in FoCaLiZe.

Definition 1 *A module is called a* ***complete module*** *when all its declared functions are defined and all its properties are proved.*

The completeness of FFML modules is an important property for FFML Compiler. Once a complete module is proved successfully, its correctness is guaranteed. It also can be executed and validated by test cases. This is an initial step to guarantee the correctness of theses modules and also the products which are constructed from such modules. Based

on the definition of an FFML module in Equation 3.1, in the next section we will present the grammar of FFML.

3.3 FFML Grammar

Our main objective in defining FFML is to propose a language close to FoCaLiZe that already allows the developer to write the feature modules from a feature model of a SPL. All artifacts are put in a single setting. FFML is inspired from FoCaLiZe, in particular concerning styles for writing specifications, code and correctness proofs. FFML and FoCaLiZe differ mainly in the way to structure and organize information. However, as we will see in this section, FFML allows the developer to focus on expressing reuse and modification of module artifacts. The syntax of FFML is presented in Section 3.3.1. We classify the properties in FFML in Section 3.3.2. In Section 3.3.3 we focus on a keyword `from` upon which the reuse and modification mechanisms of FFML are revealed. In Section 3.3.4, we show how the example of Bank Account is implemented in FFML (see Section 1.1.1.1).

3.3.1 Syntax

When we consider a SPL written in FFML, it can be defined as a collection of modules. The user-defined types related to a module are gathered in a file called a user-defined type file. Let us briefly look into a module and a user-defined type file.

- **Module.** A module can be defined as a unit that contains the artifacts to implement the module.
- **User-defined type file.** This file contains the types that are defined by the user and related to a module.

The concrete grammar of FFML is presented in Grammar 3.1. A module is introduced by a keyword `f_module` and its name. The keyword `from` expresses that a module having name *id* is extended from its parent module named *parent*, which follows the principle of FOP. Keyword `from` is very important because it enables representing the fundamental

mechanisms for reusing and modifying module artifacts in FFML. It is an optional part which is not required in the root module.

⟨*fm*⟩ ::= '`f_module`' ⟨*id*⟩ ['`from`' (⟨*parent*⟩)*,]
(⟨*sig*⟩)* (⟨*prop*⟩)* ⟨*rep*⟩ (⟨*def*⟩)*
(⟨*proof*⟩)* '`;;`'

⟨*parent*⟩ ::= ⟨*id*⟩

⟨*sig*⟩ ::= '`signature`' ⟨*func_n*⟩ '`:`' ⟨*type*⟩ '`;`'

⟨*prop*⟩ ::= ⟨*new_prop*⟩ | ⟨*inv_prop*⟩ | ⟨*ref_prop*⟩

⟨*new_prop*⟩ ::= '`contract`' ⟨*func_n*⟩
'`property`' ⟨*prop_n*⟩ '`:`' (⟨*lvar*⟩)*, [⟨*lexpr*⟩
'`->`'] ⟨*lexpr*⟩ '`;`'

⟨*inv_prop*⟩ ::= '`contract`' ⟨*func_n*⟩
'`invariant property`' ⟨*prop_n*⟩ '`:`'
(⟨*lvar*⟩)*, [⟨*lexpr*⟩ '`->`'] ⟨*lexpr*⟩ '`;`'

⟨*ref_prop*⟩ ::= '`contract`' ⟨*func_n*⟩
'`property`' ⟨*prop_n*⟩
'`refines`' ⟨*parent*⟩'`!`'⟨*prop_n*⟩
['`extends premise`' ⟨*lexpr*⟩] '`;`'

⟨*rep*⟩ ::= ⟨*bas_rep*⟩ | ⟨*ext_rep*⟩

⟨*bas_rep*⟩ ::= '`representation =`' ⟨*c_prod*⟩ '`;`'

⟨*ext_rep*⟩ ::= '`representation extends`'
⟨*parent*⟩ '`with`' ⟨*c_prod*⟩ '`;`'

⟨*def*⟩ ::= '`let`' ⟨*func_n*⟩ ['`(`' (⟨*par*⟩)*, '`)`'] '`=`'
⟨*expr*⟩ '`;`'

⟨*proof*⟩ ::= '`proof of`' ⟨*prop_n*⟩ '`=`'
'`foc proof`' ⟨*focproofbody*⟩ '`;`'

⟨*focproofbody*⟩ - FoCaLiZe_proof_body

⟨*type*⟩ - type

⟨*lvar*⟩ - variable_with_universal_quantifier

⟨*lexpr*⟩ - predicate_logical_formula

⟨*expr*⟩ - expression

⟨*c_prod*⟩ -
Cartesian_product_of_concrete_types

⟨*id*⟩ - module_name

⟨*prop_n*⟩ - property_name

⟨*func_n*⟩ - function_name

⟨*par*⟩ - function_parameter

Grammar 3.1: FFML module

A new function declaration *sig* is introduced by a keyword `signature` with its name *func_n* and its type *type*. By adhering to the basic principles of design by contract, for each function the developer can write a collection of properties that specify the expected behaviors of the function. Keyword `contract` followed by a function name *func_n* of a property definition indicates that the property is related to the function *func_n*. A new property is specified by keyword `property` with its name *prop_n*, a list of variables *lvar* (with universal quantifiers), and followed by two logical expressions *lexpr* (without quantifiers of the global variables *lvar*) as its premise and conclusion separated by the implication connective. However the other local variables in *lexpr* can be with with universal quantifiers.

Using keyword `representation`, the representation type *rep* of a module is established either from a Cartesian product *c_prod* of concrete types or an extension from the

representation type of its parent. In the extension case, two more keywords `extends` and `with` are used to describe this extension as a Cartesian product of the representation type of the parent together with the extension. After concreting the abstract type of the module by the representation type, the function, declared early (sig), are defined by definition def with expression $expr$ using keyword `let`. These function definitions follow the functional programming paradigm.

The correctness proof of a property is written using keyword `proof of` followed by the property name and a proof body description $focproofbody$. As discussed in Section 3.2.2, the proof body is a FoCaLiZe proof considered as a comment in FFML.

The elements such as $focproofbody$, $type$, $lvar$, $lexpr$, etc. are not defined in the concrete grammar of FFML because they are very close to the FoCaLiZe corresponding syntactic categories (see Grammar A.1 in Annex).

A module may be associated with a user-defined type file ($.tp$). The concrete grammar of this file is represented as follows:

⟨*ftype*⟩ ::= '`f_type`' ⟨*id*⟩ (⟨*dtype*⟩)* '`;;`'

⟨*dtype*⟩ := '`type`' ⟨*type_n*⟩ '`=`' ⟨*type*⟩ '`;`'

Grammar 3.2: User-defined type file

The file is introduced by a keyword `f_type` followed by its name id. The file has the same name as the related module. A type $dtype$ is defined in the file using a keyword `type` with its name $type\_n$ and its type $type$. Such a file appears in Poker SPL developed in Chapter 5.

In the next section we classify the properties and clarify how they can be used in FFML.

3.3.2 Classification of Properties

Before explaining the semantics for the modification and reuse in FFML, we define a property following the principle of design by contract. Given a function $f : U \rightarrow V$ in module FM, the properties P_f of f is represented by a set of all properties $p_{fi}(\overrightarrow{x_i})$ related to the function, i.e., $P_f = \{p_{f1}(\overrightarrow{x_1}), p_{f2}(\overrightarrow{x_2}), ..., p_{fn}(\overrightarrow{x_n})\}$, in which $\overrightarrow{x_i}$ is the list of universally quantified variables.

The property p_{fi} is described in Equation 3.2, where two predicates $prem_i$ and $conc_i$ are respectively the premise and the conclusion of p_{fi}. The list of variables $\overrightarrow{x_i}$ quantified by $\forall$. This property must be satisfied by the implementation of f:

$$p_{fi}(\overrightarrow{x_i}) := \quad \forall \overrightarrow{x_i} : U, prem_i(\overrightarrow{x_i}) \rightarrow conc_i(\overrightarrow{x_i}) \tag{3.2}$$

We continue to divide the properties (represented by Equation 3.2) into invariant and not invariant ones in Section 3.3.2.1 and Section 3.3.2.2.

3.3.2.1 Invariant Property

The property of a function is possibly a *universal property* defined on [Charpentier and Chandy, 2004] as a proposition that must hold for all systems. Adapting this definition to our context, we define the universal property of a module as a property that must hold for all product variants of a SPL in which the module is involved. As a result of FOP approach embedded into our work, we can observe that a universal property will never be refined in other modules. In FFML, we called such a property an *invariant property* that is defined by Definition 2. The developer uses the keyword `invariant property` (Grammar 3.1) to specify the property. The property will be inherited but can not be refined/modified by one or several modules.

Definition 2 *A property of a module is called* ***invariant property*** *if it is held for the module and for all its child modules.*

3.3.2.2 Non Invariant Property

If a property is not an invariant one, it can be a new property. It also can be a property that is refined from another. It can be specified as an extension (or modification) from a property of a *parent*. We call the property *refining property* that is defined in Definition 3. The keyword `refines property` is used for introducing this kind of property. The extension part is a new premise following the keyword `extends premise`. Due to adding a new premise into the property, the property becomes more restrictive.

Definition 3 *A property which is refined from another property by adding a new premise is called a* ***refining property****.*

Assume that from module FM we build another child module FM_1 and f is redefined in module FM_1. Properties P_{f1} associated to f in module FM_1 is a set of properties $p_{fj}(\overrightarrow{x'_j})$, i.e., $P_{f1} = \{p'_{f1}(\overrightarrow{x'_1}),\ p'_{f2}(\overrightarrow{x'_2}),\ \ldots\ p'_{fn}(\overrightarrow{x'_n}),\ p'_{f(n+1)}(\overrightarrow{x'_{n+1}}),\ \ldots,\ p'_{fm}(\overrightarrow{x_m})\ \}$. The properties $p'_{f1}(\overrightarrow{x'_1})$, $p'_{f2}(\overrightarrow{x'_2})$, ..., $p'_{fn}(\overrightarrow{x'_n})$ are kept, refined or modified from the properties of FM while $p'_{f(n+1)}(\overrightarrow{x'_{n+1}})$, ..., $p'_{fm}(\overrightarrow{x'_m})$ are new properties.

Because function f is (re)defined in module FM_1, the property specifying the expected behavior of f in module FM_1 is either a new property or a modified property from p_{fi} into p'_{fj}. If the function still keeps the behavior, p'_{fj} is the same as p_{fi}. If the behavior of the function is extended/modified, p'_{fj} is represented as in Equation 3.3 in which a new premise $prem'_j$ or new variables are added into p_{fi}. The variables $\overrightarrow{x'_j}$ are considered as an extension of $\overrightarrow{x_i}$. Adding the new premise makes the constraint p'_{fj} more restrictive than p_{fi}.

$$p'_{fj}(\overrightarrow{x'_j}) := \forall \overrightarrow{x'_j} \in U,\ prem'_{fj}(\overrightarrow{x'_j}) \rightarrow p_{fi}(\overrightarrow{x_i}) \quad \textit{where } \overrightarrow{x'_j} \textit{ is an extension of } \overrightarrow{x_i} \tag{3.3}$$

3.3.3 "From" - Reuse and Modification Mechanisms

The keyword `from` is very important for FFML by bringing reuse and modification mechanisms. In this section we summarize the possibilities of reusing and modifying artifacts. A current module can be constructed by modifying from its parent module using the keyword `from`. This keyword implies several mechanisms, such as, inheritance, modification and importation.

Inheritance

The inheritance mechanism allows a current module to be based on and use the artifacts of a parent module. Hence, the module can maintain the behavior of the parent. This mechanism is applied for the following artifacts:

- **Declaration.** The declarations in the parent module are also valid in the current one. We say they are inherited from the parent.
- **Invariant property.** The invariant properties in the parent module are inherited into the current one. The developer may not need to reprove these properties but FFML implies that the current module may use the same correctness proofs of the parent.
- **Property.** The properties in the parent which are not invariant property and not refined or modified, are kept in the current module. We say they are inherited from the parent. If the developer does not reprove them, FFML implies that the current module uses the same correctness proofs as the parent.
- **Function definition.** If a function definition is not modified in the current module, it is inherited from the parent. However, in case of changes in the representation type, a function definition may have to be adapted (this is implicit in the FFML module).
- **Correctness proofs.** A correctness proof in the parent may be kept in the current module. It can also be reused but must be provided by the developer in that case.

Modification

We describe here all possible modification cases of the artifacts in a module from its parent. The modification mechanism in FFML allows adding, refining and extending artifacts (or a part of artifacts).

- **Adding a new declaration**. The developer can declare new functions for a module. These functions have to be defined for the module to be complete.
- **Adding a new invariant property**. Similar to addition of new declarations, the developer can write new invariant properties for a module. The correctness proofs have to be written for them in the module.
- **Adding a new property**. The developer can write new properties and write correctness proofs for them.

- **Refining a property**. The developer can refine a property (not an invariant one) from the parent module and write a correctness proof for it. The refining property will be proved by the corresponding correctness proof written by the developer.

- **Extending the representation type**. The representation type of a current module can be extended from that of its parent. The structure of this representation type is a Cartesian product between the parent representation type and an extended part. If not, the module keeps the representation type of the parent.

- **Adding a new function definition**. In case of declaring a new function in the current module, the developer has to add its definition.

- **Redefining a function**. A function previously declared in a parent (direct or not direct parent), can be redefined in the current module. The definition can be a new one or is modified from the old one of the parent.

Importation

Besides the above mechanisms, FFML allows the importation of files that contain user-defined types for each module. The type definitions imported in the parent module can be used in the current module. The importation mechanism will be used in Poker SPL developed in Chapter 5.

The reuse and modification mechanisms we built for FFML, which is carried on the keyword `from`, bring flexibility.

3.3.4 Example: The Bank Account in FFML

We refer here to the feature diagram of the bank account product line given in Figure 1.4 of Section 1.1.1.1. A module BA (Bank Account) implements the root feature BA. It has three children: modules DL (Daily Limit), LL (Low Limit), and CU (Currency), mapped to the features DL, LL, and CU respectively. The modules DL, LL, CU will be defined using "`from` BA". Module CE (CurrencyExchange) implementing feature CE, is defined from the module CU.

The root module *BA* is shown in Listing 3.1. This module includes three signatures: *over* - is over limit, *get_bal* - gets the current balance value of the account and *update* - upgrades the value balance (lines 2-4). The next part of the module *BA* contains an invariant property *ba_bal_gr_over* (line 6) that concerns the function *get_bal*, saying that the balance of an account is always greater than *over*. The property *ba_upd_succ_with_over* (line 8) concerns the function *update*, saying that a customer can withdraw more money *a* from the account than available if the balance is within *over*. Then the representation type of module *BA* is defined as *int* (line 10), which means that an account is only represented by its balance. Then definitions of the functions *get_bal* and *update* are given (lines 12-15). The invariant property *ba_bal_gr_over* is admitted by the proof (line 17) which contains proof hints mentioned after the keyword `foc`. *Assumed* here means that the property is not proved but is assumed. In fact, it can not be proved, we can only prove that the different functions preserve that property [Rioboo, 2009]. The proof of the property *ba_upd_succ_with_over* (line 18) includes two proof hints: by definition of *update* and *get_bal*. This means that the proof must be done by unfolding the definitions of these two functions.

We show another example, module DL, defined according to the module *BA* using keyword `from` (line 1 of Figure 3.2). Two new declarations *limit_with* and *get_with* are added into the module (lines 2-3). The module introduces the constant *limit_with* only declared at that point. It denotes the limit of withdrawn money in a day. The module also introduces another function *get_with* that returns, for an account, the current amount of withdrawn money in a day. The functions *update*, *get_bal* and *over* defined in *parent BA* are also available in the present module. Remark that they are not even mentioned in DL. A refining property *dl_upd_succ_with_wlimit_R*1 is obtained by modifying the property *ba_upd_succ_with_over* of *parent BA* (lines 5-7) using the keyword `refines`. The modification includes a new premise following keyword `extends premise`. The refining property states that the bank allows a customer to withdraw money only if the amount of withdrawn money in a day is greater than *limit_with* (*limit_withd* are negative numbers). The representation type of module *DL* is defined as a Cartesian product of the representation type of *parent BA* and *int* (i.e. the concrete type associated with the

```
1  fmodule BA
2   signature update: BA -> int -> BA;
3   signature get_bal: BA -> int;
4   signature over: int;
5
6   contract get_bal :: invariant property
        ba_bal_gr_over: all x : BA, get_bal(x)
        >= over;
7
8   contract update :: property
        ba_upd_succ_with_over: all x : BA, all a
         : int, (get_bal(x) + a) >= over ->
        get_bal (update(x,a)) = get_bal(x) + a;
9
10  representation = int;
11
12  let get_bal (x) = x;
13  let update (x, a) =
14      if ((get_bal(x) + a) >= over) then
            get_bal(x) + a
15      else get_bal(x);
16
17  proof of ba_bal_gr_over =
18  foc proof {* assumed *};
19  proof of ba_update_succ_with_over =
20   foc proof
21   {* by definition of update, get_bal *};
22 ;;
```

Listing 3.1 Module BA in FFML

```
1  fmodule DL from BA
2   signature limit_with: int ;
3   signature get_with: DL -> int ;
4
5   contract update ::
        property dl upd succ with wlimit_R1
6    refines BA!ba_upd_succ_with_over
7    extends premise (a <= 0) /\ (get_with (x) +
         a >= limit_with);
8
9   representation extends BA with int ;
10
11  let get_with (x) = second(x);
12  let limit_with = (-70);
13  let update (x, a) =
14   if (a <= 0) then
15    if (get_with(x) + a >= limit_withd) then
16     (BA!update (x,a), get_with(x) + a)
17    else x
18   else (BA!update(x,a), get_with(x));
19
20  proof of dl_upd_succ_with_wlimit_R1 =
21  foc proof {*
22   <1>1 assume x: DL, a : int,
23   hypothesis h1: (a <= 0) /\ (get_with(x) + a
         >= limit_with),
24   prove (get_bal(x) + a) >= over -> (get_bal (
         update(x,a))) = get_bal(x) + a
25    <2>1 prove first(update(x,a)) = BA!update (
           first(x),a)
26    by definition of first, update hypothesis
           h1
27    <2>e qed by step <2>1 definition of over,
           get bal property
           BA!ba_upd_succ_with_over
28   <1>e conclude;*}
29 ;;
```

Listing 3.2 Module DL in FFML

amount of money withdrawn in a day) (line 9). The functions *get_with* and *update* are defined/redefined (lines 11-178. We can notice that the new definition of *update* calls the parent function *update* (BA!update, in lines 16 and 18). The proof is written for property *dl_upd_succ_with_wlimit_R*1 reusing property *ba_upd_succ_with_over* of parent *BA* as a proof hint (line 27). The primitive functions *first* and *second* used in this proof and the definition of function *get_with* (line 11) are the usual projections of a Cartesian product.

It is worth adding that the keyword `from` conveys different meanings in FFML. For example, `from` together with keyword `refines` is used to express the property *dl_upd_succ_with_wlimit_R*1 refining the property *ba_upd_succ_with_over* and adding a new premise using the modification mechanism. The invariant property *ba_bal_gr_over* (line 6 of Listing 3.1) is still available in the module DL because of the inheritance mechanism which is defined as a part of the meaning of `from` (line 1 of Listing 3.2). FFML allows the developer to express new artifacts and modification of artifacts. The other complete

modules of the bank account product line are shown in Annex.

3.4 Semantics

In this section, we present the semantics of FFML that is related tightly to the meanings of the keyword `from` mentioned in Section 3.3.3. We use the brackets $[\![\ ]\!]$ which is a function to display the semantics of an item, namely a module or an artifact.

$$[\![FM]\!] = \begin{cases} FM & \text{if } FM \text{ is the root} \\ (MN, PN, [\![S]\!], [\![P]\!], [\![R]\!], [\![D]\!], [\![Pf]\!]) & \text{if } FM \text{ is not the root} \end{cases} \quad (3.4)$$
$$where \;\; FM = (MN, PN, S, P, R, D, Pf)$$

The semantics of a module FM is denoted in Equation 3.4 as a tuple (MN, PN, $[\![S]\!]$, $[\![P]\!]$, $[\![R]\!]$, $[\![D]\!]$, $[\![Pf]\!]$), where MN is the name, PN is the parent name, S is a set of all the signatures, P is a set of all the properties, R is the representation type, D is a set of all the function definitions, Pf is a set of all the correctness proofs of the module. The parent module of FM is denoted by FP, a tuple (PN, PPN, S_{FP}, P_{FP}, R_{FP}, D_{FP}, Pf_{FP}) as defined by Equation 3.1 in Section 3.2.2, in which PPN is the parent name and S_{FP}, P_{FP}, R_{FP}, D_{FP}, Pf_{FP} are respectively all the signatures, all the properties, the representation type, all the function definitions and all the correctness proofs of the parent module. In the special case, if FM is the root module, $[\![FM]\!]$ is the module. We continue discussing in detail how the meanings of the module elements are calculated itself sequentially in the next sections.

3.4.1 Function Declaration

The semantics of S in Equation 3.4, the set of all the function declarations, $[\![S]\!]$ of a module FM is defined as follows.

$$[\![S]\!] = \begin{cases} S & \text{if } FM \text{ is the root} \\ [\![S_{FP}]\!]_{[PN \leftarrow MN]} \cup S & \text{if } FM \text{ is not the root} \end{cases} \quad (3.5)$$
where S_{FP} is the function declarations of the parent module FP.

If the module FM is the root, $[\![S]\!]$ is S itself, the new function definitions written into FM. By contrast (FM is not the root), according to the inheritance mechanism defined

in previous section, all the function declarations for the parent module FP, denoted by $[\![S_{FP}]\!]$, are entered into the module FM. The semantics $[\![S]\!]$ for FM, is a union of $[\![S_{FP}]\!]$ and S. The module type which is an abstract type, is renamed from PN to MN. This renaming is denoted by $[PN \leftarrow MN]$ in Equation 3.5 and later.

Listing 1 Semantics of function declarations

```
1 fmodule BA
2  signature update: BA -> int -> BA;
3  signature get_bal: BA -> int;
4  signature over: int;
5 ...
```

Listing 3.3 Signatures of module BA

```
1 fmodule LL from BA
2  signature limit_low: int;
3 ...
```

Listing 3.4 Signatures of module LL

Example. In Listing 1, we show the example of module LL which is defined from module BA. S here contains a new signature *limit_low* (Listing 3.4). Other functions *update*, *get_bal* and *over*, declared in BA (Listing 3.3), are the elements of $[\![S_{BA}]\!]$, still available in module LL thanks to the inheritance mechanism. $[\![S_{LL}]\!]$ includes all of the mentioned functions and theirs type are renamed. Hence,

$$[\![S_{LL}]\!] = \left\{ \begin{array}{ll} update : LL \rightarrow int \rightarrow LL; & get\_bal : LL \rightarrow int; \\ over : int; & limit\_low : int \end{array} \right\}$$

3.4.2 Property

As the kinds of properties are defined in Section 3.3.2, we denote the set of invariant properties by iP, the set of new properties by nP and the set of refining properties by rP. These three sets form a partition of P, the set of properties appearing in module FM. The set of all the properties, $[\![P]\!]$ for the module FM mentioned in Equation 3.4, is defined as follows:

$$[\![P]\!] = \begin{cases} P & \text{if } FM \text{ is the root} \\ ([\![P_{FP}]\!] \setminus \mathcal{P}([\![P_{FP}]\!], rP))_{[PN \leftarrow MN]} \cup P & \text{if } FM \text{ is not the root} \end{cases} \quad (3.6)$$

where P_{FP} are the properties of the parent module FP;

and $\mathcal{P}$ is the function returning the properties in $[\![P_{FP}]\!]$ that are refined by rP.

If FM is the root then $[\![P]\!]$ is P itself, in which P are the properties added or refined/modified from the parent into the module. When FM is not the root, all the properties $[\![P]\!]$ are calculated by implementing a mechanism which is more complicated than the one for function declarations. These properties include P and other properties which are calculated from $[\![P_{FP}]\!]$ by eliminating the properties refined by rP. P_{FP} is the set of the all parent properties defined as $P_{FP} = iP_{FP} \cup nP_{FP} \cup rP_{FP}$, where iP_{FP}, nP_{FP} and rP_{FP} are respectively invariant, new and refining properties of FP. A part of $([\![P_{FP}]\!] \setminus \mathcal{P}([\![P_{FP}]\!], rP))_{[PN \leftarrow MN]}$ (Equation 3.6) are all the invariant properties $[\![iP_{FP}]\!]$ inherited from FP. The remaining part of $([\![P_{FP}]\!] \setminus \mathcal{P}([\![P_{FP}]\!], rP))_{[PN \leftarrow MN]}$ are all the properties of FP kept into FM, denoted by $[\![kP]\!]$. The types in the properties have to be renamed from PN (of FP) to MN (of FM). We will discuss how we calculate these parts in the next sections.

3.4.2.1 Invariant Property

$$[\![iP]\!] = \begin{cases} iP & \text{if } FM \text{ is the root} \\ [\![iP_{FP}]\!]_{[PN \leftarrow MN]} \cup iP & \text{if } FM \text{ is not the root} \end{cases} \quad (3.7)$$

where iP_{FP} is the invariant properties of FP

Similarly to the function declarations, using the inheritance mechanism, we represent how to calculate all the invariant properties for FM (present in Equation 3.6) by Equation 3.7. If FM is the root, $[\![iP]\!]$ is iP. If FM is not the root, all the invariant properties $[\![iP]\!]$ for FM includes iP and all the invariant properties $[\![iP_{FP}]\!]$, in which the module type is renamed by MN. Using the inheritance mechanism, the invariant properties are entered into FM. For example, in the bank account product line the invariant property *ba_bal_gr_over* of the module BA (line 6 of Listing 3.1) is still available in the module DL (Listing 3.2). Thus,

$$[\![iP_{DL}]\!] = \{\, ba\_bal\_gr\_over : all\ x : DL,\ get\_bal(x) >= over;\}$$

3.4.2.2 Kept Property

The properties of FP which are neither invariant nor refined in FM (present in Equation 3.6) are kept and represented in Equation 3.8. If FM is the root, $[\![kP]\!]$ is empty set. If FM is not the root, $[\![kP]\!]$ is calculated by $([\![P_{PN}]\!] \setminus \mathcal{P}([\![P_{FP}]\!], rP))$ removing $[\![iP_{FP}]\!]$. The abstract type present in these properties must be renamed by MN.

$$[\![kP]\!] = \begin{cases} \emptyset & \text{if } FM \text{ is the root} \\ \Big(([\![P_{FP}]\!] \setminus \mathcal{P}([\![P_{FP}]\!], rP)) \setminus [\![iP_{FP}]\!]\Big)_{[PN \leftarrow MN]} & \text{if } FM \text{ is not the root} \end{cases}$$

where P_{FP} are the properties of the parent module FP;

iP_{FP} is the invariant properties of FP;

and $\mathcal{P}$ is the function returning the properties in $[\![P_{FP}]\!]$ that are refined by rP. (3.8)

For example, there are two properties *ba_upd_succ_with_over* and *ba_upd_nosucc_with_over* in module BA specified for the function *udpate*, presented in Listing 3.5. The first one is refined in module DL while the second one is not (Listing 3.5). This second one is kept and still available in DL.

Listing 2 Example of keeping a property from module BA

```
1 //fmodule BA
2 contract update:: property
      ba_upd_suc_with_over: all x : BA, all a
      : int, get_bal(x) + a) >= over ->
      get_bal (update(x,a)) = get_bal(x) + a;
3
4 contract update:: property
      ba_upd_nosucc_with_over:
5  all x : BA, all a : int, (get_bal(x) + a) <
       over -> get_bal (update(x,a)) = get_bal
       (x);
```

Listing 3.5 Module BA

```
1 //fmodule DL from BA
2 property dl_upd_succ_with_wlimit_R1 refines
      BA!ba_upd_succ_with_over
3  extends premise ((a <= 0) /\ (get_with(x) + a
       >= limit_with)) \/ (a > 0);
```

Listing 3.6 Module DL

3.4.3 Representation Type

$$[\![R]\!] = \begin{cases} R & \text{if } FM \text{ is the root} \\ [\![R_{FP}]\!] * R & \text{if } FM \text{ is not the root} \end{cases}$$

where R_{FP} is the representation type of the parent module FP;

and $*$ is the Cartesian product. (3.9)

Like in FoCaLiZe, in FFML the representation type of a module concretes the abstract date type of the specification into an explicit type. We calculate the complete representation type of FM (mentioned in Equation 3.4) by Equation 3.9. In FFML, the representation of a module is a Cartesian product of concrete types. We consider the representation type as a record of value-typed elements ordered from left to right. If FM is the root, $[\![R]\!]$ is R. If FM is not the root, $[\![R]\!]$ is the Cartesian product of $[\![R_{FP}]\!]$ and S.

Listing 3 Example of extending a representation type

```
1 //fmodule BA
2 representation = int;
```

Listing 3.7 Module BA

```
1 //fmodule DL from BA
2 representation extends BA with int;
```

Listing 3.8 Module DL

An example of extending the representation type of the module BA is described in Listing 3. The representation type of the module DL is demonstrated with BA (the parent name) and a new part *int* (an extension part), introduced in Listing 3.8. The representation type of module DL is the Cartesian product *int* * *int* which is calculated by *int* of BA (Listing 3.7) with the new part.

3.4.4 Function Definition

A function which is declared in FP can be redefined in FM with a new definition or a redefinition. We call all these function implementations rD. We denote the new functions which are declared and defined in FM by nD the set of definitions of these new functions. The sets rD and nD form a partition of D, the set of definitions appearing in FM. As mentioned in Equation 3.4, the set of all the function (re)definitions, $[\![D]\!]$ for the module FM, is represented as follows.

$$[\![D]\!] = \begin{cases} D & \text{if } FM \text{ is the root} \\ ([\![D_{FP}]\!] \setminus \mathcal{D}([\![D_{FP}]\!], rD))_{[PN \leftarrow MN]} \cup D & \text{if } FM \text{ is not the root} \end{cases} \tag{3.10}$$

where D_{FP} are the (re)definitions of the parent module FP;

$\mathcal{D}$ is the function returning the definitions in $[\![D_{FP}]\!]$ that are redefined by rD;

If FM is the root then $[\![D]\!]$ is D, where D is the (re)definitions of FM. If FM is not the root, all the (re)definitions for FM include D and others which are calcu-

lated by $[\![D_{FP}]\!]$ eliminating the ones replaced by rD. The implementations $([\![D_{FP}]\!] \setminus \mathcal{D}([\![D_{FP}]\!], rD))_{[PN \leftarrow MN]}$ of FP are kept in FM. $[PN \leftarrow MN]$ denotes that the type name PN is replaced by the type name MN.

Function Redefinition

We assume that a function f is defined in FP. Then, the function is redefined in FM with a redefinition $redef$. For simplicity we assume that this redefinition refers to the implementation of f at most once. We can then model a redefinition with the help of a term context[1] as follows:

$$redef := \mathbb{C}[FP!f(args)] \quad \text{where } \mathbb{C} \text{ is the context needed for calling } f \text{ in } FM \text{ with the arguments } args; \tag{3.11}$$

Listing 4 Example of redefining a function

```
1 //fmodule BA
2 signature update: BA -> int -> BA;
3 representation = int;
4 let update (x, a) =
5  if ((get_bal(x) + a) >= over) then get_bal(x)
       + a
6  else get_bal(x);
```

Listing 3.9 Module BA

```
1 //fmodule LL from BA
2 representation = BA;
3 let update (x, a) =
4  if ((a >= 0) || (a <= limit_low)) then
       BA!update (x, a)
5  else x;
```

Listing 3.10 Module LL

As mentioned earlier, after defining the representation type, the functions of FM must be defined before the module becomes complete. We give an example of a function *update* in Listing 4. The function *update* is declared (line 2 of Listing 3.9 and defined in the module BA (lines 4-6), and then redefined in the module LL (lines 3-5 of Listing 3.10). The signature of the function *update* is inherited from BA and available in LL as $update : LL \rightarrow int \rightarrow LL$;. While implementing, the function from BA is called in lines 4 of Listing 3.10.

Kept Function Definition

The function implementations from the parent module that are not redefined in FM are kept into the current module using the reuse mechanism, denoted by the part $([\![D_{PN}]\!]$

[1] A context is an expression with a hole where another expression, here a call to a parent function, can be plugged in. The notation $\mathbb{C}[t]$ means that t has been plugged in the hole of the context $\mathbb{C}$.

$\setminus\ \mathcal{D}([\![D_{PN}]\!], rD))_{[PN \leftarrow MN]}$ in Equation 3.10. FFML supports the reuse mechanism, that is, FM uses the same implementation as FP. Listing 5 is an example of two functions *update* and *get_bal* declared in BA (3.11). *update* is redefined in DL while *get_bal* is not. The function *get_bal* is still available in DL and its implementation is kept from BA.

Listing 5 Example of keeping a function definition

```
1 //fmodule BA
2 signature update: BA -> int -> BA;
3 signature get_bal: BA -> int;
4 let get_bal(x) = x;
5 let update (x, a) =
6  if ((get_bal(x) + a) >= over) then get_bal(x)
       + a
7  else get_bal(x);
```

Listing 3.11 Module BA

```
1 //fmodule DL from BA
2 let update (x, a) =
3  if (a <= 0) then
4   if (get_with(x) + a >= limit_with) then
5    (BA!update (x,a), get_with(x) + a)
6   else x
7  else (BA!update(x,a), get_with(x));
```

Listing 3.12 Module DL

3.4.5 Correctness Proof

For each property, the developer must write a proof (a proof may include several sub-proofs as in Listing 3.14) corresponding to it. Refining properties are proved by corresponding correctness proofs that are denoted by rPf. The proofs Pf of FM is the set rPf and new proofs nPf (the proofs for new properties nP in FM). As mentioned in Equation 3.4, the set of all the proofs, $[\![Pf]\!]$ for the module FM, is represented as follows:

$$[\![Pf]\!] = \begin{cases} Pf & \text{if } FM \text{ is the root} \\ ([\![Pf_{FP}]\!] \setminus \mathcal{P}f([\![Pf_{FP}]\!], rPf))_{[PN \leftarrow MN]} \cup Pf & \text{if } FM \text{ is not the root} \end{cases} \tag{3.12}$$

where Pf_{FP} are the proofs of the parent module FP;

and $\mathcal{P}f$ is the function returning the proofs in $[\![Pf_{FP}]\!]$ that are reproved by rPf.

If FM is the root then $[\![Pf]\!]$ is Pf, where Pf is the proofs of FM. If FM is not the root, $[\![Pf]\!]$ includes Pf and others which are calculated by $[\![Pf_{FP}]\!]$ eliminating the ones reproved by rPf. The proofs $([\![Pf_{FP}]\!] \setminus \mathcal{P}f([\![Pf_{FP}]\!], rPf))_{[PN \leftarrow MN]}$ of FP are kept in FM and the concrete module type is renamed automatically from PN to MN, denoted by $_{[PN \leftarrow MN]}$.

An example of how the proofs of BA are kept in LL is shown in Listing 6. The property *ba_upd_succ_with_over* of BA (Listing 3.13) is refined by another property

Listing 6 Example of kept proofs

```
1  //fmodule BA
2   contract update :: property
        ba_upd_succ_with_zero: all x : BA, all a
         : int,
3     (a >= 0) -> get_bal (update(x,a)) = get_bal(
        x) + a;
4
5  contract update ::  property
       ba_upd_succ_with_over : all x:BA, all a:
       int, (get_bal(x) + a) >= over -> get_bal(
       update(x,a)) = get_bal(x) + a;
6
7  proof of ba_upd_succ_with_zero =
8  foc proof {*
9   <1>1 assume x : Self, a : int,
10  prove (a >= 0) -> get_bal (update(x,a)) =
        get_bal(x) + a
11   <2>1 prove (a >= 0) -> get_bal(x) + a >=
         over + 0
12   by property int_ge_plus_plus, ba_bal_gr_over
13   <2>3 prove (a >= 0) -> get_bal(x) + a >=
         over
14   by step <2>1 property int_0_plus,
         int_plus_commute
15   <2>4 qed by step <2>3 property
         ba_upd_succ_with_over
16  <1>e conclude;*}
17
18 proof of ba_upd_succ_with_over = foc proof {*
       by definition of update, get_bal;*}
```

Listing 3.13 Module *BA*

```
1  //fmodule LL from BA
2  contract update :: property
       ll_upd_succ_with_llimit_R1
3   refines BA!ba_upd_succ_with_over
4   extends premise
5        ((a >= 0) || (a <= limit_low));
6
7  proof of ll_upd_succ_with_llimit_R1 =
8  foc proof
9  {* by definition of update, get_bal, over
       property BA!ba_upd_succ_with_over;}
```

Listing 3.14 Module *LL*

ll_upd_succ_with_llimit_R1 (Listing 3.14). The proof of the refining property is written in line 7. Another property *ba_upd_succ_with_zero* of *BA* is not refined but still available in *LL*. Its proof is kept.

Relation between a Refining Property and its Correctness Proof

As mentioned in Section 3.3.2.2, a refining property is defined in Equation 3.3 as $p'_{fj}(\overrightarrow{x'_j}) := \forall \overrightarrow{x'_j} : U,\ prem'_{fj}(\overrightarrow{x'_j}) \rightarrow p_{fi}(\overrightarrow{x_i})$, where $\overrightarrow{x'_j}$ is an extension of $\overrightarrow{x_i}$. By adding a new premise $prem'_{fj}$, the constraint p'_{fj} becomes more restrictive than p_{fi}. The property p_{fi} can be used as a proof hint in the proof of p'_{fj}. In other words, in such a case the proof of p_{fi} is reused when proving the property p'_{fj}.

The relationship between a refining property and the property refined by it while proving, is demonstrated in Listing 7. Property *ll_upd_succ_with_llimit_R1* refines property *ba_upd_succ_with_over* of *BA* (line 3 of Listing 3.16) by adding a new premise (line 4). Its correctness proof mentions *ba_upd_succ_with_over* as a property proof hint (line 9).

Listing 7 Example of relation between refining and refined property

```
1 //fmodule BA
2 contract update :: property
      ba_upd_succ_with_over: all x : BA, all a
      : int,
3   (get_bal(x) + a) >= over -> get_bal (update(
      x,a)) = get_bal(x) + a;
4
5  proof of ba_upd_succ_with_over = foc proof {*
      by definition of update, get_bal;*}
```

Listing 3.15 Module *BA*

```
1 //fmodule LL from BA
2 contract update :: property
      ll_upd_succ_with_llimit_R1
3  refines BA!ba_upd_succ_with_over
4  extends premise
5      ((a >= 0) || (a <= limit_low));
6
7 proof of ll_upd_succ_with_llimit_R1 =
8 foc proof
9 {* by definition of update, get bal, over
      property BA!ba_upd_succ_with_over;*}
```

Listing 3.16 Module *LL*

3.5 FFML Compiler into FoCaLiZe

To translate a module file *.fm* to FoCaLiZe we need a set of transformation functions implementing the translation rules that convert the abstract syntax tree (AST) of FFML into that of FoCaLiZe. The **FFML Compiler** receives a FFML file *.fm* as its input and compiles the file into a FoCaLiZe file *.fcl*.

In this section, we begin by presenting our notations to represent the translation rules of FFML. We present the abstract grammars of both FFML and FoCaLiZe by types in Section 3.5.2. Using these abstract grammars, we explain our translation functions implemented in the **FFML Compiler**. A module translation function translating an FFML module into FoCaLiZe is introduced in the Section 3.5.3. And then, we describe how the FFML Compiler translates signatures, properties, representation type, function (re)definitions and correctness proofs sequentially in next sub-sections.

3.5.1 Notation

In Table 3.1, we introduce notations, used in the translation rules of FFML Compiler. The left column describes the notations and the right column gives their corresponding informal meaning.

Notation	**Meaning**
$string$	- basic type $string$
"x"	- string x
$^{\wedge} : string \rightarrow string \rightarrow string$	- infix function that concatenates two strings.
vfm, $vparent$, $vprem$, $vconc$ $vprop\_n$ ...	- variable names
Tr_fm, Tr_props, Get_iprops, $Join\_ca$ ...	- function names
T, V, FM, $PARENT$, ...	- FFML types
$vx : T$	- variable vx having type T
Case $v : e_1 :: e_2$	- implements e_2 when v matches e_1.
SP, $SSIG$, $SPROP$, $SREP$, ...	- FoCaLiZe types
T_{op}	- $T_{op} = None \mid T$ - is optional type.
$T\ list$	- type of list of elements of type T
$vlist : T\ list$	- $vlist = [va_1; ...; va_i; ...; va_n]$ - list of elements having type T, where $va_i : T$ and $i \in \{1, ..., n\}$
$vlist[i]$	- $vlist[i]$ - the ith element of $vlist$.
$vx \in vlist$	- vx is one element of $vlist$
$@ : T\ list \rightarrow T\ list \rightarrow T\ list$	- infix function that concatenates two lists of type T
$Map : (T \rightarrow V) \rightarrow T\ list \rightarrow V\ list$	- $Map\ F\ vlist = [F\ va_1; ...; F\ va_i; ...; F\ va_n]$ - mapping function applying function F on each element of and $vlist$.
$T = T_1 * T_2 * T_3$	$vt = (vt1, vt2, vt3)$ - Cartesian product vt where $vt : T$, $vt_1 : T_1$, $vt_2 : T_2$ and $vt_3 : T_3$

Table 3.1: Notations used in the translation rules implemented in FFML Compiler

3.5.2 Abstract Syntax of FFML and FoCaLiZe

Before analyzing FFML Compiler, we describe how the concrete syntax of FFML and FoCaLiZe are abstracted into the corresponding abstract grammars. The types and construction of these abstract grammars will be used later to define the translation functions of FFML Compiler.

Table 3.2 describes the relationship between the FFML concrete syntax (presented in Grammar 3.1 in Section 3.3.1) and its corresponding abstract syntax. The types in the first column of the table are presented by the concrete constructions detailed in the second column. The definitions of these types are given in the third column. For example, the first line of the table describes that the type FM, which is the type of an FFML module in the abstract syntax grammar, is defined in the third column as a Cartesian product. The corresponding concrete syntax is given in the second column.

Similarly, we list the abstract syntax of FoCaLiZe in Table 3.3 which is related to the FoCaLiZe grammar (Grammar A.1) detailed in Annex. Each type defined in the FoCaLiZe abstract syntax grammar starts with a letter "S".

Type	**Concrete Syntax**	**Type definition**
FM	⟨*fm*⟩ ::= '`f_module`' ⟨*id*⟩ ['`from`' (⟨*parent*⟩)*,] (⟨*sig*⟩)* (⟨*prop*⟩)* ⟨*rep*⟩ (⟨*def*⟩)* (⟨*proof*⟩)* '`;;`'	$ID * ID\ list * SIG\ list * PROP\ list * REP * DEF\ list * PROOF\ list$
SIG	⟨*sig*⟩ ::= '`signature`' ⟨*func_n*⟩ '`:`' ⟨*type*⟩ '`;`'	$FUNC\_N * TYPE$
$PROP$	⟨*prop*⟩ ::= ⟨*nprop*⟩ | ⟨*iprop*⟩ | ⟨*rprop*⟩	$NPROP \mid IPROP \mid RPROP$
$NPROP$	⟨*new_prop*⟩ ::= '`contract`' ⟨*func_n*⟩ '`property`' ⟨*prop_n*⟩ '`:`' (⟨*lvar*⟩)*, [⟨*lexpr*⟩ '`->] <lexpr> ';`'	$FUNC\_N * PROP\_N * LVAR\ list * LEXPR_{op} * LEXPR$
$IPROP$	⟨*iprop*⟩ ::= '`contract`' ⟨*func_n*⟩ '`invariant property`' ⟨*prop_n*⟩ '`:`' (⟨*lvar*⟩)*, [⟨*lexpr*⟩ '`->`'] ⟨*lexpr*⟩ '`.`'	$FUNC\_N * PROP\_N * LVAR\ list * LEXPR_{op} * LEXPR$
$RPROP$	⟨*rprop*⟩ ::= '`contract`' ⟨*func_n*⟩ '`property`' ⟨*prop_n*⟩ '`refines`' ⟨*parent*⟩'`!`'⟨*prop_n*⟩ ['`extends premise`' ⟨*lexpr*⟩] '`;`'	$FUNC\_N * PROP\_N * ID * PROP\_N * LEXPR_{op}$
REP	⟨*rep*⟩ ::= ⟨*bas_rep*⟩ | ⟨*ext_rep*⟩	$BAS\_REP \mid EXT\_REP$
BAS_REP	⟨*bas_rep*⟩ ::= '`representation =`' ⟨*c_prod*⟩ '`;`'	C_PROD
EXT_REP	⟨*ext_rep*⟩ ::= '`representation extends`' ⟨*parent*⟩ '`with`' ⟨*c_prod*⟩ '`;`'	$ID * C\_PROD$
DEF	⟨*def*⟩ ::= '`let`' ⟨*func_n*⟩ ['`(`' (⟨*par*⟩)*, '`)`'] ⟨*expr*⟩ '`;`'	$FUNC\_N * PAR\ list * EXPR$
$PROOF$	⟨*proof*⟩ ::= '`proof of`' ⟨*prop_n*⟩ '`=`' '`foc proof`' ⟨*focproofbody*⟩ '`;`'	$PROP\_N * FOCPROOFBODY$
$FOCPROOFBODY$	⟨*focproofbody*⟩	*string* - FoCaLiZe proof
ID	⟨*id*⟩ (or ⟨*parent*⟩	*string* - module name
$LVAR$	⟨*lvar*⟩	variable with universal quantifier
$TYPE$	⟨*type*⟩ ::= ⟨*basicT*⟩ | ⟨*funcT*⟩ | ⟨*prodT*⟩	$BASICT \mid FUNCT \mid PRODT$
$FUNCT$	⟨*funcT*⟩ ::= ⟨*type*⟩ '`->`' ⟨*type*⟩	- function type
$PRODT$	⟨*prodT*⟩ ::= ⟨*type*⟩ '`*`' ⟨*type*⟩	- Cartesian product of types
$BASICT$	⟨*basicT*⟩	- basic type such as *int*, *string*, etc.
$LEXPR$	⟨*lexpr*⟩	FFML predicate logical formula
$EXPR$	⟨*expr*⟩	FFML expression
C_PROD	⟨*c_prod*⟩	Cartesian product of concrete types
$FUNC\_N$	⟨*func_n*⟩	*string* - function name
$PROP\_N$	⟨*proof_n*⟩	*string* - property name
PAR	⟨*par*⟩	string - parameter name of function

Table 3.2: FFML abstract syntax

Name	**Concrete Syntax**	**Type Definition**
SP	$\langle sp \rangle$ ::= '`species`' $\langle sid \rangle$ ['`(`' ($\langle sparam\_def \rangle$)*, '`)`'] '`=`' ['`inherit`' ($\langle sinh\_def \rangle$)*, '`;`'] ($\langle ssig \rangle$)* ($\langle sprop \rangle$)* [$\langle srep \rangle$] ($\langle sdef \rangle$)* ($\langle sproof \rangle$)* '`end;;`'	SID * $SPARAM\_DEF\ list$ * $SINH\_DEF\ list$ * $SSIG\ list$ * $SPROP\ list$ * $SREP_{op}$ * $SDEF\ list$ * $SPROOF\ list$
$SPARAM\_DEF$	$\langle sparam\_def \rangle$ ::= $\langle sparam\_n \rangle$ '`is`' $\langle sid \rangle$ ['`(`' ($\langle sparam\_n \rangle$)*, '`)`']	$SPARAM\_N$ * SID * $SPARAM\_N\ list$
$SINH\_DEF$	$\langle sinh\_def \rangle$ ::= $\langle sid \rangle$ ['`(`' ($\langle sparam\_n \rangle$)*, '`)`']	SID * $SPARAM\_N\ list$
$SSIG$	$\langle ssig \rangle$ ::= '`signature`' $\langle sfunc\_n \rangle$ '`:`' $\langle stype \rangle$ '`;`'	$SFUNC\_N$ * $STYPE$
$SPROP$	$\langle sprop \rangle$ ::= '`property`' $\langle sprop\_n \rangle$ '`:`' $\langle slexpr \rangle$ '`;`'	$SPROP\_N$ * $SLEXPR$
$SREP_{option}$	$\langle srep \rangle$ ::= '`representation =`' $\langle sc\_prod \rangle$ '`;`'	$None \mid SC\_PROD$
$SDEF$	$\langle sdef \rangle$ ::= '`let`' $\langle sfunc\_n \rangle$ ['`(`' ($\langle spar \rangle$)*, '`)`'] $\langle sexpr \rangle$ '`;`'	$SFUNC\_N$ * $SPAR\ list$ * $SEXPR$
$SPROOF$	$\langle sproof \rangle$ ::= '`proof of`' $\langle sproof\_n \rangle$ '`=`' $\langle sproofbody \rangle$ '`;`'	$SPROOF\_N$ * $SPROOFBODY$
$SPROOFBODY$	$\langle sproofbody \rangle$	FoCaLiZe proof body
SID	$\langle sid \rangle$	*string* - module name
$STYPE$	$\langle stype \rangle$	FoCaLiZe species type
$SLEXPR$	$\langle slexpr \rangle$	FoCaLiZe predicate logical formula
$SEXPR$	$\langle sexpr \rangle$	FoCaLiZe expression
SC_PROD	$\langle sc\_prop \rangle$	Cartesian product of FoCaLiZe concrete types
$SPARAM\_N$	$\langle sparam\_n \rangle$	*string* - parameter name of species
$SFUNC\_N$	$\langle sfunc\_n \rangle$	*string* - function name
$SPROP\_N$	$\langle sprop\_n \rangle$	*string* - property name
$SPAR$	$\langle spar \rangle$	*string* - parameter name of function
$SPROOF\_N$	$\langle sproof\_n \rangle$	*string* - proof name

Table 3.3: FoCaLiZe abstract syntax

3.5.3 A Module Translation Function

FFML Compiler first analyzes the module file *.fm* and converts it into an AST of type FM (see Table 3.2). A module translation function $tran\_fm$ of FFML Compiler receives the abstract syntax tree of type FM as its input and returns three FoCaLiZe species of type SP and one collection of type COL. This function is defined in Table 3.4. The first column of the table is the function name and its type is described in the second column. The third column is the definition of the function. $SPEC$ appearing in the second column is the type of the FFML modules defining the considered SPL. An argument $vspec$ of type $SPEC$ (in the third column) is used to retrieve the artifacts of the modules of the SPL.

Function name	Type	Definition
Tr_fm	$SPEC * FM \rightarrow SP * SP * SP$	$Tr\_fm\ (vspec, vfm) = (Tr\_sp_1\ vfm, Tr\_sp_2\ (vspec, vfm), Tr\_sp_3\ (vspec, vfm), Tr\_col\ vfm)$

Table 3.4: Module translation function

Listing 9 Example of module transformation

```
1 fmodule BA
2 ... ;;
```

Listing 3.17 Module BA in FFML

```
1  species BA_spec1
2  ... end;;
3  species BA_spec2 =
4   inherit BA_spec1;
5  ... end;;
6  species BA_imp =
7   inherit BA_spec2;
8  ... end;;
9  collection BA_col =
10  implement BA_imp;
11 end;;
```

Listing 3.19 Module BA translated into FoCaLiZe

```
1 fmodule DL from BA
2 ... ;;
```

Listing 3.18 Module DL in FFML

```
1  species DL_spec1(BA is BA_imp) =
2   inherit BA_spec1;
3  ... end;;
4  species DL_spec2(BA is BA_imp) =
5   inherit DL_spec1(BA);
6  ... end;;
7  species DL_imp(BA is BA_imp) =
8   inherit DL_spec2(BA);
9  ... end;;
10 collection DL_col =
11  implement DL_imp(BA_col);
12 end;;
```

Listing 3.20 Module DL translated into FoCaLiZe

Function Tr_fm translates module vfm into three different species and a collection. It uses three functions Tr_sp_1, Tr_sp_2 and Tr_sp_3, each of which converts sequentially

the artifacts of the module into these species. Tr_fm also uses other function Tr_col, establishing a collection in FoCaLiZe, which is built from a complete species (Section 1.2.2 of FoCaLiZe). The three separate species and the collection in FoCaLiZe are related by an inheritance hierarchy.

We explain the reasons why we divide the artifacts of a module into three separate species and a collection. In fact, their names are generated according to a strict convention that allows for an easy maintenance of the artifacts. The first species will contain the new signatures (function declarations) and invariant properties, that are abstract specifications and can be inherited. These specifications can not be modified in the children species, however, these species have to implement the declared functions and prove the properties. According to the inheritance mechanism of FoCaLiZe, the second species inherits the first species in order to reuse the specification. The new properties or refining properties are transformed into the second species, so they can be reused, refined or modified. Notice that FoCaLiZe does not contain any mechanism allowing refining or modifying properties. These properties are copied each time we use them for keeping or for defining new other properties. The third species inherits (in the FoCaLiZe sense) the second species to obtain all the artifacts from the second species. The code and correctness proof artifacts are transformed into the third species. The last translated unit is a collection that implements the third species which is a complete species (where every signature is accompanied with a definition and each property is proved).

A module can be constructed from its parent using the keyword `from` with a purpose of restricting an artifact repetition. Besides the relationships between the three species of a module, if the module is not the root one, its species are also related to the corresponding species of its parent. Via the inheritance relationships and the parametrization mechanism of FoCaLiZe, the artifacts of the parent can be reused, refined or modified. The first species of the module inherits the first species of the parent in order to obtain all the artifacts involved in the species of the module. Each species, obtained when translating a non root FFML module, receives as a parameter an implementation of its parent. Thus, each artifact of the child module may access to the functions and properties of the parent. The properties in the second species of the parent can be refined or modified by copying their

statements.

Listing 9 illustrates these translation scheme for two features BA and DL. FFML modules in the left are translated into the species in the right using the translation functions. Three species BA_spec1, BA_spec2, BA_imp and a collection BA_imp are produced by FFML Compiler while translating module BA. BA_imp inherits BA_spec2 while BA_spec2 inherits BA_spec1. BA_imp is implemented by BA_col which is used to call the artifacts of BA when executing. Similarly, module DL is also translated into three species DL_spec1, DL_spec2, DL_imp and a collection DL_imp. Because DL is constructed from BA, the first species DL_spec1 inherits BA_spec1 and BA_imp is mentioned by an object name BA as a parameter of the three species. The collection DL_col implements DL_imp with the collection BA_col as an effective parameter.

Before discussing in detail the functions appearing in Tr_fm, we also provide several supplementary functions that are necessary to get the values of an FFML module. They are listed in Tables 3.5 and 3.6.

3.5.4 Translating into the First Species

In this section we focus on explaining how the function Tr_sp_1 (Table 9) translates the module artifacts of an FFML module into the first species in FoCaLiZe. The function is presented in Table 3.7. The function also calls other functions to translate each kind of elements, i.e. Tr_id_1, $Tr\_parents$ and Tr_inh_1 for module header (presented in Section 3.5.4.1), Tr_sigs for function declarations (presented in Section 3.5.4.2) and Tr_iprops for invariant properties (presented in Section 3.5.4.3). In the next sections, we will define these mentioned functions.

3.5.4.1 Module and Parent Name

To convert an FFML module name and the name of its parent to the first species's name, parameter definitions and inheritance clauses in FoCaLiZe, we split into small cases, listed in Table 3.8. The function Tr_id_1 (row 2) translates the module name to the corresponding species's name which is built from the module name with "$\_spec1$" as a suffix. The function $Tr\_parents$ (row 3) translates the module's parent names to the parameter definitions of

Function name	**Type**	**Note**
Get_fm	$SPEC * ID \rightarrow FM$	- gets the GFML module from a module name.
Get_id	$FM \rightarrow ID$	- gets the GFML module name from a GFML module.
$Get\_parents$	$FM \rightarrow ID\ list$	- gets the list of GFML parent module names.
Get_sigs	$FM \rightarrow SIG\ list$	- gets the list of module signatures.
Get_snames	$SIG\ list \rightarrow FUNC\_N\ list$	- gets the list of function names.
Get_props	$FM \rightarrow PROP\ list$	- gets the list of module properties.
Get_iprops	$PROP\ list \rightarrow IPROP\ list$	- gets the list of the invariant properties from a list of module properties.
Get_nprops	$PROP\ list \rightarrow NPROP\ list$	- gets the list of the new properties from a list of module properties.
Get_rprops	$PROP\ list \rightarrow RPROP\ list$	- gets the list of the refining properties from a list of module properties.
Get_pname	$PROP \rightarrow PROP\_N$	- gets the name of a property.
Get_lvars	$PROP\_N * LVAR\ list * LEXPR_{op} * LEXPR \rightarrow LVAR\ list$	- gets the list of the property variables.
Get_prem	$PROP\_N * LVAR\ list * LEXPR_{op} * LEXPR \rightarrow LEXPR_{op}$	- gets the property premise from a new or invariant property.
Get_conc	$PROP\_N * LVAR\ list * LEXPR_{op} * LEXPR \rightarrow LEXPR$	- gets the property conclusion from a new or invariant property.
Get_rprop_pat	$RPROP \rightarrow ID$	- gets the parent module name from a refining property.
Get_rprop_ppname	$RPROP \rightarrow PROP\_N$	- gets the name of the property of parent module from a refining property.
Get_rnprem	$RPROP \rightarrow LEXPR_{op}$	- gets the new premise from a refining property.
Get_prop	$PROP\_N * FM \rightarrow PROP$	- finds the property through its name in a module.
Get_rep	$FM \rightarrow REP$	- gets the module representation.
Get_ext	$EXT\_REP \rightarrow C\_PROD$	- gets the extension part of an extended representation.
Get_funcs	$FM \rightarrow DEF\ list$	- gets the list of the function definitions.
Get_rfuncs	$FM \rightarrow DEF\ list$	- gets the list of the redefined functions.
Get_fname	$DEF \rightarrow FUNC\_N$	- gets the name of a function.
Get_ftype	$SPEC * FM * FUNC\_N \rightarrow TYPE$	- gets the type of a function.
Get_fexpr	$DEF \rightarrow EXPR$	- gets the expression of a function.
Get_fpars	$ID * FUNC\_N \rightarrow PAR\ list$	- finds the parameters of the function of a module.
Get_proofs	$FM \rightarrow PROOF\ list$	- gets the list of the proofs.
Get_cprops	$SPEC * FM \rightarrow PROP\ list$	- gets all the properties available in a module.
Get_cfuncs	$SPEC * FM \rightarrow FUNC\ list$	- gets all the function definitions available in a module.
$Get\_cproofs$	$SPEC * FM \rightarrow PROOF\ list$	- gets all the proofs available in a module.

Table 3.5: Supplementary functions for getting module values (1)

Function	**Type**	**Definition**	**Note**
Get_kprops	*SPEC * FM → PROP list*	*Get_kprops* (*vspec*, *vfm*) = *Remove_prop* (*vcprops*$_P$, *vrpnames*$_P$) ;	- gets all the properties kept into a module from its parent, **where** *vparent* = *Get_parent vfm*; *vcprops* = *Get_cprops* (*Get_fm vspec vparent*); - all the properties of parent *vrprops* = (*Get_rprops* (*Get_props vfm*)); - refining properties. *vrpnames*$_P$ = *Map Get_rprop_ppname vrprops*; - the properties of parent which are refined *Remove_prop* - a function removing the properties *vrpnames*$_P$ in *vcprops*.
Get_kfuncs	*SPEC * FM → FUNC list*	*Get_kfuncs* (*vspec*, *vfm*) = *Remove_func* (*vcfuncs*$_P$, *vrfuncs*$_P$) ;	- gets all the function definitions kept into a module from its parent, **where** *vparent* = *Get_parent vfm*; *vcfuncs* = *Get_cfuncs* (*Get_fm vspec vparent*); - all the function definitions of parent *vrfuncs*$_P$ = (*Get_rprops* (*Get_props vfm*)); - function re-definitions. *Remove_func* - a function removing *vrfuncs*$_P$ in *vcfuncs*.
Get_kproofs	*SPEC * FM → PROOF list*	*Get_kproofs* (*vspec*, *vfm*) = *Remove_proof* (*vcproofs*$_P$, *vrproofs*$_P$) ;	- gets all the proofs kept into a module from its parent, **where** *vparent* = *Get_parent vfm*; *vcproofs* = *Get_cproofs* (*Get_fm vspec vparent*); - all the proofs of parent *vrproofs* = (*Get_rproofs* (*Get_proofs vfm*)); - reproved proof. *Remove_proof* - a function removing *vrproofs*$_P$ in *vcproofs*.

Table 3.6: Supplementary functions for getting module values (2)

Function	Type	Definition	Note
Tr_sp_1	$FM \rightarrow SP$	$Tr\_sp_1\ vfm =$ $(Tr\_id_1\ vfm,$ $Tr\_parents\ vfm,$ $Tr\_inhs_1\ vfm,$ $Tr\_sigs\ vfm,$ $Tr\_iprops\ \ vfm,$ $None, [\,], [\,])$	- is the first species translation function translating module vfm to **the first species** in FoCaLiZe which only contains signatures and invariant properties.

Table 3.7: Function translating an FFML module to the first FoCaLiZe species

type $SPARAM\_DEF$. Each of the parameter definitions contains *vparent* (parent name) as a parameter referring to a species which is the third species obtained from the parent. This third species has a name built from the parent name with string "$\_imp$". If the current module is the root one, the first species does not contain any parameter definition. The other function Tr_inhs_1 (row 6) translates adds an inheritance clause built from the parent name. The result established from this function indicates that the first species inherits the first species translated from the parent, namely its name is obtained by concatenating the parent name with string "$\_spec1$".

Function	Type	Definition	Note
Tr_id_1	$FM \rightarrow SID$	$Tr\_id_1\ \ vfm =$ $(Get\_id\ \ vfm)\ \ ^\wedge$ "$\_spec1$"	- translates the module name of module vfm to a species name ended with "$\_spec1$".
$Tr\_parents$	$FM \rightarrow$ $SPARAM\_DEF\ \ list$	$Tr\_parents\ \ vfm =$ $[Tr\_parent$ $(Get\_parent\ \ vfm)]$	- translates the parent names of module vfm to the species parameter definitions.
Tr_parent	$ID \rightarrow SPARAM\_N *$ $SID *$ $SPARAM\_N\ list$	$Tr\_parent\ vparent =$ $(vparent,$ $vparent\ ^\wedge$ "$\_imp$", $Tr\_param\ vparent)$	- translates a parent name to a species parameter definition.
Tr_param	$ID \rightarrow$ $SPARAM\_N\ list$	$Tr\_param\ vparent =$ $Get\_parents$ $(Get\_fm\ \ vparent)$	- translates the parent names of *vparent* to parameters names.
Tr_inhs_1	$FM \rightarrow$ $SINH\_DEF\ list$	$Tr\_inhs_1\ \ vfm =$ $[Tr\_inh\ \ (Get\_parent$ $vfm)]$	- translates the parent modules of module vfm to the species inheritance definitions.
Tr_inh	$ID \rightarrow SID *$ $SPARAM\_N\ list$	$Tr\_inh\ vparent =$ $(vparent\ ^\wedge$ "$\_spec1$", $Tr\_param\ vparent)$	- translates a parent name to a species inheritance clause.

Table 3.8: Functions translating the elements of an FFML module into the first FoCaLiZe species

Example. The translation of module DL which is described from parent BA is shown in Listing 10. The first species DL_spec2 of DL receives receives a parameter named BA

that has to conform to the interface of the third species BA_imp. It also inherits the first species BA_spec1 of BA.

Listing 10 Example of translating module and parent names into the first FoCaLiZe species

```
1 fmodule BA
2 ... ;;
```

Listing 3.21 BA in FFML

```
1 species BA_spec1 =
2 ... end;;
3 species BA_spec2
4 ... end;;
5 species BA_imp
6 ... end;;
7 collection BA_col
8 ... end;;
```

Listing 3.22 BA in FoCaLiZe

```
1 fmodule DL from BA
2 ... ;;
```

Listing 3.23 DailyLimit (DL) in FFML

```
1 species DL_spec1 (BA is BA_imp) =
2 inherit BA_spec1;
3 ... end;;
```

Listing 3.24 DailyLimit (DL) in FoCaLiZe

3.5.4.2 Function Declaration

The function Tr_sigs translates all signatures of a module into FoCaLiZe signatures in the first species, presented in Table 3.9. The difference between signatures in FoCaLiZe and in FFML is that FFML signature uses an abstract type which has the same name as the module name whereas FoCaLiZe uses *Self* in that case. The function Tr_stype, called in the function Tr_sig, carries out the transformation of the abstract type. In an example shown in Listing 11, there are three functions *update*, *get_bal* and *over* declared in module BA in lines 2-4 of Listing 3.25. They are moved into the first species BA_spec1 Listing 3.26. Type BA is converted to abstract type $Self$ in this species.

Function	Type	Definition	Note
Tr_sigs	$FM \rightarrow SSIG\ list$	$Tr\_sigs\ \ vfm = Map\ \ (Tr\_sig\ vname)\ \ (Get\_sigs\ \ vfm)$	- translates FFML signatures of module vfm into the species signatures, where $vname$ is the module name.
Tr_sig	$ID \rightarrow FUNC\_N * TYPE \rightarrow SFUNC\_N * STYPE$	$Tr\_sig\ \ vname\ (vfunc\_n, vtype) = (vfunc\_n, Tr\_stype\ vtype\ vname)$	- translates a FFML signature into a FoCaLiZe signature.

Table 3.9: Function translating signatures into signatures of the first FoCaLiZe species

Listing 11 Example of translating the signatures of BA into the signatures of the first FoCaLiZe species.

```
1 //fmodule BA
2  signature update: BA -> int -> BA;
3  signature get_bal: BA -> int;
4  signature over: int;
```

Listing 3.25 BA in FFML

```
1 species BA_spec1 =
2  signature update: Self -> int -> Self;
3  signature get_bal: Self -> int;
4  signature over: int;
5 end;;
```

Listing 3.26 BA in FoCaLiZe

3.5.4.3 Invariant Property

The translation of invariant properties is done similarly and presented in Table 3.10. Function Tr_inprop transforms an FFML property with its name, variables, premise and conclusion to the corresponding property in FoCaLiZe. This function contains the function Tr_ptype translating the abstract type (as the module name $vname$) to $Self$ of FoCaLiZe. For example, invariant property $ba\_get\_bal\_gr\_over$ of module BA in Listing 3.27 is translated and added into the first species Inh_BA in Listing 3.28. The same conversion of types, BA into *Self*, is done here.

Function	Type	Definition	Note
Tr_iprops	$FM \rightarrow SPROP\ list$	$Tr\_iprops\ vfm = Map\ (Tr\_inprop\ vname)\ (Get\_iprops\ (Get\_props\ vfm))$	- translates the invariant properties of module vfm to the species properties, where $vname$ is the module name.
Tr_inprop	$ID \rightarrow PROP\_N * LVAR\ list * LEXPR_{op} * LEXPR \rightarrow SPROP\_N * SLEXPR$	$Tr\_inprop\ vname\ (vprop\_n, vlvars, vprem, vconc) = Tr\_ptype\ inprop\ vname$; **Where**: $inprop = \big(vprop\_n, Tr\_lexpr\ (vlvars, (Join\_imply\ vprem\ vconc))\big)$	- translates an invariant property (or new property) to a FoCaLiZe property.
$Join\_imply$	$LEXPR_{op} * LEXPR \rightarrow LEXPR$	$Join\_imply\ (vprem, vconc)$	- joins a premise with a conclusion with a logical implication.
Tr_lexpr	$LVAR\ list * LEXPR \rightarrow SLEXPR$	$Tr\_lexpr\ (vlvars, vlexpr)$	- joins a FFML logical formula $vlexpr$ and logical variables $vlvars$, then translates them to FoCaLiZe logical formula.

Table 3.10: Function translating invariant properties to the first FoCaLiZe species.

Listing 12 Example of translating the invariant properties of BA to the first FoCaLiZe species.

```
1 //fmodule BA
2  contract get_bal :: invariant property
       ba_get_bal_gr_over: all x : BA, get_bal(
       x) >= over;
```

Listing 3.27 BA in FFML

```
1 species BA_spec1 =
2  property ba_get_bal_gr_over: all x : Self,
       get_bal(x) >= over;
3 end;;
```

Listing 3.28 BA in FoCaLiZe

3.5.5 Translating into the Second Species

In this subsection we focus on explaining how the function Tr_sp_2 (Table 9) translates some other module artifacts into the second species in FoCaLiZe. It is presented in Table 3.11. An argument $vspec$ of type $SPEC$ is used to retrieve all the properties that are kept into module vfm from its parent (via function Get_kprops which is mentioned in Table 3.6). The function also calls several intermediate translation functions, i.e. Tr_id_2, $Tr\_parents$ and Tr_inh_2 for translating module and parent names, Tr_props for translating refining and new properties.

Function	Type	Definition	Note
Tr_sp_2	$SPEC$ * $FM \rightarrow SP$	$Tr\_sp_2\ (vspec, vfm) =$ $\big(Tr\_id_2\ vfm,$ $Tr\_parents\ vfm,$ $Tr\_inh_2\ vfm, [\,], (Map$ $(Tr\_prop\ vspec\ vname)$ $(vnrprops\ @\ vkprops)),$ $None, [\,]\big)$	- is the second species translation function translating module vfm to **the second species** in FoCaLiZe which only contains new and refining properties, where $vname$ is the module name; $vnrprops = (Get\_nprops\ vfm)$ @ $(Get\_rprops\ vfm)$; $vkprops = Get\_kprops\ vspec\ vfm$

Table 3.11: Function translating a module into the second FoCaLiZe species

3.5.5.1 Module and Parent Names

To translate a FFML module name together with its parent name into the second species name, parameter definitions and inheritance clauses in FoCaLiZe, we split them into small cases. The function Tr_id_2 (row 1 of Table 3.12) translates the module name to the corresponding species name. The second species has a name obtained from the string "$\_spec2$" and the module name. The species has the same parameter definitions as those of the first species, computed by function $Tr\_parents$ which is the same one mentioned in row 2 of Table 3.8. The other function Tr_inh_2 (row 2 of Table 3.12) produces an

inheritance clause for the second species: the second species inherits the first one.

Function	Type	Definition	Note
Tr_id_2	$FM \rightarrow SID$	$Tr\_id_2\ vfm =$ $(Get\_id\ vfm)\ ^\wedge$ "$\_spec2$"	- translates the module name of module vfm to a species name ended with "$\_spec2$".
Tr_inhs_2	$FM \rightarrow$ $SINH\_DEF\ \ list$	$Tr\_inh_2\ \ vfm = [Tr\_inh$ $((Get\_id\ \ vfm)\ ^\wedge$ "$\_spec1$")]	- creates an inherited species having name ended with "$\_spec1$". (Tr_inh - see Table 3.8)

Table 3.12: Function translating the elements of a module into the second FoCaLiZe species

Example. The translation of module DL having a parent BA is shown in Listing 13. The second species DL_spec2 of module DL receives a parameter which has to be conformed to the third species of BA, BA_imp and named BA. It also inherits the first species DL_spec1, associated to DL.

Listing 13 Example of translating the module and parent names to the second FoCaLiZe species

```
1 fmodule DL from BA
2 ... ;;
```

Listing 3.29 DL in FFML

```
1 species DL_spec1 (BA is BA_imp) =
2   inherit BA_spec1;
3   ... end;;
4 species DL_spec2 (BA is BA_imp)
5   inherit DL_spec1 (BA);
6   ... end;;
```

Listing 3.30 DL in FoCaLiZe

3.5.5.2 Refining and New Properties

The function Tr_prop (appeared Table 3.11) translates a refining or new property of a module into the second species, presented in the first row of Table 3.13. The translation of a new property is done similarly to the translation of an invariant property, by the same function Tr_inprop (in Table 3.10). For a refining property, FFML Compiler provides another function Tr_rprop to calculate its statement and transform into FoCaLiZe. The details of Tr_rprop are presented in Table 3.14.

Example. There are two properties in module DL shown in Listing 3.32. The first one is $dl\_upd\_succ\_with\_wlimit\_R1$ refined from property $ba\_upd\_succ\_with\_over$ from module BA (Listing 3.31) by adding a new premise for it, shown in line 5. The second one is $dl\_upd\_succ\_gr\_wlimit\_R1$, new property in module DL, shown in lines 7-8. The refining property is obtained by combining the statement of property $ba\_upd\_succ\_with\_over$

Function	Type	Definition	Note
Tr_prop	$SPEC \rightarrow ID \rightarrow PROP \rightarrow SPROP$	$Tr\_prop\ vspec\ vname\ vprop =$ **Case** $vprop : NPROP :: Tr\_inprop\ vname\ vprop$; **Case** $vprop : RPROP :: Tr\_rprop\ (vspec, vprop)$	- translates a FFML property (new or refining) to a FoCaLiZe property.

Table 3.13: Translating a property into the second FoCaLiZe species

Function	Type	Definition	Note
Tr_rprop	$SPEC * (PROP\_N * ID * PROP\_N * LEXPR_{op}) \rightarrow SPROP\_N * SLEXPR$	$Tr\_rprop\ (vspec, (vprop\_n, vparent, vprop\_n_P, vnprem)) = (vprop\_n, Tr\_lexpr\ (Get\_pat\_prop\ (vspec, vparent, vprop\_n_P, vnprem)))$	- translates a refining property to a FoCaLiZe property.
Get_pat_prop	$SPEC * ID * PROP\_N * LEXPR_{op} \rightarrow LVAR$ list $* LEXPR$	$Get\_pat\_prop\ (vspec, vparent, vprop\_n_P, vnprem) =$ **Case** $vprop_P : NPROP$ **::** $\big(Get\_lvars\ vprop, Join\_imply\ \big(Join\_and\ (vnprem, vprem_P), vconc_P\ \big)\big)$; **Case** $vprop_P : RPROP$ **::** $Get\_pat\_prop\ (vspec, vparent_1, vprop\_n_{P1}, Join\_and\ (vnprem, vnprem_1))$; **Where** $vfm_P = Get\_fm\ (vspec, vparent)$; $vprop_P = Get\_prop\ (vprop\_n_P, vfm_P)$;	- is a function getting the statement of a property, **Where** $vprem_P = Get\_prem\ vprop_P$; $vconc_P = Get\_conc\ vprop_P$; **and** $vparent_1 = Get\_rprop\_pat\ vprop_P$; $vprop\_n_{P1} = Get\_rprop\_ppname\ vprop_P$; $vnprem_1 = Get\_rprem\ vprop_P$;
$Join\_and$	$LEXPR_{op} * LEXPR_{op} \rightarrow LEXPR_{op}$	$Join\_and\ (vnprem, vprem)$	- joins two FFML logical formulas by a logical conjunction.

Table 3.14: Translating a refining property to the second FoCaLiZe species

with the new premise and translated into FoCaLiZe, presented in lines 5-8 of Listing 3.33. We can see here that we need to access to the artifacts of module BA to retrieve the statement of $BA!ba\_upd\_succ\_with\_over$, and thus produce the statement of the property $dl\_upd\_succ\_with\_wlimit\_R1$ in the species DL_spec2. The new property $dl\_upd\_succ\_gr\_wlimit$ is also moved to the second species DL_spec2 in line 10 of Listing 3.33.

Listing 14 Example of translating refining and new properties to the second FoCaLiZe species

```
1  //fmodule BA
2  contract update :: property
3   ba_upd_succ_with_over: all x : BA, all a :
        int, (get_bal(x) + a) >= over -> get_bal
        (update(x,a)) = get_bal(x) + a;
```

Listing 3.31 BA in FFML

```
1  //fmodule DL from BA
2  contract update :: property
3   dl_upd_succ_with_wlimit_R1:
4   refines BA!ba_upd_succ_with_over
5   extends premise
        ((a <= 0) /\ (get with)(x) + a
       >= limit_with)) \/ (a > 0);
6
7  contract update :: property
8   dl_upd_succ_gr_wlimit: all x : DL, all a :
        int, ((a <= 0) /\ (get_with(x) + a >=
        limit_with)) ->(get_with(update(x,a)) =
        get_with (x) + a;
```

Listing 3.32 DL in FFML

```
1  species DL_spec1 (BA is BA_imp) =
2  ... end;;
3  species DL_spec2 (BA is BA_imp)=
4   inherit DL spec1 (BA);
5   property dl_upd_succ_with_wlimit_R1: all x :
6     Self, all a : int,
7      (((a <= 0) /\ (get with(x) + a >=
           limit_with)) \/ (a > 0)) ->
8       ((get_bal(x) + a) >= over -> (get_bal(
            update(x,a))) = get_bal(x) + a);
9
10 property dl_upd_succ_gr_wlimit: all x : Self,
       all a : int, ((a <= 0) /\ (get_with(x) +
       a >= limit_with)) ->(get_with(update(x,a)
       ) = get_with (x) + a;
11 end;;
```

Listing 3.33 DL in FoCaLiZe

3.5.6 Translating into the Third Species

In this section, we focus on explaining how the function Tr_sp_3 (mentioned first in Table 3.4) translates the remaining artifacts of an FFML module (i.e., code and correctness proofs) into the third species in FoCaLiZe, presented in Table 3.15. The function also calls several transformation functions for each of the included elements, i.e. Tr_id_3, $Tr\_parents$ and Tr_inh_3 for the module and parent names, Tr_code for the representation type and function (re)definitions, Tr_proof for the correctness proofs. Again an argument $vspec$ of type $SPEC$ is used in Tr_code and Tr_proof to retrieve all the function (re)definitions and correctness proofs that are kept into module vfm from its parent. The function Tr_sp_3 is much more complex than the two other ones (Tr_sp_1 and Tr_sp_2) because it has to generate function (re)definitions and correctness proofs.

Function	Type	Definition	Note
Tr_sp_3	$SPEC * FM \rightarrow SP$	$Tr\_sp_3\ (vspec, vfm) =$ $(Tr\_id_3\ vfm,$ $Tr\_parents\ vfm,$ $Tr\_inh_3\ vfm, [\,], [\,], Tr\_code$ $(vspec, vfm), Tr\_proofs$ $(vspec, vfm))$;	- is the third species translation function translating module vfm to **the third species** in FoCaLiZe which contains representation, function definitions and correctness proofs.

Table 3.15: Function translating a module into the third FoCaLiZe species

3.5.6.1 Module and Parent Name

Similar to the first and second species, the functions translating a module name and its parent name into the third species are described in Table 3.16. The function Tr_id_3 is for translating the module name to the corresponding species name. The species has the same parameter definitions as the first species, presented by function $Tr\_parents$. The third species has a name built from the module name and the string "$\_imp$". The other function Tr_inhs_3 establishes an inheritance clause for the third species which explains that the third species inherits the second species associated to the module.

Function	Type	Definition	Note
Tr_id_3	$FM \rightarrow SID$	$Tr\_id_3\ vfm =$ $(Get\_id\ vfm)\ ^\wedge$ "$\_imp$"	- translates the module name of module vfm to a species name end with "$\_imp$"
Tr_inh_3	$FM \rightarrow$ $SINH\_DEF\ list$	$Tr\_inhs_3\ vfm = [Tr\_inh$ $(\ (Get\_id\ vfm)\ ^\wedge$ "$\_spec2$" $)]$	- creates an inherited species having the name ended with "$\_spec2$" (Tr_inh see Table 3.8)

Table 3.16: Function translating a module and its parent names into the third FoCaLiZe species

Example. The translation of module DL having a parent BA is shown in Listing 15. The third species DL_imp of the translation of DL has a parameter conform to the third species of BA. It also inherits the second species of DL_spec2.

Listing 15 Example of translating the module and parent names of a module to the third FoCaLiZe species

```
1 fmodule DL from BA
2 ... ;;
```

Listing 3.34 DL in FFML

```
1 species DL_spec1 (BA is BA_imp) =
2 ... end;;
3 species DL_spec2 (BA is BA_imp) =
4 inherit DL_spec1 (BA)
5 ... end;;
6 species DL_imp (BA is BA_imp)
7 inherit DL_spec2 (BA);
8 ... end;;
```

Listing 3.35 DL in FoCaLiZe

3.5.6.2 Implementation Code

The function Tr_code (appearing in Table 3.15) translates the implementation code of a module into the third species in FoCaLiZe. It distinguishes two cases according to whether the representation type of the module is either extended from the representation

type of the parent or not. The function is presented in Table 3.17.

Representation Type

In the case of extension, function Tr_erep translates the representation type to the corresponding one in FoCaLiZe using the representation type of the parent and the extended part vc_prod. The details of Tr_erep are in Table 3.18. In addition, the species is added several basic conversion functions, named $vbas\_funcs$,defined three supplementary functions in Table 3.19. If the representation type of the module is not extended, then it is translated into FoCaLiZe by function Tr_prod (see its definition in Table 3.18).

Function	Type	Definition	Note
Tr_code	$SPEC * FM \rightarrow SREP * SFUNC\ list$	$Tr\_code\ (vspec, vfm) =$ **Case** $rep : BAS\_REP ::$ $(Tr\_prod\ rep, vscfuncs\ )$ **Case** $rep : (vparent, vc\_prod)$ $:: (Tr\_erep\ (vparent, vc\_prod), vbas\_funcs\ @\ vscfuncs\ )$; **where** $vscfuncs = Map\ (Tr\_func\ (vspec, vfm))\ (vfuncs\ @\ vkfuncs)$;	- translates the implementations including function definitions and representation to FoCaLiZe, **where** $rep = Get\_rep\ vfm$; $(vparent, vc\_prod)$: EXT_REP (see Table 3.2 of FFML abstract syntax); $vfuncs = Get\_funcs\ vfm$; $vkfuncs = Get\_kfuncs\ (vspec, vfm)$; $vbas\_funcs = [Cr\_make\ vparent\ vc\_prod\ ; Cr\_first\ vparent; Cr\_second\ vc\_prod]$;

Table 3.17: Function translating the implementation code into the third FoCaLiZe species

Function	Type	Definition	Note
Tr_erep	$ID * C\_PROD \rightarrow SC\_PROD$	$Tr\_erep\ (vparent, vc\_prod) = Join\_ca\ (Tr\_tparent\ vparent, Tr\_prod\ vc\_prod)$	- translates an extended presentation to a FoCaLiZe presentation.
$Tr\_tparent$	$ID \rightarrow STYPE$	$Tr\_tparent\ vparent$	- translates $vparent$ to FoCaLiZe type.
Tr_prod	$C\_PROD \rightarrow SC\_PROD$		- translates Cartesian product in FFML to Cartesian product in FoCaLiZe
$Join\_ca$	$STYPE * SC\_PROD \rightarrow SC\_PROD$		- joins a FoCaLiZe type and a FoCaLiZe Cartesian product.

Table 3.18: Functions for translating an extended representation into FoCaLiZe

Example. The translation of representation type of module DL is shown in Listing 16. It is expressed as an extension from that of module BA with a new part *int* (line 2 of Listing 3.36). The corresponding representation type in FoCaLiZe is a Cartesian product

Function	Type	Definition	Note
Cr_make	$ID * C\_PROD \rightarrow SFUNC$	Cr_make ($vparent$, vc_prod)	- creates the function named $make$ combining two values into type $Self$ in FoCaLiZe. The code: $let\ make(vn_1 : vtype, vn_2 : vc\_prod)$: $Self = (vn_1, vn_2);;$ where vn_1, vn_2 are variable names, $vtype = Tr\_tparent\ vparent$ **and** $vc\_prod = Tr\_prod\ vc\_prod$.
Cr_first	$ID \rightarrow SFUNC$	$Cr\_first\ vparent$	- creates the function named $first$ to get the first part of a value typed $Self$ in FoCaLiZe. The code : $let\ first\ (vn : Self)$: $vtype = fst(vn);;$ **where** vn is variable name **and** $vtype = Tr\_tparent\ vparent$.
Cr_second	$C\_PROD \rightarrow SFUNC$	$Cr\_second\ vc\_prod$	- creates the function named $second$ to get the second part of a value typed $Self$ in FoCaLiZe. The code: $let\ second\ (vn : Self)$: $vprod = snd(vn);;$ **where** vn is variable name **and** $vprod = Tr\_prod\ cv\_prod$.

Table 3.19: Supplementary translation functions

of BA and *int* (line 3 of Listing 3.37). The functions following are basic functions *make*, *first* and *second* that respectively create a pair, compute the first projection or the second projection.

Listing 16 Example of translating the representation type of module DL to the third FoCaLiZe species

```
1 //fmodule DL from BA
2  representation extends BA with int;
```

Listing 3.36 DL in FFML

```
1 species DL_imp (BA is BA_imp)
2  inherit DL_spec2 (BA);
3
4  representation = BA * int;
5  let make (x: BA, a: int) : Self = (x, a);
6  let first (t: Self) : BA = fst(t) ;
7  let second (t: Self): int = snd(t);
8
```

Listing 3.37 DL in FoCaLiZe

Function Definition

In Table 3.17, we calculate two sets of function definitions. $vfuncs$ is the set of the function definitions explicitly written in the module. These implementations are copied in the third species. $vkfuncs$ is the set of all function definitions that are kept from the parent, they are calculated by the function Get_kfuncs. However, $vkfuncs$ are not

inherited in the sense of FoCaLiZe but are to be re-built from the parent which is a parameter of the third species. All of these functions are adapted to the change of the new representation type. So, $vkfuncs$ are to be generated automatically in FoCaLiZe according to the following scheme:

$$let\ f\ x\ =\ e;\ \textit{(in the parent FP)} \Rightarrow\ let\ f\ x\ =\ FP!f(first(x));\ \textit{(in the third species)}$$

in which the parameter x has the abstract type. We recall that a parameter called FP has been introduced in the third species. So $FP!f$ denotes the function f defined in FP. Furthermore the representation may have been extended, so a value of type FM is in this case a type whose first component is an element of type FP. So $FP!f(first(x))$ applies the parent function on a value of type FP.

Similarly, if $vfuncs$ are redefined, then they are updated automatically as follows (see Equation 3.11 in Section 3.4.4):

$$redef := \mathbb{C}[FP!f(arg_1, arg_2, ..., arg_n)]\ \textit{(in the parent FP)}$$
$$\Rightarrow redef := \mathbb{C}[FP!f(sarg_1, sarg_2, ..., sarg_n)]\ \textit{(in the third species)}$$

in which

$$sarg_i = \begin{cases} arg_i & \text{if } arg_i \text{ does not have the abstract type} \\ first(arg_i) & \text{if } arg_i \text{ has the abstract type} \end{cases} \quad i : 1, ..., n \qquad (3.13)$$

The function Tr_func (mentioned in Table 3.17) translates a function definition into the third species in FoCaLiZe. Function definitions in FFML are similar to those in FoCaLiZe. The translation rules for the function definitions focus mainly on adapting the function bodies to the change of the new representation type. The detail of Tr_func is described in Table 3.20. It contains the function Tr_expr that is defined for translating the expression $vexpr$ of the function $vfunc\_n$. This expression is adapted to the representation type $vrep$, in which $vpars$ are the parameters and $vftype$ is the type of the function.

Example. The function definitions of module DL in FFML are translated into FoCaLiZe, in Listing 17. Two function $limit\_with$ and get_wih are defined in module DL (lines 2-3 of Listing 3.38). The function $update$, declared previously in BA, is redefined (lines 4-9). Their translations are shown in lines 7-14 of Listing 3.39. The argument x of $update$

Function	Type	Definition	Note
Tr_func	$SPEC * FM \rightarrow FUNC\_N * PAR\ list * EXPR \rightarrow SFUNC\_N * SPAR\ list * SEXPR$	$Tr\_func\ (vspec, vfm)\ (vfunc\_n, vpars, vexpr) = (vfunc\_n, vpars, Tr\_expr\ (vpars, vexpr, Get\_ftype\ vspec\ vfm\ vfunc\_n, Get\_rep\ vfm))$	- translates a function definition to FoCaLiZe
Tr_expr	$PAR\ list * EXPR * TYPE * REP \rightarrow SEXPR$	$Tr\_expr\ (vpars, vexpr, vftype, vrep)$	- translates the expression $vexpr$ into FoCaLiZe, and adapts this expression corresponding to the representation type $vrep$.

Table 3.20: Translating a function definition to the third FoCaLiZe species

is updated automatically to $first(x)$ by Equation 3.13. The two functions get_bal, $over$, declared previously in BA, not redefined in FFML module DL, are kept and automatically generated in FoCaLiZe (lines 4-5).

Listing 17 Example of translating the functions of module DL to the third FoCaLiZe species

```
1 //fmodule DL from BA
2 let get_with (x) = second(x);
3 let limit_with = (-70);
4 let update (x,a) =
5  if (a <= 0) then
6    if (get_with(x) + a >= limit_with) then
7  (BA!update(x,a), get_with(x) + a)
8    else x
9  else (BA!update(x,a), get_with(x));
```

Listing 3.38 DL in FFML

```
1  species DL_imp (BA is BA_imp)
2   inherit DL_spec2 (BA);
3
4   let get_bal(x) = BA!get_bal(first(x));
5   let over = BA!over;
6
7   let get_with (x) = second(x);
8   let limit_with = (-70);
9
10  let update (x : Self, a : int) =
11   if (a <= 0) then
12    if (get_with(x) + a >= limit_with) then
13     (BA!update (first(x),a), get_with(x) + a)
14    else x
15   else (BA!update(first(x),a), get_with(x));
```

Listing 3.39 DL in FoCaLiZe

3.5.6.3 Correctness Proofs

The correctness proofs of a module in FFML are expressed in comments as proof scripts and copied from FoCaLiZe, presented through Tr_proofs in Table 3.15. We define the function in Table 3.21.

There are two kinds of correctness proofs in the code artifact part:

- the first kind ($vproofs$) includes the ones required for new properties or refining properties that must be done by the user. The properties, associated to the functions

which are redefined in the current module, must be reproved by the user. The translation of this proof kind is quite trivial, consisting in a simple copy.

- the second kind ($vkproofs$) concerns invariant properties and properties that correspond to the kept properties, such as, the proof of $ba\_bal\_gr\_over$. An invariant property may involve a function that has been redefined. In that case it has to be proved again. The correctness proofs, belonging to this kind, are automatically generated by the function Up_kproof, unfolding the functions and adding the proof hints being the names of the properties of the parent. It is just a heuristics. Zenon may fail in that case, asking for more hints or subproofs provided by the user.

Function	**Type**	**Definition**	**Note**
Tr_proofs	$SPEC * FM \rightarrow SPROOF\ list$	$Tr\_proofs\ (vspec, vfm) =$ Map $(Tr\_proof\ (vspec, vfm))\ (vproofs\ @\ vkproofs)$	- translate FFML proof notations to FoCaLiZe, **where** $vproofs = Get\_proofs\ vfm$; $vkproofs = Map\ (Up\_kproof\ vfm)\ Get\_kproofs\ (vspec, vfm)$;
Tr_proof	$PROP\_N * FOCPROOFBODY \rightarrow SPROOF\_N * SPROOFBODY$	$Tr\_proof\ (vrpoof\_n, vfocproof)$	- copy a FFML proof notation to FoCaLiZe.

Table 3.21: Translating proofs to the third FoCaLiZe species

Example. The correctness proof of property $ba\_upd\_succ\_with\_over$ shown in Listing 3.40 is proved in line 8. Its translation is in line 6 of Listing 3.42. However, in DL (Listing 3.41) this property is refined into the property $dl\_upd\_succ\_with\_wlimit\_R1$ (lines 2-4) and is reproved (lines 6-14). The proof of this refining property is translated into FoCaLiZe (lines 9-15 of Listing 3.43). Module BA also contains another property $ba\_bal\_gr\_over$ in lines 2-3 and its corresponding proof lines 6-7 of Listing 3.40. The property is not refined but is kept in DL. Its proof is generated automatically: the hints indicate to use the definitions of the functions get_bal and $over$, and also the property $ba\_bal\_gr\_over$ of BA (lines 5-6 of Listing 3.43). This proof is done automatically by Zenon.

Listing 18 Example of translating correctness proofs into the third FoCaLiZe species

```
1  //module BA
2  contract get_bal :: invariant property
3  ba_bal_gr_over : all x:BA, get_bal(x) >= over;
4  contract update :: property
       ba_upd_succ_with_over: all x:BA, all a:
       int, (get_bal(x) + a) >= over -> get_bal(
       update(x,a)) = get_bal(x) + a;
5
6  proof of ba_bal_gr_over =
7   foc proof { * assumed *};
8  proof of ba_upd_succ_with_over = foc proof {*
       by definition of update, get_bal *};
```

Listing 3.40 BA in FFML

```
1  //species BA_spec1 - the first species
2  property ba_bal_gr_over : all x : Self,
3   (get_bal(x) >= over);
4
5  //species BA_spec2 - the second species
6  property ba_up_succ_with_over : all x:Self,all
       a:int, ((get_bal(x) + a) >= over) -> (
       get_bal(update(x, a)) = (get_bal(x) + a))
       ;
7
8  //species BA_imp - the third species
9  proof of bal_gr_over = assumed ;
10 proof of ba_upd_succ_with_over = by definition
       of update, get_bal;
```

Listing 3.42 BA in FoCaLiZe

```
1  //module DL from BA
2  contract update ::  property
       dl_upd_succ_with_wlimit_R1
3   refines BA!ba_upd_succ_with_over
4   extends premise (a <= 0) /\ (get_with(x) + a
        >= limit_with);
5
6  proof of dl_upd_succ_with_wlimit_R1 =
7   foc proof {*
8    <1>1 assume x: DL, a : int,
9    hypothesis h1: (a <= 0) /\ (get_with(x) + a
         >= limit_with),
10   prove (get_bal(x) + a) >= over -> (get_bal (
         update(x,a))) = get_bal(x) + a
11    <2>1 prove first(update(x,a)) = BA!update (
          first(x),a)
12    by definition of first,update hypothesis h1
13    <2>e qed by step <2>1 definition of over,
          get_bal property BA!
          ba_upd_succ_with_over
14   <1>e conclude;*}
```

Listing 3.41 DL in FFML

```
1  //species DL_spec2 - the second species
2  property dl_upd_succ_with_wlimit_R1 : all x :
       Self, all a : int, (a <= 0) && ((get_with
       (x) + a) >= limit_with) -> ((get_bal(x) +
       a) >= over) -> (get_bal(update(x, a)) =
       (get_bal(x) + a));
3  //species DL_imp - the third species
4
5   proof of ba_bal_gr_over = by definition of
6    get_bal, over property BA!ba_bal_gr_over;
7
8  proof of dl_upd_succ_with_wlimit_R1 =
9   <1>1 assume x : Self, assume a : int,
10  hypothesis h1 : (a <= 0) && ((get_with(x) + a
        ) >= limit_with),
11  prove ((get_bal(x) + a) >= over) -> (get_bal(
        update(x, a)) = (get_bal(x) + a))
12   <2>1 prove (first(update(x, a)) = BA!update(
         first(x), a))
13   by definition of first, update hypothesis h1
14   <2>e qed by step  <2>1 definition of over,
         get_bal property BA!
         ba_upd_succ_with_over
15  <1>e conclude;
```

Listing 3.43 DL in FoCaLiZe

3.5.7 Generating a collection

The function Tr_col (mentioned previously in Table 3.4) is described in Table 3.22. This function creates a collection that implements the third species. An example shows how to build a collection for BA in Listing 19.

Function	Type	Definition	Note
Tr_col	$FM \rightarrow SP$	$Tr\_col\ vfm =$ $(Tr\_id_4\ vfm,$ $Tr\_imp\ vfm)$	- is a translation function translating module vfm to **a collection** in FoCaLiZe which implement the third species.
Tr_id_4	$FM \rightarrow SID$	$Tr\_id_4\ vfm =$ $(Get\_id\ vfm)\ ^\wedge$ "$\_col$"	- translates the module name of module vfm to a collection name ended with $\_col$.
Tr_imp	$ID \rightarrow SID\ *$ $SPARAM\_N\ list$	$Tr\_imp\ vparent =$ $(vparent\ ^\wedge$ "$\_col$", $Tr\_param\ vparent)$	- translates a parent name to a collection implementation clause.

Table 3.22: Function translating the module and parent names of a module into a collection

Listing 19 Example of translating a module to a collection in FoCaLiZe

```
1 fmodule DL from BA
2 ... ;;
```

Listing 3.44 DL in FFML

```
1  species DL_spec1 (BA is BA_imp) =
2  ... end;;
3  species DL_spec2 (BA is BA_imp) =
4  ... end;;
5  species DL_imp (BA is BA_imp)
6   inherit DL_spec2 (BA);
7  ... end;;
8  collection DL_col =
9   implement DL_imp(BA_col);
10 end;;
```

Listing 3.45 DL in FoCaLiZe

3.6 Correctness of FFML Translation

With the purpose of writing and reusing artifacts easily, modules are written in FFML and then translated in FoCaLiZe. As mentioned in the beginning of this chapter, the implementation of the artifacts in FoCaLiZe is more complex. FMML reduces this complexity by dissembling code repetition such as keeping the properties, the function definitions, the representation type, from a parent module. These artifacts are automatically copied, sometimes with some adaptation, or redefined into FoCaLiZe syntactic elements. Besides shorter code, FFML supports the reuse and modification mechanisms that are translated with the help of the inheritance and parametrization mechanisms available in FoCaLiZe.

Although the syntax of FFML and FoCaLiZe are close to each other, FFML supports other keywords allowing the developer to reuse existing properties for specifying new ones. To implement these in FoCaLiZe, we decided to translate an FFML module into three species with the purpose of distinguishing the artifacts according to the kind of artifacts (specification versus code and correctness proof) and also according to the variability of property may have (invariant property versus other kinds of properties).

First, to guarantee the correctness of the translation, FFML translation needs the formal semantics of FoCaLiZe [Prevosto and Doligez 2002], on which we rely. In fact, we borrowed FoCaLiZe's syntax and expanded it. Second, we use flexibly the mechanisms available in FoCaLiZe, allowing us to reuse and modify artifacts. Namely, the specification which is inherited, is translated into the first species that will be inherited by other species using the inheritance mechanism in FoCaLiZe. The second species contains the specification that will be copied in FoCaLiZe with an intention of refining or modifying. Lastly, the parameterisation mechanism is intensively used in the third species. It allows to keep the functionality, encapsulated as specification abstractions, from the parent module.

A formal proof of the correctness of our translation is beyond this thesis. It is a hard and complex task that would need more time. We have tested the translation on different FFML files, coming from the Bank Account SPL and another example (Poker SPL that will be developed in Chapter 5). Furthermore, the translated programs in FoCaLiZe have been executed on different test cases.

3.7 Summary

While comparing FFML to FoCaLiZe through their syntax (Section 3.3.1) and semantics (Sections 3.4), we can notice that FFML is close to FoCaLiZe, but focuses on building a friendly environment for the developer and does not increase encoding effort. This is considered as an advantage of FFML and explains why an FFML module is implemented in FFML more easily and in a shorter way than in FoCaLiZe.

It is worth mentioning that the keyword `from` included in FFML grammar is very flexible. Using this keyword `from`, FFML Compiler also recognizes from which a module

is extended or modified. As described by the previous translation functions, we can see that the keyword `from` does not have the same meaning as the keyword `inherit` in FoCaLiZe (Grammar A.1) which is used only for inheritance mechanism. In fact, the keyword `from` conveys itself a combination of several mechanisms for reusing and modifying artifacts.

We have formulated the semantics of the FFML language by representing how each kind of artifacts can be calculated through the mechanisms proposed for the language. Its semantics is related tightly to the meanings of the keyword `from`. The example Bank Account SPL, written in FFML, illustrates how to write the different artifacts.

We also have defined the translation rules for translating FFML modules into FoCaLiZe. The elements of an FFML module are respectively translated into three different species and a collection using translation functions. Three species allow an easier management, hence the translated artifacts can be reused in FoCaLiZe. **FFML Compiler tool**, implementing these translation functions, is built in OCaml with more than three thousands of code lines. We use the newest version of FoCaLiZe 0.9.1. FFML presents some systematic and semantic constraints, for example we require that each FFML module is complete, that a function (re)definition has the same type (or a more general one) as its signature. These constraints are verified by the FoCaLiZe compiler on the translated versions of FFML files.

As indicated in Chapter 2, to develop correct-by-construction SPLs, besides a language supporting the writing, our methodology also needs a technique for automatically generating the correct final products. This requires a connection of modules written in FFML to the generation technique, and then definitions for composing these modules according to the technique. The methodology has to be proved efficient in implementing other SPLs. With the two following chapters, we will resolve these issues. In Chapter 4, we define the composition of FFML modules, formulate the generation of FFML products and present an FFML Product Generator tool. Using our tool chain (FFML Compiler and FFML Product Generator) to implement a bigger example (Poker SPL), we evaluate our methodology in Chapter 5.

Chapter 4

FFML Product Generation

In Chapter 2, we have described a methodology which applies FOP technique for developing correct-by-construction SPLs. An FFML language has been defined for writing these SPLs in Chapter 3. In this chapter, we focus on an automated generation of the products that is proposed in the methodology.

The basic requirements of the automated generating process are discussed in Section 4.1. This part tries to answer the questions about how to make the process automated and correct. In Section 4.2, we define a binary composition operation for FFML modules. We continue to analyze how the operation behaves on each kind of artifacts involved in the modules in Section 4.3. We describe the composition rules for implementing this operation in detail in Section 4.4. In Section 4.5, we explain the generation process of final products. Using the Product Generator tool, into which the composition rules are embedded, the automated generation of the final correct products is illustrated in Section 4.6 on the Bank Account SPL. Finally, we summarize our methodology in Section 4.7.

4.1 Basic Requirements of The Automated Product Generation

Over the last decade, many methods were proposed for generating SPL products. Following FOP and design-by-contract techniques, Thüm et al. propose composition mechanisms for contracts [Thüm 2015]. The contract composition operations are formally defined, since mapped to the composite proofs which are built in Coq [Thüm et al. 2011,

2014b]. They call the proof parts, which are located somewhere and related to modules, *partial proofs*. When composing the modules, these partial proofs are also composed. However, the proof compositions are explained by cases, which may happen while composing, but not by formal definitions. Consequently, these compositions are not implemented automatically for proofs. In the context of programming language meta-theory within the Coq proof assistant, another direction for formally composing artifacts is proposed by Delaware et al. in [Delaware et al. 2011, 2013]. This approach guarantees the reuse and composition of correctness proofs but was only implemented on a limited language domain. Recently, an approach for formally composing models is presented in [Hamiaz et al. 2016]. The idea is that once elementary composition operators are formalized and verified, they can be used to define more complex operators. However, the approach was achieved on metamodels and follows component-based techniques which is not actual SPLE [Clements and Northrop 2001]. By contrast, our main goal in this direction is to develop SPLs and to define the composition operations at the level of modules which contain three kinds of artifacts.

Pursuing the main purpose of an automated generation tool, from the existed methods we give below the basic requirements which are essential to generate correct product variants.

- First, we only need a binary composition operation which is applied for a pair of modules, since composition of more modules can be done pairwise according to the module diagram. The generator tool embeds this binary composition operation.

- Second, FFML Product Generator tool needs to contain automated functions, implementing composition rules for composing two modules. These rules decide which artifacts are composed and how to compose artifacts. They are built for all kinds of artifacts: specification, code and correctness proof. The output of the composition will be a composite module, which should be also a module, that is, this composite module can be reused to write other modules. This is needed to remain the modularization and the optimization while developing SPLs in FFML. As defined in Chapter 3, the keyword `from` of FFML, which is used to express that a module is extend-

ed/modified from another, is also used here to express the relationships between the composite modules. This new semantics is introduced in Section 4.2.

- Third, it is necessary to have an implementation technique for establishing products. The generator tool also takes the hierarchy, relationships and order of the features as inputs.

Figure 4.1 shows how to obtain the expected product using the FFML Product Generator tool. The final product is built by lining up the composite modules which are denoted by the red nodes with a symbol "/" following the module names. While building the product variant, the involved modules in the module diagram have to be in turn composed by the binary operator.

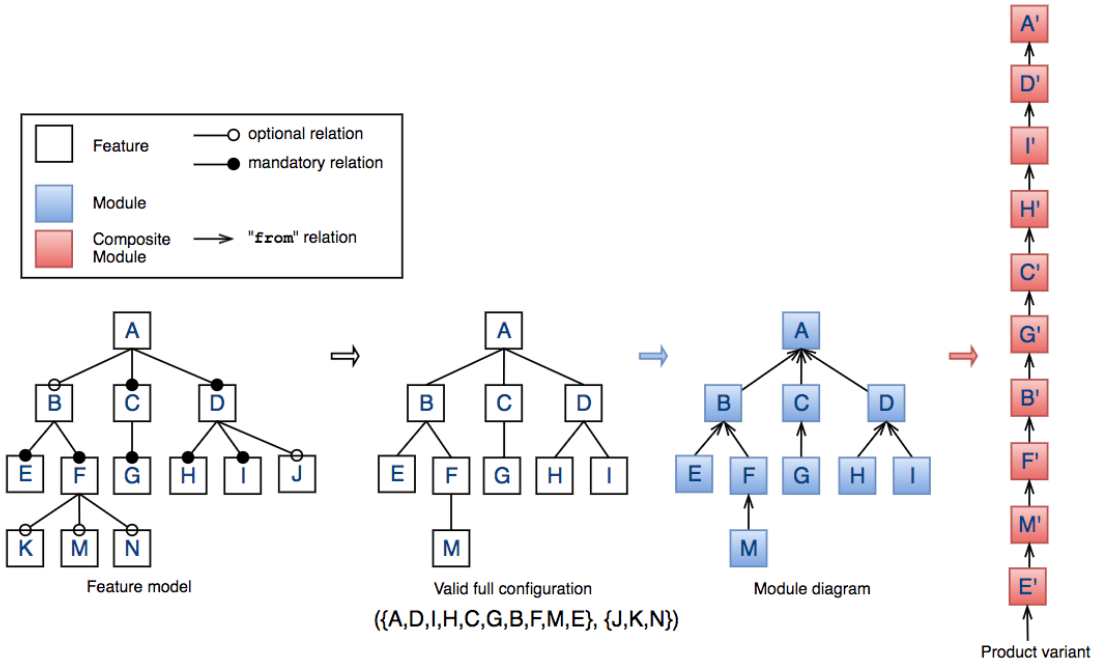

Figure 4.1: Product Building

In the next sections of this chapter we sequentially explain the solutions we have adopted for the requirements mentioned above.

4.2 Module Composition Operation

In this section, we present our binary composition operation defined for two modules. The operation is applied for computing the composition of all kinds of the artifacts (specification, code and correctness proof) contained in these two modules. The artifacts are composed by the operation which is specific to that artifact kind.

Notation 2 (Binary Composition Operation) *Given two modules FM_2 and FM_1, the composition of module FM_2 with module FM_1, represented by $FM'_2 = FM_2 \bullet FM_1$, is the binary composition operation that forms a composite module FM'_2 from module FM_2 and refers to module FM_1 as the parent of the composite module via a* `from` *relation.*

We describe a binary composition operation in Notation 2 (that is analyzed in Section 4.3, and then is defined via the function Com_fm in Table 4.1 of Section 4.4) The composition is calculated from the artifacts contained in the two modules FM_2 and FM_1. We use dot "$\bullet$" to represent the operation. Its output is a composite module, illustrated in Figure 4.2, connected to FM_1 by a relation `from`. The composite module FM'_2 refers to the name MN_1 of FM_1 meaning that FM_1 is its parent. FM'_2 is also an FFML module modeled by a 7-tuple (as defined in Section 3.2.2) whose components are sequentially calculated by the composition operation. The exact composition of the tuple elements $(MN_2 \hat{} MN_1, MN_1, S'_2, P'_2, R'_2, D'_2, Pf'_2)$ is defined in the following section. As a consequence, FM'_2 can be extended/modified to build other modules by means of FOP principle.

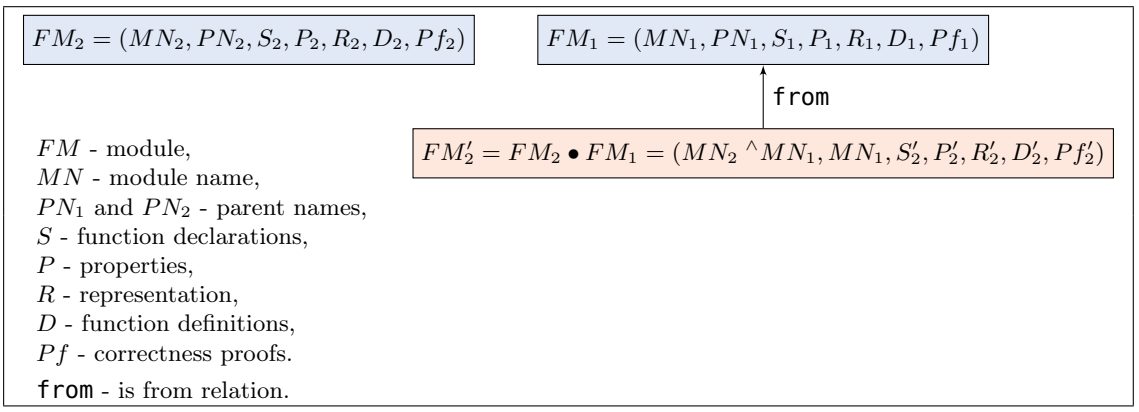

Figure 4.2: Module composition

Our composition operation is different from the others defined previously. More precisely, our operation is defined to calculate all kind of artifacts (specification, implementation code and correctness proof). In AHEAD the composition operation is achieved on the inheritance mechanism, namely *mixin*, in object-orienting programming [Batory et al. 2003]. A keyword `super` is used for representing this inheritance relation in code artifact. Another composition operation was defined in FEATUREHOUSE which uses superimposition mechanism. The keyword `original` is used to refer to the previous implementation (code artifact) [Apel et al. 2009]. Thüm improved the operation in FEATUREHOUSE for composing specification artifacts, namely for contracts, by pursuing design-by-contract approach [Thüm et al. 2014b; Thüm 2015]. For more extension, the mechanisms we define for our composition operation is represented through the keyword `from`. This keyword conveys itself more complicated mechanisms as a mix of several ones for not only extending/modifying (see in Section 3.3.3) but also composing the artifacts of modules. To understand more about how the composition operation, we continue the discussion in the next sections.

4.3 Composition Analysis

In this section, we focus on analyzing how the binary composition operation, introduced in Notation 2, achieved on specific artifact kinds. According to the notation, the composite module FM_2' is formed from FM_2 and refers to FM_1 according to the `from` relation. In fact, the module associated to the root feature is built first and does not to be composed to others. This module is involved in all products. In other words, composition only concerns the modules (namely FM_1 and FM_2) which are not the root. In the next sections, we focus on analyzing how the binary composition operation is deployed on all kinds of artifacts.

4.3.1 Signature

The signatures S_2' of FM_2' are calculated as in Equation 4.1. The signatures S_2 of FM_2 are entered into FM_2', but the type name PN_2 is replaced by the type name MN_1 and the type name MN_2 is replaced by the type name MN_2'.

$$S'_2 = (S_2)_{[PN_2 \leftarrow MN_1][MN_2 \leftarrow MN'_2]} \tag{4.1}$$

By `from` relation built for the composite module (see in Figure 4.2), the signatures S_1 of FM_1 are entered into FM'_2 (by the function Rpr_signs in Table 4.2 of Section 4.4.1). Applying Equation 3.5 in Section 3.4.1 about FFML semantics of signatures, we calculate the meaning of S'_2 as follows:

$$[\![S'_2]\!] = [\![S_1]\!]_{[MN_1 \leftarrow MN'_2]} \cup S'_2 = [\![S_1]\!]_{[MN_1 \leftarrow MN'_2]} \cup (S_2)_{[PN_2 \leftarrow MN_1][MN_2 \leftarrow MN'_2]}$$

4.3.2 Property

As described, our methodology applies the principles of the *design by contract* approach for specifying functionality of SPL. That is, for each function, the developer can write a set of properties specifying its behaviors. We imply that the composition operation only concerns the properties related to a function. In other words, for each function its related properties are composed.

The idea of composing properties is related directly to function (re)definitions in two modules FM_2 and FM_1. Each module contains the properties which are either invariant properties iP, new properties nP or refining properties rP. The composition operation manipulates all these properties and the composite properties P'_2 are computed as follows:

$$\begin{aligned} &P'_2 = (iP_2 \cup nP_2)_{[PN_2 \leftarrow MN_1][MN_2 \leftarrow MN'_2]} \cup rP'_2 \\ &\text{where } rP'_2 = (rP_2)_{[PN_2 \leftarrow MN_1][MN_2 \leftarrow MN'_2]} \bullet [\![P_1]\!]; \\ &\text{and } [\![P_1]\!] \text{ is the set of all the properties of } FM_1; \end{aligned} \tag{4.2}$$

We can see that the composite properties P'_2 are calculated by Equation 4.2. These properties consist of iP_2 and nP_2 from FM_2, and rP'_2 established by composing rP_2 from FM_2 and $[\![P_1]\!]$ from FM_1 using "$\bullet$" operation (defined in Section 4.4.2 as the function Com_props in Table 4.3). rP'_2 are also the refining properties of the composite module FM'_2. $[\![P_1]\!]$ is the set of all properties of FM_1 calculated by Equation 3.6. As with

signatures, the type names PN_2 and MN_2 are respectively replaced by the type names MN_1 and MN'_2.

The reason why rP'_2 is established by composing with $[\![P_1]\!]$ instead of P_1 is that $[\![P_1]\!]$ consists of all properties of FM_1 (calculated by Equation 3.6 in Section 3.4.2). Note that, FFML allows the user to extend a module but not to repeat the reusable artifacts. However, the FFML semantics still keeps the meaning of these artifacts. Hence, if the composition only concerns P_1, the reused property of FM_1 can be lost. In order to avoid losing these properties, the composition is calculated on $[\![P_1]\!]$.

Because of `from` relation used for connecting FM'_2 and FM_1, the properties P_1 are entered into FM'_2 via the FFML reuse mechanism. Applying Equation 3.6 in Section 3.4.2 about the semantics of property, we calculate the property meaning of P'_2 as follows.

$$[\![P'_2]\!] = ([\![P_1]\!] \setminus \mathcal{P}([\![P_1]\!], rP'_2))_{[MN_1 \leftarrow MN'_2]} \cup P'_2 \tag{4.3}$$

4.3.3 Representation Type

The representation type R'_2 of FM'_2 is calculated from that of FM_2 according to Equation 4.4 (defined by the function Rpr_rep in Table 4.6 of Section 4.4.3). The type name MN_1 replaces the type name PN_2 in R_2. The meaning of this replacement is that R'_2 reuses the representation type of FM_1 but keeps the extended part R_2 of FM_2 if this part exists.

$$R'_2 = (R_2)_{[PN_2 \leftarrow MN_1]} \tag{4.4}$$

Via the relation `from` connecting FM'_2 to FM_1, the representation type R_1 is entered into FM'_2 via the FFML reuse mechanism. We apply Equation 3.9 in Section 3.4.3 (the semantics of representation type) to calculate meaning of R'_2 as follows:

$$[\![R'_2]\!] = [\![R_1]\!] * R'_2 \tag{4.5}$$

When composing the representation types the change in the composite representation type may impact other elements of the composite module. A signature or a property

related to a function is specified with abstract types, thus they are not affected when the representation type changes. However, a (re)definition, being an implementation of the function, based on concrete type, new composite representation type forces converting between explicit types. We can understand more about this impact when composition rules for (re)definitions and correctness proofs are analyzed in the next sections.

4.3.4 (Re)definition

An FFML module may contain new definitions nD or re-definitions rD. The composition of the function (re)definitions of two module FM_2 and FM_1 is represented in Equation 4.6. The composite module FM'_2 includes nD_2 from FM_2 and rD_2 that is composed with $[\![D1]\!]$ by "$\bullet$" operation (defined in Section 4.4.4 by the function Com_funcs in Table 4.7), in which $[\![D1]\!]$ is the set of all (re)definitions of FM_1. The result of this composition is the composite re-definition rD'_2 for FM'_2. As with signatures and properties, the type names PN_2 and MN_2 are respectively replaced by the type names MN_1 and MN'_2.

$$
\begin{aligned}
&D'_2 = (nD_2)_{[PN_2 \leftarrow MN_1][MN_2 \leftarrow MN'_2]} \cup rD'_2 \\
&\text{where } rD'_2 = (rD_2)_{[PN_2 \leftarrow MN_1][MN_2 \leftarrow MN'_2]} \bullet [\![D_1]\!]; \\
&\text{and } [\![D_1]\!] \text{ is the set of all (re)definitions of } FM_1
\end{aligned}
\tag{4.6}
$$

The (re)definitions of FM_1 are entered into FM'_2 using the FFML reuse mechanism. Applying the semantic Equation 3.11 in Section 3.4.4 about FFML semantics of (re)definition, we calculate the definition meaning for FM'_2 as follows:

$$[\![D'_2]\!] = ([\![D_1]\!] \setminus \mathcal{D}([\![D_1]\!], rD'_2))_{[MN_1 \leftarrow MN'_2]} \cup D'_2$$

4.3.5 Correctness Proof

We represent how the correctness proofs of modules FM_2 and FM_1 are composed in Equation 4.7. The proofs iPf_2, nPf_2 and rPf_2 (see page 65 of Section 3.4.2) are moved into FM'_2. However, the proofs rPf_2 have to be composed with the proofs $[\![Pf_1[\![$ (the set of all the correctness proofs of FM_1) using the operation "$\bullet$" (defined by the function

Com_proofs in Table 4.10 of Section 4.4.5). We call the result of these compositions rPf'_2. the type names PN_2 and MN_2 are respectively replaced by the type names MN_1 and MN'_2.

$$
\begin{aligned}
&Pf'_2 = (nPf_2)_{[PN_2 \leftarrow MN_1][MN_2 \leftarrow MN'_2]} \cup rPf'_2 \\
&\text{where } rPf'_2 = (rPf_2)_{[PN_2 \leftarrow MN_1][MN_2 \leftarrow MN'_2]} \bullet [\![Pf_1]\!] \\
&\text{and } [\![Pf_1]\!] \text{ is the set of all the correctness proofs of} of FM_1;
\end{aligned} \tag{4.7}
$$

The proofs of FM_1 are also entered into FM'_2 using FFML reuse mechanism. Similar to other kinds of artifacts, we calculate the set of all proofs of FM'_2 by applying Equation 3.12 in Section 3.4.5 as follows.

$$[\![Pf'_2]\!] = ([\![Pf_1]\!] \setminus \mathcal{P}f([\![Pf_1]\!], rPf'_2))_{[MN_1 \leftarrow MN'_2]} \cup Pf'_2$$

In this section, we analyzed how the binary composition operation is applied to all kinds of artifacts. According to this analysis, in the next section we aim at defining the composition rules for two modules and their artifacts.

4.4 Composition Rules

In this section we focus on defining the binary composition operation for modules by expressing the functions implementing the composition rules for all the artifacts involved in two modules.

The main function Com_fm is defined for composing two modules vfm_2 and vfm_1 of a SPL, represented in the first line of Table 4.1. It corresponds to Notation 2 of the binary composition operation between two modules. The supplementary parameter $vspec$ is used to get the information from other FFML modules of the SPL. The result of the composition is a module (of type FM), called a composite module. This composite module uses the module vfm_1 (with name $vname_1$) as its parent.

The functions, used in the definition of Com_fm, realizes intermediate composition rules for the different elements into the composite module. Function Com_id is used for

creating the name of the composite module, function Rpr_sigs for defining the signatures, function Com_props for composing the properties, function Rpr_rep for composing the representation types, function Com_funcs for composing the function (re)definitions and function Com_proofs for composing the correctness proofs.

Function	Type	Definition	Note
Com_fm	$SPEC$ * FM * FM $\rightarrow FM$	$Com\_fm\ (vspec, vfm_2, vfm_1) =$ $\big(vname_{21}, vname_1,$ $Rpr\_sigs\ (vfm_2, vname_1),$ $Com\_props\ (vspec, vfm_2, vfm_1),$ $Rpr\_rep\ (vfm_2, vname_1),$ $Com\_funcs\ (vspec, vname1, vfm_2, vfm_1),$ $Com\_proofs\ (vspec, vfm_2, vfm_1)\big)$	- composes two modules vfm_2 and vfm_1 into a composite module, where $vname_2 = Get\_id\ vfm_2$; $vname_1 = Get\_id\ vfm_1$; $vname_{21} = vname_2\ ^\wedge\ vname_1$; Get_id - gets the identity of a module

Table 4.1: Composition function of two modules vfm_2 and vfm_1

In Listing 20 we present an example of a composite module DLLL (Listing 4.3) which is established by composing module DL (Listing 4.2) and module LL (Listing 4.1). $DLLL$ expresses that module LL is its parent (line 1 of Listing 4.3).

Listing 20 Example of composing the identity of two modules

```
1 fmodule LL from BA
2 ...;;
```

Listing 4.1 module LL

```
1 fmodule DLLL from LL
2 ...;;
```

Listing 4.3 module DLLL

```
1 fmodule DL from BA
2 ...;;
```

Listing 4.2 module DL

The intermediate composition functions used in Table 4.1 are further discussed in the following subsections.

4.4.1 Signature

When implementing the binary composition operation, the function declarations in module vfm_2 are reproduced into the composite module by a function Rpr_signs, represented in Table 4.2. As analyzed in Section 4.3.1, the composite function indicates module vfm_1 as its parent (using keyword `from`), the signatures in vfm_1 are still available and entered into the composite module thanks to the "from" mechanism of FFML. There is a

supplementary function Up_stype, that renames the module type $vname_2$ into the composite module type $vname_{21}$.

Function	Type	Definition	Note
Rpr_signs	$FM * ID \rightarrow SIG\ list$	$Rpr\_signs\ (vfm_2, vname_1) = Map\ (Up\_stype\ (vname_2, vname_{21}))\ vsigns_2$	- reproduces signatures into composite module; Where: $vname_2 = Get\_id\ vfm_2$; $vname_{21} = vname_2\ ^\wedge\ vname_1$; $vsigns_2 = Get\_sigs\ vfm_2$; Get_id - gets module name; Get_sigs - gets signatures;
Up_stype	$ID * ID \rightarrow SIG \rightarrow SIG$	$Up\_stype\ (vname_2, vname_{21})\ vsign_2$	- updates the module type $vname_2$ of the signature $vsig_2$ to $vname_{21}$;

Table 4.2: Concatenating the names of two modules vfm_2 and vfm_1

An example of reproducing a signature get_with, declared in module DL, is shown in Listing 21. Before composing, the function takes as inputs module type DL and returns an output type int (line 2 of Listing 4.5). In the composite module, this signature is copied in the composite module $DLLL$ with an updated type $DLLL$ (Listing 4.6).

Listing 21 Example of signature composition

```
1 fmodule LL from BA
2   signature limit_low: int;
3 ...;;
```

Listing 4.4 Module LL

```
1 fmodule DLLL from LL
2  signature get_with : (DLLL) -> int;
3 ...
```

Listing 4.6 Module $DLLL$

```
1 fmodule DL from BA
2  signature get_with: (DL) -> int;
3 ...
```

Listing 4.5 Module DL

4.4.2 Property

This section aims at describing how properties are composed. We begin by explaining the formulas representing the combination of two sets of properties in Section 4.4.2.1. The functions for composing properties are discussed in Section 4.4.2.2.

4.4.2.1 Composition rules of properties

In this section, we discuss the composition rules for two sets of properties, which are specified following the design by contract approach. According to our classification of

properties (Section 3.3.2) the properties P_f related to a function f are formulated using the keyword `contract` and formulated as $P_f = \{p_{f1}(\overrightarrow{x_1}), p_{f2}(\overrightarrow{x_2}), ..., p_{fn}(\overrightarrow{x_n})\}$, in which $\overrightarrow{x_i}$ is list of universally quantified variables.

The developer may have written properties for f in both modules FM_1 and FM_2. P_{1f} is the set of properties $p'_{fj}(\overrightarrow{x'_j})$ in FM_1, represented by $P_{f1} = \{p'_{f1}(\overrightarrow{x'_1}), p'_{f2}(\overrightarrow{x'_2}), ..., p'_{fn}(\overrightarrow{x'_n}), p'_{f(n+1)}(\overrightarrow{x'_{n+1}}), ..., p'_{fm}(\overrightarrow{x'_m})\}$. The properties $p'_{f1}(\overrightarrow{x'_1}), p'_{f2}(\overrightarrow{x'_2}), ..., p'_{fn}(\overrightarrow{x'_n})$ respectively modifies or keeps the properties $p_{f1}(\overrightarrow{x_1}), p_{f2}(\overrightarrow{x_2}), ..., p_{fn}(\overrightarrow{x_n})$ of P_f. The properties $p'_{f(n+1)}(\overrightarrow{x'_{n+1}}), ..., p'_{fm}(\overrightarrow{x'_m})$ are new ones written into FM_1.

Similarly, the set of properties in module vfm_2, is represented by $P_{2f} = \{p''_{f1}(\overrightarrow{x''_1}), p''_{f2}(\overrightarrow{x''_2}), ..., p''_{fn}(\overrightarrow{x''_n}), p''_{f(n+1)}(\overrightarrow{x''_{n+1}}), ..., p''_{fk}(\overrightarrow{x''_k})\}$. The properties $\{p''_{f1}(\overrightarrow{x''_1}), p''_{f2}(\overrightarrow{x''_2}), ..., p''_{fn}(\overrightarrow{x''_n})$ modifies or keeps the properties $p_{f1}(\overrightarrow{x_1}), p_{f2}(\overrightarrow{x_2}), ..., p_{fn}(\overrightarrow{x_n})$ of P_f. The properties $p'_{f(n+1)}(\overrightarrow{x'_{n+1}}), ..., p'_{fm}(\overrightarrow{x'_m})\}$ are new ones written into FM_2.

The composite properties P_{f21}, resulting from the composition of FM_2 and FM_1, are calculated by composing P_{f2} and P_{f1} as follows.

$$\begin{aligned} P_{f21} = P_{f2} \bullet P_{f2} = \{&p''_{f1}(\overrightarrow{x''_1}) \bullet p'_{f1}(\overrightarrow{x'_1}),\\ &p''_{f2}(\overrightarrow{x''_2}) \bullet p'_{f2}(\overrightarrow{x'_2}),\\ &...,\\ &p''_{fn}(\overrightarrow{x''_n}) \bullet p'_{fn}(\overrightarrow{x'_n}),\\ &p''_{f(n+1)}(\overrightarrow{x''_{n+1}}), ..., p''_{fk}(\overrightarrow{x''_k}),\\ &p'^{*}_{f(n+1)}(\overrightarrow{x'_{n+1}}), ..., p'^{*}_{fm}(\overrightarrow{x'_m})\} \end{aligned} \tag{4.8}$$

In Equation 4.8, the properties in both modules having index from 1 to n are composed by Equation 4.9 using the operation "$\bullet$" defined by Notation 2 in Section 4.4.2.2. The new properties of vfm_2 are copied while the new properties of vfm_1 are changed according to Equation 4.10 (marked with a symbol "*").

$$\begin{aligned} p''_k(\overrightarrow{x''_k}) \bullet p'_j(\overrightarrow{x'_j}) = \forall \overrightarrow{x_{kj}} \in U,\ &prem''_k(\overrightarrow{x''_{kj}}) \rightarrow p'_j(\overrightarrow{x'_{kj}})\\ &where\ \overrightarrow{x_{kj}},\ is\ a\ combination\ of\ \overrightarrow{x''_k}\ and\ \overrightarrow{x'_j}\\ &and\ j,k : 1, ..., n; \end{aligned} \tag{4.9}$$

Equation 4.9 represents the composition of two properties $p''_k(\overrightarrow{x''_k})$ and $p'_j(\overrightarrow{x'_j})$, both of which refines (or keeps) the same property $p_i(\overrightarrow{x_i})$. $prem''_k(\overrightarrow{x''_{kj}})$ is the premise of $p''_k(\overrightarrow{x''_k})$. The parameters $\overrightarrow{x_{kj}}$ of the composite property is a union of $\overrightarrow{x''_k}$ and $\overrightarrow{x'_j}$.

The composite property, established by Equation 4.9, is a refining property that refines property $p'_j(\overrightarrow{x'_{kj}})$ of FM_1 by adding $prem''_k(\overrightarrow{x''_{kj}})$ of FM_2 as its new premise.

While composing the properties of two modules, a special case can happen when they are invariant properties kept from Pf. As defined, these properties can not be refined or modified in other modules, since their compositions, implemented by operation "$\bullet$" in Equation 4.9, are themselves. In other words, the composite module keeps these properties.

The case of transposing a new property $p'_j(\overrightarrow{x'_j})$ with $n+1 \leq j \leq m$ of module FM_1 into the composite module is represented by Equation 4.10. The premise $prem''_k(\overrightarrow{x''_{kj}})$ (Equation 4.9) is added into $p'_j(\overrightarrow{x'_{kj}})$ to build the refining property $p^*_k(\overrightarrow{x_{kj}})$.

$$\begin{aligned} p^*_k(\overrightarrow{x_{kj}}) = \forall \overrightarrow{x_{kj}} \in U,\ prem''_k(\overrightarrow{x''_{kj}}) \rightarrow p'_j(\overrightarrow{x'_{kj}}) \\ where\ \overrightarrow{x_{kj}},\ is\ a\ combination\ of\ \overrightarrow{x''_k}\ and\ \overrightarrow{x'_j} \\ and\ j : n+1, ..., m;\ k : 1, ..., n \end{aligned} \tag{4.10}$$

4.4.2.2 Composition functions of properties

The composition rules for properties are more complicated than for signatures. The three kinds of properties in FFML, as defined in Section 3.3.2, require specific composition functions, described in Table 4.3. Function Com_props is used for composing the properties of two modules vfm_2 and vfm_1. The function $Com\_inprops$ is applied for determining the invariant properties and the new properties of the composite module. Another function Com_rprops composes the refining properties $vrprops_2$ with $vcprops_1$ in which $vcprops_1$ denotes the set of the properties of vfm_1 (denoted $[\![P_1]\!]$, see Equation 4.2).

We consider in detail the function $Com\_inprops$, applied for both invariant and new properties, presented in Table 4.4. The function provides the rules for reproducing these properties into a composite module. A function Up_ptype, used in $Com\_inprops$, renames the module type into the new one.

Function	Type	Definition	Note
Com_props	$SPEC * FM * FM \rightarrow PROP\ list$	$Com\_props\ (vspec, vfm_2\ vfm_1) =$ $Com\_inprops\ (vname_2, vname_{21}, viprops_2\ @\ vnprops_2)$ $@\ Com\_rprops\ (vspec, vname_1, vname_2, vname_{21}, vrprops_2, vcprops_1)$	- composes the properties of vfm_2 and vfm_2; **Where** $vname_1$ - name of vfm_1; $vname_2$ - name of vfm_2; $vname_{21} = vname_2\ ^\wedge\ vname_1$; $vcprops_1 = Get\_cprops\ vspec\ vfm_1$ - all properties of vfm_1 ($[\![P_1]\!]$ in Equation 4.2); $viprops_2 = Get\_iprops\ vfm_2$ - invariant properties of vfm_2; $vnprops_2 = Get\_nprops\ vfm_2$ - new properties of vfm_2; $vrprops_2 = Get\_rprops\ vfm_2$ - refining properties of vfm_2;

Table 4.3: Composition of properties

Function	Type	Definition	Note
$Com\_inprops$	$ID * ID * PROP\ list \rightarrow PROP\ list$	$Com\_inprops\ (vname_2, vname_{21}, vinprops_2) = Map\ (Up\_ptype\ (vname_2, vname_{21}))\ vinprops_2$	- composes invariant/new properties;
Up_ptype	$ID * ID \rightarrow PROP \rightarrow PROP$	$Up\_ptype\ (vname_2, vname_{21})\ vprop_2$	- updates the module type $vname_2$ of the property $vprop_2$ to $vname_{21}$;

Table 4.4: Composition of invariant and new properties

We demonstrate an example of a property $dl\_upd\_succ\_gr\_wlimit$ of module DL in Listing 22 that is a new one in the module DL. The property specifies that the total of withdrawn money must be always greater than a limit (negative number). After composing, the property is reproduced into the composite module $DLLL$ (line 2 of Listing 4.9) in which module type DL attended in the property is replaced by the module type $DLLL$ using the function Up_type_expr.

Another function Com_rprops, used in Com_props (Table 4.3), for composing refining properties is presented in Table 4.5. The function takes the refining properties $vrprops_2$ of module vfm_2 and the set of properties $vprops_1$ of vfm_1 as its inputs. The function Fi_prop is used to find for each pair of properties $vrprop_2$ (in $vrprops_2$) and $vprop_1$ (in $vrprops_1$), the property p refined by both of them. If this property p exists, then function Up_rprop composes these two properties into a composite refining property by updating $vrprop_2$. This function is an implementation of Equation 4.9, in which the composite property generated refines property named $vpname_1$ of module vfm_1. If the function

Listing 22 Example of composing new property case

```
1 fmodule LL from BA
2 ...;;
```

Listing 4.7 Module *LL*

```
1 fmodule DL from BA
2  contract update :: property
3  dl_upd_succ_gr_wlimit: all x : DL, all a :
       int, ((a <= 0) /\ (get_with(x) + a >=
       limit_with)) -> (get_with(update(x,a)) =
        get_with (x) + a );
4 ...;;
```

Listing 4.8 Module *DL*

```
1 fmodule DLLL from LL
2  contract update ::  property
3  dl_upd_succ_gr_wlimit : all x:DLLL, all a:int
       , ((a <= 0) /\ (get_with(x) + a >=
       limit_with)) -> (get_with(update(x,a)) =
        get_with(x) + a);
4 ...;;
```

Listing 4.9 Module *DLLL*

Fi_prop can not find such a property p, the composite property is established by function *Cr_rprop*. This function is an implementation of Equation 4.10 which creates a composite property refining itself with the premise $vprem_2$ of property $vprop_2$.

The detail of the function *Up_rprop* is represented in the second row of Table 4.5. The composite property, calculated by Equation 4.9, is a refining property which has a premise $vprem_2$ and refines the property $vpname_1$. To illustrate, we show an example of the function *Up_rprop* in Listing 23. For the function *update*, the property *ll_upd_succ_with_wlimit_R1* in module *LL* and *dl_upd_succ_with_llimit_R1* in module *DL* refine the same property *ba_upd_succ_with_over* of module BA (line 3 of Listing 4.10 and line 3 of Listing 4.11). They are composed into a composite property *dl_upd_succ_with_wlimit_R1* of module *DLLL* that refines property *ll_up_succ_with_llimit_R1* in module *LL* (lines 2-6 of Listing 4.12).

Listing 23 Example of composing two refining properties

```
1 fmodule LL from BA
2  contract update :: property
       ll_upd_succ_with_llimit_R1
3    refines BA!ba_upd_succ_with_over
4    extends premise
5    ((a >= 0) || (a <= limit_low));
6 ...;;
```

Listing 4.10 Module *LL*

```
1 fmodule DL from BA
2  contract update :: property
       dl_upd_succ_with_wlimit_R1
3    refines BA!ba_upd_succ_with_over
4    extends premise
5    (a <= 0)/\(get_with(x) + a >= limit_with);
6 ...;;
```

Listing 4.11 Module *DL*

```
1 fmodule DLLL from LL
2  contract update ::  property
3  dl_upd_succ_with_wlimit_R1
4    refines LL!ll_upd_succ_with_llimit_R1
5    extends premise
6   (a <= 0)/\(get_with(x) + a >= limit_with);
7 ...;;
```

Listing 4.12 Composite module *DLLL*

Function	Type	Definition	Note
Com_rprops	$SPEC * ID * ID * ID * PROP\ list * PROP\ list \rightarrow PROP\ list$	$Com\_rprops\ (spec, vname_1, vname_2, vname_{21}, vrprops_2, vcprops_1) =$ $p = Fi\_prop\ (vspec, vrpop_2, vcprop_1)$; **if** p **exists then** $Up\_ptype\ (vname_2, vname_{21})\ vcom\_props$ **else** $Up\_ptype\ (vname_2, vname_{21})\ vcom\_props^*$;	- composes refining properties; **Where** $vrprop_2 \in vrprops_2$; $vcprop_1 \in cprops_1$; Fi_prop - finds p which is refined by both $vrprop_2$ and $vcprop_1$; $vcom\_prop = Up\_rprop\ (vname_1, vrprop_2, vcprop_1)$ - the property denoted in Equation 4.9; $vcom\_prop^* = Cr\_rprop\ (vfname, vname_1, vrprop_2, vcprop_1)$ - the property denoted in Equation 4.10; ($vcom\_pros$ and $vcom\_prop^*$ will be related to proofs in Table 4.12 of Section 4.4.5)
Up_rprop	$ID * PROP * PROP \rightarrow PROP$	$Up\_rprop\ (vname_1, vrprop_2, vcprop_1) = (vfname, vrpname_2, vname_1, vcpname_1, vrprem_2)$	- updates refining property according to Equation 4.9; **Where**: $vfname$ - is method name; $vrpname_2$ - name of $vrprop_2$; $vcpname_1$ - name of $vcprop_1$; $vrprem_2 = Get\_prem\ vrprop_2$; Get_prem - gets the premise;
Cr_rprop	$ID * PROP * PROP \rightarrow PROP$	$Cr\_rprop\ (vname_1, vrprop_2, vcprop_1) = (vfname, vcpname_1, vname_1, vcpname_1, vrprem_2)$	- creates refining property according to Equation 4.10.

Table 4.5: Composition rules of refining properties

In the third row of Table 4.5, we define the function Cr_rprop. Using Equation 4.10, the composite property is built as a refining property that keeps the name $vpname_1$, refines the property $vpname_1$ and adds a premise $vprem_2$.

An example of implementation of the function Cr_rprop is described in Listing 24. Property $ll\_upd\_nosucc\_with\_ls\_llimit$ is the new one in module LL (lines 2-4 of Listing 4.13) and is used to create the composite property $ll\_upd\_nosucc\_with\_ls\_llimit\_C1$ for module $DLLL$ (lines 2-5 of Listing 4.15). This property references itself to the property of module LL (line 4 of Listing 4.15) and is extended by a new premise $(a <= 0)/\backslash(get\_with(x) + a >= limit\_with)$ (line 4 of Listing 4.15) coming from a property of function $update$ in module DL (line 5 of Listing 4.14).

4.4.2.3 Feature Interaction

Listing 24 Example of composing new property

```
fmodule LL from BA
 contract update :: property
 ll_upd_nosucc_with_ls_llimit :
 all x : LL, all a : int, (a < 0) && (a >
      limit_low) -> get_bal (update(x, a)) =
      get_bal(x);
```

Listing 4.13 Module *LL*

```
fmodule DL from BA
 contract update :: property ...
  refines ...
  extends premise
  (a <= 0)/\(get_with(x) + a >= limit_with);
...;;
```

Listing 4.14 Module *DL*

```
fmodule DLLL from LL
 contract update ::  property
ll_upd_nosucc_with_ls_llimit_C1
 refines LL!ll_upd_nosucc_with_ls_llimit
 extends premise (a <= 0)/\(get_with(x) + a >=
      limit_with);
...;;
```

Listing 4.15 Module *DLLL*

Let us assume the feature model as in Figure 4.3. In the feature F, a function f is introduced with a property P (not invariant one) which is refined both in the module F_1 and the module F_2. *P_F_1* and *P_F_2* in Listing 4.16 and 4.17 are respectively properties in the modules F_1 and F_2. So then, in the composite module F_2F_1, the composite property is established as in Listing 4.18. When translating to FoCaLiZe, the statement of this composite property will have both $x < 0$ and $x > 0$ as its premises. It is a case of feature interaction.

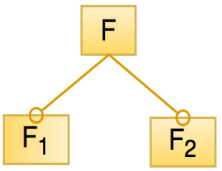

Figure 4.3: Feature diagram 1

```
fmodule F1
contract f :: property P_F_1
 refines F!P
 extends premise x > 0;
```

Listing 4.16 Module F_1

```
fmodule F2
contract f :: property P_F_2
 refines F!P
 extends premise x < 0;
```

Listing 4.17 Module F_2

```
fmodule F2F1 from F1
contract f :: property P_F_2
 refines F1!P_F1
 extends premise x < 0;
```

Listing 4.18 Composite module F_2F_1

In our approach, we assume that this case should be forbidden by the feature model. It should be in Figure 4.4, in which it is possible to select one of the features F_1 and F_2 but not both. So with this diagram, the configuration $\{F, F_1, F_2\}$ is not valid. We rely on the hypothesis that all the configurations allowed by a feature model do not lead to interactions of that kind.

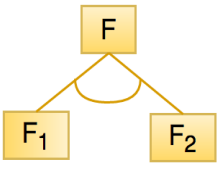

Figure 4.4: Feature diagram 2

Another problematic would be to detect such interaction and thus helps to design a good feature model. But this is out of the scope of our thesis.

4.4.3 Representation Type

Function	Type	Definition	Note
Rpr_rep	$ID * REP \rightarrow REP$	$Rpr\_rep\ (vname_1, rep_2) = Join\_prod\ (vname_1, vext_2)$	- composes representation types; **Where:** $vext_2$ - is the extension part of rep_2; $Join\_prod$ - makes a Cartesian product of $vname_1$ and $vext_2$;

Table 4.6: Composition of representation types

The principle for composing two representation types is that the composite representation type refers to the module name $vname_1$ meaning that the composite representation type is extended from that of module vfm_1 (presented in Table 4.6): the extension part $vext_2$ of rep_2 is copied. The extension can be empty, in this case it means that the module keeps the representation type of its parent. Following the semantics of FFML, this composition will be a Cartesian product of $vname_1$ and $vext_2$ that is implemented by function $Join\_prod$.

Listing 25 Example of composing representation types

```
1 fmodule LL from BA
2  representation = BA;
3 ...
```

Listing 4.19 Module *LL*

```
1 fmodule DL from BA
2  representation extends BA with int;
3 ...
```

Listing 4.20 Module *DL*

```
1 fmodule DLLL from LL
2  representation extends LL with int;
3 ...
```

Listing 4.21 Module *DLLL*

As discussed before in the previous section, the change of a representation type affects the code artifacts in the composite module. However, within the scope of our work, FFML covers all the problems automatically. We show an example in Listing 25 in which the representation type of composite module DLLL (Listing 4.21) is composed from those of module DL and LL. Instead of extending BA as in module DL, the composite representation type is constructed from LL and the extension int.

4.4.4 Function Definition

Function	Type	Definition	Note
Com_funcs	$SPEC * FM * FM \rightarrow FUNC\ list$	$Com\_funcs\ (vspec, vfm_2\ vfm_1) = Com\_nfuncs\ (vname_2, vname_{21}, vnfuncs_2)$ @ $Com\_rfuncs\ (vspec, vname_1, vname_2, vname_{21}, vrfuncs_2, vcfuncs_1)$	- composes function definitions. Where $vname_{21} = vname_2\ ^\wedge\ vname_1$ - combined name of vfm_2 and vfm_1; $vnfuncs_2 = Get\_nfuncs\ vfm_2$ - new definitions of vfm_2; $vrfuncs_2 = Get\_rfuncs\ vfm_2$ - re-definitions of vfm_2; $vcfuncs_1 = Get\_cfuncs\ vspec\ vfm_1$ - all (re)definitions of vfm_1 ($[\![D_1]\!]$ in Equation 4.6);

Table 4.7: Composition of (re)definitions

The rules for composing function (re)definitions are explained in Table 4.7. Similarly to property composition, the composition proceeds in two steps. The function Com_nfuncs is used to reproduce new definitions $vnfuncs_2$ into the composite module while another function Com_rfuncs is used to compose (re)definitions $vrfuncs_2$ of vfm_2 and $vrfuncs_1$ of vfm_1.

Function	Type	Definition	Note
Com_nfuncs	$ID * ID * FUNC\ list \rightarrow FUNC\ list$	$Com\_nfunc\ (vname_2, vname_{21}, vnfuncs_2) = Map\ (Up\_dtype\ (vname_2\ vname_{21}))\ nfuncs_2$	- composes new definitions.
Up_dtype	$ID * ID \rightarrow FUNC \rightarrow FUNC$	$Up\_dtype\ (vname_2, vname_{21})\ vfunc_2$	- updates module type.

Table 4.8: Composition of new definitions

The function Com_nfuncs moves the new definitions $vnfuncs_2$ of module vfm_2 into the composition module, presented in Table 4.8. Module type $vname_2$ must be updated to the module type $vname_{21}$ of the composite one by a function Up_dtype. We illustrate this case with a function get_with in Listing 26 whose definition is in line 2 of Listing 4.23. It is copied in the composite module $DLLL$ (Listing 4.24).

We continue with the function Com_rfuncs (Table 4.7) that allows the composition of function re-definitions. The details of the function are described in Table 4.9. The composition aims at establishing a composite re-definition from $vrfunc_2$ and refers to $vrfunc_1$ whenever both $vrfunc_2$ and $vrfunc_1$ are re-definitions of the same function. In

Listing 26 Example of composing new definitions

```
1 fmodule LL from BA
2   let limit_low = (-10);
3 ... ;
```

Listing 4.22 Definitions of module LL

```
1 fmodule DL from BA
2   let get_with (x) = second(x);
3 ... ;
```

Listing 4.23 Definitions of module BA

```
1 fmodule DLLL from LL
2   let get_with (x) = second(x);
3 ... ;
```

Listing 4.24 Definitions of module LL

that case they share the same name and the same type (since we have no overload in FFML). Function Up_rfunc updates $vrfunc_2$ in the second row of the table. By mentioning the module named $vname_1$, FFML Compiler can trace back $vrfunc_1$ in vfm_1, whatever there is a re-definition of the function in vfm_1 or not. In another case, when both the $vrfunc_2$ and $vrfunc_1$ are re-definitions of different functions, $vrfunc_2$ is kept but updated to the module type $vname_{21}$ (by function Up_dtype).

Function	**Type**	**Definition**	**Note**
Com_rfuncs	$SPEC * ID * ID * ID * FUNC\ list * FUNC\ list \rightarrow FUNC\ list$	$Com\_rfuncs\ (vspec, vname_1, vname_2, vname_{21}, vrfuncs_2, vcfuncs_1) =$ **if** $vrfunc_2, vcfunc_1$ **redefines** the same function **then** $Up\_dtype\ (vname_2, vname_{21})\ (Up\_rfunc\ (vname_1, vrfunc_2))$; **else** $Up\_dtype\ (vname_2, vname_{21})\ vrfunc_2$;	- composes re-definitions; **Where**: $vrfunc_2 \in vrfuncs_2$; $vcfunc_1 \in vcfuncs_1$;
Up_rfunc	$ID * FUNC \rightarrow FUNC$	$Up\_rfunc\ (vname_1, vrfunc_2) = (vfname, vpars, vname_1, vexpr_2)$	- updates re-definition. **Where**: $vfname$ - is function name of $vrfunc_2$; $vpars_2$ - are function parameters of $vrfunc_2$; $vexpr_2$ - the expression of $vrfunc_2$;

Table 4.9: Composition of redefinitions

Let us describe an example of composing the re-definitions of function *update* in Listing 27. The function is redefined in module LL (Listing 4.25) and in module DL (lines 2-7 of Listing 4.26). The redefinition in DL is moved into the composite module $DLLL$ but refers to the implementation of module LL instead of module BA (lines 2-7 of Listing 4.27).

Listing 27 Example of composing re-definitions

```
1  fmodule LL from BA
2   let update (x, a) =
3    if ((a >= 0) || (a <= limit_low)) then BA!
         update (x, a)
4    else x;
5  ... ;
```

Listing 4.25 Module *LL*

```
1  fmodule DL from BA
2   let update (x, a) =
3    if (a <= 0) then
4      if (get with(x) + a >= limit_with) then
5     (BA!update (x,a), get_with(x) + a)
6   else x
7    else (BA!update(x,a), get_with(x));
8  ... ;
```

Listing 4.26 Module *DL*

```
1  fmodule DLLL from LL
2   let update (x, a) =
3    if (a <= 0) then
4      if (get with(x) + a >= limit_with) then
5     (LL!update (x,a), get_with(x) + a)
6   else x
7    else (LL!update(x,a), get_with(x));
8  ... ;
```

Listing 4.27 Module *DLLL*

4.4.5 Correctness Proof

A correctness proof is written to prove that an implementation satisfies a certain property. Hence, before composing the correctness proofs we have to consider the properties related to these proofs. We start by considering the way these properties are composed, hence map to the composition cases of the proofs.

The main function Com_proofs for composing the correctness proofs of vfm_2 and vfm_1 is presented in Table 4.10. As described in Section 4.4.2 about the composition rules for properties, we distinguish the proofs $viproof_2$ and $vnproof_2$ written respectively for invariant properties ($viprop_2$) and new properties ($vnprop_2$), and the proofs $vrproofs_2$ written for refining properties ($vrprop_2$) (see Table 4.3). The function $Com\_inproofs$ is used for copying the proofs $viproofs_2$ and $vnproof_2$. The proofs $vrproofs_2$ is used for composing $vrproof_2$ with $vcproof_1$, in which $vcproof_2$ is the set of all proofs of vfm_1 that is denoted $[\![Pf_1]\!]$ (see Equation 4.7).

The detail of function $Com\_inproofs$ is presented in Table 4.11, this function is applied for the proofs $vinproofs$ (a union of $viproofs_2$ and $vnproofs_2$). As usually the types are updated in these proofs (by the function Up_pftype) and copied into the composite module.

$Com\_rproofs$ for composing the proofs $vrproofs$ is more complex than $Com\_inproofs$ because of its relationship with the composite properties established from Table 4.5. Similarly, we distinguish two cases, belonging to the property named $vpname$ is found or not.

Function	Type	Definition	Note
Com_proofs	$SPEC * FM * FM \rightarrow PROOF\ list$	$Com\_proofs\ (vspec, vfm_2\ vfm_1) =$ $Com\_inproofs\ (vname_2, vname_{21}, viproofs_2$ @ $vnproofs_2)$ @ $Com\_rproofs\ (spec, vname_1, vname_2, vname_{21}, vrproofs_2, vcproofs_1)$	- composes the proofs of vfm_2 and vfm_1; **Where** $vname_{21} = vname_2\ ^\wedge\ vname_1$ - combined name of vfm_2 and vfm_1; $vcproofs_1 = Get\_cproofs\ vspec\ vfm_1$ - all the proofs of vfm_1 ($[\![Pf_1]\!]$ in Equation 4.7); $viproofs_2 = Get\_iproofs\ vfm_2$ - proofs of invariant properties of vfm_2; $vnprops_2 = Get\_nprops\ vfm_2$ - proofs of new properties of vfm_2; $vrprops_2 = Get\_rprops\ vfm_2$ - proofs of refining properties of vfm_2;

Table 4.10: Composition of correctness proofs

Function	Type	Definition	Note
$Com\_inproofs$	$ID * ID * PROOF\ list \rightarrow PROOF\ list$	$Com\_inproof\ (vname_2, vname_{21}, vinproofs_2) = Map\ (Up\_pftype\ (vname_2, vname_{21}))\ vinproofs_2$	- composes proofs of invariant and new properties;
Up_pftype	$ID * ID \rightarrow PROOF \rightarrow PROOF$	$Up\_pftype\ (vname_2, vname_{21})\ vproof_2$	- updates the module type $vname_2$ of the proof $vproof_2$ to $vname_{21}$;

Table 4.11: Composition of correctness proofs of invariant and new properties

$vpname$ is the name of the property that is refined by two properties which are proved respectively by $vrproof_2$ and $vcproof_1$. The function Fi_pname determines the existence of $vpname$, it is similar to the function Fi_prop in Table 4.5.

We show the details of the function $Com\_rproofs$ in Table 4.12. In the case when $pname$ exits, the proof $vcom\_proof$, corresponding to the property $vcom\_prop$ (in Equation 4.9 and defined in Table 4.5), is calculated by the function Up_rproof. The types in $vcom\_proof$ are updated using the function Up_ptype, the proofs are then copied into the composite module.

Another case when $vpname$ does not exist, the two proofs $vrproof_2$ and $vcproof_1$ prove two different properties. The proof $vcom\_proof$, corresponding to $vcom\_prop^*$ in Equation 4.10 and defined in Table 4.5, is created by the function Cr_rproof.

To update the proof $vrproof_2$, we should give more information for function Up_rproof

Function	Type	Definition	Note
$Com\_rproofs$	$SPEC * ID * ID * ID * PROOF\ list * PROOF\ list \rightarrow PROOF\ list$	$Com\_rproofs\ (spec, vname_1, vname_2, vname_{21}, vrproofs_2, vcproofs_1) =$ $vpname = Fi\_pname\ (vspec, vrpoof_2, vcproof_1);$ **if** $vpname$ **exists then** $Up\_pftype\ (vname_2, vname_{21})$ $vcom\_proof;$ **else** $vpname$ **does not exist** $Up\_fptype\ (vname_2, vname_{21})$ $vcom\_proof^*;$	- composes the proofs of refining properties; **Where**: $vcproof_1 \in vcproofs_1$; $vrproof_2 \in vrproofs_2$; $vcpname_1$ - name of property $vcprop_1$; $vcprem_1$ - premise of property $vcprop_1$; Fi_pname - finds the same property refined by two properties which are proved respectively by $vrproof_2$ and $vcproof_1$; $vcom\_proof = Up\_rproof\ (vname_1, vpname, vcpname_1, vcprem_1, vcprop_1, vrproof_2)$ - the proof written for the composite property denoted as $vcom\_prop$ in Equation 4.9 and defined in Table 4.5; $vcom\_proof^* = Cr\_rproof\ (vname_1, vcpname, vcprop_1, vcprem_2)$ - the proofs written for the composite properties denoted as $vcom\_prop^*$ in Equation 4.10 and defined in Table 4.5; $vcprem_2$ - premise of property $vcom\_prop^*$;

Table 4.12: Composition rules of the correctness proofs of refining properties

such as, the premise $vcprem_1$ of the property $vcprop_1$ of proof $vcproof_1$, represented in Table 4.13. Our generator will update the proof with a $vcprem_1$ because of the update in composite property $vcom\_prop$, using function Up_prem. The property hints for the composite proof must be updated, i.e. property name $vpname$ (the same refined property) is replaced by $vcpname_1$, the property from vfm_1 (by the function Up_phint). In addition, when the composite property refers to $vcprop_1$ as its refined property, the functions are present in this refined property to be indicated in $vcom\_proof$ as the definition hints. The supplementary functions such as, $first$, $second$, $make$ are also mentioned as definition hints if they are present in $vrproof_2$. These definition hints are added by the function Ad_dhint. All updated information is necessary to give enough hints for Zenon Prover to automatically prove. But Zenon may fail because he lacks proof hints. So the process may

Function	Type	Definition	Note
Up_rpoof	$ID * ID * ID * PREM * PROP * PROOF \rightarrow PROOF$	Up_rproof ($vname_1$, $vpname$, $vcpname_1$, $vcprem_1$, $vcprop_1$, $vrproof_2$)	- updates proof. The involved functions: Up_prem - updates $vcprem_1$ into $vrproof_2$; Up_phint - replaces property hint $vpname$ by $vcpname_1$ into $vrproof_2$; Ad_dhint - adds definition hints: the redefined function names (attended in $vcprop_1$) or/and the supplementary functions ($first$, $second$, $make$) into $vrproof_2$;

Table 4.13: Updating correctness proof

require some manual help to have the proofs done by Zenon.

Example. The composition of two proofs of two refining properties is illustrated in Listing 28. Following the property composition rules, property $dl\_upd\_succ\_with\_wlimit\_R1$ in module DL is composed with property $ll\_upd\_succ\_with\_llimit\_R1$ in module LL into a refining one in module DLLL (lines 6-10 of Listing 4.30). It refines property $ll\_upd\_succ\_with\_llimit\_R1$. With the establishment of this composite property, we generate the composite proof for it in lines 17-28 of Listing 4.30. The premise of the property in module LL (line 6 of Listing 4.28) is updated into the composite proof in line 22 of Listing 4.30. The name of function $limit\_low$ is added in line 27 of the Listing 4.30 to let Zenon know that it is redefined in the composite module. The module parameter LL (lines 23 and 27 of Listing 4.30) replaces BA in the proof of module DL (lines 14 and 16 of Listing 4.29). Finally, property $ll\_upd\_succ\_with\_llimit\_R1$ in line 27 of Listing 4.30 is a property proof hint updated for the composite poof by replacing property $ba\_upd\_succ\_with\_over$ in line 16 of Listing 4.29.

Function	Type	Definition	Note
Cr_rpoof	$ID * ID * PROP * PREM \rightarrow PROOF$	Cr_rproof ($vname_1$, $vcpname_1$, $vcprop_1$, $vcprem_2$)	- creates proof. The involved functions: Cr_phint - creates property hint $vcpname_1$; Cr_dhint - creates definition hints: the redefined function names (attended in $vcprem_2$ and $vcprop_1$) or/and the supplementary functions ($first$, $second$, $make$);

Table 4.14: Creating correctness proof

Listing 28 Example of composing proofs (1)

```
1  fmodule LL from BA
2   contract update :: property
        ll_upd_succ_with_llimit_R1
3    refines BA!ba_upd_succ_with_over
4    with all x : BA, all a : int,
5    extends premise
6     ((a >= 0) || (a <= limit_low));
7
8   proof of ll_upd_succ_with_llimit_R1 = ...;
9  ...;;
```

Listing 4.28 Module *LL*

```
1  fmodule DL from BA
2   contract update :: property
3   dl_upd_succ_with_wlimit_R1
4    refines BA!ba_upd_succ_with_over
5    extends premise
6    (a <= 0)/\(get_with(x) + a >= limit_with);
7
8   proof of dl_upd_succ_with_wlimit_R1 =
9   foc proof {*
10   <1>1 assume x: DL, a : int,
11   hypothesis h1: (a <= 0) /\
12    (get_with(x) + a >= limit_with),
13   prove (get_bal(x) + a) >= over ->
14    (get_bal (update(x,a))) = get_bal(x) + a
15    <2>1 prove get1st(update(x,a)) = BA!update(
          get1st(x),a)
16    by definition of get1st,update hypothesis
          h1
17    <2>e qed by step <2>1 definition of over,
          get_bal property
          BA!ba_upd_succ_with_over
18  <1>e conclude;*}
19 ...;;
```

Listing 4.29 Module *DL*

```
1
2
3
4  fmodule DLLL from LL
5
6   contract update :: property
7    dl_upd_succ_with_wlimit_R1
8    refines LL!ll_upd_succ_with_llimit_R1
9    extends premise
10   (a <= 0)/\(get_with(x) + a >= limit_with);
11
12
13
14
15
16 // automatically generated proof
17  proof of dl_upd_succ_with_wlimit_R1 =
18  foc proof {*
19   <1>1 assume x : Self, assume a : int,
20   hypothesis h1 : (a <= 0) /\
21    (get_with(x) + a >= limit_with),
22   prove ((a >= 0) || (a <= limit_low)) ->
23    (get_bal(x) + a) >= over ->
24    (get_bal(update(x,a))) = get_bal(x) + a
25    <2>1 prove get1st(update(x,a)) = LL!update(
          get1st(x),a)
26    by definition of get1st,update hypothesis
          h1
27    <2>e qed by step <2>1 definition of over,
          get_bal definition of limit_low
          property
28    LL!ll_upd_succ_with_llimit_R1
29  <1>e conclude;*}
30 ... ;
```

Listing 4.30 Module *DLLL*

The detail of function Cr_rproof to create the proof $vcom\_proof^*$, is described in Table 4.14. The information such as, the premise $vcprem_2$ which is added to the property $vcom\_prop^*$ must be given as parameter for the function Cr_rproof. FFML Generator will create $vcom\_proof^*$ by mentioning $vcprem_2$ and adding some hints into it. The property $vcprop_1$ must be indicated as a property hint of the composite proof because of $vcom\_prop^*$ refers to this property (done by the function Cr_phint). In addition, the function present in $vcprem_2$ or $vcprop_1$ must be added as definition hints, using the function Cr_dhint. The supplementary functions are also inserted if they are used in $vcprem_2$ or $vcprop_1$.

Example. In the Listing 29, the automated property $ll\_upd\_nosucc\_ls\_llimit\_C1$ is a $vprop_1^*$ in DLLL (line 2-6 of Listing 4.33). The property refines property $ll\_upd\_nosucc\_ls\_llimit$ of module LL (lines 2-4 of Listing 4.31) and is extended by adding a premise

Listing 29 Example of Composing Correctness roofs (2)

```
1 fmodule LL from BA
2  contract update :: property
3  ll_upd_nosucc_with_ls_llimit :
4  all x : LL, all a : int, (a < 0) && (a >
       limit_low) -> get_bal (update(x, a)) =
       get_bal(x);
5
6  proof of ll_upd_nosucc_with_ls_llimit = ...;
7 ...;;
```

Listing 4.31 Module *LL*

```
1 fmodule DL from BA
2  contract update ::  property ...
3   extends premise
4   (a <= 0)/\(get_with(x) + a >= limit_with);
5 ...;;
```

Listing 4.32 Module *DL*

```
1  fmodule DLLL from LL
2   contract update ::  property
3    ll_upd_nosucc_with_ls_llimit_C1
4    refines LL!ll_upd_nosucc_with_ls_llimit
5    extends premise
6    (a <= 0)/\(get_with(x) + a >= limit_with);
7
8  // automatically generated proof
9   proof of ll_upd_nosucc_with_ls_llimit_C1 =
10  foc proof {*
11  <1>1 assume x : Self, assume a : int,
12  hypothesis h1 : (a <= 0) /\ (get_with(x) + a
        >= limit_with),
13  prove (a < 0) && (a > limit_low) -> get_bal(
        update(x,a)) = get_bal(x)
14   <2>1 prove first(update(x,a)) = LL!update(
         first(x),a)
15   by definition of update, first hypothesis h1
16   <2>e qed by step <2>1 definition of over,
         get_bal, limit_low, update
17   property
18   LL!ll_upd_nosucc_with_ls_llimit
19  <1>e conclude;*}
20 ...;;
```

Listing 4.33 Module *DLLL*

in module DL (line 4 of Listing 4.32). Referring to *ll_upd_nosucc_ls_llimit* of module LL as property hint is generated automatically by the function *Cr_phint* in Table 4.14. The function names *over*, *get_bal*, *update*, *limit_low*, are labeled as the definition hints by the function *Cr_dhint*.

4.4.6 Properties of the Binary Composition "•"

There are two basic properties of our binary composition operation, namely associativity and identity, defined as Conjecture 1 and Property 1.

Conjecture 1 (Associativity) *Given three modules FM_3, FM_2 and FM_1, two binary compositions, $(FM_3 \bullet FM_2) \bullet FM_1$ and $FM_3 \bullet (FM_2 \bullet FM_1)$ are equivalent, denoted by $(FM_3 \bullet FM_2) \bullet FM_1 \equiv FM_3 \bullet (FM_2 \bullet FM_1)$ where $\equiv$ denotes semantic equivalence.*

Property 1 (Identity) *If FM_1 refers FM_2 through the "from" relation (i.e, $FM_1 = (\_, N, \_, \_, \_, \_, \_)$ and $FM_2 = (N, \_, \_, \_, \_, \_, \_)$) then $FM_1 \bullet FM_2 \equiv FM_1$, where N is the module name of FM_2.*

Proof. The proof of Property 1 is trivial by the definitions of the composition rules.

Example. On the Bank Account SPL, applying the property of identity we can deduce

the following equalities:

$$DL \bullet BA = DL$$

$$CE \bullet CU = CE$$

4.5 Generating Final Products

As explained in the first section of this chapter, establishing the product variant from a feature model and a configuration requires the composition of all related modules of the module diagram associated to the configuration. The composition of two modules is computed by the binary composition operation, which is defined in Section 4.2 and defined formally by the related composition rules in Section 4.4. However, to compose all the related modules, we need now construction rules based on the feature diagram (such as the the order of the modules and the structure design of the feature diagram). We call this process the module diagram-based composition. In this section, we discuss how the final products are generated based on the binary operation and the module diagram-based composition process.

Module diagram-based composition. The module diagram-based composition (MDC for short) is a recursive process that uses the binary composition previously defined to compose modules and composite modules. So roughly speaking we compose two by two the modules, we impose an order for composing the modules of a module diagram to establish a final product is necessary. This order corresponds to a particular traversal of the module diagram (which is a tree obtained by pruning some branches of the initial feature model, which is also a tree): a *left-right-root* traversal (this term is a little bit

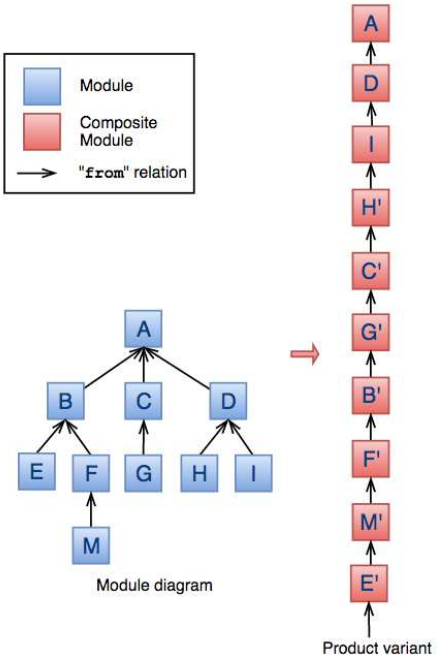

Figure 4.5: Example of a Product Generation

abusive because the tree is not binary but n-ary). This traversal produces a chain of pairwise composition, as illustrated on Figure 4.5. In this figure the arrow in the chain denotes the from relation and the prime nodes correspond to composite modules. For example, E', the expected final product results from the binary composition of the module E (taken in the asset database) and the composite module M', which is the result of the binary composition of the M and F' and so on, until reaching so top of the chain. So, the computation for E' is $E \bullet (M \bullet (F \bullet (B \bullet (G \bullet (D \bullet (H \bullet (I \bullet (D \bullet A)))))))))$. According to the identity property 1, $D \bullet A$ is D and that $I \bullet D$ is I, the computation becomes $E \bullet (M \bullet (F \bullet (B \bullet (G \bullet (D \bullet (H \bullet I))))))$.

4.6 Application on the Bank Account SPL

The FFML Product Generator tool is written in OCaml with about a thousand lines of code. The tool allows a user to select a configuration as input and returns the corresponding final product (as an FFML bunch of files, called an FFML product).

Modules	**FFML**				**FoCaLiZe**		
	iP	**nP**	**rP**	**Pf**	**P**	**Ze-Pf**	**reuse**
BA	1	3	0	4	4	4	0
DL	0	2	2	5	5	6	1
LL	0	1	2	3	4	5	2
CU	0	1	0	4	4	5	1
EX	0	2	0	2	6	7	5
iP - invariant properties **nP** - new properties **rP** - refining properties **Pf** - proofs					**P** - properties **Ze-Pf** - proofs are done **reuse** - proofs are reused		

Table 4.15: Modules of Bank Account SPL

The Bank Account SPL has been analyzed with five features and developed into the five corresponding modules (BA, DL, LL, CU and EX) by about 400 LOC in FFML. We summarize this product line in Table 4.15. The left side of the table contains qualitative information about the product line in FFML and the right side concerns the corresponding compiled FoCaLiZe code. In all the FFML files, we count one invariant property, nine new properties, four refining properties and eighteen written proofs in all FFML modules.

Configurations	FFML				FoCaLiZe				
	uP	**uPf**	**cP**	**cPf**	**P**	**Ze-Pf**	**reuse**	**auto**	**manu**
{BA,DL,LL}	2	2	3	4	6	7	1	7	0
{BA,DL,CU}	2	2	2	3	6	7	2	6	1
{BA,LL,CU}	1	1	2	2	5	6	3	6	0
{BA,DL,LL,CU}	2	2	3	4	7	8	2	7	1
{BA,LL,CU,EX}	1	1	2	2	7	8	5	7	1
{BA,DL,CU,EX}	1	1	2	2	8	9	1	7	2
{BA,DL,LL,CU,EX}	2	2	3	4	9	10	1	8	2
{BA},{BA,DL},{BA,LL},{BA,CU} and {BA,CU,EX} are in Table 4.15									
uP - properties are updated **uPf** - proofs (resp. uP) are updated **cP** - properties are composed **cPf** - proofs (resp. cP) are composed					**P** - properties **Ze-Pf** - proofs are done **reuse** - proofs are reused **auto** - automatic proofs **manu** - manual proofs				

Table 4.16: Bank Account SPL

Compared to the modules in the right side, the numbers of properties **P** and proofs **Ze-Pf** generated in FoCaLiZe are more than in FFML. The properties are reused and their proofs **reuse** are built automatically by the FFML Compiler. There are more proofs than properties because some properties are "inherited" need to be proven again because of some re-definitions.

Using our generator tool, the twelve products of Bank Account SPL have been generated automatically. Some configurations (listed in the last row of the table) are omitted because their code, both in FFML and FoCaLiZe, are just existing modules. For example, according to our composition rules and Property 1, the module corresponding to the configuration $\{BA, CU, EX\}$ is the module EX. In 4.16, we count the different properties, functions and proofs generated by the composition process. These products contains **uP** (properties obtained just by copy and type substitution) and **cP** (properties obtained by composition of other properties). The number of proofs **cPf** which are composed automatically, are more than the composed properties because of the compositions of the reused properties and the corresponding proofs in the modules. The information of the FFML products which are translated into FoCaLiZe, is in the right side. Most of the proofs after composing are done successfully. The proofs **reuse** are reused using the mechanisms of the FFML language. The proofs **auto** are done automatically by Zenon. The proofs **manu** lack some proof

hints. In fact, Zenon fails to find a proof because of internal error - this bug has been reported. However, giving a more detailed proof (with sublemmas) we can bypass this Zenon's bug (see the proofs *cu_upd_cur_succ* in line 37 of Listing A.15 of Appendix). For the moment, the proofs that have been done manually in that case are difficult to generalize and then to generate automatically.

4.7 Summary

In this chapter, we discussed the FFML Product Generation. This generation is based on the composition rules for composing two modules in Section 4.4 and the MDC process for all involved modules of products in Section 4.5. These processes are built independently and could be extended or modified for future purposes. The generation is implemented in the FFML Product Generator tool. We have applied it on the Bank Account example and shown our results in Section 4.6.

The results, obtained in Table 4.15 and Table 4.16, indicates that our methodology is applied successfully for developing Bank Account SPL. In order to check the methodology is effective and realistic, in the next chapter we run a new example, a Poker SPL that is more complex than Bank Account SPL.

Chapter 5

Evaluation

In this chapter we deal with a bigger and more complex example, Poker product line, that is developed using our methodology. We begin by explaining the case study in Section 5.1. The analysis of the modules of Poker SPL is described in Section 5.2. Based on the results obtained from both Bank Account (the running example in the thesis) and Poker product lines, we evaluate our methodology in Section 5.3.

5.1 Case Study: Poker Software Product Line

To evaluate our methodology, we develop an example which is more complex than Bank Account product line with more features and final products. We choose Poker SPL, as described on the website `http://spl2go.cs.ovgu.de/`. However, the instead of the specifications, their function definitions are simple and almost empty. The products are built using FEATUREHOUSE [Apel et al. 2009] and there is no proofs. Because of the absence of necessary information, in this section we decide to build our own Poker SPL by collecting the variants of poker game, summarizing their related rules and then designing a feature model.

To model the Poker SPL, we first define the simple rules: this first game corresponds to the feature BasicPoker, the root of our feature model. Then we define different variants formalized as different optional features, forming the feature model depicted by in Figure 5.1. In the rest of this section, we describe informally the different features.

Poker is a popular card game but having many different gaming rules related to the

number of cards and some special rules. However, in oder to analyze Poker SPL into features, we start with a feature, called BasicPoker, containing the simplest rules. From this feature, we continue to design other features and put them into a feature model that is illustrated in Figure 5.1. Poker SPL is analyzed into twelve features. We study here only seven features. The five remaining will be described in Appendix A.3. BasicPoker contains all the mandatory rules. All others are optional.

Example: Poker Product Line

We developed a Poker product line which with more features and final products than Bank Account. We find this product line on the website `http://spl2go.cs.ovgu.de/`. However, their specifications and function definitions are simple and almost empty. We collected the variants of poker game, summarized their related rules and then designed a feature model as figure 5.1.

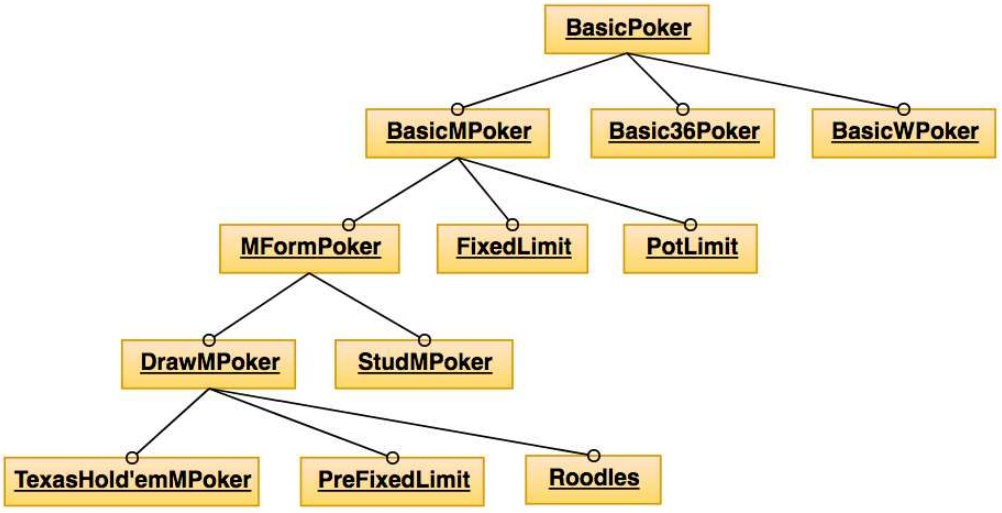

Figure 5.1: Feature Model of Poker SPL

We describes the basic rules that are applicable to all types of poker game into the root feature named BasicPoker (BP). The basic poker game is played with a standard 52-card pack of playing cards. The cards held by players are combined into ranks. The winner is a person who has the highest rank in hand. The root feature as three optional child features BasicMPoker (BMP), Basic36Poker (B36P) and BasicWPoker (BWP).

In the feature BMP, the players can use money (chips) for gambling. The addition rule is that each player put this chips into a pot when he bets. A player can "call" (put into the pot the same number of chips of previous player), "raise" (put in the pot more chips to call), or "drop" ("fold") which means the player put nos chips in the pot and is out of the game. The feature B36P allows players to play with 36 playing cards instead of 52. An addition rule of wild cards (a wild card can be seen as any card) is added into the feature BWP.

The features MFormPoker (MFP), FixedLimit (FL) and PotLimit (PL) are optional child features of BasicMPoker. MFP is added a rule dealing with "community cards" (faced up and all players can used them to combine with their held cards). A rule of each "raise" limitation and another rule relating to chips in pot are designed in FL and PL. DrawMPoker (DMP) and StudMPoker (SMP) are child features of MFP in which DMP contains the rules used popularly that allows an "ante" for the first bet. DMP has two child features TexasHold'emMPoker (THP) and PreFixedLimit (PFL). The features SMP, THP and PreFixedLimit contain other addition rules.

5.2 Analyzing and Developing Poker Software Product Line

Based on the poker rules collected and the feature model designed in the previous section, we analyze and develop the features of Poker SPL into FFML modules in this section. As previously, a module is named according to the feature it corresponds to. We describe their main artifacts and their relationships in a module diagram (Figure 5.2). In each node of the diagram, we list the new functions, the representation type and the redefined functions. The type files are also shown in the figure and they are imported into the modules relating to them. While composing to build a product, the related type files are merged into a type file unified for the product. The full code of all modules and some products of Poker SPL can be found in Appendix A.3.

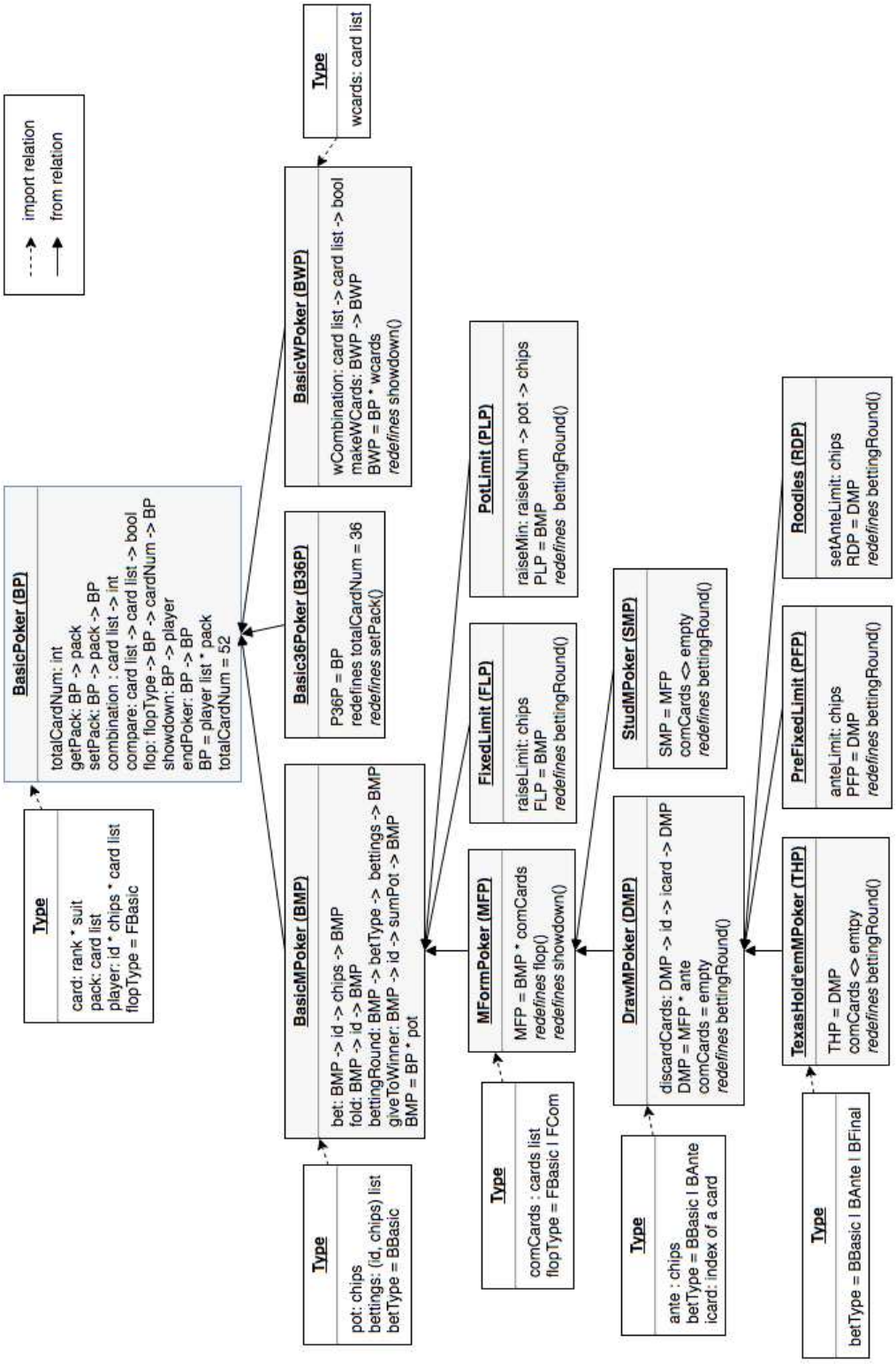

Figure 5.2: Module Diagram of Poker SPL

Module BasicPoker

The module definition is given in Listing A.17 of Appendix A.3.

- The representation type R_0 of the module BasicPoker is denoted by $BP = player\ list * pack$, meaning that the game includes a list of players and a pack (a list of 52 playing cards). A player, defined as $player = id * chips * card\ list$, has an identity *id*, *chips* and a list of hole cards (see the related type file in Figure 5.2). However, in BasicPoker the chips of each player is empty. Total of the cards held by players and put on the table, *totalCardNum*, is always 52.

- The function *getPack* gets the pack of a basic poker game.

  - The invariant property *bp_ getPack_ succ* specifies that the cards set up for the pack of a poker game can be taken out by the function *getPack* (see line 15).

- The function *setPack* sets up a pack of 52 random cards.

  - The invariant property *bp_ setPack_ totalCardNum* specifies that after setting a pack (by calling the function *setPack*), the number of cards in the pack is equal to *totalCardNum*, or the pack is not updated (see line 17).

- The function *combination* takes a list of cards (defined here xwith 5 cards) held by a player as input, counts and returns the rank of the player. The rank is in range from 1 to 9, corresponding to the ranks of poker ranking hands which are listed in the first column of Table A.1.

- The function *compare* is called when two hands have the same highest rank, corresponding to the third column of Table A.1.

- In the BasicPoker, the function *flop* deals cards (sequentially from the pack) to each player at the table with a flop type $FBasic$ (a basic dealing of cards in the module BasicPoker that $cardNum$ cards are dealt for each player; defined in the related type file in Figure 5.2). After running *flop*, each player has $cardNum$ hole cards .

  - The property *bp_flop_minusCards* states that after dealing a number of cards from a pack for all players, the cards in the pack are subtracted by all the dealt cards (see line 19).

- The function *showdown* finds out the winner of the game based on the ranks of the players, determined by the functions *combination* and *compare*.

  - An invariant property *bp_showdown_notWinner* states that if a player drops out the game, he can not be the winner (see line 21).

- The function *endPoker* resets the game.

Module BasicMPoker

The module definition is given in Listing A.18 of Appendix A.3.

- The module BasicMPoker is built from BasicPoker. Its representation type $BMP = BP * pot$, includes R_0 (the representation of BasicPoker) and an extended part *pot* containing all the bet chips of the game.

- The function *getPack* declared in the module BasicPoker is redefined. There is an invariant property added into the module.

  - The invariant property *bmp_getPack_succ* specifies that the cards set up for the pack of a basic poker game with chips can be taken out by the function *getPack* (see line 9).

- The function *bet* allows a player (with an identity *id*) to bet with an amount of chips.

  - The property *bmp_bet_upd_pot* describes that after a player makes a bet, the chips added into the *pot* are removed from the player (see line 11).

- The function *fold* allows a player to drop out of the game. This player will loose the game and not be taken into the showdown process.

- The function *bettingRound* allows players to bet with their chips sequentially. All of the bet chips are put into the *pot*. A basic bet type, defined by $betType = BBasic$ (see the related type file in Figure 5.2), determines a basic betting round which allows players to *call* or *raise*. The *bettingRound* finishes when all players at the table make "call".

  - The property *bmp_bettingground_upd_pot* defines that after a betting round, the bet chips taken out from all the players must be put into the *pot* (see line 13).

- The function *giveToWinner* is called after finding the winner by the function *showdown* (reused from BasicPoker). All the chips in the *pot* will be given to the winner of the game.

Module Basic36Poker

- The module Basic36Poker is built from BasicPoker. Its representation type is reused and denoted as $B36P = BP$. The properties of BasicPoker are reused and some of them are reproved because of some function re-definitions (see Listing A.19 of Appendix A.3).

- The total number of the cards in a pack $totalCardNum$ is updated to 36 cards.

- The function *setPack* is reused to set a pack of 36 cards.

Module BasicWPoker

The module definition is given in Listing A.20 of Appendix A.3.

- The module BasicWPoker is built from BasicPoker. Its representation type is extended from BP (the representation type of BasicPoker) to $BWP = BP * wcards$, in which $wcards$ is an extended part, a list of three wild cards.

- The function *makeWCards* deals one card, facing up on the table, and adds the other three cards having the same rank into $wcards$.

  - The property *bwp_make_wcards* describes that after dealing a wild card, the three remaining cards with the same value are put into the wild list *wcards* (see line 7).

- The function *wCombination* gets 5 hole cards of a player and *wcards* as input and returns if the player has "Five of A Kind" score (an updated hand rank of BasicWPoker mentioned in Section 5.1).

- The function *showdown* is redefined. If no player was found with "Five of A Kind" by the function *wCombination*, the function *showdown* of BasicPoker will be called. If not, the player found by *wCombination* is the winner.

Module MFormPoker

The module definition is given in Listing A.21 of Appendix A.3.

- The module MFormPoker is built from BasicMPoker. Its representation type contains community cards *comCards* (faced up and put up on the table). It is denoted by $MFP = BMP * comCards$ in which R_1 is reused from BasicMPoker.

- The function *getPack* introduced in BasicPoker is redefined in the module MFormPoker. There is a property added into it.

  - The property *mfp_getPack_succ* specifies that the cards set up for the pack of a MFormPoker poker game can be taken out by the function *getPack* (see line 5).

- The function *flop* is redefined by adding a case of flop type $flopType = FCom$, in which the cards dealt from the pack are saved into *comCards* (defined in the related type file in Figure 5.2)).

  - The property *mfp_flop_addComCards* states that after dealing some common cards from a pack, the dealt cards are put into the common card list (see line 7).

  - The refining property *mfp_flop_minusCards_R1* refines the property *bp_flop_minusCards* of the module BasicMPoker. It specifies that if the flop type is not $FCom$, the dealing is proceeded as in the module BasicMPoker (see line 9).

- The function *showdown* is redefined to find the winner from the combinations of the hole players cards and the community cards.

Module DrawMPoker

- The module DrawMPoker is built from MFormPoker. The representation type is represented as $DMP = MFP * ante$, in which *ante* is an amount of chips. The *ante* and the rule for betting with *ante* are to be set before the game starts if all players agree.

- The community card list *comCards* is empty.

- The function *discardCards* allows a player to discard some hole cards in his hand and take new cards from a pack.

- The function *bettingRound* is redefined by adding a case of bet type $betType = BAnte$ in which the amount of each bet is defined by *ante* (defined in the related type file in Figure 5.2).

  - The refining property *dmp_bettinground_upd_pot_R1* specifies that if the bet type of the function *bettingRound* is not $BAnte$, the constraint of the property *bmp_bettinground_upd_pot* of module *BasicMPoker* is still guaranteed in the module *DrawMPoker* (see lines 8 - 11 of Listing A.22 of Appendix A.3).

Module TexasHold'emMPoker

- The module TexasHold'emMPoker is built from the module DrawMPoker and has the same representation type.

- The community card list *comCards* is not empty and its cards are faced up on the table in each dealing time.

- The function *bettingRound* is redefined by adding a case where the bet type *betType* = $BFinal$. $BFinal$ is defined for the final betting round in which the players can decide to increase their bets with *raiseNum* (defined in the related type file in Figure 5.2).

Besides the seven modules that have been mentioned above, the analysis of other features, such as FixedLimit, PotLimit, StudMPoker, PreFixedLimit, Roodles can be found in the first half of Appendix A.3. At the moment, we have developed seven FFML modules corresponding to the seven analyzed features in this section. The details of these modules and the products established from them are listed in the remaining half of Appendix A.3. All of them are obtained by using our tools, the FFML Compiler and the FFML Product Generation.

```
1  fmodule BP
2   signature totalCardNum : int;
3   signature getPack : BP -> card list;
4   signature makePoker : player list -> card list -> BP;
5   ...
6   contract getPack :: invariant property bp_getPack_succ : all players : player list , all pack :
        card list , getPack(makePoker(players , pack)) = pack;
7   ...
8   representation = player list * card list;
9   let totalCardNum = 52;
10  let makePoker (players , cards) = (players , cards);
11  let getPack (x) = snd(x);
12  ...
13  proof of bp_getPack_succ = foc proof {* by definition of getPack , makePoker;*}
14  .... ;;
```

Listing 5.1: BasicPoker (BP) in FFML

A poker product is configured by selecting these features in the model 5.1. For example, a user who wants a poker game which has some functionalities as playing with 36 cards of a pack and gambling with chips will select the BP, B36P and BMP features to make a valid configuration $\{BP, B36P, BMP\}$. The development of the configuration is generated from the associated FFML modules (same names with features). We show these modules partially in Listings 5.1, 5.2 and 5.3 (We just give small parts of their code). A function *makePoker* describe a game with a list of player, a list of playing cards. Total of all cards are 52 describes by *totalCardNum*. A function *getPack* is used for getting cards in a current pack. The property *bp_getPack_succ* in BP, related to *getPack*, specifies a user can get the cards in the pack. This property is inherited and reproved in both BMP and B36P.

```
fmodule BMP from BP
 signature makeMPoker : BP -> int -> BMP; // a poker combined BasicPoker and a pot (int)
 ...
 representation extends BP with int; (* BP and pot *)
 let makeMPoker (poker, pot) = (poker, pot);
 ...
 proof of bp_getPack_succ =
 foc proof {*
  <1>1 assume players : player list, desk : card list,
       prove getPack(makePoker(players, desk)) = desk
    <2>1 prove getPack(makePoker(players, desk)) = BP!getPack(BP!makePoker(players, desk))
         by definition of getPack, makePoker, get1st
    <2>e qed by step <2>1 property BP!bp_getPack_succ
 <1>e conclude; *}
 ... ;;
```

Listing 5.2: BasicMPoker (BMP) in FFML

```
fmodule B36P from BP
 representation = BP;
 let totalCardNum = 36;
 proof of bp_getPack_succ =
 foc proof {*
  <1>1 assume players : player list, assume desk : card list,
 prove (getPack(makePoker(players, desk)) = desk)
   <2>1 prove (getPack(makePoker(players, desk)) = BP!getPack(BP!makePoker(players, desk)))
        by definition of getPack, makePoker
   <2>e qed by step <2>1 property BP!bp_getPack_succ
 <1>e conclude; *}
 ... ;;
```

Listing 5.3: Basic36Poker (B36P) in FFML

The three FFML modules are composed into a module, called B36PBMP, by FFML Product Generator. The generated product that is shown in Listing 5.4 inherits BMP. However, the total of playing cards is kept (36 cards) and the proof of the property *bp_getPack_succ* is updated from B36P. The object name BP in the proof is changed to BMP.

```
fmodule B36PBMP from BMP
 representation = BMP;
 let totalCardNum  = 36;
 proof of bp_getPack_succ =
 foc proof {*
  <1>1 assume players : player list, assume desk :  card list,
       prove (getPack(makePoker(players,desk)) = desk)
       <2>1 prove (getPack(makePoker(players,desk)) = BMP!getPack(BMP!makePoker(players,desk)))
            by definition of getPack, makePoker
       <2>e qed by step  <2>1 property BMP!bp_getPack_succ
  <1>e conclude;*}
 ... ;;
```

Listing 5.4: B36PBMP in FFML

5.3 Evaluation

In this section, we focus on analyzing the results obtained from developing Poker SPL. The validity and the limitation of our methodology are also discussed.

Poker SPL

The Poker SPL has been analyzed with twelve features, seven of which have been developed into the seven corresponding modules: BasicPoker (BP), BasicMPoker (BMP), Basic36Poker (B36P), BasicWPoker (BWP), MFormPoker (MFP), DrawMPoker (DMP), TexasHold'emMPoker (THP) by about 800 LOC in FFML. These seven modules have been translated into FoCaLiZe by FFML Compiler. We summarize the statistics of these modules in Table 5.1. The left side of the table contains qualitative information about the product line in FFML and the right side concerns the corresponding compiled FoCaLiZe code. Similar to the Bank Account SPL, there are more proofs than properties because some properties, which are "inherited", need to be proven again because of some re-definitions. **Ze-Pf** are the proofs obtained by translating the modules in which **reuse** are proofs built automatically. For example, the module BP (line 1) implementing the feature BasicPoker is defined with 4 properties in FFML, three of which are invariant. The four corresponding proofs are written for these properties. This module translated into FoCaLiZe contains 4 properties and 4 proofs. No properties or proofs are reused because BasicPoker is the root feature. The module BMP (line 2) implementing the feature BasicMPoker is built from BP. This module reuses all the properties of BP and is added 3 properties, one of which is invariant. There are five proofs written in BMP, three of which are for these added properties and the two remaining ones reprove the properties reused from BP. The module BMP translated into FoCaLiZe, contains 4 properties and 7 proofs. We can see that the translated module has 1 property and 2 proofs more than the FFML module. This has happened because of the reuse mechanism in FFML which is implemented in the FFML Compiler.

Table 5.2 presents the results we obtained by implementing some other configurations selected from the seven developed modules. Using our tool, the FFML Product Generation, the corresponding products in FFML are generated automatically. The right side of the table is the FFML products translated into FoCaLiZe. **cPf** are the proofs built from the composing process. Most of their proofs, after being composed, are done successfully. A half of these proofs (**reuse**) are reused and the remaining proofs (**auto**) are done automatically by Zenon. The proofs **manu** that lack some proof hints are more numerous than in the

Modules	**FFML**				**FoCaLiZe**		
	iP	**nP**	**rP**	**Pf**	**Fo-P**	**Ze-Pf**	**reuse**
BP	3	1	0	4	4	4	0
BMP	1	2	0	5	4	7	2
B36P	0	0	0	3	1	4	1
BWP	0	1	0	4	2	5	1
MFP	0	2	1	9	5	9	0
DMP	0	0	1	7	5	9	2
THP	0	0	0	6	5	8	2
iP - invariant properties **nP** - new properties **rP** - refining properties **Pf** - proofs in FFML					**Fo-P** - properties in FoCaLiZe **Ze-Pf** - proofs are done by Zenon **reuse** - proofs are reused		

Table 5.1: Modules of Poker SPL

Bank Account SPL. As discussed previously, Zenon fails to find a proof because of an internal error. However, giving a more detailed proof (with sublemmas) we can bypass this Zenon's bug.

Configurations	**FFML**		**FoCaLiZe**					
	P	**Pf**	**Fo-P**	**Ze-Pf**	**cPf**	**reuse**	**auto**	**manu**
{BP,B36P,BMP}	0	3	6	7	4	3	5	2
{BP,B36P,BWP}	0	3	5	5	4	1	4	1
{BP,BMP,BWP}	3	4	7	8	4	4	7	1
{BP,BMP,MFP,BWP}	1	5	10	10	4	6	7	3
P - properties in FFML **Pf** - proofs in FFML			**Fo-P** - properties in FoCaLiZe **Ze-Pf** - proofs are done by Zenon **cPf** - proofs are composed **reuse** - proofs are reused **auto** - automatic proofs **manu** - manual proofs					

Table 5.2: Configurations of Poker SPL

We explian an example of the configuration $\{BP, B36P, BMP\}$ (line 1 of Table 5.2). To generate the product corresponding to this configuration, the two modules B36P and BMP (lines 2-3 in Table 5.1) are composed into a module B36PBMP (see Listing A.24 in Appendix A.3). The composed module does not contain any properties but reuses all the properties of BMP. The translattion of the module B36PBMP contains 120 LOC while the module B36PBMP in FFML is written by only 30 LOC. Seven proofs in the translated

module are generated by the FFML Product Generator tool, four of them are composed automatically and three of them are reused via the reuse mechanism of FFML. Five of these seven proofs are automatic while two of them lack proof hints. These **manu** proofs appeared due to Zenon's bug. Similar to the Bank Account SPL, we can bypass this bug by giving a more detailed proof. For example, Listing 5.5 shows the proof *bmp_ getPack_ succ* (line 2) of the module B36PBMP that is fixed manually by a proof with more details in lines 4-12.

```
1  //The generated proof:
2  //proof of bmp_getPack_succ = by definition of getPack, makeMPoker property BMP!bmp_getPack_succ;
3  //  is fail by Zenon's bug. This proof is fixed by the following:
4   proof of bmp_getPack_succ =
5    <1>1 assume p0 : BP, assume pot : int,
6         prove (getPack(makeMPoker(p0, pot)) = BP!getPack(p0))
7         <2>1 prove getPack (makeMPoker (p0, pot)) = BMP!getPack(BMP!makeMPoker(p0, pot))
8              by definition of getPack, makeMPoker
9         <2>2 prove BMP!getPack(BMP!makeMPoker(p0, pot)) = BP!getPack(p0)
10             by property BMP!bmp_getPack_succ
11        <2>e conclude
12   <1>e conclude;
```

Listing 5.5: An example of bypassing Zenon's bug

Based on the results obtained after developing the Poker SPL and the Bank Account SPL (in the previous chapter), we evaluate our methodology by its validity and limitation.

Validity of the Methodology

- According to the results obtained from the statistics tables of the Bank Account SPL and the Poker SPL, we can see that these two SPLs have been developed successfully. Namely, Table 4.15 and Table 5.1 show the developed modules of the two SPLs. Table 4.16 and Table 5.2 contains the products generated from these modules. The generated products are correct-by-construction. The artifacts, i.e., the properties and the proofs, are composed automatically by our tools. Although some proofs lack their proof hints, most of the proofs are done automatically.

- The development of the modules of SPLs using FFML is easier than in FoCaLiZe. The developer will write less LOC. This can be seen when comparing the modules of the Bank Account SPL in FFML and FoCaLiZe in Appendix A.2, or the product B36PBMP of the Poker SPL in Listing A.24 and Listing A.28 in Appendix A.3.

- By reducing the complexity and saving the efforts from the automated generation of the correct products while developing the two SPLs, our methodology is proven to

be effective and reliable.

- Although a product line becomes more complex when more features are added, the reuse and modification mechanisms in FFML allow reducing the complexity of the code and its redundancy (for example, the developer must not redefine a function if he/she intends to reuse its definition). This is shown by the fact that less artifacts are written for the modules in lower levels of the Bank Account and Poker SPLs.

- The generation of the combinations is automated and requires no intervention from user or developer. This meets the aim of SPL development that it is easier to generate new products as the combinations or configurations are automatically generated.

- The products of a SPL developed using our methodology are correct-by-construction even if some proofs remain to be done manually. The properties and proofs of these products are composed automatically. These product can be proved to be correct with respect to their specifications by Zenon.

Limitation of the Methodology

Besides the advantages mentioned above, we indicate some limitations:

- Our examples, the two product lines (Bank Account SPL and Poker SPL) developed using our methodology, are not much complex. We need more the developments of practical SPLs to evaluate exactly the methodology.

- The composite modules become big piece of code for some configurations when the SPL contains many features. We have realized this issue when developing the Poker SPL which has more features than the Bank Account SPL.

- Our tools only work with an assumption that a proof is written for each property and has a defined structure. In fact, it is possible that a property can be proved by several proofs. The tool has to cover this case and implement further proof heuristics.

- One of the advantages of FoCaLiZe is to handle proofs by their proof hints that is how they are done. In other languages, it is not the case. Therefore, the reuse of proofs will be much more difficult.

Conclusion

In this chapter, we briefly summarize our work on the thesis and give the contributions. Then we discuss potential future work on developing correct-by-construction product lines.

Summary

We have proposed in Chapter 2 a methodology to develop product lines such that the generated products are correct-by-construction. Our main intention is that a user does not need to know the product generation process but can receive a correct final product from selecting a configuration of features. Using the methodology, the final products are generated automatically and their correctness is guaranteed. Following this proposal, we have moved in Chapter 3 to define the FFML language that is used for writing modules. The reuse and modification mechanism, defined for the language and applied to all kinds of artifacts (specification, code and correctness proof), reduce the programming effort. In Chapter 4, we have focused on defining the composition mechanisms for composing FFML modules and embedded them into the FFML Product Generator tool. The evaluation of our methodology is performed through the development of two software product lines, the Bank Account SPL and the Poker SPL, the latter being a bit more complex than the former. In the evaluation, we have highlighted the advantages and the limitation of our methodology.

Contributions

- The main contribution of this thesis refers to an effective methodology for developing SPLs and generating automatically correct final products using an approach close to CbyC approach.

- The methodology has been implemented using FoCaLiZe and FOP techniques. The results obtained by using our tools to generate the correct final products of the Bank Account SPL and the Poker SPL demonstrate the applicability of our methodology.
- The modification mechanisms of all kinds of artifacts (specification, code and correctness proof) have been defined. Especially, a property can be refined from an existing one, hence reusing the corresponding proof. The proof was actually easier to realize. These mechanisms help to reuse artifacts and reduce effort for artifact writing.
- A composition operation for all kinds of artifacts at the module level has been defined and implemented in our tool. This tool can generate automatically products without user intervention. Verification effort is significantly reduced by means of proof reuse.
- A tool chain with FFML Compiler and FFML Product Generator has been developed. It supports both the developer and the user when developing SPLs and generating the correct-by-construction products.

Future work

We discuss the future work that could lead to further improvements of our methodology when developing SPLs. We list them below going from short term perspectives to more ambitious improvements:

- Complete the development of the Poker SPL. More properties and their corresponding proofs should be written. More products will be generated automatically by selecting these remaining features.
- Analyze the properties whose proofs are done manually (see Table 4.16 and Table 5.2), hence define new proof composition rules in order to support these properties. FFML Product Generator tool will be updated to take into account these new rules and thus make Zenon able to do the proofs automatically.
- Make FFML involve in order to take into account new ways of reusing such as introducing new parameters in a refining property. The presence of these parameters would allow the developer to write more complex properties.

- Develop a Graphic User Interface (GUI) for our tool chain and integrate to it a tool that supports checking the validity of configurations. This would permit a user to select the desired features for a configuration from the system and tell him/her the validity of the selected configuration.

- Adapt our methodology to other languages, such as B or Java language instead of FoCaLiZe. The adaptation is necessary to prove the independence of the methodology.

- Transform FOP, the implementation technique used in our methodology, to other ones, for instance DOP technique (based on the concept of program deltas) [Schaefer et al. 2010]. DOP is more flexible than FOP with allowance of removing artifacts. This will extend the characteristic for the methodology and is an evolution of the FFML language. A transformation of implementation technique or an adaptation to other languages can require more effort and difficulties. The mechanisms of modifications and compositions might be updated or new ones must be defined. For example, in DOP technique an artifact can be removed before entering into another module, we need to define a mechanism for removing the artifact and analyze the impacts of this modification.

Bibliography

Abrial, J.-R. (2005). *The B-book - assigning programs to meanings*. Cambridge University Press. 58, 59

Abrial, J.-R. (2010). *Modeling in Event-B - System and Software Engineering*. Cambridge University Press. 59

Apel, S., Batory, D. S., Kästner, C., and Saake, G. (2013a). *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer. 10, 39, 56, 74

Apel, S., Kästner, C., and Lengauer, C. (2009). FEATUREHOUSE: Language-independent, automated software composition. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 221–231. IEEE. 44, 54, 55, 62, 127, 153

Apel, S., Kästner, C., and Lengauer, C. (2013b). Language-Independent and Automated Software Composition: The FeatureHouse Experience. *IEEE Trans. Software Eng.*, 39(1):63–79. 41, 55

Apel, S., Lengauer, C., Möller, B., and Kästner, C. (2010). An algebraic foundation for automatic feature-based program synthesis. *Sci. Comput. Program.*, 75(11):1022–1047. 55

Apel, S., Rhein, A. v., Thüm, T., and Kästner, C. (2013c). Feature-interaction detection based on feature-based specifications. *Computer Networks*, 57(12):2399–2409. 55

Apel, S., Rhein, A. v., Wendler, P., slinger, A. G., and Beyer, D. (2013d). Strategies for product-line verification: case studies and experiments. In Notkin, D., Cheng, B. H. C., and Pohl, K., editors, *35th International Conference on Software Engineering, ICSE '13,*

San Francisco, CA, USA, May 18-26, 2013, pages 482–491. IEEE Computer Society. 41, 55, 57, 62

Apel, S., Speidel, H., Wendler, P., Rhein, A. v., and Beyer, D. (2011). Detection of feature interactions using feature-aware verification. In Alexander, P., Pasareanu, C. S., and Hosking, J. G., editors, *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 372–375. IEEE Computer Society. 57

Aversano, L., Penta, M. D., and Baxter, I. D. (2002). Handling Preprocessor-Conditioned Declarations. In *2nd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002), 1 October 2002, Montreal, Canada*, pages 83–92. IEEE Computer Society. 41

Ayrault, P., Hardin, T., and Pessaux, F. (2009). Development of a Generic Voter under FoCal. In Dubois, C., editor, *Tests and Proofs, Third International Conference, TAP 2009, Zurich, Switzerland, July 2-3, 2009. Proceedings*, volume 5668 of *Lecture Notes in Computer Science*, pages 10–26. Springer. 60

Back, R. J. (1978). On the correctness of refinement steps in program development. 14, 47, 59

Back, R. J. and Wright, J. (1998). *Refinement calculus*. Springer. 14, 59

Barnes, J., Chapman, R., Johnson, R., Widmaier, J., Cooper, D., and Everett, B. (2006). Engineering the Tokeneer enclave protection software. In *Proceedings of IEEE International Symposium on Secure Software Engineering*. 58

Bass, L., Clements, P., and Kazman, R. (1998). *Software Architecture in Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 11, 39, 48, 53

Batory, D. S. (2005a). Feature Models, Grammars, and Propositional Formulas. In Obbink, J. H. and Pohl, K., editors, *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer. 49, 52, 53, 69

Batory, D. S. (2005b). A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. In Lämmel, R., Saraiva, J., and Visser, J., editors, *Generative and Transformational Techniques in Software Engineering, International Summer School, GTTSE 2005, Braga, Portugal, July 4-8, 2005. Revised Papers*, volume 4143 of *Lecture Notes in Computer Science*, pages 3–35. Springer. 40

Batory, D. S. (2015). A theory of modularity for automated software development (keynote). In France, R. B., Ghosh, S., and Leavens, G. T., editors, *Companion Proceedings of the 14th International Conference on Modularity, MODULARITY 2015, Fort Collins, CO, USA, March 16 - 19, 2015*, pages 1–10. ACM. 10, 41, 74

Batory, D. S. and O'Malley, S. W. (1992). The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Trans. Softw. Eng. Methodol.*, 1(4):355–398. 52

Batory, D. S., Sarvela, J. N., and Rauschmayer, A. (2003). Scaling Step-Wise Refinement. In Clarke, L. A., Dillon, L., and Tichy, W. F., editors, *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, pages 187–197. IEEE Computer Society. 40, 54, 127

Benavides, D., Martín-Arroyo, P. T., and Cortés, A. R. (2005). Automated Reasoning on Feature Models. In Pastor, O. and Cunha, J. F. e., editors, *Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005, Proceedings*, volume 3520 of *Lecture Notes in Computer Science*, pages 491–503. Springer. 50, 53

Benavides, D., Segura, S., and Cortés, A. R. (2010). Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636. 12, 49, 50, 51, 52

Benavides, D., Segura, S., Trinidad, P., and Cortés, A. R. (2007). FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In Pohl, K., Heymans, P., Kang, K. C., and Metzger, A., editors, *First International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS 2007, Limerick, Ireland, January 16-18, 2007. Proceedings*, volume 2007-01 of *Lero Technical Report*, pages 129–134. 53

Benduhn, F., Thüm, T., Lochau, M., Leich, T., and Saake, G. (2015). A Survey on Modeling Techniques for Formal Behavioral Verification of Software Product Lines. In Schmid, K., Haugen, O., and Müller, J., editors, *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS '15, Hildesheim, Germany, January 21-23, 2015*, page 80. ACM. 39

Berre, D. L. and Parrain, A. (2010). The Sat4j library, release 2.2. *JSAT*, 7(2-3):59–6. 53

Bonichon, R., Delahaye, D., and Doligez, D. (2007). Zenon : An Extensible Automated Theorem Prover Producing Checkable Proofs. In Dershowitz, N. and Voronkov, A., editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings*, volume 4790 of *Lecture Notes in Computer Science*, pages 151–165. Springer. 43, 60

Bracha, G. and Cook, W. R. (1990). Mixin-based Inheritance. In Yonezawa, A., editor, *Conference on Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming (OOPSLA/ECOOP), Ottawa, Canada, October 21-25, 1990, Proceedings*, pages 303–311. ACM. 54

Braga, R. T. V., Branco, K. R. L. J. C., Júnior, O. T., Masiero, P. C., Neris, L. d. O., and Becker, M. (2012). The ProLiCES Approach to Develop Product Lines for Safety-Critical Embedded Systems and its Application to the Unmanned Aerial Vehicles Domain. *CLEI Electron. J.*, 15(2). 41

Burdy, L., Cheon, Y., Cok, D. R., Ernst, M. D., Kiniry, J. R., Leavens, G. T., Leino, K. R. M., and Poll, E. (2005). An overview of JML tools and applications. *STTT*, 7(3):212–232. 62

Chapman, R. (2006). Correctness by construction: a manifesto for high integrity software. In *Proceedings of the 10th Australian workshop on Safety critical systems and software-Volume 55*, pages 43–46. Australian Computer Society, Inc. 58

Charpentier, M. and Chandy, K. M. (2004). Specification transformers: a predicate transformer approach to composition. *Acta Inf.*, 40(4):265–301. 82

Classen, A., Cordy, M., Heymans, P., Legay, A., and Schobbens, P.-Y. (2014). Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Sci. Comput. Program.*, 80:416–439. 57

Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional. 12, 39, 48, 53, 124

Czarnecki, K. (2002). Generative Programming: Methods, Techniques, and Applications. In Gacek, C., editor, *Software Reuse: Methods, Techniques, and Tools, 7th International Conference, ICSR-7, Austin, TX, USA, April 15-19, 2002, Proceedings*, volume 2319 of *Lecture Notes in Computer Science*, pages 351–352. Springer. 56

Czarnecki, K. and Eisenecker, U. W. (1999). Components and Generative Programming. In Nierstrasz, O. and Lemoine, M., editors, *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings*, volume 1687 of *Lecture Notes in Computer Science*, pages 2–19. Springer. 10, 40

Czarnecki, K., Helsen, S., and Eisenecker, U. W. (2005). Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29. 49, 52, 69

Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R., editors (1972). *Structured Programming*. Academic Press Ltd., London, UK, UK. 47

Damiani, F., Owe, O., Dovland, J., Schaefer, I., Johnsen, E. B., and Yu, I. C. (2012). A Transformational Proof System for Delta-oriented Programming. In *Proceedings of the 16th International Software Product Line Conference - Volume 2*, SPLC '12, pages 53–60, New York, NY, USA. ACM. 57

Delaware, B., Cook, W. R., and Batory, D. S. (2011). Product lines of theorems. In Lopes, C. V. and Fisher, K., editors, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications,*

OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011, pages 595–608. ACM. 41, 57, 63, 74, 124

Delaware, B., Keuchel, S., Schrijvers, T., and Oliveira, B. C. d. S. (2013). Modular monadic meta-theory. In Morrisett, G. and Uustalu, T., editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 319–330. ACM. 41, 124

Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition. 14, 59

Djebbi, O., Salinesi, C., and Diaz, D. (2007). Deriving Product Line Requirements: the RED-PL Guidance Approach. In *14th Asia-Pacific Software Engineering Conference (APSEC 2007), 5-7 December 2007, Nagoya, Japan*, pages 494–501. IEEE Computer Society. 53

Douglas, M. (1968). Mass produced software components. In *Proc. NATO Conf. Software Engineering*, pages 138–155. Springer. 47

Dubois, C., Levy, N., and Pham, T. (2016). Vers un développement formel non incrémental. In *AFADL'2016*, pages 106–113, Besançon, France. In //hal.inria.fr/hal-01326694/file/AFADL2016.pdf. 73

Fisler, K. and Krishnamurthi, S. (2001). Modular verification of collaboration-based software designs. In Tjoa, A. M. and Gruhn, V., editors, *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001*, pages 152–163. ACM. 41, 57

Flatt, M., Krishnamurthi, S., and Felleisen, M. (1998). Classes and Mixins. In MacQueen, D. B. and Cardelli, L., editors, *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, pages 171–183. ACM. 54

Goguen, J. A., Wagner, E. G., and Thatcher, J. W. (1976). An initial algebra approach

to the specification, correctness, and implementation of abstract data types. Technical Report RC 06487, IBM US Research Centers (Yorktown,San Jose,Almaden, US). 76

Hall, A. and Chapman, R. (2002). Correctness by construction: Developing a commercial secure system. *IEEE software*, 19(1):18–25. 10, 41, 58, 59

Hamiaz, M. K., Pantel, M., Thirioux, X., and Combemale, B. (2016). Correct-by-construction model driven engineering composition operators. *Formal Asp. Comput.*, 28(3):409–440. 124

Harhurin, A. and Hartmann, J. (2008). Towards Consistent Specifications of Product Families. In Cuéllar, J., Maibaum, T. S. E., and Sere, K., editors, *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*, volume 5014 of *Lecture Notes in Computer Science*, pages 390–405. Springer. 57

Hähnle, R. and Schaefer, I. (2012). A Liskov Principle for Delta-Oriented Programming. In Margaria, T. and Steffen, B., editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I*, volume 7609 of *Lecture Notes in Computer Science*, pages 32–46. Springer. 55, 57

Hähnle, R., Schaefer, I., and Bubel, R. (2013). Reuse in Software Verification by Abstract Method Calls. In Bonacina, M. P., editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 300–314. Springer. 55

Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A. (1990). Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. 9, 12, 40, 48, 49, 50

Kolesnikov, S. S., Rhein, A. v., Hunsen, C., and Apel, S. (2013). A comparison of product-based, feature-based, and family-based type checking. In Järvi, J. and Kästner, C.,

editors, *Generative Programming: Concepts and Experiences, GPCE'13, Indianapolis, IN, USA - October 27 - 28, 2013*, pages 115–124. ACM. 55

Kourie, D. G. and Watson, B. W. (2012). *The Correctness-by-Construction Approach to Programming*. Springer. 10, 41, 59

Linden, F. v. d., Schmid, K., and Rommes, E. (2007). *Software product lines in action - the best industrial practice in product line engineering*. Springer. 39

Liu, J., Basu, S., and Lutz, R. R. (2011). Compositional model checking of software product lines using variation point obligations. *Autom. Softw. Eng.*, 18(1):39–76. 57

Mannion, M. (2002). Using First-Order Logic for Product Line Model Validation. In Chastek, G. J., editor, *Software Product Lines, Second International Conference, SPLC 2, San Diego, CA, USA, August 19-22, 2002, Proceedings*, volume 2379 of *Lecture Notes in Computer Science*, pages 176–187. Springer. 53

Mendonça, M., Wasowski, A., and Czarnecki, K. (2009). SAT-based analysis of feature models is easy. In Muthig, D. and McGregor, J. D., editors, *Software Product Lines, 13th International Conference, SPLC 2009, San Francisco, California, USA, August 24-28, 2009, Proceedings*, volume 446 of *ACM International Conference Proceeding Series*, pages 231–240. ACM. 53

Mendonça, M., Wasowski, A., Czarnecki, K., and Cowan, D. D. (2008). Efficient compilation techniques for large scale feature models. In Smaragdakis, Y. and Siek, J. G., editors, *Generative Programming and Component Engineering, 7th International Conference, GPCE 2008, Nashville, TN, USA, October 19-23, 2008, Proceedings*, pages 13–22. ACM. 53

Meyer, B. (1992). Applying "Design by Contract". *IEEE Computer*, 25(10):40–51. 75

Morgan, C. (1993). The refinement calculus. In *Program Design Calculi*, pages 3–52. Springer Berlin Heidelberg. 14, 59

Parnas, D. L. (1976). On the Design and Development of Program Families. *IEEE Trans. Software Eng.*, 2(1):1–9. 39, 47

Pham, T.-K.-Z., Dubois, C., and Lévy, N. (2015). Towards correct-by-construction product variants of a software product line: GFML, a formal language for feature modules. In Atlee, J. M. and Gnesi, S., editors, *Proceedings 6th Workshop on Formal Methods and Analysis in SPL Engineering, FMSPLE@ETAPS 2015, London, UK, 11 April 2015*, volume 182 of *EPTCS*, pages 44–55. 57, 73, 74

Plath, M. and Ryan, M. (2001). Feature integration using a feature construct. *Sci. Comput. Program.*, 41(1):53–84. 57

Pohl, K., Böckle, G., and Linden, F. v. d. (2005). *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer. 9, 12, 39, 40, 48, 49, 50, 53

Poppleton, M. (2007). Towards Feature-Oriented Specification and Development with Event-B. In Sawyer, P., Paech, B., and Heymans, P., editors, *Requirements Engineering: Foundation for Software Quality, 13th International Working Conference, REFSQ 2007, Trondheim, Norway, June 11-12, 2007, Proceedings*, volume 4542 of *Lecture Notes in Computer Science*, pages 367–381. Springer. 41

Post, H. and Sinz, C. (2008). Configuration Lifting: Verification meets Software Configuration. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pages 347–350. IEEE Computer Society. 54

Prehofer, C. (1997). Feature-Oriented Programming: A Fresh Look at Objects. In *ECOOP*, pages 419–443. 10, 41, 43, 54

Prehofer, C. (2001). Feature-oriented programming: A new way of object composition. *Concurrency and Computation: Practice and Experience*, 13(6):465–501. 54

Prévosto, V. (2003). *Conception et implantation du langage FoC pour le développement de logiciels certifiés*. PhD thesis, Université Pierre et Marie Curie (UPMC). Type : Thèse de Doctorat – Soutenue le : 2003-09-15 – Dirigée par : Hardin, Thérèse. 60

Prevosto, V. and Doligez, D. (2002). Algorithms and Proofs Inheritancey in the FOC Language. *J. Autom. Reasoning*, 29(3-4):337–363. 14, 43, 60, 121

Rioboo, R. (2009). Invariants for the FoCaL language. *Ann. Math. Artif. Intell.*, 56(3-4):273–296. 86

Schaefer, I., Bettini, L., Bono, V., Damiani, F., and Tanzarella, N. (2010). Delta-Oriented Programming of Software Product Lines. In Bosch, J. and Lee, J., editors, *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings*, volume 6287 of *Lecture Notes in Computer Science*, pages 77–91. Springer. 10, 13, 18, 41, 55, 75, 171

Schmid, K., Rabiser, R., and Grünbacher, P. (2011). A Comparison of Decision Modeling Approaches in Product Lines. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '11, pages 119–126, New York, NY, USA. ACM. 9, 40, 50

Thompson, S. (1991). *Type theory and functional programming*. International computer science series. Addison-Wesley. 41, 58

Thüm, T. (2015). *Product-line specification and verification with feature-oriented contracts*. PhD thesis, Otto von Guericke University Magdeburg. 41, 62, 123, 127

Thüm, T., Apel, S., Kästner, C., Schaefer, I., and Saake, G. (2014a). A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.*, 47(1):6:1–6:45. 39, 41, 55, 56, 57

Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., and Leich, T. (2014b). FeatureIDE: An extensible framework for feature-oriented software development. *Sci. Comput. Program.*, 79:70–85. 10, 41, 44, 55, 124, 127

Thüm, T., Meinicke, J., Benduhn, F., Hentschel, M., Rhein, A. v., and Saake, G. (2014c). Potential synergies of theorem proving and model checking for software product lines. In Gnesi, S., Fantechi, A., Heymans, P., Rubin, J., Czarnecki, K., and Dhungana, D., editors, *18th International Software Product Line Conference, SPLC '14, Florence, Italy, September 15-19, 2014*, pages 177–186. ACM. 57

Thüm, T., Schaefer, I., Hentschel, M., and Apel, S. (2012a). Family-based deductive verification of software product lines. In Ostermann, K. and Binder, W., editors, *Generative*

Programming and Component Engineering, GPCE'12, Dresden, Germany, September 26-28, 2012, pages 11–20. ACM. 50, 62

Thüm, T., Schaefer, I., Kuhlemann, M., and Apel, S. (2011). Proof Composition for Deductive Verification of Software Product Lines. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings*, pages 270–277. IEEE Computer Society. 41, 62, 63, 74, 123

Thüm, T., Schaefer, I., Kuhlemann, M., Apel, S., and Saake, G. (2012b). Applying Design by Contract to Feature-Oriented Programming. In Lara, J. d. and Zisman, A., editors, *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7212 of *Lecture Notes in Computer Science*, pages 255–269. Springer. 75

Tsang, E. P. K. (1993). *Foundations of constraint satisfaction*. Computation in cognitive science. Academic Press. 53

Watson, B. W., Kourie, D. G., and Cleophas, L. G. (2015). Experience with correctness-by-construction. *Sci. Comput. Program.*, 97:55–58. 10, 41

Whaley, J. (2016). JavaBDD. `http://javabdd.sourceforge.net`. Accessed June-2016. 53

White, J., Schmidt, D. C., Benavides, D., Trinidad, P., and Cortés, A. R. (2008). Automated Diagnosis of Product-Line Configuration Errors in Feature Models. In *Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings*, pages 225–234. IEEE Computer Society. 53

Appendix A

Appendix

We give the supplementary information that is referred in the chapters of the thesis. Our tools and two implemented product lines (Bank Account SPL and Poker SPL) are publicly available at `https://files.fm/u/hqpyuhgr`. In Section A.1 we present the FoCaLiZe grammar in Grammar A.1. The encodings of the modules and the products of the Bank Account SPL are supplemented in Section A.2. Last, we describe the code parts of the Poker SPL in Section A.3.

A.1 FoCaLiZe

FoCaLiZe grammar, which is used to compared to FFML's (see Grammar 3.1 in Section 3.3.1), is represented in Grammar A.1.

⟨*sp*⟩ ::= '`species`' ⟨*sid*⟩ ['`(`' (⟨*sparam_def*⟩)*, '`)`'] '`=`' ['`inherit`' (⟨*sinh_def*⟩)*, '`;`'] (⟨*ssig*⟩)* (⟨*sprop*⟩)* [⟨*srep*⟩] (⟨*sdef*⟩)* (⟨*sproof*⟩)* '`end;;`'

⟨*sparam_def*⟩ ::= ⟨*sparam_n*⟩ '`is`' ⟨*sid*⟩ ['`(`' (⟨*sparam_n*⟩)*, '`)`']

⟨*sinh_def*⟩ ::= ⟨*sid*⟩ ['`(`' (⟨*sparam_n*⟩)*, '`)`']

⟨*ssig*⟩ ::= '`signature`' ⟨*sfunc_n*⟩ '`:`' ⟨*stype*⟩ '`;`'

⟨*sprop*⟩ ::= '`property`' ⟨*sprop_n*⟩ '`:`' ⟨*slexpr*⟩ '`;`'

⟨*srep*⟩ ::= '`representation =`' ⟨*sc_prod*⟩ '`;`'

⟨*sdef*⟩ ::= '`let`' ⟨*sfunc_n*⟩ ['`(`' (⟨*spar*⟩)*, '`)`'] ⟨*sexpr*⟩ '`;`'

⟨*sproof*⟩ ::= '`proof of`' ⟨*sproof_n*⟩ '`=`' ⟨*sproofbody*⟩ '`;`'

⟨*sproofbody*⟩ - FoCaLiZe_proof_body

⟨*stype*⟩ - species type

⟨*slexpr*⟩ - FoCaLiZe_predicate_logical_formula

⟨*sexpr*⟩ - FoCaLiZe_expression

⟨*sc_prod*⟩ - Cartesian_product_of_FoCaLiZe_concrete_types

⟨*sid*⟩ - species_name

⟨*sparam_n*⟩ - parameter_name

⟨*sfunc_n*⟩ - function_name

⟨*sprop_n*⟩ - property_name

⟨*spar*⟩ - function_parameter

⟨*sproof_n*⟩ - FoCaLiZe_proof_name

Grammar A.1: FoCaLiZe Grammar

A.2 Bank Account SPL

The features of Bank Account SPL models bank account management into basic concepts, such as simple withdrawal and deposition, a withdrawal limitation or currency exchange. We supplement the details of the following modules: BA (BanckAccount) in Listing A.1, LL (LowLimit) in Listing A.2, DL (DailyLimit) in Listing A.3, CU (Currency) in Listing A.4, EX (CurrencyExchange) in Listing A.5 and the final products: $DLLL$ in Listing A.6 (built from the configuration $\{BA, DL, LL\}$), $DLCU$ in Listing A.7 (built from the configuration $\{BA, DL, CU\}$), $LLCU$ in Listing A.8 (built from the configuration $\{BA, LL, CU\}$), $LLEX$ in Listing A.9 (built from the configuration $\{BA, LL, EX\}$), $DLLLCU$ in Listing A.10 (built from the configuration $\{BA, DL, LL, CU\}$) and $DLLLEX$ in Listing A.11 (built from the configuration $\{BA, DL, LL, EX\}$). The translations of some modules and products are supplemented in Listings A.12, A.13, A.14, A.15 and A.16.

```
1  fmodule BA
2   signature update: BA -> int -> BA;
3   signature get_bal: BA -> int;
4   signature over: int;
5   signature makeBA : int -> BA;
6
7   contract get_bal :: invariant property ba_bal_gr_over: all x : BA, get_bal(x) >= over;
8   contract update :: property ba_upd_succ_with_over: all x : BA, all a : int,
9    (get_bal(x) + a) >= over -> get_bal (update(x,a)) = get_bal(x) + a;
10  contract update :: property ba_upd_nosucc_with_over: all x : BA, all a : int,
11   (get_bal(x) + a) < over -> get_bal (update(x,a)) = get_bal(x);
```

```
12  contract update :: property ba_upd_succ_with_zero: all x : BA, all a : int,
13   (a >= 0) -> get_bal (update(x,a)) = get_bal(x) + a;
14
15  representation = int; (* amount *)
16
17  let get_bal(x) = x;
18  let over = 0;
19  let makeBA (x) = x;
20  let update (x, a) = if ((get_bal(x) + a) >= over) then get_bal(x) + a else get_bal(x);
21
22  proof of ba_bal_gr_over = foc proof {* assumed; *}
23  proof of ba_upd_succ_with_over = foc proof {* by definition of update, get_bal;*}
24  proof of ba_upd_nosucc_with_over = foc proof {* by property int_ge_le_eq definition of get_bal,
        update;*}
25  proof of ba_upd_succ_with_zero =
26  foc proof {*
27   <1>1 assume x : BA, a : int,
28        prove (a >= 0) -> get_bal (update(x,a)) = get_bal(x) + a
29        <2>1 prove (a >= 0) -> get_bal(x) + a >= over + 0
30             by property int_ge_plus_plus, ba_bal_gr_over
31        <2>3 prove (a >= 0) -> get_bal(x) + a >= over
32             by step <2>1 property int_0_plus, int_plus_commute
33        <2>4 qed by step <2>3 property ba_upd_succ_with_over
34   <1>e conclude;*}
35 ;;
```

Listing A.1: BA

```
1  fmodule LL from BA
2   signature limit_low: int;
3
4   contract update :: property ll_upd_succ_with_llimit_R1
5     refines BA!ba_upd_succ_with_over
6     extends premise ((a >= 0) || (a <= limit_low));
7   contract update :: property ll_upd_nosucc_with_llimit_R1
8     refines BA!ba_upd_nosucc_with_over
9     extends premise ((a >= 0) || (a <= limit_low));
10  contract update :: property ll_upd_nosucc_with_ls_llimit : all x : LL, all a : int,
11     (a < 0) && (a > limit_low) -> get_bal (update(x, a)) = get_bal(x);
12
13  representation = BA ;
14
15  let limit_low = (-10);
16  let update (x, a) = if ((a >= 0) || (a <= limit_low)) then BA!update (x, a) else x;
17
18  proof of ll_upd_succ_with_llimit_R1 = foc proof {* by definition of update, get_bal, over
        property BA!ba_upd_succ_with_over;*}
19  proof of ll_upd_nosucc_with_llimit_R1 = foc proof {* by definition of update, get_bal, over
        property BA!ba_upd_nosucc_with_over;*}
20  proof of ll_upd_nosucc_with_ls_llimit = foc proof {* by definition of update, get_bal property
        int_ge_le_eq, int_ge_le_eq2;*}
21 ;;
```

Listing A.2: LL

```
1  fmodule DL from BA
2   signature limit_with: int ;
3   signature get_with: DL -> int ;
4
5   contract update :: property dl_upd_succ_with_wlimit_R1
6     refines BA!ba_upd_succ_with_over
7     extends premise (a <= 0) /\ (get_with(x) + a >= limit_with);
8   contract update :: property dl_upd_nosucc_with_wlimit_R1
9     refines BA!ba_upd_nosucc_with_over
10    extends premise (a <= 0) /\ (get_with(x) + a >= limit_with);
11  contract update :: property dl_upd_nosucc_ls_wlimit: all x : DL, all a : int,
12    ((a <= 0) /\ (get_with(x) + a < limit_with)) ->
13    get_with(update(x,a)) = get_with (x);
14  contract update :: property dl_upd_succ_gr_wlimit: all x : DL, all a : int,
15    ((a <= 0) /\ (get_with(x) + a >= limit_with)) ->
16    (get_with(update(x,a)) = get_with (x) + a );
17
18  representation extends BA with int;
19
20  let limit_with = 70;
21  let get_with (x) = snd(x);
22  let makeBA (amount) = (BA!makeBA(amount), 0);
23  let update (x, a) =
24    if (a <= 0)  then
25      if (get_with(x) + a >= limit_with) then (BA!update (x,a), get_with(x) + a) else x
26    else (BA!update(x,a), get_with(x));
27
28  proof of dl_upd_succ_with_wlimit_R1 =
```

```
 foc proof {*
  <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, update hypothesis h1
       <2>e qed by step <2>1 definition of over, get_bal property BA!ba_upd_succ_with_over
  <1>e conclude;*}
 proof of dl_upd_nosucc_with_wlimit_R1 =
 foc proof {*
  <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)
       <2>1 prove first(update(x,a)) = BA!update (first(x),a)
            by definition of update, first hypothesis h1
       <2>e qed by step <2>1 definition of over, get_bal property BA!ba_upd_nosucc_with_over
  <1>e conclude;*}
 proof of ba_upd_succ_with_zero =
 foc proof {*
  <1>1 assume x : DL, a : int,
       hypothesis H1: a >= 0,
       prove get_bal (update(x,a)) = get_bal(x) + a
      <2>1 prove first(update(x,a)) = BA!update(first(x), a)
           by definition of update, first property int_ge_le_eq3 hypothesis H1
      <2>2 prove BA!get_bal (BA!update(first(x),a)) = BA!get_bal(first(x)) + a
           by property BA!ba_upd_succ_with_zero hypothesis H1
      <2>e qed by step <2>1, <2>2 definition of get_bal
  <1>e conclude; *}
 proof of dl_upd_nosucc_ls_wlimit = foc proof {* by definition of update, get_with property
     int_ge_le_eq;*}
 proof of dl_upd_succ_gr_wlimit = foc proof {* by definition of update, get_with;*}
;;
```

Listing A.3: DL

```
1  //type cur = USD | EUR | VND;;
2  fmodule CU from BA
3   signature get_cur: CU -> cur;
4   signature upd_cur : CU -> cur  -> CU;
5   signature makeCU : BA -> cur -> CU;
6
7   contract upd_cur ::  property cu_upd_cur_succ : all x: CU, all c : cur, (get_cur(upd_cur(x,c)) =
        c) && (get_bal(upd_cur(x,c)) = get_bal(x) + get_bal (x) * (convert_cur(c) - convert_cur(
       get_cur(x))));
8
9   representation extends BA with cur;
10
11  let get_cur (x) = snd(x);
12  let makeBA (amount) = (BA!makeBA(amount), VND);
13  let makeCU (ba, c) = (ba,c);
14  let update (x, a) = (BA!update (x,a), get_cur(x));
15  let upd_cur (x, c) = let a = get_bal (x) * (convert_cur(c) - convert_cur(get_cur(x))) in
16    ((BA!makeBA(get_bal(x) + a)), c);
17
18  proof of ba_upd_succ_with_over =
19  foc proof {*
20   <1>1 assume x: CU, a : int,
21        prove (get_bal(x) + a) >= over -> (get_bal (update(x,a))) = get_bal(x) + a
22        <2>1 prove first(update(x,a)) = BA!update (first(x),a)
23             by definition of first, update
24        <2>e qed by step <2>1 definition of over, get_bal property BA!ba_upd_succ_with_over
25   <1>e conclude;*}
26  proof of ba_upd_nosucc_with_over =
27  foc proof {*
28   <1>1 assume x: CU, a : int,
29        prove (get_bal(x) + a) < over -> (get_bal (update(x,a))) = get_bal(x)
30        <2>1 prove first(update(x,a)) = BA!update (first(x),a)
31             by definition of update, first
32        <2>e qed by step <2>1 definition of over, get_bal property BA!ba_upd_nosucc_with_over
33   <1>e conclude;*}
34  proof of ba_upd_succ_with_zero =
35  foc proof {*
36   <1>1 assume x : CU, a : int,
37        hypothesis H1: a >= 0,
38        prove get_bal (update(x,a)) = get_bal(x) + a
39       <2>1 prove first(update(x,a)) = BA!update(first(x), a)
40            by definition of update, first
41       <2>2 prove BA!get_bal (BA!update(first(x),a)) = BA!get_bal(first(x)) + a
42            by property BA!ba_upd_succ_with_zero hypothesis H1
43       <2>e qed by step <2>1, <2>2 definition of get_bal
44   <1>e conclude;*}
45  proof of cu_upd_cur_succ =
46  foc proof {*
47  <1>1 assume  x: CU, c : cur, a : int,
48       hypothesis H1: a = get_bal (x) * (convert_cur(c) - convert_cur(get_cur(x))),
49       prove (get_cur(upd_cur(x,c)) = c) && (get_bal(upd_cur(x,c)) = get_bal(x) + a)
```

```
50        <2>1 prove BA!get_bal(BA!makeBA(get_bal(x) + a)) = get_bal(x) + a
51        assumed
52        <2>e qed by step <2>1 hypothesis H1 definition of upd_cur, get_cur, get_bal, first
53   <1>e conclude;*}
54 ;;
```

Listing A.4: CU

```
1  fmodule EX from CU
2   signature exchange_cur : EX -> int -> cur -> EX * int;
3
4   contract upd_cur ::  property ex_exchange_cur_succ : all x: EX, all a : int, all c : cur,
        get_bal (fst(exchange_cur (x, a, c))) = get_bal (x) + a;
5   contract upd_cur ::  property ex_exchange_cur_nosucc : all x: EX, all a : int, all c : cur, a>=
        0 -> (snd(exchange_cur (x, a, c))) = 0;
6
7   representation = CU;
8
9   let update (x, a) = CU!update(x, a) ;
10  let exchange_cur (x, a, c) =
11   if a < 0 then
12    (CU!makeCU(BA!makeBA(get_bal(x) + a), get_cur(x)), a * ratio (get_cur(x), c))
13   else (x,0);
14
15  proof of ex_exchange_cur_succ = foc proof {* assumed;}
16  proof of ex_exchange_cur_nosucc = foc proof {* by definition of exchange_cur property
        int_ge_le_eq;*}
17 ;;
```

Listing A.5: EX

```
1  fmodule DLLL from LL
2   signature limit_with : int;
3   signature get_with : DLLL -> int;
4
5   contract update ::  property ll_upd_nosucc_with_ls_llimit_C1
6    refines LL!ll_upd_nosucc_with_ls_llimit
7    extends premise (a <= 0) /\ (get_with(x) + a >= limit_with);
8   contract update ::  property dl_upd_succ_with_wlimit_R1
9    refines LL!ll_upd_succ_with_llimit_R1
10   extends premise (a <= 0) /\ (get_with(x) + a >= limit_with);
11  contract update ::  property dl_upd_nosucc_with_wlimit_R1
12   refines LL!ll_upd_nosucc_with_llimit_R1
13   extends premise (a <= 0) /\ (get_with(x) + a >= limit_with);
14  contract update ::  property dl_upd_nosucc_ls_wlimit : all x:DLLL, all a:int, ((a <= 0) /\ (
        get_with(x) + a < limit_with)) -> get_with(update(x,a)) = get_with(x);
15  contract update ::  property dl_upd_succ_gr_wlimit : all x:DLLL, all a:int, ((a <= 0) /\ (
        get_with(x) + a >= limit_with)) -> (get_with(update(x,a)) = get_with(x) + a);
16
17  representation extends LL with int;
18
19  let limit_with  = 70;
20  let get_with (x) = snd(x);
21  let makeBA (amount) = (LL!makeBA(amount),0);
22  let update (x , a) =
23   if (a <= 0) then
24    if (get_with(x) + a >= limit_with) then (LL!update(x,a),get_with(x) + a) else x
25   else (LL!update(x,a),get_with(x));
26
27  proof of ll_upd_nosucc_with_ls_llimit_C1 =
28  foc proof {*
29  <1>1 assume x : DLLL, assume a : int,
30       hypothesis h1 : (a <= 0) /\ (get_with(x) + a >= limit_with),
31       prove (a < 0) && (a > limit_low) -> get_bal(update(x,a)) = get_bal(x)
32       <2>1 prove first(update(x,a)) = LL!update(first(x),a)
33            by definition of update, first hypothesis h1
34       <2>e qed by step  <2>1 definition of over, get_bal, limit_low, update property LL!
             ll_upd_nosucc_with_ls_llimit
35  <1>e conclude;*}
36  proof of dl_upd_succ_with_wlimit_R1 =
37  foc proof {*
38   <1>1 assume x : , assume a : int,
39        hypothesis h1 : (a <= 0) /\ (get_with(x) + a >= limit_with),
40        prove ((a >= 0) || (a <= limit_low)) -> (get_bal(x) + a) >= over -> (get_bal(update(x,a)))
               = get_bal(x) + a
41       <2>1 prove first(update(x,a)) = LL!update(first(x),a)
             by definition of first, update hypothesis h1
43       <2>e qed by step  <2>1 definition of over, get_bal definition of limit_low property LL!
             ll_upd_succ_with_llimit_R1
44  <1>e conclude;*}
45  proof of dl_upd_nosucc_with_wlimit_R1 =
46  foc proof {*
47   <1>1 assume x : DLLL, assume a : int,
```

```
48          hypothesis h1 : (a <= 0) /\ (get_with(x) + a >= limit_with),
49          prove ((a >= 0) || (a <= limit_low)) -> (get_bal(x) + a) < over -> (get_bal(update(x,a)))
                = get_bal(x)
50          <2>1 prove first(update(x,a)) = LL!update(first(x),a)
51               by definition of update, first hypothesis h1
52          <2>e qed by step  <2>1 definition of over, get_bal definition of limit_low property LL!
                ll_upd_nosucc_with_llimit_R1
53   <1>e conclude;*}
54  proof of ba_upd_succ_with_zero =
55  foc proof {*
56   <1>1 assume x : DLLL, assume a : int,
57          hypothesis H1 : a >= 0,
58          prove get_bal(update(x,a)) = get_bal(x) + a
59          <2>1 prove first(update(x,a)) = LL!update(first(x),a)
60               by definition of update, first property int_ge_le_eq3 hypothesis H1
61          <2>2 prove LL!get_bal(LL!update(first(x),a)) = LL!get_bal(first(x)) + a
62               by property LL!ba_upd_succ_with_zero hypothesis H1
63          <2>e qed by step  <2>1, <2>2 definition of get_bal
64   <1>e conclude;*}
65  proof of dl_upd_nosucc_ls_wlimit = foc proof {* by definition of update, get_with property
        int_ge_le_eq;*}
66  proof of dl_upd_succ_gr_wlimit = foc proof {* by definition of update, get_with;*}
67 ;;
```

Listing A.6: DLLL

```
1  fmodule DLCU from CU
2   signature limit_with : int;
3   signature get_with : DLCU -> int;
4
5   contract update ::  property dl_upd_succ_with_wlimit_R1
6    refines CU!ba_upd_succ_with_over
7    extends premise (a <= 0) /\ (get_with(x) + a >= limit_with);
8   contract update ::  property dl_upd_nosucc_with_wlimit_R1
9    refines CU!ba_upd_nosucc_with_over
10   extends premise (a <= 0) /\ (get_with(x) + a >= limit_with);
11  contract update ::  property dl_upd_nosucc_ls_wlimit : all x:DLCU, all a:int, ((a <= 0) /\ (
        get_with(x) + a < limit_with)) -> get_with(update(x,a)) = get_with(x);
12  contract update ::  property dl_upd_succ_gr_wlimit : all x:DLCU, all a:int, ((a <= 0) /\ (
        get_with(x) + a >= limit_with)) -> (get_with(update(x,a)) = get_with(x) + a);
13
14  representation extends CU with int;
15
16  let limit_with  = 70;
17  let get_with (x) = snd(x);
18  let makeBA (amount) = (CU!makeBA(amount),0);
19  let update (x , a) =
20   if (a <= 0) then
21    if (get_with(x) + a >= limit_with) then (CU!update(x,a),get_with(x) + a) else x
22   else (CU!update(x,a),get_with(x));
23
24  proof of dl_upd_succ_with_wlimit_R1 =
25  foc proof {*
26  <1>1 assume x : DLCU, assume a : int,
27       hypothesis h1 : (a <= 0) /\ (get_with(x) + a >= limit_with),
28       prove (get_bal(x) + a) >= over -> (get_bal(update(x,a))) = get_bal(x) + a
29       <2>1 prove first(update(x,a)) = CU!update(first(x),a)
30            by definition of first, update hypothesis h1
31       <2>e qed by step  <2>1 definition of over, get_bal property CU!ba_upd_succ_with_over
32   <1>e conclude;*}
33  proof of dl_upd_nosucc_with_wlimit_R1 =
34  foc proof {*
35  <1>1 assume x : DLCU, assume a : int,
36       hypothesis h1 : (a <= 0) /\ (get_with(x) + a >= limit_with),
37       prove (get_bal(x) + a) < over -> (get_bal(update(x,a))) = get_bal(x)
38       <2>1 prove first(update(x,a)) = CU!update(first(x),a)
39            by definition of update, first hypothesis h1
40       <2>e qed by step  <2>1 definition of over, get_bal property CU!ba_upd_nosucc_with_over
41   <1>e conclude;*}
42  proof of ba_upd_succ_with_zero =
43  foc proof {*
44  <1>1 assume x : DLCU, assume a : int,
45       hypothesis H1 : a >= 0,
46       prove get_bal(update(x,a)) = get_bal(x) + a
47       <2>1 prove first(update(x,a)) = CU!update(first(x),a)
48            by definition of update, first property int_ge_le_eq3 hypothesis H1
49       <2>2 prove CU!get_bal(CU!update(first(x),a)) = CU!get_bal(first(x)) + a
50            by property CU!ba_upd_succ_with_zero hypothesis H1
51       <2>e qed by step  <2>1, <2>2 definition of get_bal
52   <1>e conclude;*}
53  proof of dl_upd_nosucc_ls_wlimit = foc proof {* by definition of update, get_with property
        int_ge_le_eq;*}
54  proof of dl_upd_succ_gr_wlimit = foc proof {* by definition of update, get_with;*}
55 ;;
```

Listing A.7: DLCU

```
fmodule LLCU from CU
 signature limit_low : int;

 contract update :: property ll_upd_succ_with_llimit_R1
  refines CU!ba_upd_succ_with_over
  extends premise ((a >= 0) || (a <= limit_low));
 contract update :: property ll_upd_nosucc_with_llimit_R1
  refines CU!ba_upd_nosucc_with_over
  extends premise ((a >= 0) || (a <= limit_low));
 contract update :: property ll_upd_nosucc_with_ls_llimit : all x:LLCU, all a:int, (a < 0) && (a
      > limit_low) -> get_bal(update(x,a)) = get_bal(x);

 representation = CU;

 let limit_low  = (-10);
 let update (x , a) =  if ((a >= 0) || (a <= limit_low)) then CU!update(x,a) else x;

 proof of ll_upd_succ_with_llimit_R1 = foc proof {* by definition of update, get_bal, over
      property CU!ba_upd_succ_with_over;*}
 proof of ll_upd_nosucc_with_llimit_R1 = foc proof {* by definition of update, get_bal, over
      property CU!ba_upd_nosucc_with_over;*}
 proof of ll_upd_nosucc_with_ls_llimit = foc proof {* by definition of update, get_bal property
      int_ge_le_eq, int_ge_le_eq2;}
;;
```

Listing A.8: LLCU

```
fmodule LLEX from EX
 signature limit_low : int;

 contract update :: property ll_upd_succ_with_llimit_R1
  refines EX!ba_upd_succ_with_over
  extends premise ((a >= 0) || (a <= limit_low));
 contract update :: property ll_upd_nosucc_with_llimit_R1
  refines EX!ba_upd_nosucc_with_over
  extends premise ((a >= 0) || (a <= limit_low));
 contract update :: property ll_upd_nosucc_with_ls_llimit : all x:LLEX, all a:int, (a < 0) && (a
      > limit_low) -> get_bal(update(x,a)) = get_bal(x);

 representation = EX;

 let limit_low  = (-10);
 let update (x , a) =  if ((a >= 0) || (a <= limit_low)) then EX!update(x,a) else x;

 proof of ll_upd_succ_with_llimit_R1 = foc proof {* by definition of update, get_bal, over
      property EX!ba_upd_succ_with_over;*}
 proof of ll_upd_nosucc_with_llimit_R1 = foc proof {* by definition of update, get_bal, over
      property EX!ba_upd_nosucc_with_over;*}
 proof of ll_upd_nosucc_with_ls_llimit = foc proof {* by definition of update, get_bal property
      int_ge_le_eq, int_ge_le_eq2;*}
;;
```

Listing A.9: LLEX

```
fmodule DLLLCU from LLCU
 signature limit_with : int;
 signature get_with : DLLLCU -> int;

 contract update :: property ll_upd_nosucc_with_ls_llimit_C1
  refines LLCU!ll_upd_nosucc_with_ls_llimit
  extends premise (a <= 0) /\ (get_with(x) + a >= limit_with);
 contract update :: property dl_upd_succ_with_wlimit_R1
  refines LLCU!ll_upd_succ_with_llimit_R1
  extends premise (a <= 0) /\ (get_with(x) + a >= limit_with);
 contract update :: property dl_upd_nosucc_with_wlimit_R1
  refines LLCU!ll_upd_nosucc_with_llimit_R1
  extends premise (a <= 0) /\ (get_with(x) + a >= limit_with);
 contract update :: property dl_upd_nosucc_ls_wlimit : all x:DLLLCU, all a:int, ((a <= 0) /\ (
      get_with(x) + a < limit_with)) -> get_with(update(x,a)) = get_with(x);
 contract update :: property dl_upd_succ_gr_wlimit : all x:DLLLCU, all a:int, ((a <= 0) /\ (
      get_with(x) + a >= limit_with)) -> (get_with(update(x,a)) = get_with(x) + a);

 representation extends LLCU with int;

 let limit_with  = 70;
 let get_with (x) = snd(x);
 let makeBA (amount) = (LLCU!makeBA(amount),0);
 let update (x , a) =  if (a <= 0)
 then  if (get_with(x) + a >= limit_with)
 then (LLCU!update(x,a),get_with(x) + a)
 else x
 else (LLCU!update(x,a),get_with(x));

```

```
28  proof of ll_upd_nosucc_with_ls_llimit_C1 =
29  foc proof {*
30  <1>1 assume x : DLLLCU, assume a : int,
31       hypothesis h1 : (a <= 0) /\ (get_with(x) + a >= limit_with),
32       prove (a < 0) && (a > limit_low) -> get_bal(update(x,a)) = get_bal(x)
33       <2>1 prove first(update(x,a)) = LLCU!update(first(x),a)
34            by definition of update, first hypothesis h1
35       <2>e qed by step  <2>1 definition of over, get_bal, limit_low, update property LLCU!
             ll_upd_nosucc_with_ls_llimit
36  <1>e conclude;*}
37  proof of dl_upd_succ_with_wlimit_R1 =
38  foc proof {*
39  <1>1 assume x : DLLLCU, assume a : int,
40       hypothesis h1 : (a <= 0) /\ (get_with(x) + a >= limit_with),
41       prove ((a >= 0) || (a <= limit_low)) -> (get_bal(x) + a) >= over -> (get_bal(update(x,a)))
             = get_bal(x) + a
42      <2>1 prove first(update(x,a)) = LLCU!update(first(x),a)
43           by definition of first, update hypothesis h1
44      <2>e qed by step  <2>1 definition of over, get_bal definition of limit_low property LLCU!
             ll_upd_succ_with_llimit_R1
45  <1>e conclude;*}
46  proof of dl_upd_nosucc_with_wlimit_R1 =
47  foc proof {*
48  <1>1 assume x : DLLLCU, assume a : int,
49       hypothesis h1 : (a <= 0) /\ (get_with(x) + a >= limit_with),
50       prove ((a >= 0) || (a <= limit_low)) -> (get_bal(x) + a) < over -> (get_bal(update(x,a))) =
              get_bal(x)
51       <2>1 prove first(update(x,a)) = LLCU!update(first(x),a)
52            by definition of update, first hypothesis h1
53       <2>e qed by step  <2>1 definition of over, get_bal definition of limit_low property LLCU!
             ll_upd_nosucc_with_llimit_R1
54  <1>e conclude;*}
55  proof of ba_upd_succ_with_zero =
56  foc proof {*
57  <1>1 assume x : DLLLCU, assume a : int,
58       hypothesis H1 : a >= 0,
59       prove get_bal(update(x,a)) = get_bal(x) + a
60       <2>1 prove first(update(x,a)) = LLCU!update(first(x),a)
61            by definition of update, first property int_ge_le_eq3 hypothesis H1
62       <2>2 prove LLCU!get_bal(LLCU!update(first(x),a)) = LLCU!get_bal(first(x)) + a
63            by property LLCU!ba_upd_succ_with_zero hypothesis H1
64       <2>e qed by step  <2>1, <2>2 definition of get_bal
65  <1>e conclude;*}
66  proof of dl_upd_nosucc_ls_wlimit = foc proof {*
67 by definition of update, get_with property int_ge_le_eq;*}
68  proof of dl_upd_succ_gr_wlimit = foc proof {*
69 by definition of update, get_with;*}
70 ;;
```

Listing A.10: DLLLCU

```
1  fmodule DLLLEX from LLEX
2   signature limit_with : int;
3   signature get_with : DLLLEX -> int;
4
5   contract update ::  property ll_upd_nosucc_with_ls_llimit_C1
6    refines LLEX!ll_upd_nosucc_with_ls_llimit
7    extends premise (a <= 0) /\ (get_with(x) + a >= limit_with);
8   contract update ::  property dl_upd_succ_with_wlimit_R1
9    refines LLEX!ll_upd_succ_with_llimit_R1
10   extends premise (a <= 0) /\ (get_with(x) + a >= limit_with);
11  contract update ::  property dl_upd_nosucc_with_wlimit_R1
12   refines LLEX!ll_upd_nosucc_with_llimit_R1
13   extends premise (a <= 0) /\ (get_with(x) + a >= limit_with);
14  contract update ::  property dl_upd_nosucc_ls_wlimit : all x:DLLLEX, all a:int, ((a <= 0) /\ (
        get_with(x) + a < limit_with)) -> get_with(update(x,a)) = get_with(x);
15  contract update ::  property dl_upd_succ_gr_wlimit : all x:DLLLEX, all a:int, ((a <= 0) /\ (
        get_with(x) + a >= limit_with)) -> (get_with(update(x,a)) = get_with(x) + a);
16
17  representation extends LLEX with int;
18
19  let limit_with  = 70;
20  let get_with (x) = snd(x);
21  let makeBA (amount) = (LLEX!makeBA(amount),0);
22  let update (x , a) =
23   if (a <= 0) then
24    if (get_with(x) + a >= limit_with)
25  then (LLEX!update(x,a),get_with(x) + a) else x
26  else (LLEX!update(x,a),get_with(x));
27
28  proof of ll_upd_nosucc_with_ls_llimit_C1 =
29  foc proof {*
30  <1>1 assume x : Self, assume a : int,
31       hypothesis h1 : (a <= 0) /\ (get_with(x) + a >= limit_with),
32       prove (a < 0) && (a > limit_low) -> get_bal(update(x,a)) = get_bal(x)
33       <2>1 prove first(update(x,a)) = LLEX!update(first(x),a)
```

```
34              by definition of update, first hypothesis h1
35        <2>e qed by step  <2>1 definition of over, get_bal, limit_low, update property LLEX!
                ll_upd_nosucc_with_ls_llimit
36  <1>e conclude;*}
37  proof of dl_upd_succ_with_wlimit_R1 =
38  foc proof {*
39  <1>1 assume x : Self, assume a : int,
40        hypothesis h1 : (a <= 0) /\ (get_with(x) + a >= limit_with),
41        prove ((a >= 0) || (a <= limit_low)) -> (get_bal(x) + a) >= over -> (get_bal(update(x,a)))
                = get_bal(x) + a
42        <2>1 prove first(update(x,a)) = LLEX!update(first(x),a)
43              by definition of first, update hypothesis h1
44       <2>e qed by step  <2>1 definition of over, get_bal definition of limit_low property LLEX!
                ll_upd_succ_with_llimit_R1
45  <1>e conclude;*}
46  proof of dl_upd_nosucc_with_wlimit_R1 =
47  foc proof {*
48  <1>1 assume x : Self, assume a : int,
49        hypothesis h1 : (a <= 0) /\ (get_with(x) + a >= limit_with),
50        prove ((a >= 0) || (a <= limit_low)) -> (get_bal(x) + a) < over -> (get_bal(update(x,a))) =
                 get_bal(x)
51        <2>1 prove first(update(x,a)) = LLEX!update(first(x),a)
52              by definition of update, first hypothesis h1
53        <2>e qed by step  <2>1 definition of over, get_bal definition of limit_low property LLEX!
                ll_upd_nosucc_with_llimit_R1
54  <1>e conclude;*}
55  proof of ba_upd_succ_with_zero =
56  foc proof {*
57  <1>1 assume x : Self, assume a : int,
58        hypothesis H1 : a >= 0,
59        prove get_bal(update(x,a)) = get_bal(x) + a
60        <2>1 prove first(update(x,a)) = LLEX!update(first(x),a)
61              by definition of update, first property int_ge_le_eq3 hypothesis H1
62        <2>2 prove LLEX!get_bal(LLEX!update(first(x),a)) = LLEX!get_bal(first(x)) + a
63              by property LLEX!ba_upd_succ_with_zero hypothesis H1
64        <2>e qed by step  <2>1, <2>2 definition of get_bal
65  <1>e conclude;*}
66  proof of dl_upd_nosucc_ls_wlimit = foc proof {* by definition of update, get_with property
        int_ge_le_eq;*}
67  proof of dl_upd_succ_gr_wlimit = foc proof {* by definition of update, get_with;*}
68  ;;
```

Listing A.11: DLLLEX

```
1  species BA_spec1 =
2   signature update : Self -> int -> Self;
3   signature get_bal : Self -> int;
4   signature over : int;
5   signature makeBA : int -> Self;
6   property ba_bal_gr_over : all x : Self, (get_bal(x) >= over);
7  end;;
8
9  species BA_spec2 =
10  inherit BA_spec1;
11  property ba_upd_succ_with_over : all x : Self, all a : int, ((get_bal(x) + a) >= over) -> (
        get_bal(update(x, a)) = (get_bal(x) + a));
12  property ba_upd_nosucc_with_over : all x : Self, all a : int, ((get_bal(x) + a) < over) -> (
        get_bal(update(x, a)) = get_bal(x));
13  property ba_upd_succ_with_zero : all x : Self, all a : int, (a >= 0) -> (get_bal(update(x, a)) =
        (get_bal(x) + a));
14 end;;
15
16 species BA_imp =
17  inherit BA_spec2;
18  representation = int;
19  let get_bal (x) = x;
20  let over = 0;
21  let makeBA (x) = x;
22  let update (x, a) =
23     if ((get_bal(x) + a) >= over) then
24    (get_bal(x) + a)
25   else get_bal(x);
26
27  proof of ba_bal_gr_over = assumed;
28  proof of ba_upd_succ_with_over = by definition of update, get_bal;
29  proof of ba_upd_nosucc_with_over = by property int_ge_le_eq definition of get_bal, update;
30  proof of ba_upd_succ_with_zero =
31   <1>1 assume x : Self, assume a : int,
32         prove (a >= 0) -> (get_bal(update(x, a)) = (get_bal(x) + a))
33         <2>1 prove (a >= 0) -> ((get_bal(x) + a) >= (over + 0))
34               by property int_ge_plus_plus, ba_bal_gr_over
35        <2>3 prove (a >= 0) -> ((get_bal(x) + a) >= over)
36 by step  <2>1 property int_0_plus, int_plus_commute
37        <2>4 qed by step  <2>3 property ba_upd_succ_with_over
38   <1>e conclude;
39 end;;
```

```
40
41 collection BA_col =
42   implement BA_imp;
43 end;;
```

Listing A.12: BA in FoCaLiZe

```
1  species DL_spec1 (BA is BA_imp) =
2   inherit BA_spec1;
3   signature limit_with : int;
4   signature get_with : Self -> int;
5  end;;
6
7  species DL_spec2 (BA is BA_imp) =
8   inherit DL_spec1 (BA);
9   property ba_upd_succ_with_zero : all x : Self, all a : int, (a >= 0) -> (get_bal(update(x, a)) =
        (get_bal(x) + a));
10  property dl_upd_succ_with_wlimit_R1 : all x : Self, all a : int, (a <= 0) && ((get_with(x) + a)
      >= limit_with) -> ((get_bal(x) + a) >= over) -> (get_bal(update(x, a)) = (get_bal(x) + a));
11  property dl_upd_nosucc_with_wlimit_R1 : all x : Self, all a : int, (a <= 0) && ((get_with(x) + a
      ) >= limit_with) -> ((get_bal(x) + a) < over) -> (get_bal(update(x, a)) = get_bal(x));
12  property dl_upd_nosucc_ls_wlimit : all x : Self, all a : int, ((a <= 0) && ((get_with(x) + a) <
      limit_with)) -> (get_with(update(x, a)) = get_with(x));
13  property dl_upd_succ_gr_wlimit : all x : Self, all a : int, ((a <= 0) && ((get_with(x) + a) >=
      limit_with)) -> (get_with(update(x, a)) = (get_with(x) + a));
14 end;;
15
16 species DL_imp (BA is BA_imp) =
17  inherit DL_spec2 (BA);
18  representation = BA* int;
19  let make (x1: BA, x2: int): Self = (x1, x2);
20  let get1st (x1: Self): BA = fst(x1);
21  let get2nd (x1: Self): int = snd(x1);
22  let get_bal (x) = BA!get_bal(get1st(x));
23  let over = BA!over;
24  let limit_with = 70;
25  let get_with (x) = snd(x);
26  let makeBA (amount) = (BA!makeBA(amount), 0);
27
28  let update (x, a) =a
29   if (a <= 0) then
30    if ((get_with(x) + a) >= limit_with) then (BA!update(get1st(x), a), (get_with(x) + a)) else x
31   else (BA!update(get1st(x), a), get_with(x));
32
33  proof of ba_bal_gr_over = by definition of get_bal, over property BA!ba_bal_gr_over;
34  proof of dl_upd_succ_with_wlimit_R1 =
35   <1>1 assume x : Self, assume a : int,
36        hypothesis h1 : (a <= 0) && ((get_with(x) + a) >= limit_with),
37        prove ((get_bal(x) + a) >= over) -> (get_bal(update(x, a)) = (get_bal(x) + a))
38       <2>1 prove (get1st(update(x, a)) = BA!update(get1st(x), a))
39            by definition of get1st, update hypothesis h1
40       <2>e qed by step  <2>1 definition of over, get_bal property BA!ba_upd_succ_with_over
41   <1>e conclude;
42  proof of dl_upd_nosucc_with_wlimit_R1 =
43   <1>1 assume x : Self, assume a : int,
44        hypothesis h1 : (a <= 0) && ((get_with(x) + a) >= limit_with),
45        prove ((get_bal(x) + a) < over) -> (get_bal(update(x, a)) = get_bal(x))
46        <2>1 prove (get1st(update(x, a)) = BA!update(get1st(x), a))
47             by definition of update, get1st hypothesis h1
48        <2>e qed by step  <2>1 definition of over, get_bal property BA!ba_upd_nosucc_with_over
49   <1>e conclude;
50  proof of ba_upd_succ_with_zero =
51   <1>1 assume x : Self, assume a : int,
52        hypothesis H1 : (a >= 0),
53        prove (get_bal(update(x, a)) = (get_bal(x) + a))
54       <2>1 prove (get1st(update(x, a)) = BA!update(get1st(x), a))
55            by definition of update, get1st property int_ge_le_eq3 hypothesis H1
56       <2>2 prove (BA!get_bal(BA!update(get1st(x), a)) = (BA!get_bal(get1st(x)) + a))
57            by property BA!ba_upd_succ_with_zero hypothesis H1
58       <2>e qed by step  <2>1, <2>2 definition of get_bal
59   <1>e conclude;
60  proof of dl_upd_nosucc_ls_wlimit = by definition of update, get_with property int_ge_le_eq;
61  proof of dl_upd_succ_gr_wlimit = by definition of update, get_with;
62 end;;
63
64 collection DL_col =
65   implement DL_imp (BA_col);
66 end;;
```

Listing A.13: DL in FoCaLiZe

```
species CU_spec1 (BA is BA_imp) =
 inherit BA_spec1;
 signature get_cur : Self -> cur;
 signature upd_cur : Self -> cur -> Self;
 signature makeCU : BA -> cur -> Self;
end;;

species CU_spec2 (BA is BA_imp) =
 inherit CU_spec1 (BA);
 property ba_upd_succ_with_zero : all x : Self, all a : int, (a >= 0) -> (get_bal(update(x, a)) =
      (get_bal(x) + a));
 property ba_upd_nosucc_with_over : all x : Self, all a : int, ((get_bal(x) + a) < over) -> (
     get_bal(update(x, a)) = get_bal(x));
 property ba_upd_succ_with_over : all x : Self, all a : int, ((get_bal(x) + a) >= over) -> (
     get_bal(update(x, a)) = (get_bal(x) + a));
 property cu_upd_cur_succ : all x : Self, all c : cur, ((get_cur(upd_cur(x, c)) = c) && (get_bal(
     upd_cur(x, c)) = (get_bal(x) + (get_bal(x) * (convert_cur(c) - convert_cur(get_cur(x)))))))
     ;
end;;

species CU_imp (BA is BA_imp) =
 inherit CU_spec2 (BA);
 representation = BA* cur;
 let make (x1: BA, x2: cur): Self = (x1, x2);
 let first (x1: Self): BA = fst(x1);
 let secon (x1: Self): cur = snd(x1);
 let get_bal (x) = BA!get_bal(first(x));
 let over = BA!over;
 let get_cur (x) = snd(x);
 let makeBA (amount) = (BA!makeBA(amount), VND);
 let makeCU (ba, c) = (ba, c);
 let update (x, a) = (BA!update(get1st(x), a), get_cur(x));
 let upd_cur (x, c) =
   let a = (get_bal(x) * (convert_cur(c) - convert_cur(get_cur(x))))  in
 (BA!makeBA((get_bal(x) + a)), c);

 proof of ba_bal_gr_over = by definition of get_bal, over property BA!ba_bal_gr_over;
 proof of ba_upd_succ_with_over =
  <1>1 assume x : Self, assume a : int,
       prove ((get_bal(x) + a) >= over) -> (get_bal(update(x, a)) = (get_bal(x) + a))
      <2>1 prove (get1st(update(x, a)) = BA!update(get1st(x), a))
           by definition of get1st, update
      <2>e qed by step  <2>1 definition of over, get_bal property BA!ba_upd_succ_with_over
  <1>e conclude;
 proof of ba_upd_nosucc_with_over =
  <1>1 assume x : Self, assume a : int,
       prove ((get_bal(x) + a) < over) -> (get_bal(update(x, a)) = get_bal(x))
       <2>1 prove (get1st(update(x, a)) = BA!update(get1st(x), a))
            by definition of update, get1st
       <2>e qed by step  <2>1 definition of over, get_bal property BA!ba_upd_nosucc_with_over
  <1>e conclude;
 proof of ba_upd_succ_with_zero =
  <1>1 assume x : Self, assume a : int,
       hypothesis H1 : (a >= 0),
       prove (get_bal(update(x, a)) = (get_bal(x) + a))
       <2>1 prove (get1st(update(x, a)) = BA!update(get1st(x), a))
            by definition of update, get1st
       <2>2 prove (BA!get_bal(BA!update(get1st(x), a)) = (BA!get_bal(get1st(x)) + a))
            by property BA!ba_upd_succ_with_zero hypothesis H1
       <2>e qed by step  <2>1, <2>2 definition of get_bal
  <1>e conclude;
 proof of cu_upd_cur_succ =
  <1>1 assume x : Self, assume c : cur, assume a : int,
       hypothesis H1 : (a = (get_bal(x) * (convert_cur(c) - convert_cur(get_cur(x))))),
       prove ((get_cur(upd_cur(x, c)) = c) && (get_bal(upd_cur(x, c)) = (get_bal(x) + a)))
       <2>1 prove (BA!get_bal(BA!makeBA((get_bal(x) + a))) = (get_bal(x) + a))
            assumed
       <2>e qed by step  <2>1 hypothesis H1 definition of upd_cur, get_cur, get_bal, get1st
  <1>e conclude;
end;;

collection CU_col =
  implement CU_imp (BA_col);
end;;
```

Listing A.14: CU in FoCaLiZe

```
species DLCU_spec1 (BA is BA_imp, CU is CU_imp (BA)) =
 inherit CU_spec1 (BA);
 signature limit_with : int;
 signature get_with : Self -> int;
end;;

species DLCU_spec2 (BA is BA_imp, CU is CU_imp (BA)) =
 inherit DLCU_spec1 (BA, CU);
```

```
9   property cu_upd_cur_succ : all x : Self, all c : cur, ((get_cur(upd_cur(x, c)) = c) && (get_bal(
        upd_cur(x, c)) = (get_bal(x) + (get_bal(x) * (convert_cur(c) - convert_cur(get_cur(x)))))))
        ;
10  property ba_upd_succ_with_zero : all x : Self, all a : int, (a >= 0) -> (get_bal(update(x, a)) =
        (get_bal(x) + a));
11  property dl_upd_succ_with_wlimit_R1 : all x : Self, all a : int, (a <= 0) && ((get_with(x) + a)
        >= limit_with) -> ((get_bal(x) + a) >= over) -> (get_bal(update(x, a)) = (get_bal(x) + a));
12  property dl_upd_nosucc_with_wlimit_R1 : all x : Self, all a : int, (a <= 0) && ((get_with(x) + a
        ) >= limit_with) -> ((get_bal(x) + a) < over) -> (get_bal(update(x, a)) = get_bal(x));
13  property dl_upd_nosucc_ls_wlimit : all x : Self, all a : int, ((a <= 0) && ((get_with(x) + a) <
        limit_with)) -> (get_with(update(x, a)) = get_with(x));
14  property dl_upd_succ_gr_wlimit : all x : Self, all a : int, ((a <= 0) && ((get_with(x) + a) >=
        limit_with)) -> (get_with(update(x, a)) = (get_with(x) + a));
15 end;;
16
17 species DLCU_imp (BA is BA_imp, CU is CU_imp (BA)) =
18  inherit DLCU_spec2 (BA, CU);
19  representation = CU* int;
20  let make (x1: CU, x2: int): Self = (x1, x2);
21  let get1st (x1: Self): CU = fst(x1);
22  let get2nd (x1: Self): int = snd(x1);
23  let get_bal (x) = CU!get_bal(get1st(x));
24  let over = CU!over;
25  let get_cur (x) = CU!get_cur(get1st(x));
26  let makeCU (ba, c) = (CU!makeCU(ba, c), 0);
27  let upd_cur (x, c) = (CU!upd_cur(get1st(x), c), get2nd(x));
28  let limit_with = 70;
29  let get_with (x) = snd(x);
30  let makeBA (amount) = (CU!makeBA(amount), 0);
31  let update (x, a) =
32   if (a <= 0) then
33    if ((get_with(x) + a) >= limit_with) then (CU!update(get1st(x), a), (get_with(x) + a)) else x
34   else (CU!update(get1st(x), a), get_with(x));
35
36 //****proof is reused and automatically generated from CU. Zenon error: cannot infer a value of
       type _p_CU_T.
37  proof of cu_upd_cur_succ = by definition of get_cur, upd_cur, get_bal, convert_cur, get1st
        property CU!cu_upd_cur_succ;
38 // The proof is done manually as follows:
39 (* proof of cu_upd_cur_succ =
40    <1>1 assume x: Self, c : cur, a : int,
41         hypothesis H1: a = get_bal (x) * (convert_cur(c) - convert_cur(get_cur(x))),
42         prove (get_cur(upd_cur(x,c)) = c) && (get_bal(upd_cur(x,c)) = get_bal(x) + a)
43      <2>1 prove CU!get_cur(CU!upd_cur(get1st(x), c)) = c && CU!get_bal(CU!upd_cur(get1st(x), c))
             = CU!get_bal(get1st(x)) + a
44      by hypothesis H1 definition of get_bal, get_cur property CU!cu_upd_cur_succ
45      <2>e qed by step <2>1 hypothesis H1 definition of upd_cur, get_cur, get_bal, get1st
46    <1>e conclude;
47 *)
48
49  proof of ba_bal_gr_over = by definition of get_bal, over property CU!ba_bal_gr_over;
50  proof of dl_upd_succ_with_wlimit_R1 =
51   <1>1 assume x : Self, assume a : int,
52        hypothesis h1 : (a <= 0) && ((get_with(x) + a) >= limit_with),
53        prove ((get_bal(x) + a) >= over) -> (get_bal(update(x, a)) = (get_bal(x) + a))
54        <2>1 prove (get1st(update(x, a)) = CU!update(get1st(x), a))
55             by definition of get1st, update hypothesis h1
56        <2>e qed by step  <2>1 definition of over, get_bal property CU!ba_upd_succ_with_over
57   <1>e conclude;
58  proof of dl_upd_nosucc_with_wlimit_R1 =
59   <1>1 assume x : Self, assume a : int,
60        hypothesis h1 : (a <= 0) && ((get_with(x) + a) >= limit_with),
61        prove ((get_bal(x) + a) < over) -> (get_bal(update(x, a)) = get_bal(x))
62        <2>1 prove (get1st(update(x, a)) = CU!update(get1st(x), a))
63             by definition of update, get1st hypothesis h1
64        <2>e qed by step  <2>1 definition of over, get_bal property CU!ba_upd_nosucc_with_over
65   <1>e conclude;
66  proof of ba_upd_succ_with_zero =
67   <1>1 assume x : Self, assume a : int,
68        hypothesis H1 : (a >= 0),
69        prove (get_bal(update(x, a)) = (get_bal(x) + a))
70        <2>1 prove (get1st(update(x, a)) = CU!update(get1st(x), a))
71             by definition of update, get1st property int_ge_le_eq3 hypothesis H1
72        <2>2 prove (CU!get_bal(CU!update(get1st(x), a)) = (CU!get_bal(get1st(x)) + a))
73 by property CU!ba_upd_succ_with_zero hypothesis H1
74        <2>e qed by step  <2>1, <2>2 definition of get_bal
75   <1>e conclude;
76  proof of dl_upd_nosucc_ls_wlimit = by definition of update, get_with property int_ge_le_eq;
77  proof of dl_upd_succ_gr_wlimit = by definition of update, get_with;
78 end;;
79
80 collection DLCU_col =
81   implement DLCU_imp (BA_col, CU_col);
82 end;;
```

Listing A.15: DLCU in FoCaLiZe

```
1  species DLLLCU_spec1 (BA is BA_imp, CU is CU_imp (BA), LLCU is LLCU_imp (BA, CU)) =
2   inherit LLCU_spec1 (BA, CU);
3   signature limit_with : int;
4   signature get_with : Self -> int;
5  end;;
6
7  species DLLLCU_spec2 (BA is BA_imp, CU is CU_imp (BA), LLCU is LLCU_imp (BA, CU)) =
8   inherit DLLLCU_spec1 (BA, CU, LLCU);
9   property cu_upd_cur_succ : all x : Self, all c : cur, ((get_cur(upd_cur(x, c)) = c) && (get_bal(
        upd_cur(x, c)) = (get_bal(x) + (get_bal(x) * (convert_cur(c) - convert_cur(get_cur(x)))))))
        ;
10  property ba_upd_succ_with_zero : all x : Self, all a : int, (a >= 0) -> (get_bal(update(x, a)) =
        (get_bal(x) + a));
11  property ll_upd_nosucc_with_ls_llimit_C1 : all x : Self, all a : int, (a <= 0) && ((get_with(x)
        + a) >= limit_with) -> ((a < 0) && (a > limit_low)) -> (get_bal(update(x, a)) = get_bal(x))
        ;
12  property dl_upd_succ_with_wlimit_R1 : all x : Self, all a : int, (a <= 0) && ((get_with(x) + a)
        >= limit_with) -> ((a >= 0) || (a <= limit_low)) -> ((get_bal(x) + a) >= over) -> (get_bal(
        update(x, a)) = (get_bal(x) + a));
13  property dl_upd_nosucc_with_wlimit_R1 : all x : Self, all a : int, (a <= 0) && ((get_with(x) + a
        ) >= limit_with) -> ((a >= 0) || (a <= limit_low)) -> ((get_bal(x) + a) < over) -> (get_bal
        (update(x, a)) = get_bal(x));
14  property dl_upd_nosucc_ls_wlimit : all x : Self, all a : int, ((a <= 0) && ((get_with(x) + a) <
        limit_with)) -> (get_with(update(x, a)) = get_with(x));
15  property dl_upd_succ_gr_wlimit : all x : Self, all a : int, ((a <= 0) && ((get_with(x) + a) >=
        limit_with)) -> (get_with(update(x, a)) = (get_with(x) + a));
16 end;;
17
18 species DLLLCU_imp (BA is BA_imp, CU is CU_imp (BA), LLCU is LLCU_imp (BA, CU)) =
19  inherit DLLLCU_spec2 (BA, CU, LLCU);
20  representation = LLCU* int;
21  let make (x1: LLCU, x2: int): Self = (x1, x2);
22  let get1st (x1: Self): LLCU = fst(x1);
23  let get2nd (x1: Self): int = snd(x1);
24  let get_bal (x) = LLCU!get_bal(get1st(x));
25  let over = LLCU!over;
26  let get_cur (x) = LLCU!get_cur(get1st(x));
27  let makeCU (ba, c) = (LLCU!makeCU(ba, c), 0);
28  let upd_cur (x, c) = (LLCU!upd_cur(get1st(x), c), get2nd(x));
29  let limit_low = LLCU!limit_low;
30  let limit_with = 70;
31  let get_with (x) = snd(x);
32  let makeBA (amount) = (LLCU!makeBA(amount), 0);
33  let update (x, a) =
34   if (a <= 0) then
35    if ((get_with(x) + a) >= limit_with) then (LLCU!update(get1st(x), a), (get_with(x) + a)) else
          x
36   else (LLCU!update(get1st(x), a), get_with(x));
37
38 //**** proof is reused and automatically generated from LLCU. Zenon error: cannot infer a value
       for a variable of type _p_LLCU_T.
39  proof of cu_upd_cur_succ = by definition of get_cur, upd_cur, get_bal, convert_cur, get1st
       property LLCU!cu_upd_cur_succ;
40 //The proof is done manually
41 (* proof of cu_upd_cur_succ =
42    <1>1 assume  x: Self, c : cur, a : int,
43         hypothesis H1: a = get_bal (x) * (convert_cur(c) - convert_cur(get_cur(x))),
44         prove (get_cur(upd_cur(x,c)) = c) && (get_bal(upd_cur(x,c)) = get_bal(x) + a)
45      <2>1 prove LLCU!get_cur(LLCU!upd_cur(get1st(x), c)) = c && LLCU!get_bal(LLCU!upd_cur(get1st(
             x), c)) = LLCU!get_bal(get1st(x)) + a
46       by hypothesis H1 definition of get_bal, get_cur property LLCU!cu_upd_cur_succ
47       <2>e qed by step <2>1 hypothesis H1 definition of upd_cur, get_cur, get_bal, get1st
48    <1>e conclude;
49  *)
50
51  proof of ba_bal_gr_over = by definition of get_bal, over property LLCU!ba_bal_gr_over;
52  proof of ll_upd_nosucc_with_ls_llimit_C1 =
53   <1>1 assume x : Self, assume a : int,
54        hypothesis h1 : (a <= 0) && ((get_with(x) + a) >= limit_with),
55        prove ((a < 0) && (a > limit_low)) -> (get_bal(update(x, a)) = get_bal(x))
56        <2>1 prove (get1st(update(x, a)) = LLCU!update(get1st(x), a))
57             by definition of update, get1st hypothesis h1
58        <2>e qed by step  <2>1 definition of over, get_bal, limit_low, update property LLCU!
               ll_upd_nosucc_with_ls_llimit
59   <1>e conclude;
60  proof of dl_upd_succ_with_wlimit_R1 =
61   <1>1 assume x : Self, assume a : int,
62        hypothesis h1 : (a <= 0) && ((get_with(x) + a) >= limit_with),
63        prove ((a >= 0) || (a <= limit_low)) -> ((get_bal(x) + a) >= over) -> (get_bal(update(x, a
             )) = (get_bal(x) + a))
64        <2>1 prove (get1st(update(x, a)) = LLCU!update(get1st(x), a))
65             by definition of get1st, update hypothesis h1
66        <2>e qed by step  <2>1 definition of over, get_bal definition of limit_low property LLCU!
               ll_upd_succ_with_llimit_R1
67   <1>e conclude;
68  proof of dl_upd_nosucc_with_wlimit_R1 =
69   <1>1 assume x : Self, assume a : int,
```

```
70          hypothesis h1 : (a <= 0) && ((get_with(x) + a) >= limit_with),
71          prove ((a >= 0) || (a <= limit_low)) -> ((get_bal(x) + a) < over) -> (get_bal(update(x, a)
                ) = get_bal(x))
72          <2>1 prove (get1st(update(x, a)) = LLCU!update(get1st(x), a))
73               by definition of update, get1st hypothesis h1
74          <2>e qed by step  <2>1 definition of over, get_bal definition of limit_low property LLCU!
                ll_upd_nosucc_with_llimit_R1
75    <1>e conclude;
76   proof of ba_upd_succ_with_zero =
77    <1>1 assume x : Self, assume a : int,
78          hypothesis H1 : (a >= 0),
79          prove (get_bal(update(x, a)) = (get_bal(x) + a))
80          <2>1 prove (get1st(update(x, a)) = LLCU!update(get1st(x), a))
81               by definition of update, get1st property int_ge_le_eq3 hypothesis H1
82          <2>2 prove (LLCU!get_bal(LLCU!update(get1st(x), a)) = (LLCU!get_bal(get1st(x)) + a))
83  by property LLCU!ba_upd_succ_with_zero hypothesis H1
84          <2>e qed by step  <2>1,  <2>2 definition of get_bal
85    <1>e conclude;
86   proof of dl_upd_nosucc_ls_wlimit = by definition of update, get_with property int_ge_le_eq;
87   proof of dl_upd_succ_gr_wlimit = by definition of update, get_with;
88  end;;
89
90  collection DLLLCU_col =
91    implement DLLLCU_imp (BA_col, CU_col, LLCU_col);
92  end;;
```

Listing A.16: DLLLCU in FoCaLiZe

A.3 Poker Product Line

We developed the product line of poker game from scratch. The features are designated in order to cover all practical variants of the game. We describe here all features of Poker SPL. We also give the details of the models and some products of this product line in FFML.

BasicPoker

Although Poker has many variants, they share some basic rules. We specify the simple rules that are applicable to all types of poker game into the BasicPoker feature. The basic poker game is played with a standard 52-card pack of playing cards. Poker is totally a gambling game, however at BasicPoker the players do not need to use money. Poker is ideally played with 4-7 players. A pack is being dealt one time for each player in turn with 5 cards (by default). The cards held by a player are called "the hole cards" that are distributed face down and not seen by the other players.

The winner is determined according to the score of Poker hands. The various combinations of Poker hands are listed in Table A.1 with ranks from one pair to five of a kind (the highest rank). The case when two hands have the same rank are described in the last column of the table. In such cases, the ranking of the other cards in the hand determines who wins. For example: 9, 9, 7, 4, 2 beats 9, 9, 5, 3, 2 which is decided by the third card.

BasicMPoker

The feature BasicMPoker is built as a child of BasicPoker. The players joining the game can use money (chips) for gambling. A betting round takes place each time before or after a dealing in which the players have an opportunity to bet on their hands. The rule is that a player puts his chips into a pot then the next player to the left, in turn, must either "call" that bet by putting into the pot the same number of chips; or "raise", which means that the next player puts in the pot more chips to call; or "drop" ("fold"), which means that the next player puts no chips in the pot, discards his hand, and is out of the game. Lastly, the player having the highest score will be the winner and receives all the chips in

Ranks of Hand	**Description**	**If two hands show the same rank**
One Pair	2 cards of the same value	The hand with the higher value pair wins. If they are the same, then the highest value card in the remaining 3 cards determines who wins. If they are also the same, the highest value card in the remaining 2 cards determines who wins and so on.
Two Pairs	2 different value pairs	The hand with the highest value pair wins. If they are the same, then the value of the second pair determines who wins. If they are also the same, then the value of the remaining card determines who wins.
Three of a kind	3 cards of the same value	The hand with the higher 3 cards wins.
Straight	A run of 5 cards, regardless of suit	The Straight that starts with the highest value card wins.
Flush	5 cards of the same suit	The hand with the highest value card wins or if the highest cards are the same, the value of the second highest cards determines the winner and so on.
Full House	3 cards of the same value and 2 cards of the same different value.	The hand with the higher 3 cards wins.
Four of a kind	4 cards of the same value	The hand with the higher matching 4 cards wins.
Straight Flush	A run of 5 cards of the same suit	The Straight Flush that starts with the highest value card wins.
Royal Flush	A run of 5 cards of the same suit starting with a ten.	n/a

Table A.1: Poker Ranking Hands

the pot.

Basic36Poker

Basic36Poker is built as a child of BasicPoker, it is played with a 36-card pack of playing cards in which the cards from 2s to 5s are removed. In simple words, the feature Basic36Poker is built from BasicPoker but only the total cards in a pack is updated. The number of total cards is declared as an integer number in BasicPoker then defined with 52. However, in Basic36Poker the declaration of this number is inherited and the total cards are redefined by 36. So, Basic36Poker can be chosen together with any of the other features modifying the number of cards of the pack.

BasicWPoker

BasicWPoker is built as a child of BasicPoker. After the fourth round of dealing cards, a card is put on the table and this card is seen by all players. The other three cards having the same rank as the card are wild. The last final round of cards is dealt to give the fifth card to the players. For example, if a card 9 of hearts is the one showed up on the table, others namely 9 of clubs, 9 of diamonds and 9 of spades are wild cards. Betting rounds can be set up after showing the wild cards and after dealing the last round.

The ranking of a hand is determined by the cards held by a player. Wild cards can be seen as any card. The card combination is similar to BasicPoker's but the last combination, the highest possible hand, is "Five of a Kind" instead of "The Royal Flush". This can occur only when at least one card is wild. For example, "Five of a Kind" would be four cards 10s and a wild card.

MFormPoker

MFormPoker is built as a child of BasicMPoker. In the game, after dealing cards to each player, some cards (1, 2 or 3) continue to be taken from the pack and faced up on the table. These cards are called "community cards". They are seen by all players and combined with the hole cards held by each player to calculate his/her score. Two activities which are added into a betting round, are "check" that allows a player to follow the game

but not to put chips into the pot and "all in" that allows a player to put all chips he has into the pot.

DrawMPoker

DrawPoker is a child of MFormPoker in which the community card list is empty. Players can join a betting round before the first dealing. The first bet is determined by a number, called "ante" and decided by all players. The next player will put down another bet, double the ante. Instead of dealing community cards, some dealing rounds allow a player to discard one to three cards (1, 2 or 3).

TexasHold'emMPoker

A popular poker is TexasHold'emMPoker that is designated as a child of DrawMPoker. The first dealing turn, called "preflop", gives 2 hole cards to each player. The second dealing turn, called "flop", deals 2 community cards on the table. The third dealing turn, called "turn", gives one more community card. The forth dealing turn, called "river", give the last community card. After each dealing turn, there is a betting round. However, in the final betting round, the number of chips for raising can be increased by players.

Other features of Poker SPL, such as FixedLimit, PotLimit, StudMPoker, PreFixedLimit and Roodles are described in Appendix A.3.

FixedLimit

FixedLimit is a child of BasicMPoker that allows fixing a limit of a *raise* for each bet time.

- The module FixedLimit is built from BasicMPoker. Its representation type is the same as BasicMPoker's.

- The function *bettingRound* is redefined by adding a raise limit *raiseLimit*.

PotLimit

BasicPoker has one more child called PotLimit that is added a rule for each *raise*. The rule is defined each time when a player bets based on the amount of the chips in the *pot*.

- The module FixedLimit is built from BasicMPoker and keeps the representation type.
- The function *bettingRound* is redefined by adding a new function *raiseMin* that defines the bet rule relating to the chips in the *pot*.

StudMPoker

StudMPoker is also a child of MFormPoker in which there are community cards. The two cards, dealt in the first dealing round for each player, are hole cards, but one is faced down and one faced up. The faced-up card decides who makes the first bet of the first betting round. The second, third and fourth dealing turns give totally 3 community cards on the table (one card for each turn).

- The module StudMPoker is built from and has the same representation type as MFormPoker.
- The community card list *comCards* is not empty.
- The function *bettingRound* is redefined with activating the case that the first faced-up hole card decides who makes the first bet of the first betting round.

PreFixedLimit

In fact, in DrawWPoker a bet can be limited by an "**ante**". We design PreFixedLimit as a child feature of DrawWPoker in which the pre-betting round has an ante limit which is decided by all the player.

- The module PreFixedLimit is built from and has the same representation type as DrawMPoker.
- The function *bettingRound* is redefined with a case of bet type $betType = BAnte$ in which the chips of a bet are limited by an ante limit $anteLimit$.

Roodles

Roodles is a child of DrawWPoker whose rule is applied when a lucky player wants to bet more in the game. He can make a "**roodle**" by setting a rule, such as multiplication of "**ante**".

- The module Roodles is built from and has the same representation type as DrawMPoker.
- A function *setAnteLimit* sets up a new rule for *ante* (such as multiplication).
- The function bettingRound is redefined by adding a new rule in which the chips of a bet are limited by the function *setAnteLimit*.

Module Implementations in FFML

The analyzed modules of Poker SPL are implemented in FFML. We show here the modules BasicPoker (BP) in Listing A.17, BasicMPoker (BMP) in Listing A.18, Basic36Poker (B36P) in Listing A.19, BasicWPoker (BWP) in Listing A.20, MFormPoker (MFP) in Listing A.21, DrawMPoker (DMP) in Listing A.22 and TexasHold'emMPoker (TMP) in Listing A.23.

```
1  fmodule BP
2   signature totalCardNum : int;
3   signature flop : flopType -> BP -> int -> BP;
4   signature combination : card list -> int;
5   signature showdown : BP -> player;
6   signature endPoker : BP -> BP;
7   signature setPack : BP -> card list -> BP;
8   signature getPack : BP -> card list;
9   signature makePoker : player list -> card list -> BP;
10  signature playerGetsCards : player list -> card list -> int ->  player list;
11  signature getPlayers : BP -> player list;
12  signature isPlaying : player -> BP -> bool;
13  signature getPlayerId : BP -> int -> player;
14
15 contract getPack :: invariant property bp_getPack_succ : all players : player list , all pack :
       card list , getPack(makePoker(players , pack)) = pack;
16
17 contract setPack :: invariant property bp_setPack_totalCardNum : all x : BP, all cards : card
       list ,  getLength (getPack(setPack(x, cards))) = totalCardNum || getLength (getPack(setPack(x
       , cards))) = getLength (getPack(x));
18
19 contract flop :: property bp_flop_minusCards : all x : BP, all cardNum : int , all flopT :
       flopType , all sumNum : int , flopT = FBasic -> cardNum * getLength(getPlayers (x)) = sumNum
       -> getLength(getPack(flop(flopT , x, cardNum))) = getLength(removeElements(getPack(x), sumNum
       ));
20
21 contract showdown :: invariant property bp_showdown_notWinner : all x : BP, all p : player , ~(
       isPlaying (p, x)) -> ~(showdown (x) = p);
22
23  representation = player list * card list;
24
```

```
25  let totalCardNum = 52;
26  let setPack (x, cards) = if getLength (cards) = totalCardNum then (fst(x),cards) else x;
27
28  let combination (cards) =
29   if haveRoyalFlush (cards) then 9
30   else if haveStraightFlush (cards) then 8
31   else if haveFourOfAKind (cards) then 7
32   else if haveFullHouse (cards) then 6
33   else if haveFlush (cards) then 5
34   else if haveCardRun (cards) then 4
35   else if haveTriple (cards) then 3
36   else if have2Pairs (cards) then 2
37   else if have1Pair (cards) then 1
38   else 0;
39
40  let compare (cards1, cards2) = ...; // compare 2 card lists that show the same rank.
41
42  let flop (x, cardNum, turnNum) =
43   let playerNum = getLength (fst(x)) in
44   let newplayers = playerGetsCards (fst(x), snd(x), cardNum) in
45  (newplayers, removeElements(pack, cardNum * playerNum));
46
47  let showdown (x) =
48   let players = fst(x) in
49   let player0 = getPlayer(players) in
50   let rec loop (players : player list, player0) = match players with
51    | FNil -> player0
52    | FCons (p, l) ->
53     begin
54      let pcombine = combination (snd(p)) in
55      let pcombine0 = combination (snd(player0)) in
56   if  pcombine > pcombine0 then loop(l, p)
57   else if pcombine = pcombine0 then
58    if compare(snd(p), snd(player0)) then loop(l, p)
59    else loop(l, player0)
60      else loop(l, player0)
61     end in loop (players, player0);
62
63  let endPoker (x) = ...; // set the game back to the initial state.
64
65 proof of bp_getPack_succ = foc proof {* by definition of getPack, makePoker;*}
66
67 proof of bp_setPack_totalCardNum =
68 foc proof {*
69  <1>1 assume x: Self, cards : card list,
70       prove getLength (getPack(setPack(x, cards))) = totalCardNum || getLength (getPack(setPack(x
             , cards))) = getLength (getPack(x))
71  <2>1 hypothesis H1: getLength (cards) = totalCardNum,
72      prove getLength (getPack(setPack(x, cards))) = totalCardNum
73      by definition of getPack, setPack hypothesis H1
74  <2>2 hypothesis H2: ~(getLength (cards) = totalCardNum),
75      prove getLength (getPack(setPack(x, cards))) = getLength (getPack(x))
76      by definition of getPack, setPack hypothesis H2
77  <2>3 conclude
78  <1>e conclude;*}
79
80 proof of bp_flop_minusCards =
81 foc proof {*
82   <1>1 assume x: Self, cardNum : int, flopT : flopType, sumNum : int,
83        hypothesis H1: flopT = FBasic,
84        hypothesis H2: cardNum * getLength(getPlayers (x)) = sumNum,
85        prove getLength(getPack(flop(flopT, x, cardNum))) = getLength(removeElements(getPack(x),
              sumNum))
86        by definition of flop, getPack, makePoker hypothesis H1, H2
87   <1>e conclude; *)
88
89 proof of bp_showdown_notWinner = foc proof {* assumed; *}   // time limited to complete this
       proof
90 ;;
```

Listing A.17: BasicPoker (BP) in FFML

```
1  fmodule BMP from BP
2   signature getPot : BMP -> int;
3   signature bet : BMP -> int -> int -> BMP;
4   signature fold : BMP -> int -> BMP;
5   signature makeMPoker : BP -> int -> BMP;
6   signature bettingRound : BMP -> betType -> int -> int -> (int * int) list -> BMP;
7   signature giveToWinner : BMP -> int -> int -> BMP;
8
9   contract getPack :: invariant property bmp_getPack_succ : all p0 : BP, all pot : int, getPack(
        makeMPoker(p0, pot)) = BP!getPack(p0);
10
11  contract bet :: property bmp_bet_upd_pot : all x1 : BMP, all id : int, all amount : int, all x2
        : BMP,  all player1 : player, all player2 : player, x2 = bet (x1, id, amount) -> player1 =
```

```
      getPlayerId(x1, id) -> player2 = getPlayerId(x2, id) -> getPot(x2) = getPot(x1) + amount &&
       getAmount(player2) = getAmount(player1) - amount;
12
13  contract bettinground :: property bmp_bettinground_upd_pot : all x:BMP, all bType : betType, all
       raiseNum : int, all callRepeat : int, all bettings : (int * int) list, all x2:BMP, x2 =
      bettingRound (x, bType, raiseNum, callRepeat, bettings) -> getPot(x2) = getPot(x) + sumBets
      (bettings);
14
15  representation extends BP with int; (* BP and pot *)
16
17  let makeMPoker (poker, pot) = (poker, pot);
18  let getPot (x) = snd (x);
19
20  let bet (x, id, amount) =
21   let players = getPlayers(x) in
22   let pack = getPack(x) in
23   let rec loop (players) = match players with
24    | FNil -> FNil
25    | FCons (p, l) -> if fst(fst(p)) = id then FCons (minusAmount(p, amount), loop(l))
26                      else FCons(p, loop(l)) in
27   makeMPoker(BP!makePoker(loop(players), pack), snd(x) + amount);
28
29  let fold (x, id) =
30   let players = getPlayers(x) in
31   let pack = getPack(x) in
32   let rec loop (players : player list) = match players with
33    | FNil -> FNil
34    | FCons (p, l) -> if fst(fst(p)) = id then loop(l)
35                      else FCons(p, loop(l)) in
36   makeMPoker(BP!makePoker(loop(players), pack),snd(x));
37
38  let rec secondBetting (x, raiseNum, callRepeat, lastid, list)  = ...; // a local function is
      used to check a bet
39
40 let bettingRound (x, betType, raiseNum, callRepeat, bettings) =
41 if betType = BBasic then
42  begin
43   if getLength(getPlayers(x)) <= 1 then x
44   else
45    match bettings with
46     | FNil -> x
47     | FCons ((id, betAmount), l) ->
48       if ~(inPlayers (id, getPlayers(x))) then x
49       else if ~(betAmount > 0) then x
50       else  secondBetting (bet(x,id,betAmount), raiseNum, callRepeat, id, l)
51  end else x;
52
53 let giveToWinner (x, id, sumPot) =
54   let rec loop (players) = match players with
55    | FNil -> FNil
56    | FCons (p, l) ->
57      if fst(fst(p)) = id then FCons (minusAmount(p, 0 - sumPot), loop(l))
58      else FCons(p, loop(l)) in makeMPoker(BP!makePoker(loop( getPlayers(x)), getPack(x)), snd(x)
           );
59
60 proof of bmp_getPack_succ =
61 foc proof {*
62   <1>1 assume p0 : BP, pot : int,
63        prove getPack(makeMPoker(p0, pot)) = BP!getPack(p0)
64     by definition of getPack, makeMPoker, get1st
65   <1>e conclude; *}
66
67 proof of bmp_bet_upd_pot =
68 foc proof {*
69   <1>1 assume x1 : Self, id : int, amount : int, x2 : Self, player1 : player, player2 : player,
70        hypothesis H1: x2 = bet (x1, id, amount),
71        hypothesis H2: player1 = getPlayerId(x1, id),
72        hypothesis H3: player2 = getPlayerId(x2, id),
73        prove getPot(x2) = getPot(x1) + amount && getAmount(player2) = getAmount(player1) - amount
74        <2>1 prove getPot(x2) = getPot(x1) + amount
75             assumed (* by hypothesis H1 definition of bet, makeMPoker, getPot*)
76        <2>2 prove getAmount(player2) = getAmount(player1) - amount
77             assumed
78        <2>e conclude
79   <1>e conclude; *)
80
81 proof of bmp_bettinground_upd_pot = foc proof {* assumed;*}
82
83 proof of bp_getPack_succ =
84 foc proof {*
85   <1>1 assume players : player list, desk : card list,
86       prove getPack(makePoker(players, desk)) = desk
87     <2>1 prove getPack(makePoker(players, desk)) = BP!getPack(BP!makePoker(players, desk))
88          by definition of getPack, makePoker, get1st
89     <2>e qed by step <2>1 property BP!bp_getPack_succ
90  <1>e conclude; *}
91
92 proof of bp_flop_minusCards =
```

```
93  foc proof {*
94    <1>1 assume x: Self, cardNum : int, flopT : flopType, sumNum : int,
95         prove flopT = FBasic -> cardNum * getLength(getPlayers (x)) = sumNum -> getLength(getPack(
                 flop(flopT, x, cardNum))) = getLength(removeElements(getPack(x), sumNum))
96         <2>1 prove getPack (flop(flopT, x, cardNum)) = BP!getPack(BP!flop(flopT, get1st(x),
                cardNum))
97              by definition of flop, get1st, getPack
98         <2>e qed by step <2>1 definition of getPlayers, getPack property BP!bp_flop_minusCards
99    <1>e conclude; *}
100 ;;
```

Listing A.18: BasicMPoker (BMP) in FFML

```
1   fmodule B36P from BP
2    representation = BP;
3    let totalCardNum = 36;
4
5    proof of bp_setPack_totalCardNum = foc proof {* assumed;*}
6
7    proof of bp_getPack_succ =
8    foc proof {*
9     <1>1 assume players : player list, assume desk : card list,
10   prove (getPack(makePoker(players, desk)) = desk)
11     <2>1 prove (getPack(makePoker(players, desk)) = BP!getPack(BP!makePoker(players, desk)))
12          by definition of getPack, makePoker
13     <2>e qed by step <2>1 property BP!bp_getPack_succ
14   <1>e conclude; *}
15
16   proof of bp_flop_minusCards =
17   foc proof {*
18    <1>1 assume x : Self, assume cardNum : int, assume flopT : flopType, assume sumNum : int,
19         prove (flopT = FBasic) -> ((cardNum * getLength(getPlayers(x))) = sumNum) -> (getLength(
                 getPack(flop(flopT, x, cardNum))) = getLength(removeElements(getPack(x), sumNum)))
20      <2>1 prove (getPack(flop(flopT, x, cardNum)) = BP!getPack(BP!flop(flopT, x, cardNum)))
21           by definition of flop, getPack
22      <2>e qed by step  <2>1 definition of getPlayers, getPack property BP!bp_flop_minusCards
23   <1>e conclude; *}
24  ;;
```

Listing A.19: Basic36Poker (B36P) in FFML

```
1   fmodule BWP from BP
2    signature makeWCards : BWP -> BWP;
3    signature getWCards : BWP -> card list;
4    signature wCombination : card list -> card list -> bool * int;
5    signature makeWPoker : BP -> card list -> BWP;
6
7    contract makeWCards :: property bwp_make_wcards : all x : BWP, all x2 : BWP, x2 = makeWCards(x)
         -> isAKind (FCons (getElement(getPack(x),1), getWCards (x2)));
8
9    representation extends BP with card list; // BP and wild cards
10
11   let makeWCards (x) = .. ; // make wild cards
12   let getWCards (x) = snd (x);
13   let makeWPoker (poker, wCards) = (poker, wCards);
14   let wCombination (cards, wcards) = ...; // combination of cards and wild cards
15
16   let showdown(x) =
17     let players = getPlayers(x) in
18     let player0 = getPlayer(players) in
19     let rec loop (players : player list, player0) = match players with
20      | FNil -> player0
21      | FCons (p, l) ->
22       begin
23        if fst(wCombination (snd(p), wCards)) then
24          if snd(wCombination (snd(p), wCards)) > snd(wCombination(snd(player0), getWCards(x)))
                then  loop(l, p)
25       else loop (l, player0)
26        else loop (l, player0)
27       end in
28     if (fst(wCombination(snd(loop(players, player0)), wCards)) = false) then BP!showdown(x)
29     else loop(players, player0);
30
31   proof of bwp_make_wcards = foc proof {* assumed;*}
32
33   proof of bp_getPack_succ =
34   foc proof {*
35     <1>1 assume players : player list, desk : card list,
36          prove getPack(makePoker(players, desk)) = desk
37       <2>1 prove getPack(makePoker(players, desk)) = BP!getPack(BP!makePoker(players, desk))
38            assumed (* by definition of getPack, makePoker, get1st *)
39       <2>e qed by step <2>1 property BP!bp_getPack_succ
40    <1>e conclude; *}
```

```
41
42 proof of bp_flop_minusCards =
43 foc proof {*
44   <1>1 assume x: Self, cardNum : int, flopT : flopType, sumNum : int,
45        prove flopT = FBasic -> cardNum * getLength(getPlayers (x)) = sumNum -> getLength(getPack(
             flop(flopT, x, cardNum))) = getLength(removeElements(getPack(x), sumNum))
46        <2>1 prove getPack (flop(flopT, x, cardNum)) = BP!getPack(BP!flop(flopT, get1st(x),
             cardNum))
47             by definition of flop, get1st, getPack
48        <2>e qed by step <2>1 definition of getPlayers, getPack property BP!bp_flop_minusCards
49   <1>e conclude; *}
50
51  proof of bp_showdown_notWinner =
52  foc proof {*
53   <1>1 assume x : Self, p : player, players: player list, player0 : player, wCards : card list,
54        hypothesis H1 : wCards = getWCards(x),
55        hypothesis H2 : players = getPlayers(x),
56        hypothesis H3 : player0 = getPlayer(players),
57        prove  ~ (isPlaying (p, x)) -> ~ (showdown (x) = p)
58        <2>1 hypothesis H21: (fst(wCombination(snd(findPlayer(players, player0, wCards)), wCards))
               = false),
59             prove  ~ (isPlaying (p, x)) -> ~ (showdown (x) = p)
60
61          <3>1 prove showdown(x) = BP!showdown(get1st(x))
62            by definition of showdown hypothesis H1, H2, H3, H21
63             <3>2 prove  isPlaying (p, x) =  BP!isPlaying (p, get1st(x))
64            by definition of isPlaying
65             <3>3 prove  ~ (BP!isPlaying (p, get1st(x))) -> ~ (BP!showdown (get1st(x)) = p)
66            by property BP!bp_showdown_notWinner
67             <3>4 prove  (~ (BP!isPlaying (p, get1st(x))) -> ~ (BP!showdown (get1st(x)) = p)) ->
                    (~ (isPlaying (p, x)) -> ~ (showdown (x) = p))
68            (* by property update_equal step <3>1, <3>2*) assumed
69          <3>e conclude
70        <2>2 hypothesis H22: ~ (fst(wCombination(snd(findPlayer(players, player0, wCards)), wCards
              )) = false),
71             prove  ~ (isPlaying (p, x)) -> ~ (showdown (x) = p)
72          assumed
73        <2>e conclude
74   <1>e conclude; *}
```

Listing A.20: BasicWPoker (BWP) in FFML

```
1  fmodule MFP from BMP
2   signature makeMPokerForm : BMP -> card list -> MFP; (* BMP -> common cards -> MFP *)
3   signature getCommonCards : MFP -> card list; (* MFP -> common  cards *)
4
5  contract getPack :: property mfp_getPack_succ : all p1 : BMP, all ccards : card list, getPack(
       makeMPokerForm(p1, ccards)) = BMP!getPack(p1);
6
7  contract flop :: property mfp_flop_addComCards : all x : MFP, all cardNum : int, all flopT :
       flopType, all x2: MFP, flopT = FCom -> flop (flopT, x, cardNum) = x2 -> getPack (x2) =
       removeElements(getPack(x), cardNum) &&  getCommonCards(x2) = getNumCards(getPack(x), cardNum
       );
8
9  contract flop :: property mfp_flop_minusCards_R1
10  refines BMP!bp_flop_minusCards
11  extends premise flopT <> FCom;
12
13  representation extends BMP with card list; (* BMP x common cards *)
14
15  let makeMPokerForm (mpoker, commonCards) = (mpoker, commonCards);
16  let getCommonCards (x) = snd (x);
17
18  let flop (flopType, x, cardNum) =
19   if flopType = FCom then
20    begin
21     let pack = removeElements(getPack(x), cardNum) in
22     makeMPokerForm(BMP!makeMPoker
23        (BP!makePoker(getPlayers(x),
24        removeElements(getPack(x), cardNum)), getPot(x))
25     , getNumCards(getPack(x), cardNum))
26    end
27   else (BMP!flop(flopType,x, cardNum), getPot(x));
28
29  let showdown (x) =
30   let players = getPlayers(x) in
31   let player0 = getPlayer(players) in
32   let rec loop (players : player list , player0) = match players with
33     | FNil -> player0
34     | FCons (p, l) ->
35        begin
36         let pcombine = combination (join2Lists(snd(p), getCommonCards(x))) in
37      let pcombine0 = combination (join2Lists(snd(player0), getCommonCards(x))) in
38      if  pcombine > pcombine0 then loop(l, p)
39      else if pcombine = pcombine0 then
```

```
40          if BP!compareCombination(join2Lists(snd(p), getCommonCards(x)) , join2Lists(snd(player0),
                getCommonCards(x))) then loop(l, p)
41      else loop(l, player0)
42          else loop(l, player0)
43          end in
44  loop (players, player0);
45
46  proof of mfp_flop_minusCards_R1 =
47  foc proof {*
48      <1>1 assume x: MFP, cardNum : int, flopT : flopType,
49          hypothesis H1: flopT <> FCom,
50          hypothesis H2: flopT = FBasic,
51          prove getLength(getPack(flop(flopT, x, cardNum))) = getLength(getPack(x)) - cardNum *
                getLength(getPlayers(x))
52          <2>1 prove fst(flop(flopT, x, cardNum)) = P1!flop(flopT, get1st(x), cardNum)
53               by definition of get1st, flop hypothesis H1 property flopType_neq
54          <2>2 prove getLength(P1!getPack(P1!flop(flopT, get1st(x), cardNum))) = getLength( P1!
                getPack(get1st(x))) - cardNum * getLength(P1!getPlayers(get1st(x)))
55       by property P1!bmp_flop_minusCards_K1 hypothesis H2
56          <2>e qed by step <2>1, <2>2 definition of getPlayers, getPack, get1st
57      <1>e conclude; *}
58
59  proof of mfp_getPack_succ = foc proof {* by definition of getPack, makeMPokerForm, get1st; *}
60
61  proof of mfp_flop_addComCards =
62  foc proof {*
63   <1>1 assume x: Self, cardNum : int, flopT : flopType, x2 : Self,
64        hypothesis H1: flopT = FCom,
65        hypothesis H2: x2 = flop (flopT, x, cardNum),
66        prove getPack (x2) = removeElements(getPack(x), cardNum) && getCommonCards(x2) =
              getNumCards(getPack(x), cardNum)
67        <2>1 prove getCommonCards(x2) = getNumCards(getPack(x), cardNum)
68             by definition of flop, makeMPokerForm, getCommonCards hypothesis H1, H2
69        <2>2 prove getPack (x2) = removeElements(getPack(x), cardNum)
70             <3>1 prove getPack(x2) = BMP!getPack(BMP!makeMPoker (BP!makePoker(getPlayers(x),
                    removeElements(getPack(x), cardNum)), getPot(x)))
71            by property mfp_getPack_succ definition of flop, getPack, makeMPokerForm, get1st
                  hypothesis H1, H2
72           <3>2 prove BMP!getPack(BMP!makeMPoker (BP!makePoker(getPlayers(x), removeElements(getPack
                  (x), cardNum)), getPot(x))) = BP!getPack(BP!makePoker(getPlayers(x), removeElements(
                  getPack(x), cardNum)))
73             by property BMP!bmp_getPack_succ
74           <3>3 prove BP!getPack(BP!makePoker(getPlayers(x), removeElements(getPack(x), cardNum))) =
                   removeElements(getPack(x), cardNum)
75            by property BP!bp_getPack_succ
76              <3>e conclude
77        <2>3 conclude
78   <1>e conclude; *}
79
80  proof of mfp_flop_minusCards_R1 =
81  foc proof {*
82      <1>1 assume x: Self, cardNum : int, flopT : flopType, sumNum : int,
83          hypothesis H1: flopT <> FCom,
84          prove flopT = FBasic -> cardNum * getLength(getPlayers (x)) = sumNum -> getLength(getPack(
                flop(flopT, x, cardNum))) = getLength(removeElements(getPack(x), sumNum))
85          <2>1 prove getPack (flop(flopT, x, cardNum)) = BMP!getPack(BMP!flop(flopT, get1st(x),
                cardNum))
86               by definition of flop, get1st, getPack hypothesis H1 property flopType_neq
87          <2>e qed by step <2>1 hypothesis H1 definition of getPlayers, getPack property BMP!
                bp_flop_minusCards
88       <1>e conclude; *}
89
90  // done but ERROR of FoC
91  proof of bp_getPack_succ = foc proof {* assumed; *}
92  (* proof of bp_getPack_succ =
93        <1>1 assume players : player list, desk : card list,
94             prove getPack(makePoker(players, desk)) = desk
95      <2>1 prove getPack(makePoker(players, desk)) = P1!getPack(P1!makePoker(players, desk))
96       by definition of getPack, makePoker, get1st
97      <2>e qed by step <2>1 propety P1!bp_getPack_succ
98        <1>e conclude; *)
99
100 // error of Zenon
101 proof of bmp_bettingground_upd_pot =
102 foc proof {*
103   <1>1 assume x:Self, bType : betType, raiseNum : int, callRepeat : int, bettings : (int * int)
          list, x2:Self,
104         hypothesis H1 : x2 = bettingRound (x, bType, raiseNum, callRepeat, bettings),
105         prove getPot(x2) = getPot(x) + sumBets(bettings)
106         <2>1 prove get1st(x2) = BMP!bettingRound(get1st(x), bType, raiseNum, callRepeat, bettings)
107              by definition of bettingRound, get1st hypothesis H1
108         <2>2 prove getPot(x2) = BMP!getPot(get1st(x2))
109              by definition of getPot
110         <2>3 prove getPot(x) = BMP!getPot(get1st(x))
111              by definition of getPot
112         <2>4 assume x2BMP : BMP, xBMP : BMP,
113              hypothesis H2 : x2BMP = get1st(x2),
114              hypothesis H3 : xBMP = get1st(x),
```

```
115       prove x2BMP = BMP!bettingRound(xBMP, bType, raiseNum, callRepeat, bettings) -> BMP!getPot(
              x2BMP) = BMP!getPot(xBMP) + sumBets(bettings)
116             by property BMP!bmp_bettinground_upd_pot
117         <2>e conclude
118   <1>e conclude; *}
119
120 proof of bmp_bet_upd_pot =
121   <1>1 assume x1 : Self, id : int, amount : int, x2 : Self, player1 : player, player2 : player,
122         hypothesis H1: x2 = bet (x1, id, amount),
123         hypothesis H2: player1 = getPlayerId(x1, id),
124         hypothesis H3: player2 = getPlayerId(x2, id),
125         prove getPot(x2) = getPot(x1) + amount && getAmount(player2) = getAmount(player1) - amount
126         <2>1 assume x1BMP : BMP, x2BMP : BMP, player11 : player, player22 : player,
127               hypothesis H10 : x1BMP = get1st(x1),
128            hypothesis H11 : x2BMP = BMP!bet (x1BMP, id, amount),
129               hypothesis H12 : player11 = BMP!getPlayerId(x1BMP, id),
130               hypothesis H13 : player22 = BMP!getPlayerId(x2BMP, id),
131            prove getPot(x2) = getPot(x1) + amount && getAmount(player2) = getAmount(player1) -
                   amount
132           <3>1 prove get1st(x2) = x2BMP
133             <31>1 prove  get1st(x2) = BMP!bet (get1st(x1), id, amount)
134                   by definition of get1st, bet hypothesis H1
135             <31>2 prove BMP!bet (get1st(x1), id, amount) = x2BMP
136                   by hypothesis H11, H10
137             <31>e conclude
138           <3>2 prove player1 = player11
139             by definition of getPlayerId hypothesis H10, H2, H12
140           <3>3 prove player2 = player22
141             by definition of getPlayerId hypothesis H10, H3, H13 step <3>1
142         <3>10 prove BMP!getPot(x2BMP) = BMP!getPot(x1BMP) + amount && getAmount(player22) =
                  getAmount(player11) - amount
143        (* by property BMP!bmp_bet_upd_pot hypothesis H11, H12, H13 *) assumed
144         <3>e qed by hypothesis H10 step <3>1, <3>2, <3>3, <3>10 definition of getPot
145          <2>e conclude
146   <1>e conclude; *}
147
148 proof of bmp_getPack_succ =
149 foc proof {*
150   <1>1 assume p0 : BP, pot : int,
151         prove getPack(makeMPoker(p0, pot)) = BP!getPack(p0)
152      assumed
153      (* FoC + Zenon error:
154      <2>1 prove getPack(makeMPoker(p0, pot)) = BMP!getPack(BMP!makeMPoker(p0, pot))
155           by definition of getPack, get1st, makeMPoker
156         <2>2 prove BMP!getPack(BMP!makeMPoker(p0, pot)) = BP!getPack(p0)
157           by property BMP!bmp_getPack_succ
158         <2>e conclude *)
159   <1>e conclude; *}
160
161  proof of bp_setPack_totalCardNum =
162  foc proof {*
163    <1>1 assume x: Self, cards : card list,
164          prove getLength (getPack(setPack(x, cards))) = totalCardNum || getLength (getPack(setPack
                (x, cards))) = getLength (getPack(x))
165          by definition of totalCardNum, getPack, setPack, get1st property BMP!
                bp_setPack_totalCardNum
166    <1>e conclude; *}
```

Listing A.21: MFormPoker (MFP) in FFML

```
1  fmodule DMP from MFP
2   signature makeDMPoker : MFP -> int -> DMP;
3   signature discardCards : DMP -> int -> list (int) -> DMP;
4   signature getAnte : DMP -> int;
5   signature setAnte : DMP -> int -> DMP;
6   signature ante : int;
7
8   contract bettingRound :: property dmp_bettinground_upd_pot_R1
9   refines MFP!bmp_bettinground_upd_pot
10  extends premise bType <> BAnte;
11
12  representation extends MFP with int; (* MFP x ante *)
13
14  let makeDMPoker (mfpoker, ante) = (mfpoker, ante);
15  let getAnte (x) = snd(x);
16  let setAnte (x, ante) = makeDMPoker(fst(x), ante);
17
18  let checking_betting_chips (x, betlist : (int * int) list) = match betlist with
19    | FNil -> false
20    | FCons ((_, chips1), FNil) -> false
21    | FCons ((_, chips1), FCons ((_, chips2), list)) ->
22      if chips1 = getAnte(x) && chips2 = 2 * chips1 then
```

```
23        begin
24         let rec loop (list : (int * int) list, chips2) = match list with
25           | FNil -> true
26         | FCons ((_, chips3), l) -> (chips3 = 2 * chips2) && loop (l, chips3)
27         in loop (list, chips2)
28        end
29      else false;
30
31  let bettingRound (x, betType, raiseNum, callRepeat, bettings) =
32   if betType = BAnte then
33     if getAnte(x) >= 0 then
34        if checking_betting_chips (x, bettings) = true then
35           (MFP!bettingRound (x, betType, raiseNum, callRepeat, bettings), snd(x)) else x
36     else x
37   else (MFP!bettingRound (x, betType, raiseNum, callRepeat, bettings), snd(x));
38
39  let discardCards (x, playerId, places) =
40    let players = getPlayers (x) in
41    let pack = getPack (x) in
42    let rec loop_players (players : player list, pack : card list) = match players with
43     | FNil -> FNil
44     | FCons(((id, amount), playerCards), l) ->
45     begin
46      if playerId = id then
47       begin
48        let result = discard (pack, playerCards, places) in
49        FCons(((id, amount), snd(result)), loop_players (l, fst(result)))
50       end
51      else FCons(((id, amount), playerCards), loop_players (l, pack))
52     end in
53    let rec loop_pack (players : player list, pack : card list) = match players with
54     | FNil -> pack
55     | FCons(((id, amount), playerCards), l) ->
56     begin
57      if playerId = id then
58       begin
59        let result = discard (pack, playerCards, places) in
60        fst(result)
61       end
62      else (loop_pack (l, pack))
63     end in
64    let newPack = loop_pack (players, pack) in
65    let updatePlayers = loop_players (players, pack) in
66    makeDMPoker(MFP!makeMPokerForm(BMP!makeMPoker
67          (BP!makePoker(updatePlayers, newPack),getPot(x)), getCommonCards(x)),snd(x));
68
69  proof of dmp_bettinground_upd_pot_R1 =
70  foc proof {*
71   <1>1 assume x:Self, bType : betType, raiseNum : int, callRepeat : int, bettings : (int * int)
         list, x2:Self,
72        hypothesis H1 : (bType <> BAnte),
73        hypothesis H2 : (x2 = bettingRound(x, bType, raiseNum, callRepeat, bettings)),
74        prove (getPot(x2) = (getPot(x) + sumBets(bettings)))
75
76        <2>1 assume x1MFP : MFP, x2MFP : MFP,
77             hypothesis H3 : x1MFP = get1st(x),
78             hypothesis H4 : (x2MFP = MFP!bettingRound(get1st(x), bType, raiseNum, callRepeat,
                  bettings)),
79          prove (getPot(x2) = (getPot(x) + sumBets(bettings)))
80
81          <3>1 prove getPot(x2) = MFP!getPot(get1st(x2))
82               by definition of getPot
83          <3>2 prove getPot(x) = MFP!getPot(get1st(x))
84               by definition of getPot
85             <3>3 prove x2MFP = MFP!bettingRound(x1MFP, bType, raiseNum, callRepeat, bettings)
86                by definition of bettingRound, get1st hypothesis H1,H2, H3, H4 property
                     betType_neq
87             <3>4 prove get1st(x2) = x2MFP
88               by hypothesis H2, H4 definition of bettingRound, get1st hypothesis H1 property
                    betType_neq
89
90             <3>5 prove MFP!getPot(x2MFP)  = (MFP!getPot(x1MFP) + sumBets(bettings))
91                  assumed (* by property MFP!bmp_bettinground_upd_pot step <3>3 *)
92             <3>e qed by step <3>1, <3>2, <3>3, <3>4, <3>5 hypothesis H3
93        <2>e conclude
94   <1>e conclude; *}
95
96  proof of mfp_flop_addComCards =
97  foc proof {*
98    <1>1 assume x: Self, cardNum : int, flopT : flopType, x2 : Self,
99         hypothesis H1: flopT = FCom,
100        hypothesis H2: x2 = flop (flopT, x, cardNum),
101        prove getPack (x2) = removeElements(getPack(x), cardNum) && getCommonCards(x2) =
             getNumCards(getPack(x), cardNum)
102        <2>1 assume xMFP : MFP, x2MFP : MFP,
103             hypothesis H21N1: xMFP = get1st(x),
104          hypothesis H21N2: x2MFP = get1st(x2),
105          prove getPack (x2) = removeElements(getPack(x), cardNum) && getCommonCards(x2) =
```

```
                getNumCards(getPack(x), cardNum)
106           <3>1 prove getPack (x2) = removeElements(getPack(x), cardNum)
107             <4>1 prove get1st(x2) = MFP!flop(flopT, get1st(x), cardNum)
108                    by definition of flop, get1st hypothesis H2
109                <4>2 prove x2MFP = MFP!flop(flopT, xMFP, cardNum)
110                    by hypothesis H21N1, H21N2 step <4>1
111             <4>3 prove getPack(x) = MFP!getPack(xMFP)
112                    by definition of getPack hypothesis H21N1
113                <4>4 prove getPack(x2) = MFP!getPack(x2MFP)
114                    by definition of getPack hypothesis H21N2
115             <4>5 prove MFP!getPack(x2MFP) = removeElements(MFP!getPack(xMFP), cardNum)
116                    by hypothesis H1 step <4>2 property MFP!mfp_flop_addComCards
117             <4>e conclude
118           <3>2 prove getCommonCards(x2) = getNumCards(getPack(x), cardNum)
119             <4>1 prove get1st(x2) = MFP!flop(flopT, get1st(x), cardNum)
120                    by definition of flop, get1st hypothesis H2
121                <4>2 prove x2MFP = MFP!flop(flopT, xMFP, cardNum)
122                 by hypothesis H21N1, H21N2 step <4>1
123                 <4>3 prove getCommonCards(x2) = MFP!getCommonCards(x2MFP)
124                 by definition of getCommonCards hypothesis H21N2
125                   <4>4 prove getPack(x) = MFP!getPack(xMFP)
126                 by definition of getPack hypothesis H21N1
127                   <4>5 prove MFP!getCommonCards(x2MFP) = getNumCards( MFP!getPack(xMFP), cardNum)
128                 by hypothesis H1 step <4>2 property MFP!mfp_flop_addComCards
129               <4>e conclude
130         <3>e conclude
131       <2>e conclude
132     <1>e conclude; *}
133
134  proof of bmp_getPack_succ =
135  foc proof {*
136    <1>1 assume p0 : BP, pot : int,
137         prove getPack(makeMPoker(p0, pot)) = BP!getPack(p0)
138      <2>1 prove getPack(makeMPoker(p0, pot)) = MFP!getPack(MFP!makeMPoker(p0, pot))
139         by definition of getPack, get1st, makeMPoker
140         <2>2 prove MFP!getPack(MFP!makeMPoker(p0, pot)) = BP!getPack(p0)
141           by property MFP!bmp_getPack_succ
142         <2>e conclude
143    <1>e conclude;*}
```

Listing A.22: DrawMPoker(DMP) in FFML

```
1  fmodule THP from DMP
2   representation = DMP;
3   let bettingRound (x, betType, raiseNum, callRepeat, bettings) =
4    if betType = BFinal then x
5    else DMP!bettingRound (x, BBasic, raiseNum, callRepeat, bettings);
6  ;;
```

Listing A.23: TexasHold'emMPoker (THP) in FFML

Products of Poker SPL

We show the products generated automatically by FFML Product Generation tool. Listing A.24 is the implementation of the configuration $\{BP, B36P, BMP\}$. Listing A.25 is the implementation of the configuration $\{BP, BMP, BWP\}$. Listing A.26 is the implementation of the configuration $\{BP, B36P, BWP\}$. Listing A.27 is the implementation of the configuration $\{BP, BMP, MFP, BWP\}$. At the end of this section we give the details of the product B36PBMP of the configuration $\{BP, B36P, BMP\}$ translated into FoCaLiZe in Listing A.28.

```
1  fmodule B36PBMP from BMP
2   representation = BMP;
3   let totalCardNum  = 36;
4
5  proof of bp_getPack_succ =
6  foc proof {*
```

```
 <1>1 assume players : player list, assume desk : card list,
      prove (getPack(makePoker(players,desk)) = desk)
      <2>1 prove (getPack(makePoker(players,desk)) = BMP!getPack(BMP!makePoker(players,desk)))
           by definition of getPack, makePoker
      <2>e qed by step <2>1 property BMP!bp_getPack_succ
 <1>e conclude;*}

proof of bp_flop_minusCards =
foc proof {*
 <1>1 assume x : Self, assume cardNum : int, assume flopT : flopType, assume sumNum : int,
      prove (flopT = FBasic) -> ((cardNum * getLength(getPlayers(x))) = sumNum) -> (getLength(
          getPack(flop(flopT,x,cardNum))) = getLength(removeElements(getPack(x),sumNum)))
      <2>1 prove (getPack(flop(flopT,x,cardNum)) = BMP!getPack(BMP!flop(flopT,x,cardNum)))
           by definition of flop, getPack
      <2>e qed by step <2>1 definition of getPlayers, getPack property BMP!bp_flop_minusCards
 <1>e conclude; *}
;;
```

Listing A.24: B36PBMP in FFML

```
fmodule BMPBWP from BWP
 signature getPot : BMPBWP -> int;
 signature bet : BMPBWP -> int -> int -> BMPBWP;
 signature fold : BMPBWP -> int -> BMPBWP;
 signature makeMPoker : BWP -> int -> BMPBWP;
 signature bettingRound : BMPBWP -> betType -> int -> int -> (int * int) list -> BMPBWP;
 signature giveToWinner : BMPBWP -> int -> int -> BMPBWP;

 contract getPack :: invariant property bmp_getPack_succ : all p0:BWP, all pot:int, getPack(
     makeMPoker(p0,pot)) = BWP!getPack(p0);

 contract bet :: property bmp_bet_upd_pot : all x1:BMPBWP, all id:int, all amount:int, all x2:
     BMPBWP, all player1:player, all player2:player, x2 = bet(x1,id,amount) -> player1 =
     getPlayerId(x1,id) -> player2 = getPlayerId(x2,id) -> getPot(x2) = getPot(x1) + amount &&
     getAmount(player2) = getAmount(player1) - amount;

 contract bettinground :: property bmp_bettinground_upd_pot : all x:BMPBWP, all bType:betType,
     all raiseNum:int, all callRepeat:int, all bettings: (int * int) list, all x2:BMPBWP, x2 =
     bettingRound(x,bType,raiseNum,callRepeat,bettings) -> getPot(x2) = getPot(x) + sumBets(
     bettings);

 representation extends BWP with int;
 let getPot (x) = snd(x);

 let bet (x , id , amount) =
  let players = getPlayers(x) in
  let pack = getPack(x) in
  let rec loop (players) = match players with
   | FNil -> FNil
   | FCons(p,l) -> if fst(fst(p)) = id then FCons(minusAmount(p,amount),loop(l))
                   else FCons(p,loop(l)) in
  makeMPoker(BWP!makePoker(loop(players),pack),getPot(x) + amount);

 let fold (x , id) =
  let players = getPlayers(x) in
  let pack = getPack(x) in
  let rec loop (players : player list) = match players with
  | FNil -> FNil
  | FCons(p,l) -> if fst(fst(p)) = id then loop(l)
                  else FCons(p,loop(l)) in
  makeMPoker(BWP!makePoker(loop(players),pack),getPot(x));

 let rec secondBetting (x , raiseNum , callRepeat, lastid, list) = ... ; // check the bet

 let bettingRound (x , betType , raiseNum , callRepeat , bettings) =
  if betType = BBasic then
   begin
    if getLength(getPlayers(x)) <= 1 then x
    else match bettings with
    | FNil -> x
    | FCons((id,betAmount),l) ->
     if ~(((inPlayers(id,getPlayers(x))))) then x
     else if ~(((betAmount > 0))) then x
     else secondBetting(bet(x,id,betAmount),raiseNum,callRepeat,id,l)
   end
 else x;

 let giveToWinner (x , id , sumPot) =
  let rec loop (players) = match players with
   | FNil -> FNil
   | FCons(p,l) -> if fst(fst(p)) = id then FCons(minusAmount(p,0 - sumPot),loop(l))
                   else FCons(p,loop(l)) in
  makeMPoker(BWP!makePoker(loop(getPlayers(x)),getPack(x)),getPot(x));

proof of bp_flop_minusCards =
foc proof {*
```

```
60  <1>1 assume x : Self, assume cardNum : int, assume flopT : flopType, assume sumNum : int,
61       prove flopT = FBasic -> cardNum * getLength(getPlayers(x)) = sumNum -> getLength(getPack(
             flop(flopT,x,cardNum))) = getLength(removeElements(getPack(x),sumNum))
62       <2>1 prove getPack(flop(flopT,x,cardNum)) = BWP!getPack(BWP!flop(flopT,get1st(x),cardNum))
63            by definition of flop, get1st, getPack
64       <2>e qed by step  <2>1 definition of getPlayers, getPack property BWP!bp_flop_minusCards
65  <1>e conclude; *}
66
67  proof of bmp_bet_upd_pot =
68  foc proof {*
69  <1>1 assume x1 : Self, assume id : int, assume amount : int, assume x2 : Self, assume player1 :
         player, assume player2 : player,
70       hypothesis H1 : x2 = bet(x1,id,amount),
71       hypothesis H2 : player1 = getPlayerId(x1,id),
72       hypothesis H3 : player2 = getPlayerId(x2,id),
73       prove getPot(x2) = getPot(x1) + amount && getAmount(player2) = getAmount(player1) - amount
74      <2>1 prove getPot(x2) = getPot(x1) + amount
75           assumed
76      <2>2 prove getAmount(player2) = getAmount(player1) - amount
77           assumed
78      <2>e conclude
79  <1>e conclude; *}
80
81  proof of bmp_bettinground_upd_pot = foc proof {* assumed; *}
82
83  proof of bp_getPack_succ =
84  foc proof {*
85  <1>1 assume players : player, assume desk : card list,
86       prove getPack(makePoker(players,desk)) = desk
87       <2>1 prove getPack(makePoker(players,desk)) = BWP!getPack(BWP!makePoker(players,desk))
88            by definition of getPack, makePoker, get1st
89       <2>e qed by step  <2>1 property BWP!bp_getPack_succ
90  <1>e conclude; *}
91
92  proof of bmp_getPack_succ =
93  foc proof {*
94  <1>1 assume p0 : BWP, assume pot : int,
95       prove getPack(makeMPoker(p0,pot)) = BWP!getPack(p0)
96       by definition of getPack, makeMPoker, get1st
97  <1>e conclude; *}
98  ;;
```

Listing A.25: BMPBWP in FFML

```
1   fmodule B36PBWP from BWP
2    representation = BWP;
3    let totalCardNum  = 36;
4
5    proof of bp_getPack_succ =
6    foc proof {*
7     <1>1 assume players : player list, assume desk : card list,
8          prove (getPack(makePoker(players,desk)) = desk)
9          <2>1 prove (getPack(makePoker(players,desk)) = BWP!getPack(BWP!makePoker(players,desk)))
10              by definition of getPack, makePoker
11         <2>e qed by step  <2>1 property BWP!bp_getPack_succ
12    <1>e conclude;*}
13
14   proof of bp_flop_minusCards =
15   foc proof {*
16    <1>1 assume x : Self, assume cardNum : int, assume flopT : flopType, assume sumNum : int,
17         prove (flopT = FBasic) -> ((cardNum * getLength(getPlayers(x))) = sumNum) -> (getLength(
               getPack(flop(flopT,x,cardNum))) = getLength(removeElements(getPack(x),sumNum)))
18         <2>1 prove (getPack(flop(flopT,x,cardNum)) = BWP!getPack(BWP!flop(flopT,x,cardNum)))
19              by definition of flop, getPack
20        <2>e qed by step  <2>1 definition of getPlayers, getPack property BWP!bp_flop_minusCards
21    <1>e conclude; *}
22  ;;
```

Listing A.26: B36PBWP in FFML

```
1   fmodule BWPMFP from MFP
2    signature makeWCards : BWPMFP -> BWPMFP;
3    signature getWCards : BWPMFP -> card list;
4    signature wCombination : card list -> card list -> bool * int;
5    signature makeWPoker : MFP -> card list -> BWPMFP;
6
7    contract makeWCards ::  property bwp_make_wcards : all x:BWPMFP, all x2:BWPMFP, x2 = makeWCards(
         x) -> isAKind(FCons(getElement(getPack(x),1),getWCards(x2)));
8
9    representation extends MFP with card list;
10   let makeWCards (x) = ...; // make wild cards
11
12   let getWCards (x) = snd(x);
```

```
13  let makeWPoker (poker , wCards) = (poker,wCards);
14  let wCombination (cards , wcards) =  ... ; // combination of cards and wild cards
15
16  let showdown (x) =
17   let wCards  = getWCards(x) in
18   let players  = getPlayers(x) in
19   let player0  = getPlayer(players) in
20   if (fst(wCombination(snd(findPlayer(players,player0,wCards)),wCards)) = false) then MFP!
         showdown(x)
21   else findPlayer(players,player0,wCards);
22
23 proof of bp_getPack_succ =
24 foc proof {*
25  <1>1 assume players : player list, assume desk : card list,
26       prove getPack(makePoker(players,desk)) = desk
27       <2>1 prove getPack(makePoker(players,desk)) = MFP!getPack(MFP!makePoker(players,desk))
28            assumed
29       <2>e qed by step  <2>1 property MFP!bp_getPack_succ
30  <1>e conclude; *}
31
32 proof of bp_flop_minusCards =
33 foc proof {*
34  <1>1 assume x : Self, assume cardNum : int, assume flopT : flopType, assume sumNum : int,
35       prove flopT <> FCom -> flopT = FBasic -> cardNum * getLength(getPlayers(x)) = sumNum ->
            getLength(getPack(flop(flopT,x,cardNum))) = getLength(removeElements(getPack(x),sumNum
            ))
36       <2>1 prove getPack(flop(flopT,x,cardNum)) = MFP!getPack(MFP!flop(flopT,get1st(x),cardNum))
37            by definition of flop, get1st, getPack
38       <2>e qed by step  <2>1 definition of getPlayers, getPack property MFP!
            mfp_flop_minusCards_R1
39  <1>e conclude; *}
40
41 proof of bwp_make_wcards = foc proof {* assumed; *}
42
43 proof of bp_showdown_notWinner =
44 foc proof {*
45  <1>1 assume x : Self, assume p : player, assume players : player list, assume player0 : player,
      assume wCards : card list,
46       hypothesis H1 : wCards = getWCards(x),
47       hypothesis H2 : players = getPlayers(x),
48       hypothesis H3 : player0 = getPlayer(players),
49       prove  ~ (isPlaying(p,x)) ->  ~ (showdown(x) = p)
50       <2>1 hypothesis H21 : (fst(wCombination(snd(findPlayer(players,player0,wCards)),wCards)) =
            false),
51            prove  ~ (isPlaying(p,x)) ->  ~ (showdown(x) = p)
52            <3>1 prove showdown(x) = MFP!showdown(get1st(x))
53                 by definition of showdown hypothesis H1, H2, H3, H21
54            <3>2 prove isPlaying(p,x) = MFP!isPlaying(p,get1st(x))
55                 by definition of isPlaying
56            <3>3 prove  ~ (MFP!isPlaying(p,get1st(x))) ->  ~ (MFP!showdown(get1st(x)) = p)
57                 by property MFP!bp_showdown_notWinner
58            <3>4 prove ( ~ (MFP!isPlaying(p,get1st(x))) ->  ~ (MFP!showdown(get1st(x)) = p)) -> (
                 ~ (isPlaying(p,x)) ->  ~ (showdown(x) = p))
59                 assumed
60            <3>e conclude
61      <2>2 hypothesis H22 :  ~ (fst(wCombination(snd(findPlayer(players,player0,wCards)),wCards))
            = false),
62            prove  ~ (isPlaying(p,x)) ->  ~ (showdown(x) = p)
63            assumed
64      <2>e conclude
65  <1>e conclude; *}
66 ;;
```

Listing A.27: BWPMFP in FFML

```
1  open "bp";;
2  open "bmp";;
3  species B36PBMP_spec1 (BP is BP_imp, BMP is BMP_imp (BP)) =
4   inherit BMP_spec1 (BP);
5  end;;
6
7  species B36PBMP_spec2 (BP is BP_imp, BMP is BMP_imp (BP)) =
8   inherit B36PBMP_spec1 (BP, BMP);
9
10  property bmp_bettinground_upd_pot : all x : Self, all bType : betType, all raiseNum : int, all
       callRepeat : int, all bettings : ( int* int ) list , all x2 : Self, (x2 = bettingRound(x,
       bType, raiseNum, callRepeat, bettings)) -> (getPot(x2) = (getPot(x) + sumBets(bettings)));
11
12  property bmp_bet_upd_pot : all x1 : Self, all id : int, all amount : int, all x2 : Self, all
       player1 : player, all player2 : player, (x2 = bet(x1, id, amount)) -> (player1 =
       getPlayerId(x1, id)) -> (player2 = getPlayerId(x2, id)) -> ((getPot(x2) = (getPot(x1) +
       amount)) && (getAmount(player2) = (getAmount(player1) - amount)));
13
14  property bmp_getPack_succ : all p0 : BP, all pot : int, (getPack(makeMPoker(p0, pot)) = BP!
       getPack(p0));
15
```

```
16  property bp_flop_minusCards : all x : Self, all cardNum : int, all flopT : flopType, all sumNum
        : int, (flopT = FBasic) -> ((cardNum * getLength(getPlayers(x))) = sumNum) -> (getLength(
        getPack(flop(flopT, x, cardNum))) = getLength(removeElements(getPack(x), sumNum)));
17
18  property bp_showdown_notWinner : all x : Self, all p : player, ~ isPlaying(p, x) -> ~ (
        showdown(x) = p);
19
20  property bp_getPack_succ : all players : player list, all pack : card list , (getPack(makePoker(
        players, pack)) = pack);
21
22  property bp_setPack_totalCardNum : all x : Self, all cards : card list, ((getLength(getPack(
        setPack(x, cards))) = totalCardNum) || (getLength(getPack(setPack(x, cards))) = getLength(
        getPack(x))));
23 end;;
24
25 species B36PBMP_imp (BP is BP_imp, BMP is BMP_imp (BP)) =
26  inherit B36PBMP_spec2 (BP, BMP);
27
28  representation = BMP;
29
30  let maxPlayers = BMP!maxPlayers;
31  let combination (listc) = BMP!combination(listc);
32  let setPack (x, cards) = BMP!setPack(x, cards);
33  let addPlayer (x, player) = BMP!addPlayer(x, player);
34  let getPlayerCards (player) = BMP!getPlayerCards(player);
35  let addCards (cards, pack, cardNum, count) = BMP!addCards(cards, pack, cardNum, count);
36  let playerGetsCards (players, pack, cardNum) = BMP!playerGetsCards(players, pack, cardNum);
37  let flop (flopType, x, cardNum) = BMP!flop(flopType, x, cardNum);
38  let showdown (x) = BMP!showdown(x);
39  let getPlayers (x) = BMP!getPlayers(x);
40  let getPack (x) = BMP!getPack(x);
41  let inPlayers (id, players) = BMP!inPlayers(id, players);
42  let isPlaying (p0, x) = BMP!isPlaying(p0, x);
43  let getPlayerId (x, id) = BMP!getPlayerId(x, id);
44  let makePoker (players, cards) = BMP!makePoker(players, cards);
45  let endPoker (x) = BMP!endPoker(x);
46  let makeMPoker (poker, pot) = BMP!makeMPoker(poker, pot);
47  let getPot (x) = BMP!getPot(x);
48  let bet (x, id, amount) = BMP!bet(x, id, amount);
49  let fold (x, id) = BMP!fold(x, id);
50  let bettingRound (x, betType, raiseNum, callRepeat, bettings) = BMP!bettingRound(x, betType,
        raiseNum, callRepeat, bettings);
51  let giveToWinner (x, id, sumPot) = BMP!giveToWinner(x, id, sumPot);
52  let totalCardNum = 36;
53
54 // The generated proof:
55 //proof of bmp_getPack_succ = by definition of getPack, makeMPoker property BMP!bmp_getPack_succ;
56 // is failed by Zenon's bug. It is fixed by the following:
57  proof of bmp_getPack_succ =
58   <1>1 assume p0 : BP, assume pot : int,
59        prove (getPack(makeMPoker(p0, pot)) = BP!getPack(p0))
60     <2>1 prove getPack (makeMPoker (p0, pot)) = BMP!getPack(BMP!makeMPoker(p0, pot))
61          by definition of getPack, makeMPoker
62     <2>2 prove BMP!getPack(BMP!makeMPoker(p0, pot)) = BP!getPack(p0)
63          by property BMP!bmp_getPack_succ
64     <2>e conclude
65   <1>e conclude;
66
67  proof of bp_showdown_notWinner = by definition of isPlaying, showdown property BMP!
        bp_showdown_notWinner;
68
69  proof of bp_getPack_succ =
70   <1>1 assume players : player list,    assume desk : card list,
71        prove (getPack(makePoker(players, desk)) = desk)
72        <2>1 prove (getPack(makePoker(players, desk)) = BMP!getPack(BMP!makePoker(players, desk)))
73             by definition of getPack, makePoker
74        <2>e qed by step  <2>1 property BMP!bp_getPack_succ
75  <1>e conclude;
76
77  proof of bp_flop_minusCards =
78   <1>1 assume x : Self, assume cardNum : int, assume flopT : flopType, assume sumNum : int,
79        prove (flopT = FBasic) -> ((cardNum * getLength(getPlayers(x))) = sumNum) -> (getLength(
             getPack(flop(flopT, x, cardNum))) = getLength(removeElements(getPack(x), sumNum)))
80        <2>1 prove (getPack(flop(flopT, x, cardNum)) = BMP!getPack(BMP!flop(flopT, x, cardNum)))
81             by definition of flop, getPack
82        <2>e qed by step  <2>1 definition of getPlayers, getPack property BMP!bp_flop_minusCards
83  <1>e conclude;
84  ...
85 end;;
86
87 collection B36PBMP_col =
88   implement B36PBMP_imp (BP_col, BMP_col);
89 end;;
```

Listing A.28: B36PBMP in FoCaLiZe

le cnam

Thi-Kim-Dung PHAM

Development of Correct-by-Construction Software using Product Lines

le cnam

Résumé :
Aujourd'hui, la diversité des logiciels pose des difficultés à de nombreuses entreprises et organisations. Alors que l'ingénierie des lignes de produits logiciels est considérée comme une solution possible et utilisée dans de nombreux domaines depuis des décennies, la problématique du développement de lignes de produits corrects par construction est toujours d'actualité. Cette thèse commence par une présentation de quelques techniques existantes appliquées pour développer et garantir la correction des lignes de produits logiciels. Nous proposons une solution basée sur la conception et la mise en oeuvre d'un langage FFML (Formal Feature Module Language) inspiré du langage FoCaLiZe et fournissant des mécanismes pour exprimer la réutilisation et la variabilité. Ce langage permet de spécifier, implanter une fonctionnalité et prouver sa correction en donnant des indications au prouveur automatique de théorèmes Zenon. Nous développons un compilateur de FFML en FoCaLiZe. Nous fournissons également un mécanisme de composition qui, appliqué à une configuration valide fournie par l'utilisateur, produit automatiquement un produit final correct-par-construction, ce qui signifie que le code produit est correct par rapport à ses spécifications, elles-aussi obtenues par composition des spécifications des caractéristiques impliquées dans la configuration de l'utilisateur. Enfin, nous évaluons notre méthodologie en construisant une ligne de produits logiciels pour le poker.

Mots-clefs:
Développement correct-par-construction, ligne de produits logiciels, variablilité, spécification formelle, preuve formelle, FoCaLiZe, Zenon

Abstract :
Nowadays diversity of software raises difficulties for many companies and organizations. While software product line engineering is considered as a solution and used in many domains for decades, research about the development of correct-by-construction software product lines is still up-to-date and necessary. We begin this thesis with an overview of how existing techniques were applied to develop and guarantee the correctness of software product lines. We propose a solution based on the design and implementation of a language FFML (Formal Feature Module Language) inspired by the FoCaLiZe language and providing mechanisms for expressing reuse and variability. This language allows to specify, implement a feature and prove correctness by giving hints to the automatic theorem prover Zenon. We develop a compiler for FFML to FoCaLiZe. We also provide a composition mechanism which applied to a valid user configuration automatically computes a final product correct-by-construction, meaning that the code of the product is correct with respect to its specifications. The specifications of the final product are obtained by composing the specifications of the features involved in the user configuration, the code is obtained by composing the code of the features and the proofs are also produced by composition. Finally, we evaluate our methodology by building a poker software product line.

Keywords :
Correct-by-Construction Development, Software Product Lines, variability, formal specification, Formal proof, FoCaLiZe, Zenon