N° d'ordre:

N° de série:

République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

UNIVERSITE ECHAHID HAMMA LAKHDAR EL OUED
FACULTE DES SCIENCES EXACTES
DEPARTEMENT D'INFORMATIQUE

Mémoire de fin d'étude

MASTER ACADEMIQUE

Domaine : Mathématique et InformatiqueFilière : Informatique

Spécialité : Systèmes Distribués et Intelligence Artificielle (SDIA)

Thème

Réalisation d'un outil de transformation automatique de diagramme de classes UML en FoCaLiZe

Présenté par

BOUSBIA LAICHE Abdelhamid TERCHA Abdallah

Soutenu devant le jury composé de

M.	Mohamed Amine Yagoub	Président	Univ.d'ElOued
M.	Abbas Messaoud	Rapporteur	Univ.d'ElOued
M.	Meftah M. Charaf-Eddine	Examinatrice	Univ.d'ElOued

Année universitaire 2020/2021

Remerciements

Nous remercions tout d'abord notre Dieu qui nous a donné la forceet la volonté pour élaborer ce travail.

Nous voudrions remercier notre encadreur **Mr ABBAS Messaoud** qui nous a aidé et nous a orienté tout au long de la préparation de ce mémoire. Il est toujours présent pour clarifier certains aspects de nos lacunes et de discuter de nos pensées. Ses critiques, commentaires et conseils ont certainement permis à cettemémoire d'être comme ce qu'elle est.

Nous remercions sincèrement ceux qui ont bien voulu prendre partà ce jury.

Nous tenons également à remercier toutes les personnes qui ont coopéré à notre formation, en particulier les enseignants du département informatique de l'Université d'El-Oued.

Bousbia Laiche Abdelhamid & Tercha Abdallah

Dédicaces

Je dédie ce travail

A l'âme de mon père

A Ma chère mère

A Ma femme

A Mes chers enfants

A Mes frères

Et

A Mon Encadreur Abbas Messaoud.

Et tous mes amis

ABDELHAMID

Dédicaces

Je dédie ce modeste travail

A Mes très chers parents

A Mes grands-parents

A Mes frères et sœurs

Et

A Mon Encadreur ***Abbas Messaoud.***

Et tous mes amis.

ABDALLAH

Abstract

The Unified Modeling Language (UML) is gradually establishing itself as the standard fact when it comes to modeling software systems using an object-oriented approach. But, despite its great richness, this language has semantics that remain informal and poorly defined which makes it difficult to use formal methods directly.

On the other hand, FoCaLiZe is a formal object-oriented development environment, allowing the realization of certified applications. This approach fits into the framework of model engineering (IDM, "Model-driven Engineering"), which aims at the production of software by automatic refinements of models, from abstract specifications to concrete implementations.

In this thesis, we envisage the realization of an Automatic Transformation Tool of UML / OCL structural aspect into FoCaLiZe. The proposed implementation supports directly the most of UML features such as inheritance, multiple inheritance, dependency, UML Template and Template binding etc.

We illustrate our implementation with concrete examples.

Keywords: UML, FoCaLiZe, Transformation, Modeling, Model Driven Engineering (MDE).

Résumé

L’Unified Modeling Language (UML) est progressivement devenu le standard de fait pour la modélisation de systèmes logiciels à l'aide de méthodes orientées objet. Cependant, malgré sa richesse, la sémantique de ce langage est encore informelle et mal définie, ce qui rend difficile l'utilisation directe de méthodes formelles.

D'autre part, FoCaLiZe est un environnement de développement formel orienté objet qui permet la mise en œuvre d'applications certifiées. Cette méthode appartient au cadre de l'Ingénierie des Modèles (IDM, " Ingénierie Dirigée par les Modèles "), qui vise de produire des logiciels par affinement automatiques de modèles, des spécifications abstraites aux implémentations concrètes.

Dans ce mémoire, nous envisageons la réalisation d’un outil de transformation automatique de diagramme de classes UML en FoCaLiZe . L'implémentation proposée supporte directement la plupart des fonctionnalités d’UML telles que l'héritage, l'héritage multiple, la dépendance, le Template UML et le Template binding etc.

Nous illustrons notre implémentation par des exemples concrets.

Mots-clés: UML, FoCaLiZe, Transformation, Modélisation, Ingénierie Dirigée par les Modèles (IDM).

Sommaire:

Listes de figure

Liste de Tableau

Introduction

Introduction

La modélisation est un processus nécessaire dans le développement de logiciels, car un modèle est conçu pour envisager les résultats du codage. Un modèle est une représentation abstraite d'un système destiné à en faciliter l'étude, la documentation et l'implémentation du système.

De plus, les systèmes développés sont devenus de plus en plus complexes et volumineux, ce qui rend leur compréhension et leur maîtrise très difficiles. La modélisation permet une meilleure compréhension du logiciel, apporte une grande rigueur et permet de comparer les conceptions avant le développement. Cette méthode est basée sur un langage de modélisation et permet de surmonter les limitations du langage d'implémentation.

Depuis la normalisation de l'Object Management Group (OMG) en 1997, le langage de modélisation unifié (UML) s'est progressivement imposé comme un standard pour la modélisation de systèmes orientés objet. UML est un ensemble de symboles graphiques conçus pour représenter, spécifier, construire et documenter systèmes logiciels. Ses deux objectifs principaux sont d'utiliser la technologie orientée objet pour modéliser les systèmes, de la conception à la maintenance, et de créer des langages abstraits que les humains peuvent comprendre et les machines peuvent interpréter (Benoît CHARROUX, 2009).

UML est convenable à tout le personnel responsable de la production, du déploiement et du suivi des logiciels (analystes, développeurs, chefs de projet, architectes, etc.), mais il peut également être utilisé pour communiquer avec les clients et les utilisateurs de logiciels. Il est convenable à tous les domaines d'application et à tous les supports. Il permet de construire plusieurs modèles du système, chaque modèle mettant en évidence différents aspects : fonctionnels, statiques, dynamiques et organisationnels.

L'OCL (Object Constraint Language) (OMG, 2020)est un langage formel de la famille OMG. Il permet de spécifier les éléments des diagrammes UML par des contraintes (exigences). En réalité, le langage naturel (non formel) utilisé au début avec UML a été remplacé par le langage OCL pour mettre des annotions sur les éléments des diagrammes.

Malheureusement, UML n'est qu'un langage semi-formel, et sa syntaxe abstraite est

définitivement précise (elle est basée sur le méta-modèle UML et se fait avec des contraintes structurelles exprimées en OCL). Cependant, sa sémantique est ambiguë et surtout informelle. Afin de donner une vraie sémantique à UML, de nombreux travaux de recherche sont actuellement en cours.

Le langage UML/OCL ne permet que la description du système et la satisfaction de contraintes. Ils n'ont aucun moyen de détecter les erreurs de modélisation ou de vérifier la satisfaction des contraintes OCL. Par conséquent, lorsqu'il s'agit d'applications critiques nécessitant un haut degré de sécurité (contrôle du trafic aérien, centrales nucléaires, etc.), l'utilisation (en combinaison avec UML/OCL) de méthodes formelles permettant l'analyse et la vérification de modèles est pertinente. Ces méthodes fournissent une notation mathématique qui permet une spécification très précise et rigoureuse des propriétés du système en cours de construction. Comparé à UML, le modèle est moins intuitif, mais il peut être vérifié à l'aide de techniques de preuve formelle. L'objectif de la méthode formelle est de guider le programmeur pour l'amener à donner, puis prouver les attributs nécessaires à la cohérence de son modèle. (ABBAS, 2018).

Dans le cadre de ce mémoire, nous considérons une formalisation de modèles UML/OCL supportant un certain nombre des fonctionnalités UML/OCL, en utilisant FoCaLiZe (Thérèse Hardin, 2016). Ce dernier est une méthode formelle, initiée par Thérèse Hardin et Renaud Rioboo au sein des laboratoires LIP6, C DRIC et INRIA à la fin des années 90.

FoCaLiZe, issue de l'assistant d'aide la preuve Coq (INRIA, 2016) est un environnement de développement complet, comprenant un langage de programmation (fonctionnel), un langage de spécification des exigences et un langage de preuve. L'environnement FoCaLiZe partage la plupart de l'architecture et des fonctions conceptuelles avec UML/OCL, telles que l'héritage (généralisation/spécialisation) et l'héritage multiple, la redéfinition des fonctions, la liaison tardive, les dépendances, etc., et prend en charge la plupart des exigences imposées par la cycle de vie du logiciel (Philippe Ayrault, 2009) (David Delahaye J.-F. É.-G., 2008) (Fechter, 2005).

Bien que David Delahaye et son équipe aient analysé la correspondance sémantique entre les éléments UML et les diagrammes de classes FoCaLiZe (David Delahaye J.-F. É.-G., 2008), la transformation d'UML/OCL vers FoCaLiZe n'a pas suffisamment encore traitée.

Pour aboutir cet objectif, nous avons choisi une approche directe (transformationnelle). D'une façon précise, nous voulons supporter les fonctionnalités UML/OCL suivantes :

- Héritage (multiple) avec redéfinition d'opérations et liaison tardive ;
- Dépendance (multiple) ;
- Classes paramétrées par d'autres classes ;
- Remplacements de paramètres formels de classes (paramétrées) par des paramètres effectifs afin de générer des modèles liées ("Template binding") ;
- Associations binaires ;
- Les contraintes OCL de type invariant.

Ainsi, nous allons étudier la sémantique la majorité des éléments du diagramme de classes UML, des contraintes OCL et des spécifications FoCaLiZe; puis nous étudions les règles de transformations d'UML/OCL vers FoCaLiZe réalisés par M.ABBAS (ABBAS, 2018).

La transformation proposée (d'UML/OCL vers FoCaLiZe) peut être utilisée comme un outil de validation des modèles UML/OCL car elle peut être utilisée pour générer du code exécutable dans le contexte de MDE. En termes de vérification, le passage d'UML/OCL à FoCaLiZe va nous permettre d'identifier les défauts de modélisation (dès la première étape de développement), d'établir des preuves de contraintes OCL, et de vérifier les contraintes des associations. La génération du code exécutable est effectuée par raffinements successifs de la spécification FoCaLiZe dérivée.

Un développeur FoCaLiZe complète la spécification générée et prouve toutes ses propriétés (en utilisant les outils de preuve de FoCaLiZe, Zenon et Coq) jusqu'à l'étape finale qui consiste à générer un fichier exécutable (OCaml) certifié (ABBAS, 2018).

Plan du mémoire

Voici un bref survol des chapitres de ce document :

Dans le chapitre 1, nous introduisons les diagrammes UML et les expressions OCL supportés par la démarche de transformation. En particulier, nous détaillons la syntaxe des sous-ensembles des diagrammes de classes, et des contraintes OCL utilisés.

Dans le chapitre 2, nous présentons les concepts de base de l'environnement FoCaLiZe et les différents mécanismes de preuves ainsi que le modèle de développement adopté par FoCaLiZe.

Dans le chapitre 3, nous présentons la démarche de transformation des diagrammes de classes UML en spécifications FoCaLiZe. Nous détaillons d'abord la transformation des membres d'une classe (attributs et opérations), puis nous décrivons la transformation des fonctionnalités UML réunissant plusieurs classes, comme l'héritage, la paramétriation, la dérivation de classes liées à partir d'une classe paramétrée ("Template-binding"), le mécanisme de dépendance et les associations binaires.

Le chapitre 4 sera consacré à la conception de l'architecture en basant sur l'approche de programmation orienté objets.

Le chapitre 5 présente la réalisation et l'implémentation de cette architecture de notre modèle de transformation d'UML/OCL en FoCaLiZe par l'exploitation de toutes les techniques et les règles de transformations développées par M.ABBAS.

CHAPITRE I:

Modèles UML/OCL

I.1. Introduction

Les méthodes objets sont indispensables dans le contexte du développement de systèmes logiciels complexes et peuvent suivre le développement continu des exigences technologiques et applicatives. Cependant, la programmation objet n'est pas aussi intuitive que la programmation fonctionnelle. En fait, il est plus naturel de décomposer les problèmes informatiques selon des fonctions que d'interagir avec des ensembles d'objets. Par conséquent, la méthode objet doit être modélisée avant la conception.

À mesure que l'échelle des applications objet augmente, il est nécessaire de trouver un moyen de décrire et de développer le système, en tenant compte des données et du traitement.

Au milieu des années 90, il y avait des dizaines de méthodes objet disponibles, mais aucune ne domine. L'unification et la normalisation des trois méthodes dominantes, à savoir Booch (Booch, 1991), du nom de son auteur, OOSE (Ivar Jacobson, 1994) (*Object Oriented Software Engineering*), d'Ivan Jacobson et OMT (James R Rumbaugh, 1990) (*Object Modeling Technique*), de James Rumbaugh, sont à l'origine de la création du langage UML (*Unified Modeling Language*).

UML comble une lacune importante des technologies objet. Il permet d'exprimer et d'élaborer des modèles objet, indépendamment de tout langage de programmation.

Le langage OCL est un langage du standard OMG (OMG, OCL Object Constraint Language 2.4, 2020) qui permet la description des contraintes (propriétés) sur les modèles UML. Une contrainte OCL est une spécification précise qui peut être attachée à n'importe quel élément UML. OCL permet principalement de définir des invariants de classes et des pré et post-conditions pour les opérations de classes.

Le langage UML (OMG, 2020) permet la modélisation et la conception d'un système en utilisant un ensemble de diagrammes graphiques. Certains diagrammes UML permettent de modéliser l'aspect statique d'un système et d'autres diagrammes servent à la modélisation de l'aspect dynamique (aspect comportemental).

Le but de ce chapitre est de donner une description de diagramme fondamental d'UML, mais nous nous intéresserons ici qu'aux diagrammes de classes.

I.2. Modèles UML/OCL

Le modèle UML/OCL est une description conceptuelle du système réel utilisant des symboles graphiques UML et OCL. UML permet de décrire le modèle à travers différentes vues, chaque vue étant représentée par un ou plusieurs diagrammes et contraintes OCL. Les diagrammes UML sont représentés par des diagrammes de connexion, où les sommets sont des arcs d'éléments et de relations.

La version actuelle d'UML (UML 2.5) utilise jusqu' 23 types de diagrammes différents (OMG, 2020) pour décrire les aspects structurels (décrire les composants du système et leurs relations, indépendamment du temps) et le comportement des aspects (afficher le comportement du système, l'interaction d'objets et leur évolution dans le temps). Ces types de diagrammes sont divisés en deux groupes principaux (Roques, 2008) **:**

Diagrammes de structure

- Diagramme de classes : Il décrit la structure du système à travers un ensemble de classes et de relations (héritage, association, dépendance, etc.). C'est le diagramme le plus courant dans la modélisation UML. Ce n'est pas seulement l'axe de base de la modélisation objet, mais aussi le symbole le plus riche de tous les diagrammes UML
- Diagramme d'objets : Il représente une vue statique des instances d'éléments qui apparaissent dans le diagramme de classes. Il se compose d'un ensemble d'objets et de leurs relations à un moment donné.
- Diagramme de cas d'utilisation : Il présente le comportement du système face aux utilisateurs sous la forme d'un ensemble de scénarios et de participants. Le diagramme comprend des réponses aux questions sur qui utilise le système et dans quelles circonstances.
- **Diagramme de composants** : Un composant correspond généralement une ou plusieurs classes ou interfaces. Le diagramme de composants consiste regrouper sous forme de paquetage les différents composants d'un système. Il s'intéresse généralement l'implémentation statique et logique du système.
- Diagramme de déploiements : Il montre la disposition physique des di rentes ressources d'une architecture en phase d'exécution. En particulier, il décrit les ressources de calcul, leurs configurations et le lien entre l'exécution et les ressources de calcul.

Diagrammes de comportement

- Diagramme de collaborations : Il représente les interactions entre les dif f é rents objets qui constituent le système et les messages qu'ils changent entre eux.
- Diagramme de séquences : Il décrit l'échange de messages entre les objets intervenant en tenantcompte du séquencement chronologique de messages.
- Diagramme d'états-transitions : C'est un automate d'états fini, composé d'états, de transitionset d'événements d'activités. Ce diagramme décrit les états et le comportement des instances d'une classe en réponse des évènements extérieurs.
- Diagrammes d'activités : Il décrit l'enchainement des activités (en séquence, en parallèle ou sélectif) d'un systè me en fonction de ses é tats et ses é vènements. Le diagramme d'activité est également utilisé pour décrire un flux de travail ("workflow").

Dans le cadre de notre étude, nous utilisons les diagrammes de classes pour représenter la vue statique (structurelle) d'un système. Les contraintes OCL considérées sont les invariants de classes et les pré /post- conditions pour les opérations de classes.

I.3. Diagramme de classe

Nous montrons les composants de diagramme de classes à travers leurs méta-modèles UML tels qu'ils sont spécifiés dans UML 2.5, et les expliquons avec des exemples. Un diagramme de classes est une instance de classifieur (figure I-1), donne un aperçu d'un système en montrant ses classes et les relations entre elles telles que les héritages, les associations, les dépendances et autres. Le diagramme de classe affiche ce qui interagit mais pas ce qui se passent quand ils interagissent.

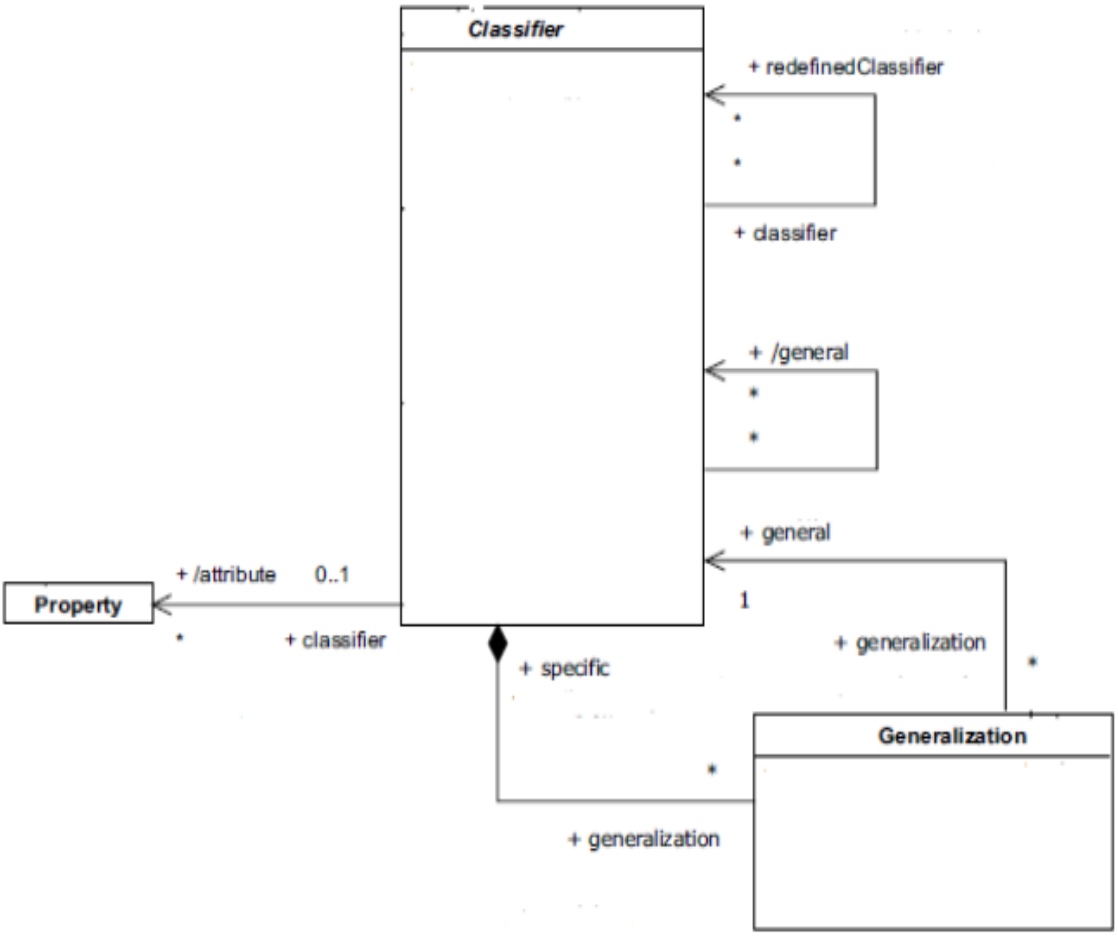


Figure I.1 : méta-model de diagramme de classe

I.3.1. Classes

Les classes sont la pierre angulaire la plus importante de tout système orienté objet. En UML, une classe est un classificateur (voir Figure I-2) qui décrit un ensemble d'objets, qui partagent les mêmes spécifications en termes de fonctionnalités, de contraintes et de sémantique. Une classe peut avoir une visibilité (nous nous limitons à n'utiliser que la visibilité de classe publique) et peut inclure une contrainte, un attribut et une opération.

Un **stéréotype** (**stéréotype** de classe), il représente une sous-classe d'un élément de

modélisation existant avec la même forme (attributs et relations) mais avec une intention différente. Généralement, un stéréotype représente une distinction d'usage et représente l'un des mécanismes d'extensibilité intégrés d'UML. Par exemple, modéliser un type d'énumération à l'aide de l'énumération de **stéréotype**.

Une classe peut utiliser la définition d'une autre classe pour définir ses propres attributs et opérations. Cette relation d'utilisation entre les classes est appelée **dépendance** (dépendante).

Une classe peut être définie par héritage d'autres classes. L'héritage est généralement défini comme un mécanisme par lequel des classes plus spécifiques (sous-classes) incorporent la structure et le comportement de classes plus générales (superclasses).

Une classe (modèle) peut être spécifié avec une liste de paramètres de modèle formel, une classe peut également contenir modèle des liaisons (Lié) à d'autres modèles de classes. Ces liaisons de modèle spécifient comment la classe est construite en remplaçant les paramètres de modèle formels des classes de modèle cible par des paramètres réels.

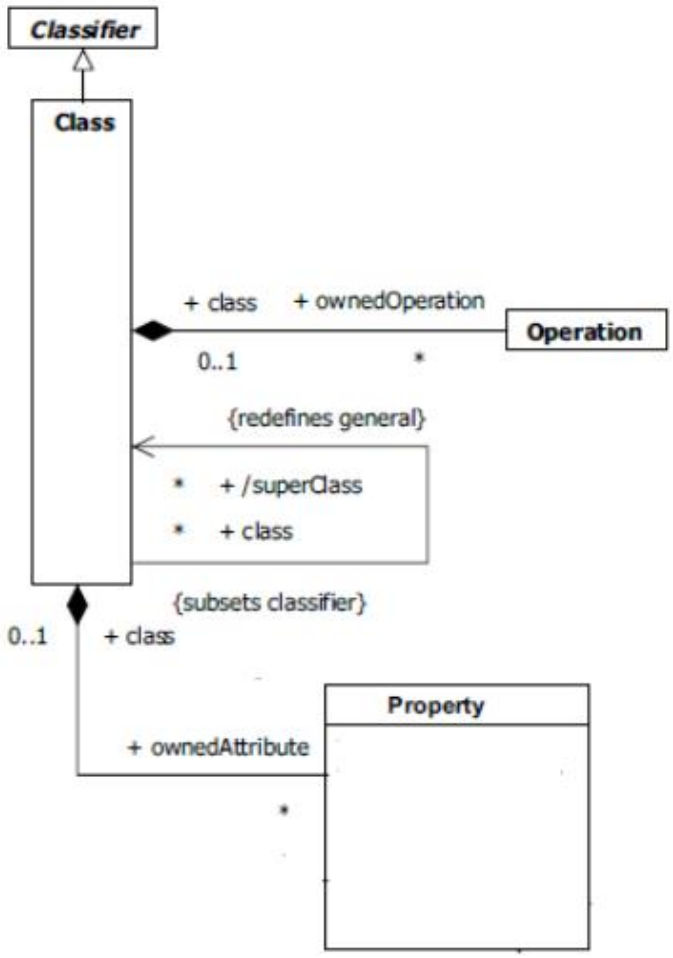


Figure I.2 : Méta-model de classe

I.3.2. Attributs

Un attribut est identifié par un nom et spécifié par un type. Le type d'un attribut peut être primitif (*Integer*, *Boolean*, *String*, *Real* et *UnlimitedNatural*) ou un type classe du modèle.

La multiplicité d'un attribut, si elle vaut 1 ([1..1]), indique que l'attribut est composé d'une valeur unique. Dans le cas contraire, elle indique que l'attribut est composé d'une collection d'éléments.

La visibilité d'un attribut définit les droits d'accès à l'attribut selon que l'on y accède par une opération de la classe elle-même, d'une classe héritière, ou bien d'une classe quelconque.

On distingue quatre niveaux de visibilités :

- *public*(+) : Les opérations de toutes les classes peuvent accéder aux données. Les données ne sont pas protégées.
- *private* (-) : L'accès aux données est limité aux opérations de la classe.
- *protected* (#) : L'accès aux données est uniquement réservé aux opérations de la classe et aux opérations héritées.

package(~) : Seule une opération déclarée dans une classe de même paquetage (regroupement de classes) peut accéder à l'attribut.

Dans le cadre de notre étude, nous utilisons les visibilités *public* et *private*.

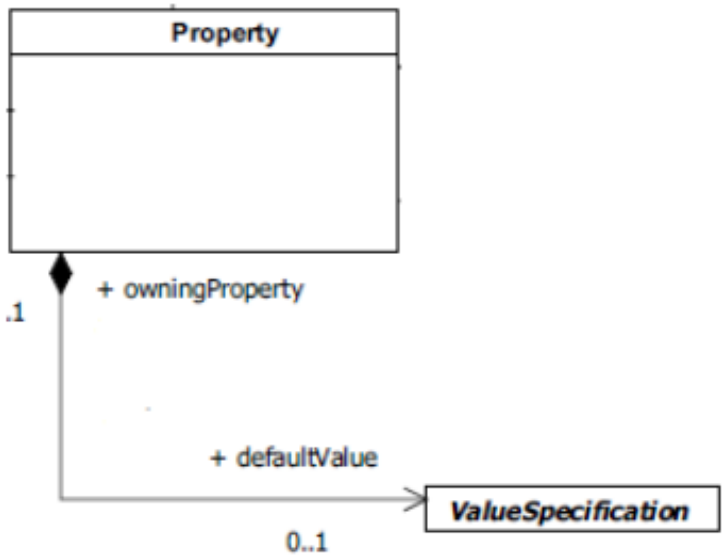


Figure I.3 : Méta-model d'attribut

I.3.2.1. Opérations

L'opération représente un élément de comportement des objets, défini de manière globale dans la classe.

Une opération est une fonctionnalité assurée par une classe. La description des opérations peut préciser les paramètres d'entrée et de sortie ainsi que les actions élémentaires à exécuter.

La liste des opérations de la classe permet de manipuler les attributs de la classe et de changer les états de ses instances. Une opération est introduite par sa signature.

Chaque opération est identifiée par son nom et possède une visibilité, similaire aux visibilités des attributs), un stéréotype, une liste de paramètres formels et un type de retour.

La direction d'un paramètre indique son orientation, s'il s'agit d'un paramètre d'entrée on utilise la direction in (la direction par défaut) et dans le cas d'un paramètre de sortie on utilise la direction out.

Exemple : La classe *Product* représente les informations relatives à un produit. Chaque produit est défini par ses attribues (*ProductId, ProductPrice, ProductType*) et trois opérations *(AddProduct(), ModifyProdect() et SelectProduct)*.

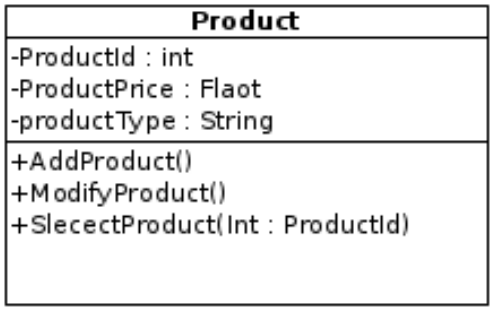
Notation UML	Description textuelle
	**public class** Product = **attribute** ProductId : **int** **attribute** ProductPrice : **Float** **attribute** ProductType : **String** **operation** AddProduct() **operation** ModifyProduct() **operation** SelectProduct(int:ProductId) **end**

Tableau I. 1: Classe Product

I.3.2.2. Classe paramétrée ("Template")

Une classe paramétrée est une classe spécifiée par une liste de paramètres formels. Elle est utilisée pour modéliser une classe générique qui pourrait être spécialisée en substituant ses paramètres formels par des paramètres effectifs.

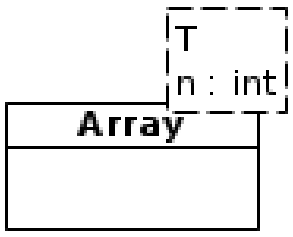
Notation UML	Description textuelle
	**public class Array(T : Class, n : int) =** **end**

Tableau I. 2: Méta-model de Template

I.3.2.3. Énumérations et Intervalles

Comme nous l'avons indiqué plus haut, les stéréotypes permettent d'étendre le vocabulaire d'UML pour introduire des nouvelles structures. Les stéréotypes les plus utilisés sont :

«*enumeration*» et «*intervalType*».

Le stéréotype : «*enumeration*» permet la définition d'un type en énumérant ses différentes valeurs

Exemple. Voir la spécification de l'énumération *Alignement* (voir tableau I.3) qui modélise les les positions d'alignement.

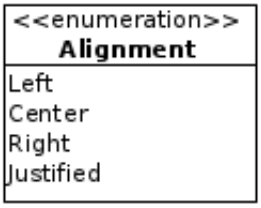
Notation UML	Description textuelle
	**public class** «enumeration» Alignment = Left Center Right Justified **end**

Tableau I. 3: Énumérations

Le stéréotype «*intervalType*» permet la définition d'un type intervalle en fournissant ses limites inférieure et supérieure

Exemple. Voir la spécification de l'intervalle *Floor* (voir tableau I.4) qui modélise les étages d'un Bâtiment.

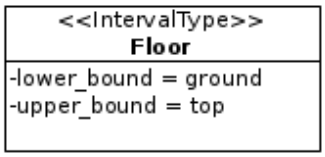
Notation UML	Description textuelle
	**public class** «IntervalType» Floor = lower_bound = ground upper_bound = top **end**

Tableau I. 4: Le type d'intervalle Floor

I.3.3. Relations de classes

Les classes UML (qui décrivent des objets du monde réel) peuvent être reliées par différentes sortes de relations. Dans notre sous-ensemble d'UML, nous supportons les relations suivantes :

- Héritage (simple et multiple), avec redéfinitions des opérations et liaison tardive ;
- Dépendance ;
- Génération de modèles liés aux classes paramétrées ("*Template binding*") ;
- Associations binaires

I.3.3.1. Héritage

L'héritage est le mécanisme qui permet à une nouvelle sous-classe d'acquérir tous les attributs, les opérations et les propriétés de ses super-classes. De plus, la sous-classe peut définir ses propres attributs et opérations, comme elle peut redéfinir des attributs et des opérations déjà définies au niveau de super-classes.

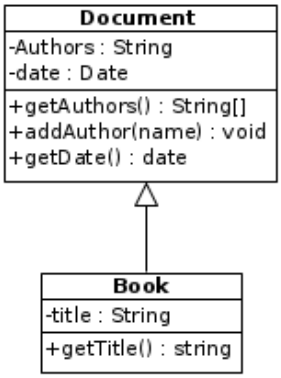
Notation UML	Description textuelle
	**public class** Book **inherits** Document = **attribute** title : **String** ; **operation** getTitle**() : String ;** **end**

Tableau I. 5: Héritage simple

Notons que l'héritage multiple (voir tableau I.6) est considéré comme une extension du modèle d'héritage simple, où l'on autorise une classe à posséder plusieurs super-classes.

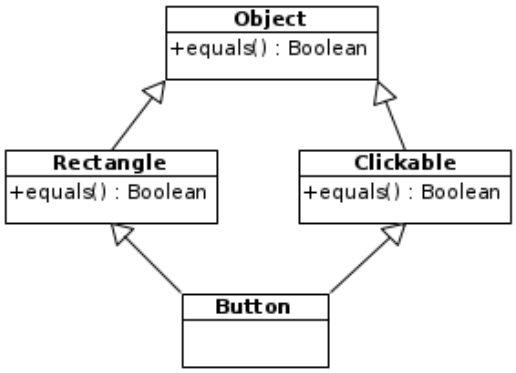
Notation UML	Description textuelle
	**public class** Clickable **inherits** Object = **operation** equals**() : Boolean ;** **end** **public class** Rectangle **inherits** Object = **operation** equals**() : Boolean ;** **end** **public class** Button **inherits** Clickable , Rectangle = **operation** equals**() : Boolean ;** **end**

Tableau I.6 : Héritage multiple

I.3.3.2. Dépendance

Une relation de dépendance spécifie qu'une classe cliente a besoin des classes fournisseurs pour définir ses propres attributs et opérations. Ceci implique que la définition de la classe cliente est sémantiquement et structurellement dépendante de la définition de classes fournisseurs :

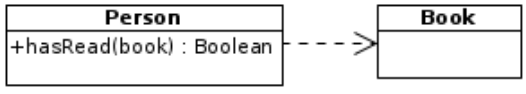
Notation UML	Description textuelle
	**public class** Person **depend** Book = **operation** hasRead**(**book**) : Boolean ;** **end**

Tableau I. 7: Exemple d'une relation de dépendance

I.3.3.3. Génération des classes liées ("Template binding")

Une classe liée ("bound class") peut être dérivée d'une classe paramétrée en substituant ses paramètres formels par des paramètres effectifs. Ces substitutions sont spécifiées dans une relation particulière qui relie les paramètres effectifs de la classe liée aux paramètres formels de la classe paramétrée :

Exemple. Une classe paramétrée *Array* peut servir comme classe générique pour définir toute Array avec un nombre fini d'éléments. Par exemple, la classe Année (voir tableau I.8) est un modèle lié dérivé de la classe paramétrée Array, en substituant ses paramètres formels par des paramètres effectifs (*T -> Customer, n -> 24*).

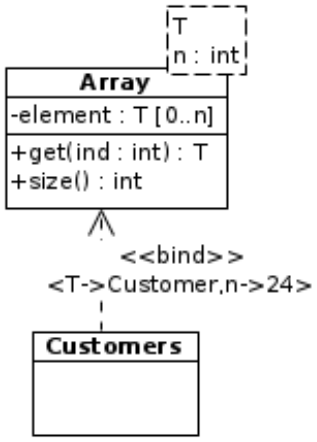
Notation UML	Description textuelle
	**public class Array(T : Class, n : int) =** **attribute** element : **String** ; **operation** get**(int) : T** ; **end** **public class** Person **depend** Book = **end**

Tableau I. 8: Exemple Génération des classes liées

I.3.3.4. Associations

L'association est la relation la plus courante et la plus riche du point de vue sémantique. Une association est une relation statique n-aire (le plus souvent : elle est binaire) : c'est-à-dire qu'elle relie plusieurs classes entre elles.

L'association existe entre les classes et non entre les instances : elle est introduite pour montrer une structure et non pour montrer des échanges de données.

Une association n-aire possède n rôles qui sont les points terminaux de l'association ou terminaisons.

Chaque classe qui participe à l'association joue un rôle. Les rôles sont définis par 2 propriétés :

- la multiplicité : elle définit le nombre d'instances de l'association pour une instance de la classe. La multiplicité est définie par un nombre entier ou un intervalle de valeurs.

- la navigabilité: La navigabilité n'a rien à voir avec le sens de lecture de l'association. Une navigabilité placée sur une terminaison cible indique si ce rôle est accessible à partir de la source. Par défaut les associations sont navigables dans les 2 sens. Dans certains cas, une seule direction de navigation est utile : l'extrémité d'association vers laquelle la navigation est possible porte alors une flèche

.

Exemple. Le processus d'écriture des livres peut être modélisé par une association binaire *Wrote* (voir tableau I.9) entre les classes *Professor* et Book.

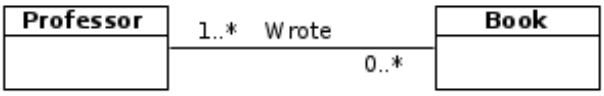
Notation UML	Description textuelle
	**association** *Wrote* **Both** *Professor* **[1..Infini]** *Book* **[0.. Infini]** **End**

Tableau I. 9: Exemple Associations

I.4. Contraintes OCL

Les contraintes permettent de rendre compte des détails que n'expriment pas les éléments du langage UML afin de restreindre la portée du modèle. En UML, les contraintes sont vues comme un mécanisme d'extension. Elles sont exprimées par une expression mathématique, un langage de programmation, le langage naturel, etc. Cependant, un langage d'expression des contraintes est maintenant associé à UML et permet d'exprimer la plupart des contraintes : il s'agit du langage OCL. OCL permet d'exprimer principalement deux types de contraintes : les invariants, les pré/post conditions.

I.4.1. Invariant

Un invariant est une assertion à propos d'une classe qui doit être vérifiée par toutes les instances de la classe, à tout moment. Étant donné une classe, un invariant attaché à la classe est défini comme suit :

$$context\ c\_n\ inv : E_{inv}$$

Où E_{inv} est une formule OCL.

I.4.2. Pré/Post Conditions

La pré-condition d'une opération de classe décrit une contrainte qui doit être vraie avant l'exécution de l'opération. Autrement dit, elle permet de préciser une condition sous laquelle l'appel d'une opération aboutira à un comportement correct.

La post-condition d'une opération de classe décrit une contrainte qui doit être satisfaite après l'exécution de l'opération. En d'autres termes, elle énonce comment devra être le système après

I.5. Conclusion

Dans ce chapitre, nous avons présenté le méta modèle des ensembles UML/OCL supportés par notre étude. Il permet la modélisation et la spécification des aspects statiques et dynamique de systèmes, en utilisant des diagrammes de classes et des contraintes OCL.

Il est à noter que sont pris en charge les fonctionnalités d'UML comme l'héritage multiple, la liaison tardive, la redéfinition des méthodes et la substitution des paramètres formels d'une classes paramétrée ("Template") par des paramètres effectifs ("Template binding").

Ces fonctionnalités UML/OCL seront prises en compte dans notre démarche de formalisation de modèles UML/OCL, alors qu'elles sont rarement prises en compte dans des études similaires.

CHAPITRE II:
L'environnement FoCaLiZe

II.1. Introduction

Actuellement, les grands dispositifs industriels sont quasiment impossibles à contrôler sans utiliser des outils informatiques, souvent développés avec des langages de programmation comme C, C++, Java, C#, etc. Cependant, lorsqu'il s'agit de systèmes de haute criticité (avions, centrales nucléaires, trains à grande vitesse, . . .) les erreurs de programmation peuvent causer des dégâts énormes et des accidents mortels. Ceci a conduit au développement d'environnements de développements intégrés (IDE) permettant d'exprimer formellement les spécifications, de décrire l'architecture, de produire du code et de vérifier que les exigences de spécification sont bien satisfaites par ce code. Ces outils de développement, dits méthodes formelles, sont basés sur les mathématiques et la logique (ABBAS, 2018).

L'environnement FoCaLiZe (anciennement FoCal) (Thérèse Hardin, 2016) est un IDE qui permet le développement de programmes certifiés pas à pas, de la spécification à l'implantation. Cet environnement propose un langage également nommé FoCaLiZe et des outils d'analyse de code, de preuve (Zenon) (Doligez, 2015) et des compilateurs vers Ocaml, Coq et Dedukti.

Le langage FoCaLiZe permet d'écrire des propriétés en logique du premier ordre, d'écrire des signatures et des fonctions dans un style fonctionnel, et enfin d'écrire des preuves dans un style déclaratif.

Ce langage offre également des mécanismes inspirés par la programmation orientée objets (David Delahaye C. D.-N., 2010), tels que l'héritage, la liaison retardée et la redéfinition afin de faciliter la modularisation et la réutilisation. FoCaLiZe vous permet d'hériter des spécifications, des codes et des preuves. Dans le cas de la redéfinition, le calcul peut déterminer quelles preuves sont encore valides et quelles preuves sont invalides en raison de la redéfinition, elles doivent donc être refaites. Le mécanisme de paramétrage complète la collection.

Dans ce chapitre, nous allons présenter en détail les différents concepts de l'environnement FoCaLiZe en commençant par la spécification, puis en décrivant les aspects d'implémentation et de preuve.

II.2. Spécification (espèce)

La brique de base du langage FocCaLiZe est le concept d'« espèce », qui a beaucoup en commun avec le concept de classes dans le paradigme objet. Une espèce peut être considérée comme une liste de trois méthodes (Tollitte, 2013) :

- Type support, appelé "représentation", qui est le type d'entité manipulée par la

fonction de l'espèce ; la représentation peut être abstraite ou concrète ;

- les fonctions, qui dénotent les opérations autorisés sur les entités de la représentation ; les fonctions peuvent être soit des définitions (lorsque un corps est fourni), soit des déclarations (quand seul le type est donné) ;
- les propriétés, qui doivent être vérifiées par toute implantation de l'espèce ; les propriétés peut être un simple énoncé ou un théorème (lorsque la preuve est également donnée).

La syntaxe d'une espèce est la suivante :

```
species <name> =
representation [= <type>] ;          (* représentation*)
signature <name> : <type>;            (*déclaration*)
let <name> = <body>;                  (*définition*)
property <name> : <prop>;             (*propriétés*)
theorem <name> : <prop> ;             (*théorème*)
proof = <proof>;
end ; ;
```

où <*name*> est un nom, <*type*> une expression de type, <*body*> un corps de fonction (dans une syntaxe proche du noyau purement fonctionnel de OCaml), <*prop*> une proposition logique (du premier ordre) et <*proof*> une preuve (exprimée au moyen d'un langage de preuve déclaratif ou d'un script Coq). Dans le langage des types, le mot-clé "*Self* " fait référence au type de la représentation et peut être utilisée partout dans une espèce sauf dans la représentation elle-même.

II.2.1. Représentation

La représentation d'une espèce définit la structure des données manipulées par les fonctions et les propriétés de l'espèce. C'est un type, défini par une expression de type en FoCaLiZe (AYRAULT, 2011): variables de types, types primitifs, types inductifs, enregistrements, etc.

Exemple: Nous définissons l'espèce *Point* qui modélise les points du plan. La représentation de cette espèce est exprimée par un couple de réels. Le premier représente l'abscisse et le deuxième désigne l'ordonnée d'un point :

```
species Point  =
representation = float * float
...
end ; ;
```

La représentation d'une espèce peut demeurer abstraite (la représentation n'est pas donnée) le long de quelques étapes de la spécification, mais elle doit être explicitée par une expression de type avant l'implémentation finale de l'espèce. Une fois la représentation d'une espèce donnée par une expression de type, elle ne peut plus être redéfinie. Ceci exprime le fait que les fonctions et les propriétés d'une espèce opèrent sur un type de données unique.

Entité d'espèce : est un élément de type représentation de l'espèce, créée par une fonction (constructeur) de l'espèce. Par exemple, la paire (0.5, 2.0) est une entité de l'espèce Point.

II.2.2. Fonctions

Les fonctions d'une espèce permettent de manipuler les entités d'une espèce et de décrire leurs comportements. Dans l'environnement FoCaLiZe existe deux catégories de fonctions :

1. Une fonction déclarée (**signature**) sert à spécifier une fonction sans rentrer dans les détails de son implémentation.
2. Une fonction définie (**let**) est une fonction complètement définie via son corps calculatoire.

II.2.2.1. Signature

Une signature est une fonction énoncée uniquement par son type fonctionnel. Elle permet de décrire la vue abstraite d'une fonction, qui sera concrétisée par les héritières de l'espèce (Messaoud Abbas, 2014)**.**

Nous complétons notre espèce **Point** par la signature de la fonction constructrice **deplacer_point** (pour déplacer un point d'une position vers une autre) et la fonction binaire **egal** (pour décider de l'égalité de deux points) comme suit :

```
species Point =
representation = float * float;
signature deplacer_point : Self -> float-> float -> Self;
signature egal : Self -> Self -> bool;
...
end;;
```

Le mot clé **Self** (utilisé dans les types des fonctions **deplacer_point** et **egal**) désigne une entité de l'espèce en cours de spécification, même si la représentation d'une espèce n'est pas définie (abstraite).

Le type **Self** -> **float**-> **float** -> **Self** de la fonction **deplacer_point** spécifie une fonction paramétrée par une entité de l'espèce Point (le premier **Self**) et par deux réels, et retournant une nouvelle entité de l'espèce (le dernier **Self**).

Une fois une signature énoncée, elle peut être utilisée pour spécifier d'autres fonctions et propriétés.

C'est le type fonctionnel de la signature qui permet à FoCaLiZe de contrôler qu'elle est correctement utilisée.

II.2.2.2. Fonction définie

Une fonction définie est une fonction énoncée parallèlement avec son corps calculatoire (Khadija, 2015). Le corps calculatoire d'une fonction est exprimé par des instructions fonctionnelles comme : la liaison let (**let**...**in**), le filtrage (**match**...**with**), la conditionnelle (**if**... **then**...**else**), etc.

Une fonction définie est soit nouvellement énoncée (pour la première fois), ou bien la définition d'une signature énoncée au niveau des ascendants de l'espèce ou même la redéfinition d'une fonction déjà définie.

Continuons avec notre espèce **Point**, les fonctions **get_x** (récupérer l'abscisse d'un point) et **get_y** (récupérer l'ordonnée d'un point) sont définies en utilisant les fonctions **fst** (retournant le premier composant d'une paire) et **snd** (retournant le deuxième composant d'une paire) de la bibliothèque basics de FoCaLiZe. La fonction **different** (décidant de l'inégalité de deux points) est définie à partir de la signature **egal** (donnée ci-dessus) :

```
species Point =
...
let get_x(p:Self) = fst(p);
let get_y(p:Self) = snd(p);
let different(a, b) = ~~(egal(a, b));
...
end;;
```

Le symbole ~~ désigne la négation logique (booléenne) dans FoCaLiZe, tandis que le symbole ~ désigne la négation dans le contexte d'une propriété.

II.2.3. Propriétés

Les propriétés sont des contraintes à satisfaire par les entités de l'espèce. FoCaLiZe permet deux niveaux de spécification de propriétés. Une propriété est soit uniquement déclarée (**property**), sans se préoccuper de sa preuve (niveau abstrait), soit définie (**theorem**) avec sa preuve (niveau concret)**.**

II.2.3.1. Propriétés déclarées

Une propriété déclarée est une formule de la logique du premier ordre énoncée sans fournir sa preuve. Elle permet d'exprimer une exigence de sûreté sur les entités de l'espèce dès les premières étapes de la spécification, même avant la définition de la représentation et des fonctions de l'espèce. La preuve d'une telle propriété devra être donnée au niveau des héritiers de l'espèce**.**

Pour définir une structure respectant une relation d'équivalence sur l'opération **egal** de l'espèce Point, nous n'introduisons les trois propriétés suivantes:

```
species Point =
...
property egal_reflexive : all p:Self, egal(p, p);
property egal_symetrique : all p1 p2: Self, egal(p1, p2) -> egal(p2, p1) ;
property egal_transitive : all p1 p2 p3: Self, egal(p1, p2) -> egal(p2, p3)
                                                              - > egal(p1, p3);
...
end;;
```

Le mot clé **all** dénote le quantificateur universel **(∀)** et le symbole **->** désigne le connecteur d'implication.

II.2.3.2. Théorèmes

Un théorème est une propriété énoncée parallèlement avec sa preuve. Un théorème peut être nouvellement énoncé, ou produit la preuve d'une propriété énoncée au niveau des ascendants de l'espèce.

Un théorème peut faire référence à n'importe quelle fonction ou propriété reconnue dans le contexte de l'espèce courante, comme il peut être utilisé pour spécifier d'autres propriétés de la même espèce ou de ses héritières.

En utilisant le langage FPL (FoCaLiZe Proof Language), la preuve de théorème *egal_est_non_different ((a = b)) => ¬(a ≠ b))* est acceptée par **assumed**, comme suit :

```
species Point =
...
theorem egal_est_non_different : all p1 p2:Self, egal(p1, p2) -> ~(different(p1, p2))
proof = assumed;
...
end;;
```

II.2.4. Héritage et liaison tardive

Nous pouvons créer une nouvelle espèce par héritage d'une espèce déjà définie, et rendre cette nouvelle espèce plus « complexe » en ajoutant de nouvelles opérations et propriétés, ou nous pouvons la rendre plus concrète en fournissant des définitions aux signatures et des preuves aux propriétés, sans ajouter de nouvelles fonctionnalités.

Une espèce peut hériter des déclarations et définitions d'une ou plusieurs espèces déjà définies et est libre de définir ou de redéfinir une méthode héritée tant que cette (ré) définition ne change pas le type de la méthode.

Lors de la construction par héritage multiple, certaines signatures peuvent être remplacées par des fonctions et des propriétés par des théorèmes. Il est également possible d'associer une définition de fonction à une signature ou une preuve à une propriété. Dans le même ordre, il est possible de redéfinir une méthode même si elle est déjà utilisée par une méthode existante. Toutes ces caractéristiques sont pertinentes pour un mécanisme connu sous le nom de liaison tardive.

Exemple de l'héritage en FoCaLiZe

```
type couleur = | Rouge | Vert | Bleu ;
species PointColore = inherit Point;
 representation = (float * float) * couleur;
let get_color(p:Self):couleur = snd(p);
let creerPointColore(x:float, y:float, c:couleur):Self = ((x, y),
c);
let get_x(p) = fst(fst(p));
let get_y(p) = snd(fst(p));
let deplacer(p, dx, dy) = ((get_x(p) +f dx, get_y(p) +f dy ),
get_color(p) );
let affiche_point (p:Self):String =
let affiche_couleur (c:couleur) = match c with
| Rouge -> "Rouge"
| Vert -> "Vert"
| Bleu -> "Bleu"
in ( " X = " ^ string_of_float(get_x(p)) ^ " Y = " ^
string_of_float(get_y(p)) ^
" COULEUR = " ^ affiche_couleur(get_color(p)));*)
end;;
```

Pour manipuler des points colorés (points graphiques), une nouvelle espèce **PointColore** est créée en héritant de l'espèce **Point**. L'espèce **PointColore** a besoin de manipuler la couleur d'un point en plus de ses cordonnées. Par conséquent, elle hérite de toutes les méthodes de l'espèce **Point** et apporte les enrichissements suivants : Elle définit la représentation par l'expression **(float * float)**

*** couleur** (le type couleur modélise la couleur d'un point), fournit des corps calculatoires aux signatures **get_x**, **get_y** et **deplacer** et introduit la fonction **get_color** pour récupérer la couleur d'un point. Elle définit aussi la fonction **affiche_point**, pour permettre l'affichage de la couleur dupoint en plus de ses cordonnées.

II.2.5. Espèce complète

Une espèce est dite complète si toutes ses déclarations ont reçu une définition et toutes ses propriétés des preuves**,** Cette notion implique que:

- La représentation a été associée à une définition de type.
- Chaque déclaration est associée à une définition.
- Une preuve est donnée pour chaque propriété.

II.3. Abstraction (collections)

Quand une espèce est complète, elle peut être utilisée pour créer et manipuler ses entités. Cependant, l'utilisation d'une espèce complète doit passer d'abord par la création d'une **collection**

II.3.1. Collection.

Les représentations d'espèces complètes sont encapsulées à travers des interfaces d'espèces. L'interface d'une espèce complète est la liste des déclarations de ses méthodes d'espèce sous-jacentes (signatures de fonctions et déclarations logiques). Cela correspond au point de vue de l'utilisateur final, qui veut savoir quelles fonctions il peut utiliser et quelles propriétés ont ces fonctions, mais ne se soucie pas des détails de l'implémentation. Une fois terminée, une espèce peut être mise en œuvre à travers la création d'une collection. Ainsi, une collection est une sorte de type de données abstrait, utilisable uniquement via les méthodes de son interface, avec la garantie supplémentaire que toutes les déclarations ont été définies et toutes les déclarations ont été prouvées.

La syntaxe d'une collection est la suivante:

collection *<name>* = ***implement*** *<name> (<pars>)* ***end*** *; ;*

Nous implémentons les espèces complètes **PointAbstrait** comme suit:

```
species PointAbstrait = signature get_x : Self -> float; signature
get_y : Self -> float;

let get_x(p:Self) = fst(p);
let get_y(p:Self) = snd(p);
let distance (p1:Self, p2: Self):float = #sqrt( ( get_x(p1) -f get_x(p2)
) *f ( get_x(p1) –f get_x(p2) ) +f ( get_y(p1) -f get_y(p2) ) *f (
get_y(p1) -f get_y(p2) ) );

Proof of distance = assumed;
end;;
collection PointAbstrait Collection = implement PointAbstrait;
end;;
```

II.3.2. Paramétrage

Comme l'héritage, le paramétrage est un mécanisme qui permet de créer une nouvelle espèce en utilisant des espèces existantes. Le paramétrage permet à une espèce d'utiliser les

méthodes et les entités des autres espèces. FoCaLiZe permet deux formes de paramétrage:

- Une espèce peut être paramétrée par collections (des espèces spécifiées).
- Une espèce peut être paramétrée par entités de collections.

II.3.2. 1. Paramètres de collections

Un paramètre de collection est une espèce (fournisseur) utilisée par une autre espèce (la cliente)comme type abstrait, par l'intermédiaire de son interface.

Un paramètre de collection est introduit par un nom (**parameter_name**) et une interfacedésignée par le nom d'une espèce (**supplier_species_name**), en utilisant la syntaxe suivante:

species *client_species_name* (*parameter_name* is *supplier_species_name*, . . .)
. . . **end**;;

Le nom du paramètre (**parameter_name**) est utilisé par l'espèce cliente comme une variable de type, et pour appeler les méthodes de l'espèce fournisseur. En géométrie, un cercle est défini par son centre (un point) et son diamètre. Une espèce Cercle (modélisant les cercles) peut être créée en utilisant l'espèce **PointAbstrait** comme paramètre de collection :

```
species Cercle (P is PointAbstrait) = representation = P * float ;
let get_centre(c:Self):P = fst(c);
 let get_rayon(c:Self):float = snd(c);
let creeCercle(centre:P, rayon:float):Self = (centre, rayon);
let appartient(p:P, c:Self): Bool = (P!distance(p,
get_centre(c)) =get_rayon(c));
theorem appartient_spec : all c:Self, all p:P,
appartient(p, c) <->(P!distance(p, get_centre(c)) =
get_rayon(c))
proof = assumed ; end;;
```

II.3.2.2. Paramètres d'entités

Un paramètre d'entité est une entité d'une espèce existante (espèce fournisseur), utilisée par l'espèce cliente pour servir en tant que valeur effective au sein du développement de ses propres méthodes.

Un paramètre d'entité est introduit par le nom d'une entité (**entity_name**) et le nom d'une collection (**collection_name**), en utilisant la syntaxe :

```
species client_species_name (entity_name in collection_name, . . .) . . .
```

La collection **collection_name** peut servir comme type dans l'espèce, comme elle peut être utilisée pour invoquer les fonctions de l'espèce complète sous-jacente.

Dans cet exemple de la géométrie, l'origine d'un repère orthogonal est un point particulier (valeur) qui peut être une entité de la collection **PointConcretCollection** donnée ci-dessus. Ainsi, une espèce **RepereOrthogonal** qui modélise les repères orthogonaux utilise un paramètre d'entité de la collection **PointConcretCollection** comme suit :

```
species RepereOrthogonal (origine in PointConcretCollection) =
...end;;
 let o=PointConcretCollection!creerPointConcret(0.0, 0.0);;
collection RepereOrthogonalZero = implement
RepereOrthogonal(o);end;;
```

FoCaLiZe ne permet pas la spécification d'un paramètre d'entité de type primitif (int, string, bool . . .). Cela signifie que les types primitifs doivent être intégrés dans des collections pour être utilisés comme paramètres d'entités.

II.4. Preuves en FoCaLiZe

Mener un développement FoCaLiZe requiert de prouver que les propriétés ou les théorèmes énoncés sont valides (ABBAS, 2018).

FoCaLiZe dispose de trois modes de preuve de théorèmes :

1. Soit en utilisant le mot clé *assumed* pour admettre un théorème sans donner sa preuve

(axiome) ;

2. Soit en introduisant manuellement un script de preuve en Coq ;

3. Soit en utilisant le langage de preuve de FoCaLiZe (FPL, FoCaLiZe Proof Language) pour écrire une preuve qui sera directement déchargée par le prouveur automatique de théorèmes de FoCaLiZe : Zenon.

La syntaxe générale pour introduire la preuve d'un théorème ou d'une propriété est présentée comme suit :

```
theorem theorem_name : theorem_specification
 proof = proof ;                                   (* cas d'un théorème *)
 proof of property_name = proof ;          (* cas d'une propriété déjà déclarée*)
```

II.5. Compilation

La compilation d'un fichier source FoCaLiZe génère deux codes cibles: un code Ocaml et un code Coq. Le code Ocaml représente l'exécutable du programme, et le code Coq sert à vérifier les preuves et la cohérence du modèle. La compilation est réalisée, en quatre étapes (ABBAS, 2018):

1. Compilation du fichier source FoCaLiZe (source.fcl) : Le compilateur FoCaLiZe (focalizec) lit le fichier source (source.fcl) et génère un code OCaml (source.ml) et un code Coq intermédiaire (source.zv) qui contient une traduction du code FoCaLiZe (y compris l'aspect calculatoire) vers Coq. A la place des preuves de propriétés, nous trouvons des emplacements vides (trous) dans source.zv qui sera complétés par les preuves que Zenon va produire.

2. Compilation du code OCaml : A cette étape, FoCaLiZe invoque le compilateur ocamlc pour produire un exécutable à partir du code OCaml (source.ml), généré dans l'étape précédente.

3. Compilation du code Coq intermédiaire (source.zv) : FoCaLiZe invoque la commande zvtov pour produire un code Coq complet (source.v) à partir du code intermédiaire (source.zv) généré dans la première étape. Tous les vides laissés dans source.zv sont remplis par des preuves Coq effectives réalisées par Zenon.

4. Compilation du code Coq : Enfin, FoCaLiZe invoque la commande coqc pour compiler le code Coq (source.v), généré dans l'étape précédente.

II.6. Conclusion

FoCaLiZe est le résultat d'un travail collectif de plusieurs chercheurs, lorsqu'il se caractérise parplusieurs spécifications telles que:

- ✓ Est un atelier de développement et de la programmation orienté objet.
- ✓ Est un environnement de développement de logiciels sûrs.
- ✓ Couvre toutes les phases du cycle de vie du logiciel de la spécification de besoin jusqu'au la génération du code exécutable.
- ✓ Est une langue basée sur des résultats théoriques fermes avec une sémantique claire etfournit une mise en œuvre efficace.
- ✓ Est une méthode formelles qui permet d'exprimer des propriétés et prouve, que cespropriétés sont satisfaits vis-à-vis les spécifications.

CHAPITRE III:

De UML vers FoCaLiZe

III.1. Introduction

L'aspect structurel de notre modèle UML se compose de classes et de relations entre classes. Chaque classe peut être spécifiée par une liste d'attributs et d'opérations, éventuellement vide. Généralement, les relations entre classes sont les héritages, les dépendances et les associations binaires. Une classe peut également être spécifiée avec une liste de paramètres de modèles formels, il peut également s'agir d'un modèle de liaison dérivé des liaisons de modèles à partir d'autres modèles de classes.

Dans ce chapitre, nous décrivons les règles de transformation automatique des diagrammes de classes UML en FoCaLiZe. Pour justifier les règles de transformation, nous faisons d'abord une comparaison entre les caractéristiques de conception des concepts UML présentés au chapitre 1 et les concepts FoCaLiZe présentés au chapitre 2. Dans cette comparaison, nous décrivons les similitudes ainsi que les différences conceptuelles entre les diagrammes de classes UML et FoCaLiZe.

Les transformations présentées sont basées sur les études et travaux présentés en (ABBAS, 2018)

III.2. Comparaison entre les concepts UML et FoCaLiZe

A partir des études ci-dessus des concepts UML et FoCaLiZe (voir chapitres 1 et 2), nous concluons les similitudes suivantes entre les concepts UML et FoCaLiZe :

- Une classe UML correspond à une espèce FoCaLiZe.
- Les attributs de classe jouent le même rôle que les représentations des espèces FoCaLiZe.
- Les opérations des classes UML correspondent aux fonctions des espèces FoCaLiZe.
- Les relations entre les classes UML sont également similaires aux relations entre les espèces FoCaLiZe.
- La dépendance entre les classes UML correspond au paramétrage au sein de FoCaLiZe.
- L'héritage multiple est pris en charge par UML et FoCaLiZe.
- FoCaLiZe comme UML, permet la redéfinition des méthodes et les mécanismes de liaison tardive.
- Les modèles UML (classes paramétrées) sont équivalents aux espèces paramétrées.
- La liaison de modèle (la substitution des paramètres formels de classe par des paramètres réels) correspond à la substitution des paramètres d'espèce dans FoCaLiZe.

Le tableau suivant résume les principales correspondances entre UML et FoCaLiZe:

UML	FoCaLiZe
Class	Species
Attributes	Representations
Operations	Functions
Instances(object)	Instances(entities)
Inheritance + multiple inheritance	Inheritance + multiple inheritance
Parametrized class	Parametrized species
Late binging	Late binging
Data encapsulation	Data encapsulation

Tableau III. 1: Comparaison entre les concepts UML et FoCaLiZe

III.2.1. Transformation d'une classe

Une classe UML est transformée en une espèce FoCaLiZe comme suit :

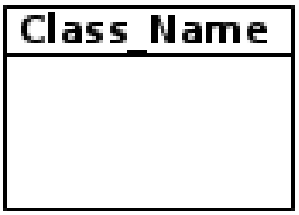
Classe UML	Code FoCaLiZe
	***Species*** *Class_Name =* *…* ***end****;;*

Tableau III. 2: Transformation de classe UML.

Une classe peut contenir une liste d'attributs et des opérations. Ensuite, nous allons décrire latransformation de ces derniers.

III.2.2. Transformation d'attributs

Chaque attribut de classe est formalisé par une fonction "getter" (qui renvoie la valeur del'attribut) dans les espèces correspondantes.

Pour un attribut **attr: attr_type**, la fonction qui modélise l'attribut **attr** dans les espèces correspondantes est:

Signature get_attr: Self -> [attr_type];

Où **[attr_type]** est le type correspond à **attr: attr_type** en FoCaLiZe.

Il existe deux possibilités pour traduire le type d'attributs:

2.I.1 Si **attr_type** est un UML types primitifs (entier, réel, String et Boolean), nous utilisons directement les types primitifs en FoCaLiZe (int, float, string et bool).

2.I.2 Si **attr_type** n'est pas un type primitif (un type de classe), nous utilisons les espèces correspondantes.

Le tableau suivant montre la transformation d'une classe UML avec des attributs:

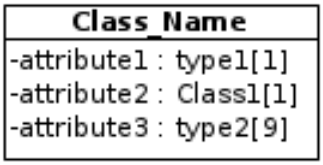
Les attributs de classe	Code FoCaLiZe
	***species*** *Class_Name (C is Class1)=* ***signature*** *get_attribute1: Self -> attr_type1 ;* ***signature*** *get_ attribute2: Self -> C ;* ***signature*** *get_ attribute3: Self ->list(type2) ;* ***end;;***

Tableau III. 3: Transformation d'une classe UML avec des attributs.

Si la multiplicité des **attr_type** est supérieur à un (comme l'attribut **attr3** dans le tableau IV.2): nous utilisons le type liste dans FoCaLiZe:

Signature get_attr: Self ->list (**Foc_attr_type)**;

III.2.2.1. Transformation des opérations

Les opérations de classe seront transformées en fonctions (signature) des espèces correspondantes, la signature dérivée d'une opération d'UML dépend des paramètres de l'opération (d'entrée et de sortie), les types des paramètres peuvent être des types primitifs ou de type classe, la transformation des types d'opérations est similaire à la transformation des types d'attributs, nous distinguons les cas suivants:

- Si l'opération a plusieurs paramètres "**in**" et un paramètre "**out**":

Op_name (p_name1: type 1..., p_name n: typen): return_type, sa transformation est:

Signature op_name: self -> type1->… ->typen->return_type**;**

- Si l'opération a plusieurs paramètres "**in**" et sans paramètres "**out**":

Op_name (p_name 1: type 1, ..., p_name n: type n), sa transformation sera:
Signature op_name: self -> type1->… ->typen->self**;**

- Si l'opération est sans paramètres:

Op_name (), elle se transforme en une signature des espèces correspondantes:

Signature op_name: self ->self;

Le tableau suivant montre la transformation d'une classe avec des opérations :

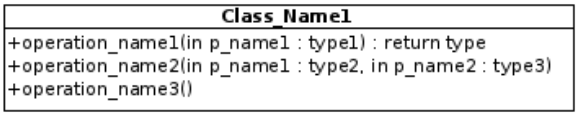
Les opérations de classe	Code FoCaLiZe
	***Species*** *Class_Name1 =* ***signature*** *operation_name1: Self -> type -> return_type;* ***signature*** *operation_name2: Self ->type2 ->type3 -> Self;* ***signature*** *operation _name3: Self ->Self;* ***end****;;*

Tableau III. 4: Transformation d'une classe UML avec des opérations.

III.2.2.2. Transformation de type énumération

Un type d'énumération peut uniquement être utilisé pour typer les méthodes d'espèces, il ne peut pas être utilisé pour typer les paramètres d'une espèce. De plus, les paramètres d'espèces peuvent uniquement être de type collection et/ou de type entité d'une autre espèce

Le tableau suivant montre un exemple de transformation d'énumération :

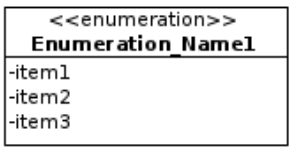
Type d'Enumeration classe	Code FoCaLiZe
	***type*** *enumeration_Name = | Item1 | Item2 | Item3 | ... ;;* ***species*** *Enumeration_Name =* ***representation*** *= enumeration_Name;* ***signature*** *new_ Enumeration_Name : enum_Name -> Self;* ***signature*** *to_ Enumeration_Name : Self -> enumeration_Name;* ***end;;***

Tableau III. 5: Transformation d'une classe UML d'énumération

III.2.3. Transformation de la relation de dépendance

Une relation de dépendance entre deux classes est modélisée par le paramétrage dans FoCaLiZe.

Par exemple, pour une classe C1 dépendant de la classe C2, l'espèce C1 (dérivée de la classe C1) sera paramétrée par l'espèce C2 (dérivée de la classe C2).

Le tableau suivant présente un exemple de transformation de dépendance :

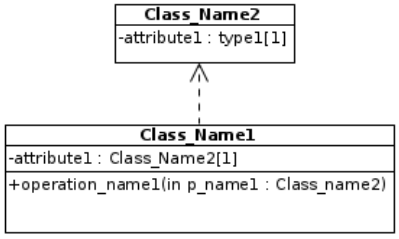
Relation de dependence	Code FoCaLiZe
	***species*** *Class_Name1(C is Class_Name2) =* ***signature*** *get_attribute1: Self -> C ;* ***signature*** *operation_name1 : Self -> C -> Self ;* ***new*** *Class_Name: C -> self;* ***end*** *;;*

Tableau III. 6: Transformation de la relation de dépendance

III.2.4. Transformation de l'héritage

Parce que la spécification en FoCaLiZe prend en charge l'héritage et l'héritage multiple (voir chapitre 02), nous transformons directement la relation d'héritage UML en relation d'héritage FoCaLiZe. L'héritage multiple entre classes se transforme en héritage multiple entre espèces.

Le tableau suivant montre un exemple de transformation d'héritage UML :

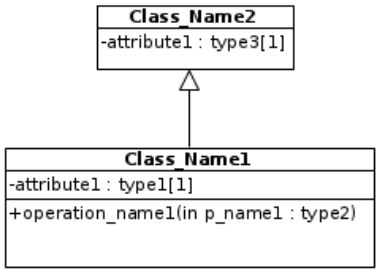
UML Simple heritage	Code FoCaLiZe
	***species*** *Class_Name1 =* ***inherit*** *Class_Name2;* ***signature*** *get_attribute1: Self -> type1 ;* ***signature*** *operation_name1: Self -> type1 -> Self;* ***signature*** *new_ Class_Name1:type1 -> Self;* ***end****;;*

Tableau III. 7: Transformation de l'héritage simple

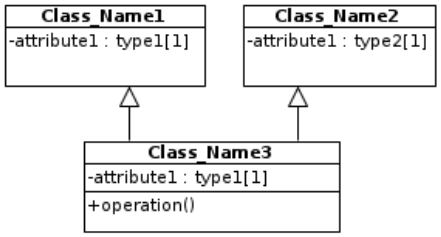
UML Multiple heritage	Code FoCaLiZe
	***species*** *Class_Name3 = inherit Class_Name1, Class_Name2 ;* ***signature*** *get_attribute1: Self -> type1 ;* ***signature*** *operation_name1: Self -> Self;* ***signature*** *new_ Class_Name1:type1 -> Self ;* ***end****;;*

Tableau III. 8: Transformation de l'héritage Multiple

III.2.5. Transformation de classe Paramétré (Classe Template)

Une classe paramétrée (Template,) spécifiée par une liste de paramètres formels est transformée en une espèce paramétrée, où chaque paramètre du Template est transformé en paramètre de l'espèce correspondante.

Le tableau suivant montre un exemple de transformation du classe Template :

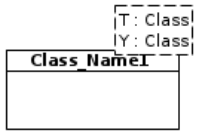
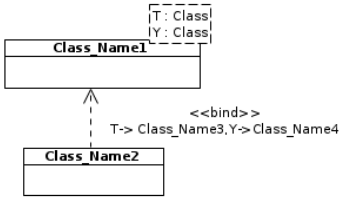
Classe paramétré	Code FoCaLiZe
	***Species*** *Class_Name1(T is Basic_object, Y is Basic_object) =* ***inherit*** *Basic_object ;* ... ***end****;;*
Template Binding	**Parameter substitution**
	***Species*** *Class_Name2(T is Class_name3, Y is Class_name4) =* ***inherit*** *Class_Name1(T , Y) ;* ***end****;;*

Tableau III. 9: Transformation du classe Template

III.2.6. Transformation des associations

Contrairement aux autres fonctionnalités d'UML, les associations binaires n'ont pas d'équivalents directs en FoCaLiZe. Pour cette raison, nous avons développé plusieurs espèces générales qui seront instanciées pour formaliser les différents aspects des associations binaires. En particulier, nous développons les espèces suivantes :

_ L'espèce Association, modélisant la structure générale d'une association binaire ;

_ L'espèce Direction, modélisant la direction de navigabilité d'une association binaire ;

_ L'espèce Range, modélisant les multiplicités aux extrémités gauche et droite d'une association binaire.

Le tableau suivant montre la transformation d'une association UML vers FoCaLiZe.

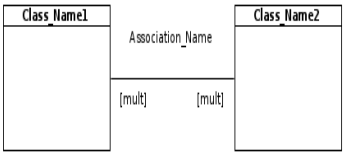
Association	Code FoCaLize
	***species*** *Association_name* *(Left is Class_Name1, Left_Set is Set(Left),Right is Class_Name2, Right_Set is Set(Right))* *=* ***inherit*** *Association (Range_collection, Left, Left_Set, multl,Right, Right_Set, multr, Direction_collection, d);* *end;;* ***let*** *d = Direction_collection!make_direction(Both);; let multl = Range_collection!make_range(Unlimited_Nat!make(I(1)), Unlimited_Nat!make(I(1)));;* ***let*** *multr = Range_collection!make_range(Unlimited_Nat!make(I(1)),* *Unlimited_Nat!make(I(1)));;*

Tableau III. 10: Transformation d'association UML

III.3. Conclusion

Dans ce chapitre, nous avons cité la transformation formelle des diagrammes de classes UML en FoCaLiZe. A partir d'un diagramme de classes UML réalisé par M.ABBAS, qui a généré une spécification textuelle qui est conforme à la syntaxe proposée pour UML et structurée suivant les dépendances syntaxiques et sémantiques entre les éléments du modèle. Ensuite, il a produit une spécification FoCaLiZe, où chaque classe correspond à une espèce, les attributs de classe ont modélisés par leurs fonctions getters, les opérations de classe sont transformées en fonctions déclarées (signatures) de l'espèce correspondante et les relations de classes sont formalisées par des relations équivalentes entre les espèces.

CHAPITRE IV:

Conception

IV.1. Introduction

Lors de ce chapitre, nous présentons le schéma global de modèle de transformation automatique d'UML/OCL en FoCaLiZe inspiré du modèle systématique de transformation UML/OCL en FoCaLiZe proposé dans l'article (Abbas, 2020)

Nous allons aussi identifier les diagrammes de classes réalisée pour mettre en œuvre l'architecture d'outil de transformation automatique l'aspect structurel d'UML en FoCaLiZe. La motivation fondamentale de la modélisation est de fournir une démarche antérieure afin de réduire la complexité du système étudié lors de la conception et d'organiser la réalisation du projet en définissant les modules et les étapes de la réalisation. Plusieurs démarches de modélisation sont utilisées. Nous adoptons dans notre travail une approche objet basée sur un outil de modélisation UML.

IV.2. Schéma globale :

Ce schéma nous montre une vue globale du modèle de transformation automatique utilisé lors de l'implémentation :

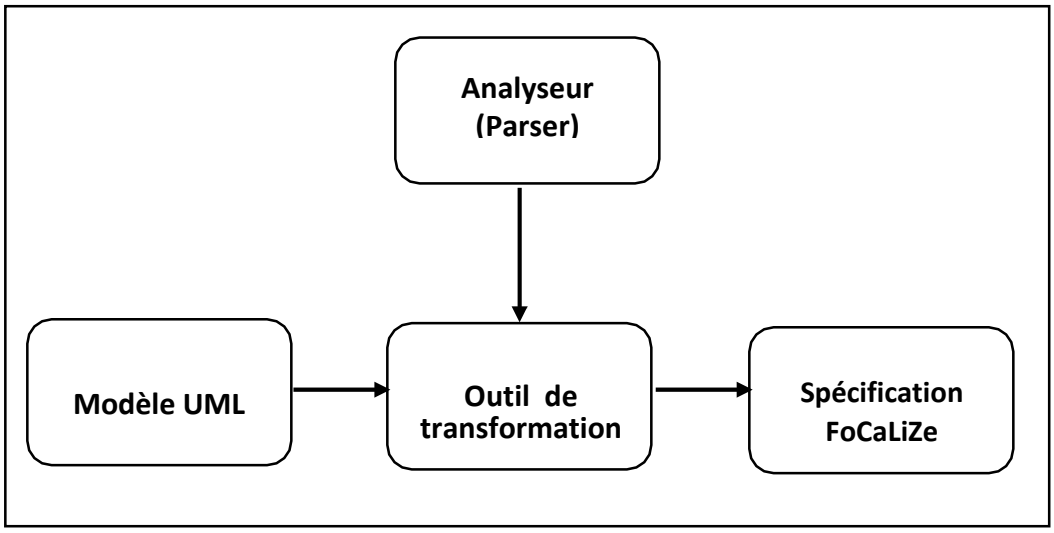


Figure IV.1: schéma globale de modèle de transformation

IV.3. Conception de l'architecture

IV.3.1. Identification des diagrammes

Pour modéliser l'architecture de l'outil de transformation, nous allons identifier deux diagrammes:

● Le diagramme d'activités décrit la succession des activités et leurs interactions dans un

processus du système.

● Un diagramme de classes est une collection d'éléments de modélisations statiques (classes, attributs...), qui montre la structure d'un modèle. Les classes sont liées entre elles par des associations. Une association permet d'exprimer une connexion sémantique bidirectionnelle entre deux classes.

IV.3.2. Diagramme d'activités

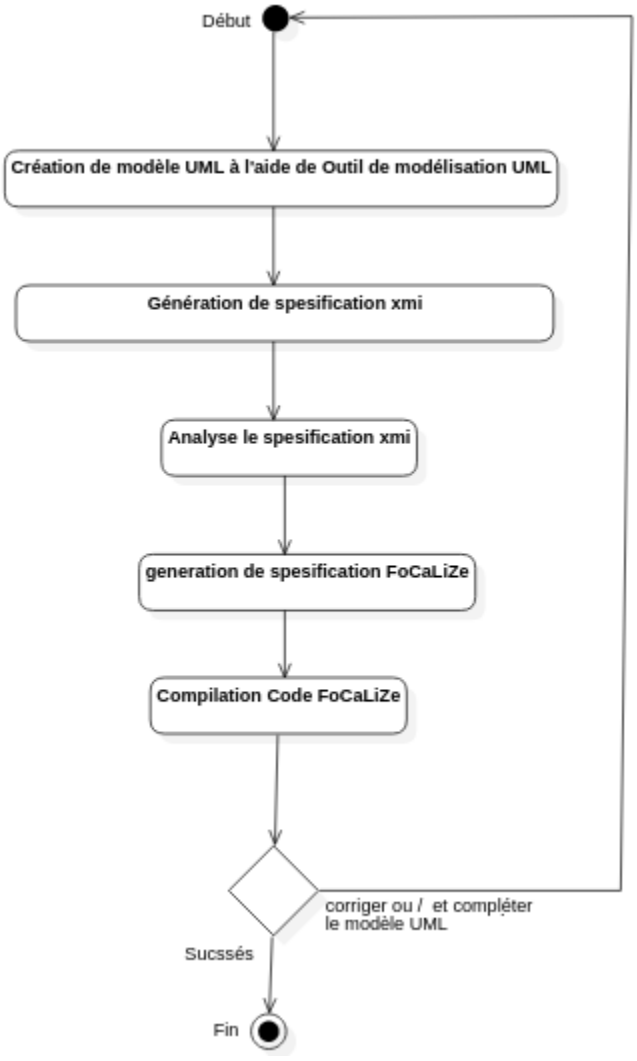


Figure IV.2: Diagramme d'activités représentant l'interaction entre les différents modules

Cinq étapes sont définies (figure IV. 2) :

- **Première étape : Création de modèle UML**

Cette étape consiste à créer un modèle UML à aide d'un outil de modélisation UML qui

respect le standard ouvert contrôlé par l'OMG.

- **Deuxième étape : Génération de spécification XMI**

Cette étape exploite le résultat de étape précédente pour générer le format XMI correspond au modèle UML crée ; le résultat de cette étape est la génération du fichier de format XMI.

- **Troisième étape : Analyse de spécification XMI**

Cette étape consiste à l'utilisation d'un Parser afin d'analyser la spécification XMI et extraire les éléments constituant l'aspect structurel du diagramme de classe UML, et mettre ces éléments dans une liste.

- **Quatrième étape : Génération de la spécification FoCaLiZe**

FoCaLiZe exige le respect d'ordre des dépendances entre les Classes, pour cela nous trions la liste des éléments constituant l'aspect structurel du diagramme de classe UML en fonction de leurs dépendances, A partir de la liste généré, on applique les règles de transformation réalisés par M.ABBAS (voir chapitre III) pour générer une spécification FoCaLiZe.

- **Cinquième étape : Compilation code FoCaLiZe**

Lors de l'étape de compilation, le compilateur FoCaLiZe vérifie l'éventuelle présence des erreurs, On doit retourner à la première étape pour corriger le modèle UML/OCL en cas de présence d'erreurs sinon nous adoptons ce modèle.

IV.3.3. Diagramme de classes

Le diagramme de classes représenté dans la figure IV-3 suivante décrit la structure et les relations entre les classes et ceci afin de déterminer les dépendances entre les différentes classes.

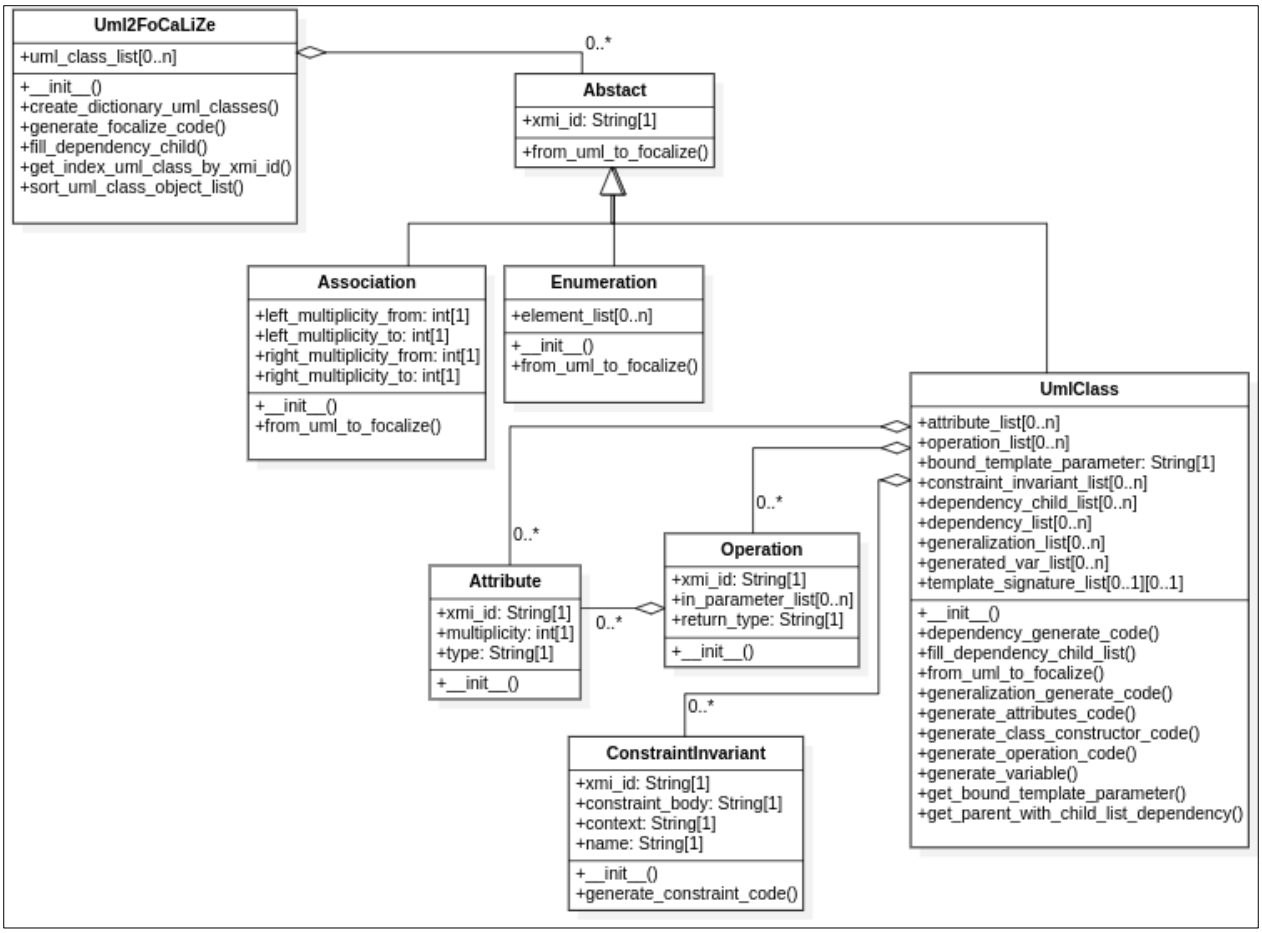


Figure IV. 3: Diagramme de classes d'outil de transformation

La figure ci-dessus présente l'architecture détaillée de l'outil de transformation à partir d'analyse de spécification jusqu'à la génération de la spécification FoCaLiZe :

- L'analyseur (Parser) parcours la spécification XMI obtenue du deuxième étape cité dans le diagramme d'activités : le constructeur de classe **UML2FOC __ini__()** est chargé d'analyser le fichier XMI.

- Le résultat obtenue est une liste **uml_class_list** composée d'objets (de type **UmlClass** , **Association** et **Enumeration**), ces objets décrit l'aspects structurels d' éléments du modèle UML/OCL.

- Par l'appel du méthode **sort_uml_class_object_list()** qui sert au tri du liste d'objets **uml_class_list** suivant l'ordre de dépendance selon les cas suivantes :

- une classe hérite d'une autre classe;
- une classe est paramétrée par une autre classe ;
- une classe est dépendante d'autre classe;
- une classe est une classe liée obtenue par la substitution des paramètres formels d'une autre classe par des paramètres effectifs.

Ces dépendances définissent un graphe de dépendances qui va déterminer l'ordre de traitement des classes (ABBAS, 2018).

- La méthode **generarate_focalize_code**() consiste à générer le code FoCaLiZe par l'exécution du méthode **from_uml_to_focalize()** (qu'est une méthode des éléments de la liste **uml_class_list**).

- Dans le tableau suivant nous présentons les méthodes et attributs de chaque classe dans notre modèle proposé.

Structure de données	**Rôles**
dictionary_xmi_id_element	**Dictionnaire {clé :: xmi_id , Valeur :: Nom}** contient tous les composant de modèle UML
generated_variable_list	List global des variables générés lors de la création de code FoCaLiZe
Uml2FoCaLiZe	
Attribute	
uml_class_list	Liste des objets de type **UmlClass** ou **Enumeration** ou **Association**
Méthodes	
__ini__()	Constructeur charger de extrait les données depuis le fichier XMI en créant les listes **constraint_invariant_list** et **uml_class_list**
create_dictionary_uml_classes()	Cree un dictionnaire des class contient que les class UML et leur **xmi_Id**
Generate_focalize_code()	Générer le code FoCaLiZe par l'appel des méthodes **from_uml_to_focalize()** des objets de

	liste **uml_class_list**
fill_dependency_child()	Charger la liste de dependeny_cheild_list pour chaque classe
get_index_uml_class_by_xmi_id()	Retune index of Uml object correspond aux **xmi_Id** dans la liste **uml_class_list**
sort_uml_class_object_list	Trier la liste uml_class_list selon la dépendance entre les classes
Abstract	
Attribute	
xmi_id	Identificateur de composant dans fichier XMI
Operation	
generalization_generate_code()	Abstract méthode pour generation de code
UmlClass	
Attribute	
attribute_list	Liste des objets de type **Attribute**
operation_list	Liste des objets de type **Operation**
bound_Template_parametre	List des paramètres de substitution
constraint_invariant_list	Liste des contraintes de classe
dependeny_cheild_list	Liste de **xmi_id** des classes des utilisé par les classes fils de cette classe
dependency_list	Liste des **xmi_id** des classes utilisé par cette classe dépendance
generalization_list	Liste des **xmi_id** des classes père de cette classe
generated_var_list	Liste des variables généré par la classe lors de la génération de code FoCaLiZe
Template_signature_list	Liste de paramètre de Template
Operation	
__init__()	Constructeur appelé par **Uml2FoCaLiZe** constructeur méthode pour l'initialisation de leurs attributs depuis le fichier XMI

dependency_generate_code()	Générer le code selon le dépendance
fill_dependancy_child_list()	Remplit la liste des dépendances des fils
from_uml_to_focalize()	Génère et retourne le code FoCaLiZe de classe
generalization_generate_code()	Génère le code correspond aux héritage
generate_attributes_code()	Génère de code de liste des attribues
generate_class_constructor_code()	
generate_operation_code()	généré le code de liste des opérations
generate_variable()	généré des variable lors de la génération de code
get_bound_Template_parameter()	Récupérer les paramètres de Template
get_parent_with_child_list_dependency()	Récupérer la liste des dépendances
Enumeration	
Attribute	
element_list	Liste des elements
Méthodes	
__init__()	Constructeur appelé par **Uml2FoCaLiZe** constructeur méthode pour l'initialisation de leurs attributs depuis le fichier XMI
from_uml_to_focalize	Génère et retourne le code FoCaLiZe de l'**Enumeration**
Association	
Attribute	
left_multiplicity_from	La valeur minimum de multiplicité de gauche
left_multiplicity_to	La valeur maximum de multiplicité de gauche
right_multiplicity_fom	La valeur minimum de multiplicité de droite
right_multiplicity_to	La valeur maximum de multiplicité de droite
Méthodes	
__init__()	Constructeur appelé par **Uml2FoCaLiZe** constructeur méthode pour l'initialisation de leurs attributs depuis le fichier XMI

from_uml_to_focalize	Génère et retourne le code FoCaLiZe de l'**Association**
Attribute	
Attribute	
xmi_id	Identificateur de composant dans fichier XMI
type	**xmi_id** de type
multiplicity	La multiplicité de variable
Méthodes	
__init__()	Constructeur appelé par **UmlClass** constructeur ou **Operation** constructeur, méthode pour l'initialisation de leurs attributs depuis le fichier XMI
Operation	
Attribute	
xmi_id	Identificateur de composant dans fichier XMI
return_type	**xmi_id** de type de reture
multiplicity	La valeur minimum de multiplicité de droite
Méthodes	
__init__()	Constructeur appelé par **UmlClass** constructeur ou **Operation** constructeur, méthode pour l'initialisation de leurs attributs depuis le fichier XMI

Tableau IV. 1: les structures de données et leurs rôles

IV.4. Conclusion

Dans ce chapitre, nous avons présenté le schéma global de modèle de transformation automatique d'UML/OCL en FoCaLiZe et on a identifié les diagrammes d'activités et de classes pour faciliter la réalisation de notre prototype.

Dans le chapitre suivant nous montrerons les étapes, plus en détails, que nous avons suivies pour implémenter et réaliser notre solution.

CHAPITRE V:

Implémentation

V.1. Introduction

L'implémentation est la phase la plus importante après celle de la conception Le choix des outils de développement a un impact énorme sur le coût du temps de programmation et la flexibilité du produit à réaliser.

Cette étape comprend la conversion du modèle conceptuel précédemment établi en composants logiciels qui forment notre système.

Dans ce chapitre, nous allons commencer par décrire l'environnement de travail, puis identifier et développer les composants de notre système..

Nous décrivons aussi les différentes étapes de notre implémentation d'outil de transformation automatique de diagramme de clases UML en FoCaLiZe.

V.2. L'environnement de travail

Pour mettre en œuvre notre processus de transformation, nous avons utilisé:

- L'outil Visual Paradigm version 16.2 pour la création de modèles.
- L'environnement Pycharm pour développer l'outil de transformation.

V.2.1. Visual Paradigm

Visual Paradigm (VP-UML) est un outil UML CASE qui prend en charge UML 2.5, SysML et Business Process Modeling Notation (BPMN) de l'Object Management Group (OMG). En plus du support de modélisation, il fournit également des fonctions de reporting et d'ingénierie de code, y compris la génération de code. Il peut générer des diagrammes à partir du code et fournir une ingénierie aller-retour pour divers langages de programmation. (https://www.visual-paradigm.com/download/)

V.2.2. Pycharm

Pycharm est un environnement de développement intégré pour la programmation Python. Il permet l'analyse de code et inclut un débogueur graphique. Il permet également la gestion des tests unitaires, un logiciel de gestion de version intégré et prend en charge l'utilisation de Django pour le développement Web.

Il est développé par la société tchèque JetBrains et est un logiciel multiplateforme qui fonctionne sous Windows, Mac OS X et Linux. Il dispose d'une version professionnelle, distribuée sous licence propriétaire, et d'une version communautaire distribuée sous licence Apache. (https://www.jetbrains.com/pycharm/download)

V.3. L'implémentation

Cette étape décrit le processus d'implémentation d'outil de transformation (l'installation et utilisation).

V.3.1. L'installation

L'installation de notre plugin sera mise en place 2 composants:

- Visual paradigm.
- Compilateur FoCaLiZe.

Pour utilise notre outil de transformation il faut installer les softwares suivantes :

V.3.1.1. Installation de Visual Paradigm

Pour installer l'outil de modélisation **Visual Paradigm** voir le site Web suivant:

https://www.visual-paradigm.com/

V.3.1.2 Installation de focalize

Pour installer FoCaLiZe, voir le lien :

http://FoCaLiZe.inria.fr/download/

Il requiert l'installation des outils externes suivants dans la racine de répertoire de FoCaLiZe(habituellement « Focalize »).

- **OCaml** (toute version récente => 3.11) :

Commande d'installation (sous ubunto): **#sudo apt-get install ocaml**

- **Coq** (toute version récente > = 8.1pl5):

Commande d'installation (sous ubunto): **#sudo apt-get install coq**

- **Zenon:** Il est disponible dans le site

http://FoCaLiZe.inria.fr/zenon.git.

V.3.2. Utilisation de l'outil

Dans cette étape, nous allons expliquer les techniques d'utilisation de notre outil de

transformation.

- Premièrement: à l'aide de Visual Paradigm on crie un modèle UML et on l'exporte en format XMI

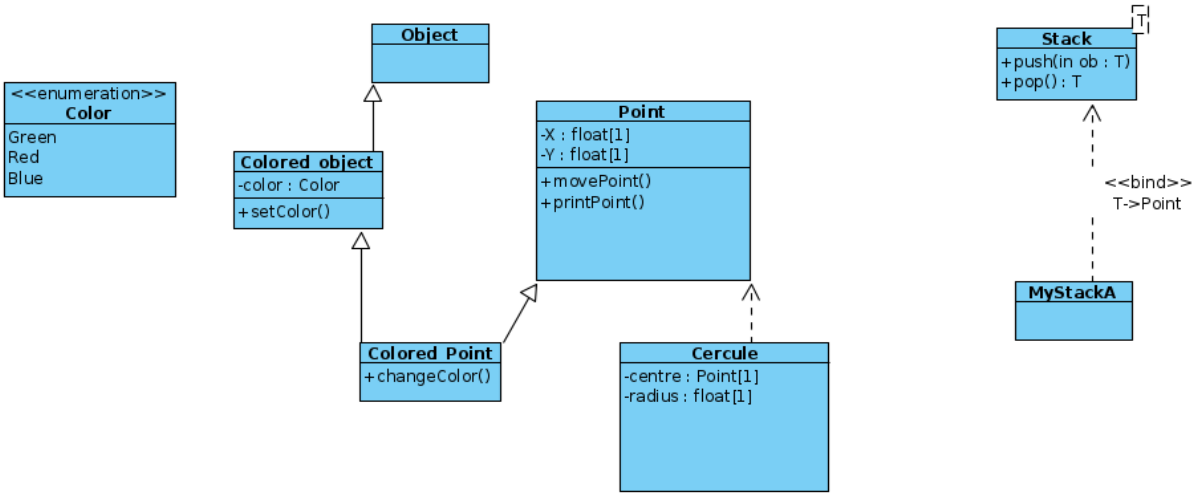


Figure V.1: Modèle UML créer par Visual Paradigm.

```
1     <?xml version="1.0" encoding="UTF-8"?>
2     <xmi:XMI xmi:version="2.1" xmlns:uml="http://schema.omg.org/spec/UML/2.0" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
3       <xmi:Documentation exporter="Visual Paradigm" exporterVersion="7.0.2">
8       <xmi:Extension extender="Visual Paradigm">
222     <uml:Model name="untitled" xmi:id="w8.hx56D.AACAQkO">
19106   <uml:Diagram diagramType="ClassDiagram" documentation="" name="Class Diagram1" toolName="Visual Paradigm" xmi:id="2yh
20053   </xmi:XMI>
```

Figure V.2: fichier de format XMI exporté depuis Visual Paradigm.

Deuxièmement: on utilise notre outil de transformation afin de généré la spécification FoCaLiZe

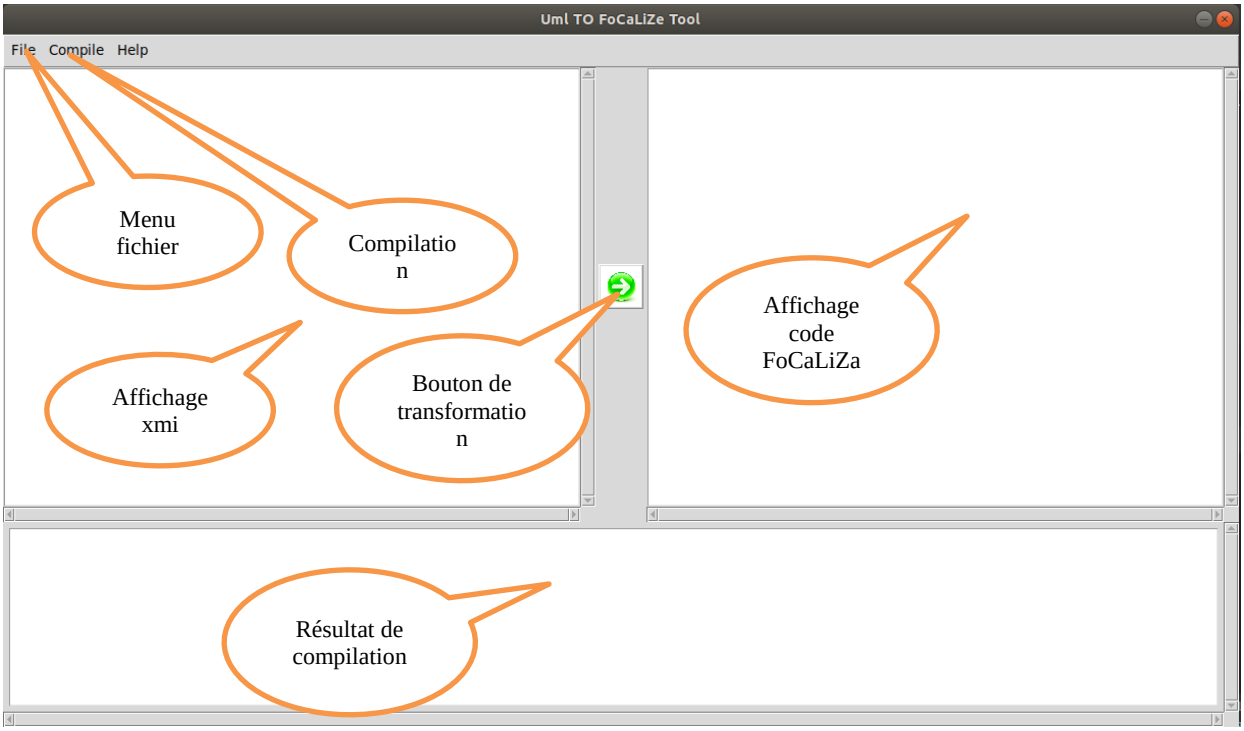


Figure V.3: interface de l'outil de transformation.

Pour transformer le modèle UML vers une spécification FoCaLiZe d'abord, on ouvre le fichier de format XMI exporté depuis l'outil visual Paradigm en utilisant la commande **Open** dans le menu **File** comme mentionné dans la **figure V.4**.

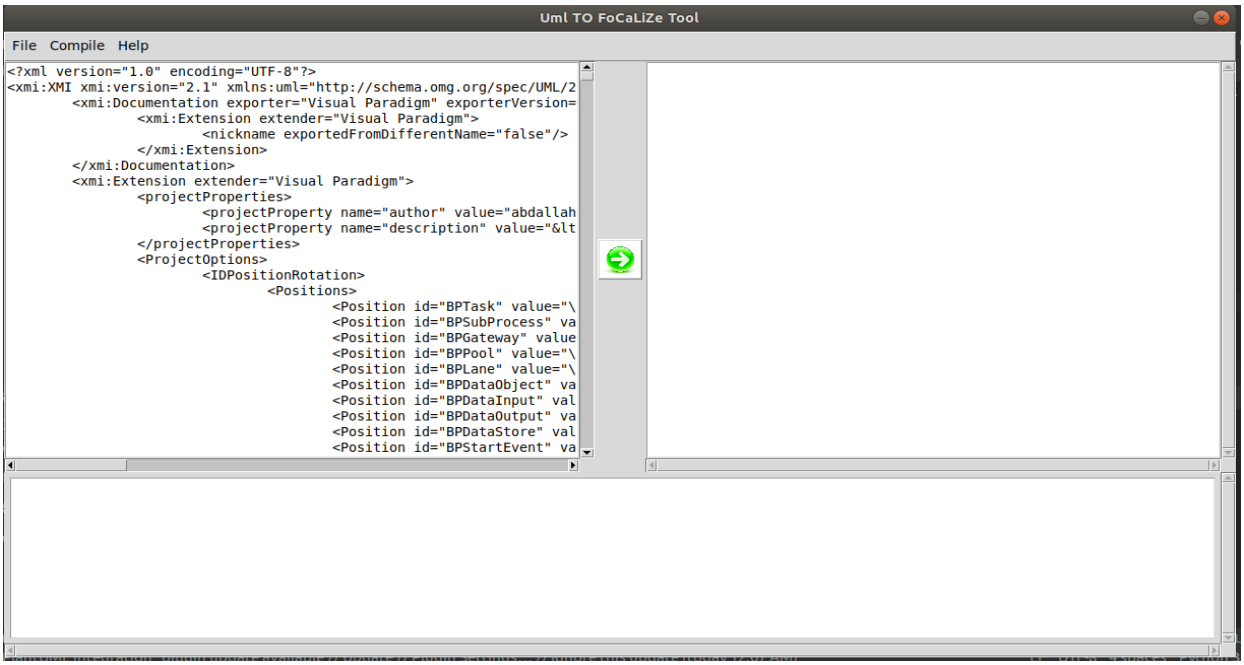


Figure V.4: ouvrir le fichier de format xmi.

Ensuite on appuyant sur le bouton de génération de spécification FoCaLiZe, le résultat de cette étape est présenté en **Figure V. 5**

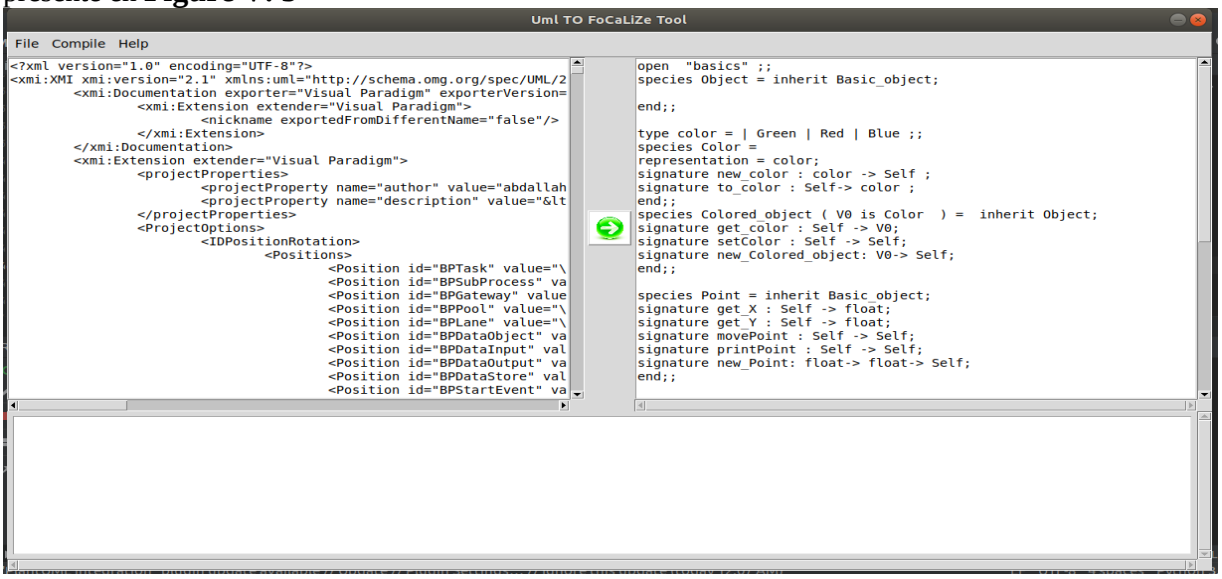


Figure V.5: génération de spécification FoCaLiZe.

En fin, on passe à l'étape de compilation, le résultat de compilation est présenté dans la **Figure V. 6**

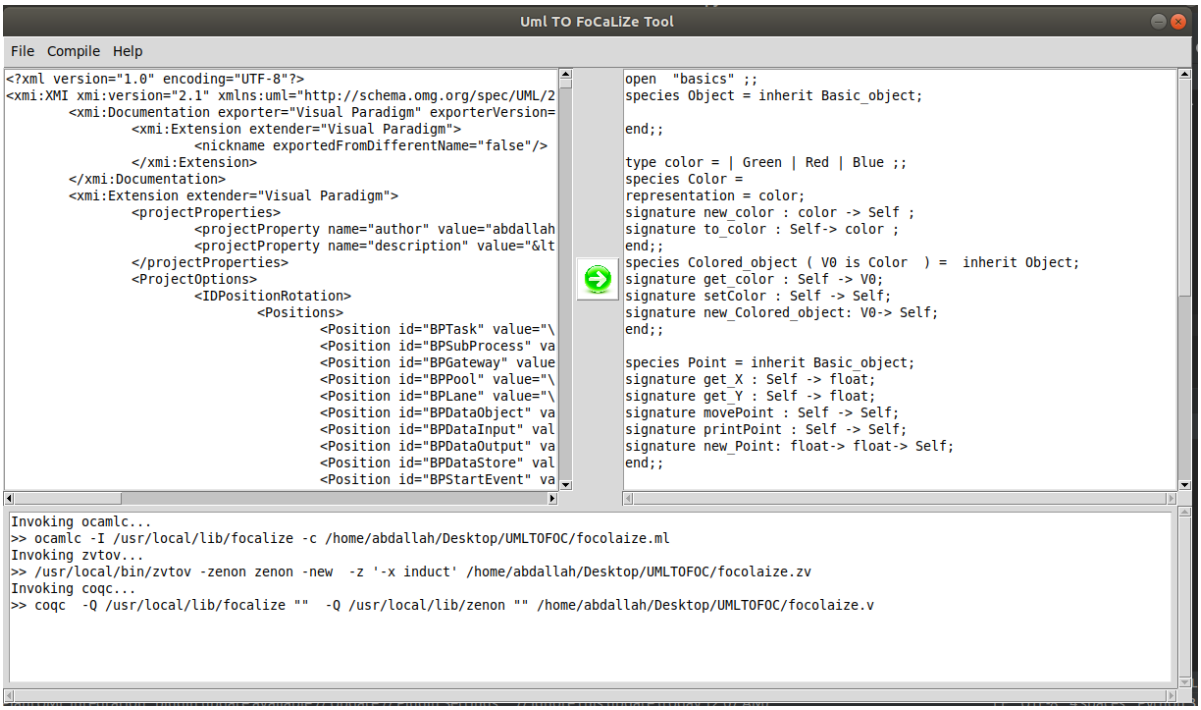


Figure V.6: résultat de la compilation.

Lors de l'étape de compilation, le compilateur FoCaLiZe détecte la présence éventuelle des erreurs:

- Si le compilateur FoCaLiZe trouve des erreurs dans le code, il indique les lignes correspondant aux erreurs.
- Si non, il nous renvoi ce message (**voir tableau V.7**) :

```
Invoking ocamlc...

>> ocamlc -I /usr/local/lib/focalize -c /home/abdelaziz/workspace/ATM/model.ml

Invoking zvtov...

>> zvtov -zenon zenon -new /home/abdelaziz/workspace/ATM/model.zv

Invoking coqc...

>> coqc  -I /usr/local/lib/focalize  -I /usr/local/lib/zenon /home/abdelaziz/workspace/ATM/model.v
```

Figure V.7: résultat de compilation

V.4. Conclusion

Dans ce chapitre, nous avons implémenté les démarches de transformation d'outil de transformation de diagramme de classes UML en FoCaLiZe. Le processus de transformation utilise un outil graphique UML, l'environnement FoCaLiZe et les règles de transformation.

L'approche proposée permet une transformation systématique du modèle UML/OCL en FoCaLiZe. Ensuite, un développeur FoCaLiZe peut compléter la spécification abstraite obtenue et fournir des preuves pour ses propriétés jusqu'à atteindre le code exécutable.

L'implémentation du processus de transformation se fait par l'analyse du spécification XMI et l'extraction des données qui décrit le diagramme de classes UML du modèle UML, ces données sont indispensable pour l'application des règles de transformation.

Conclusion général

Conclusion général

Dans ce mémoire, nous avons établi une autre alternative pour la formalisation de diagramme de classes, en utilisant l'environnement FoCaLiZe.

Pour mettre en œuvre la transformation proposée, nous avons utilisé un analyseur (Parser) qui nous permet de récupérer les données indispensable pour l'application d'outil de transformation de diagramme de classes UML exprimé dans la spécification XMI (généré par l'outil Visual Paradigm) dans FoCaLiZe.

Cette formalisation s'applique naturellement avec la transformation des diagrammes de classes,elle supporte les fonctionnalités d'UML comme l'héritage multiple et la paramétrisation.

Néanmoins, ce prototype n'est pas celui qui contient parfaitement toutes les fonctionnalités qu'un outil de transformation automatique de diagramme de classes UML en FoCaLiZe devrait avoir. Il est d'ailleurs en cours de développement et pourrait être amélioré selon plusieurs points de vue :

- Développer un analyseur (Parser) universel pour tous les Outils des modélisations
- L'intégration des modèles de transformation pour d'autres types des diagrammes UML dans notre prototype.
- Le traitement des contraintes OCL qui ne sont pas pris en compte lors de la réalisation de notre prototype

Bibliographies

Bibliographies

- ABBAS, M. (2018). L'environnement FoCaLiZe. Alger: Université USTHB.

- AYRAULT, P. (2011). Développement de logiciel critique en FoCaLiZe méthodologie et outils pour l'évaluation de conformité. Paris: Thèse de doctorat, Université Pierre et Marie Curie-6.

- Benoît CHARROUX, A. O.-M. (2009). UML2 Pratique de la modélisation. Paris: Pearson Education France.

- Booch, G. (1991). objct-oriental design with applications. CA: Cummings, Redwood City .

- David Delahaye, C. D.-N. (2010). Génération de code fonctionnel cérite à partir de spécifications inductives dans l'environnement Focalize .

- David Delahaye, J.-F. É.-G. (2008). Theoretical Aspects of Software Engineering, 121-124.

- David Delahaye, J.-F. É.-G. (2008). A formal and sound transformation from Focal to UML : an application to airport security regulations. Innovations in Systems and Software Engineering, 267-274.

- Doligez, D. (2015). the Zenon tool. Software and documentations freely available at http://focal.inria.fr/zenon/.

- Fechter, S. (2005). Sémantique des traits orientés objet de Focal. PARIS: Université PARIS 6.

- Ivar Jacobson, M. C. (1994). The cost method : a usecase-driven approach. Object development methods, 247-270.

- James R Rumbaugh, M. R. (1990). Object-oriented modeling and design.

- Khadija, G. M. (2015). Automatic transformation tool of UML classe diagrams into FoCaLiZe . Thése de Master UNIVERSITE ECHAHID HAMMA LAKHDAR EL OUED.

- Messaoud Abbas, C.-B. B.-Y. (2014). Generating FoCaLiZe Specifications from UML Models. The International Conference on Advanced Aspects of Software Engineering . Constantine Algeria: ICAASE .

- OMG. (2014). OCL : Object Constraint Language 2.4.
http://www.omg.org/spec/OCL/2.4/PDF.

- OMG. (2015). Unified Modeling Language Version 2.5.
http://www.omg.org/spec/UML/2.5/PDF.

- Philippe Ayrault, T. H. (2009). Development life-cycle of critical software under Focal.

- Roques, P. (2008). Les cahiers du programmeur UML 2.0 modéliser une application web. Eyrolles.

- Thérèse Hardin, P. F. (2016). FoCaLiZe :Tutorial and Reference Manual, version 0.9.1. CNAM/INRIA/LIP6.