

N° d'ordre : 04/2018-D/INF

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et de la Technologie Houari Boumedienne
Faculté d'Electronique et Informatique



Thèse présentée pour l'obtention du grade de Docteur en science

En : **Informatique**

Spécialité : **Programmation et Systèmes**

Par

ABBAS MESSAOUD

Thème

L'environnement FoCaLiZe au service d'UML/OCL

Soutenue publiquement, le 03 Novembre 2018, devant le jury composé de :

M. A. BELKHIR	Professeur à l'USTHB - LSI (Alger)	Président
M. C.B. BEN-YELLES	Professeur à l'USTHB/UGA - LCIS (Grenoble, France)	Directeur
M. R. RIOBOO	Professeur à l'ENSIE - SAMOVAR (Paris, France)	Co-directeur
M. M. AHMED-NACER	Professeur à l'USTHB - LSI (Alger)	Examinateur
Mme. K. BENATCHBA	Professeur à l'ESI - LMCS (Alger)	Examinatrice
M. M. MEZGHICHE	Professeur à l'UMBB - LIMOSE (Boumerdès)	Examinateur

Remerciements

Je remercie vivement le Professeur Abdelkader BELKHIR qui me fait l'honneur de présider ce jury et d'avoir bien voulu juger ce travail.

Je suis très honoré que les professeurs Mohamed AHMED-NACER, Karima BENATCHBA et Mohamed MEZGHICHE aient accepté d'examiner cette thèse, et les en remercie chaleureusement.

Des remerciements tous particuliers à mes directeurs de thèse, Choukri-Bey BEN-YELLES et Renaud RIOBOO pour leur encadrement et soutien tout au long de ce travail. Leurs conseils et encouragements constants m'ont été d'un grand apport. Qu'ils trouvent ici l'expression de ma profonde gratitude.

J'exprime aussi ma profonde reconnaissance et mes vifs remerciements à Karim BENABADJI (Qu'Allah lui fasse miséricorde), Chouhed BOUABANA et Lies KADDOURI pour m'avoir fait découvrir FoCaLiZe et intégré dans leur équipe au sein du laboratoire LSI.

Je remercie également les membres du projet FoCaLiZe au sein du laboratoire CEDRIC, en particulier les professeurs David DELAHAYE et Catherine DUBOIS, pour leurs conseils et leurs remarques surtout pendant le stage de 18 mois passé à l'ENSIIE.

J'adresse également mes sincères remerciements aux membres du laboratoire LCIS, en particulier le professeur Michel OCELLO pour m'avoir accueilli plusieurs fois avec des échanges toujours fructueux.

Enfin je tiens à remercier tous les collègues que j'ai côtoyés au département Informatique de l'USTHB, à l'université d'Ouargla, à l'ENSIIE puis dans mon université d'accueil d'El Oued.

Table des matières

Remerciements	1
Introduction générale	8
I Concepts et état de l’art	13
1 L’environnement FoCaLiZe	14
1.1 Niveau spécification : l’Espèce	15
1.1.1 Représentation	16
1.1.2 Fonctions	17
1.1.2.1 Signatures	17
1.1.2.2 Fonctions définies	18
1.1.3 Propriétés	18
1.1.3.1 Propriétés déclarées	18
1.1.3.2 Théorèmes	19
1.1.4 Héritage et liaison tardive	20
1.1.5 Espèce complète et interface :	22
1.2 Niveau Abstraction : la Collection	22
1.2.1 Collection	23
1.2.2 Paramétrage	23
1.2.2.1 Paramètres de collections	23
1.2.2.2 Paramètres d’entités	25
1.3 Preuves en FoCaLiZe	26
1.3.1 Utilisation de assumed	26
1.3.2 Preuves en FPL	27
1.3.3 Preuves par Coq	29
1.4 Compilation et mise en œuvre	30
1.5 Documentation	31
1.6 Méthode de développement en FoCaLiZe :	32
1.7 Conclusion	35
2 Sous-ensembles UML/OCL	36
2.1 Modèles UML/OCL	37
2.2 Diagramme de classes	40
2.2.1 Classe	40
2.2.1.1 Attributs (variables d’instances)	41

2.2.1.2	Opérations	41
2.2.1.3	Classe paramétrée ("template")	42
2.2.1.4	Énumérations et Intervalles	43
2.2.2	Relations de classes	44
2.2.2.1	Héritage	44
2.2.2.2	Dépendance	45
2.2.2.3	Génération des classes liées ("template binding")	45
2.2.2.4	Associations	46
2.3	Diagrammes d'états-transitions	48
2.3.1	État	48
2.3.2	Transition	48
2.4	Contraintes OCL	50
2.4.1	Invariant	50
2.4.2	Pré/Post Conditions	52
2.5	Étude de cas	52
2.5.1	Tableaux et piles finis	53
2.5.2	Contrôle d'un passage à niveau	54
2.6	Conclusion	57
3	Formalisation et vérification de modèles UML/OCL : Etat de l'art	58
3.1	Attribution d'une sémantique formelle pour UML	59
3.2	Transformation en méthodes formelles	60
3.2.1	Transformations vers B	61
3.2.2	Transformation vers Alloy	63
3.2.3	Traduction vers HOL	64
3.2.4	Transformation vers des méthodes basées model-checker	64
3.3	Vérification par génération de tests	66
3.4	Vérification par réseau de Petri	67
3.5	Transformation inverse : du formel vers UML	67
3.6	Comparaison et Conclusion	69
II	De UML/OCL vers FoCaLiZe	71
4	Transformation de diagrammes de classes	73
4.1	Attributs et opérations	74
4.1.1	Expressions de types	74
4.1.1.1	Types primitifs	74
4.1.1.2	Type d'énumérations	75
4.1.1.3	Type d'intervalles	76
4.1.1.4	Expressions de types en général	77
4.1.2	Attributs	77
4.1.3	Opérations	78
4.2	Classes sans relations (cas particulier)	79
4.3	Relations de classes	80
4.3.1	Paramètres de classes	81

4.3.2	Substitution de paramètres de classes	82
4.3.3	Héritage	83
4.3.4	Dépendances	84
4.3.5	Associations binaires	85
4.4	Classes avec relations (cas général)	88
4.5	Conclusion	90
5	Transformation des contraintes OCL	92
5.1	Expressions OCL sur les types primitifs	93
5.1.1	Expressions OCL sur le type Integer	93
5.1.2	Expressions OCL sur le type Real	93
5.1.3	Expressions OCL sur le type String	94
5.1.4	Expressions OCL sur le type Boolean	95
5.2	Expressions OCL de type Collection	96
5.3	Contraintes OCL	98
5.3.1	Invariant	98
5.3.2	Pré/post-conditions	99
5.3.3	Post-condition utilisant @pre.	100
5.4	Conclusion	101
6	Génération et vérification de code FoCaLiZe	102
6.1	Processus de vérification	102
6.2	Etude de cas	104
6.3	Déploiement	109
6.4	Conclusion	110
7	Transformation de diagrammes d'états-transitions	111
7.1	Transformation d'états	111
7.2	Transformation de transitions	112
7.3	Détection d'erreurs de diagrammes d'états-transitions	114
7.4	Conclusion	116
	Conclusion générale	117
	Bibliographie	120
	Annexes	127
A	Sources FoCaLiZe	127
B	Transformation des expressions OCL	130
B.1	Définition des opérations OCL sur le type Real	130
B.2	Définition des opérations OCL sur le type String	133
B.3	Définition des opérations OCL de collection	134
C	Espèces supports pour la transformation des associations	137

Table des figures

1.1	Exemple d'héritage en FoCaLiZe	21
1.2	Exemple d'une hiérarchie de développement en FoCaLiZe	25
1.3	Exemple d'un document HTML dérivé d'une source FoCaLiZe	32
1.4	Hiérarchie de développement du produit $\mathbb{Z}\mathbf{x}\mathbb{Z}$ en FoCaLiZe	33
2.1	Sous-ensemble UML	38
2.2	Sous-ensemble OCL	51
2.3	La classe <i>Person</i>	52
2.4	Diagramme de classes - Tableaux et piles finis -	53
2.5	Diagramme de classes - Contrôle d'un passage à niveau	55
2.6	Diagrammes d'états-transitions de classes Control , Barrier et Light	55
6.1	Processus de vérification des modèles UML/OCL par FoCaLiZe	103
6.2	Transformation systématique d'UML/OCL vers FoCaLiZe	110

Liste des tableaux

1.1	Syntaxe générale d’une espèce	16
1.2	Exemple de l’héritage en FoCaLiZe	20
2.1	Description de la classe <i>Point</i>	42
2.2	La classe paramétrée <i>Liste_finie</i>	42
2.3	Le type d’énumération <i>Day</i>	43
2.4	Le type d’intervalle <i>Floor</i>	43
2.5	Exemple de l’héritage simple	44
2.6	Exemple de l’héritage multiple	45
2.7	Exemple d’une relation de dépendance	45
2.8	Exemple d’une classe paramétrée et d’une relation de “binding” en UML	46
2.9	L’association <i>Review</i> entre les classes <i>Person</i> et <i>Paper</i>	47
2.10	Diagramme d’états-transitions de la classe <i>Person</i>	49
4.1	Convergences entre les concepts UML et les concepts FoCaLiZe	88
5.1	Transformation des expressions OCL sur le type Integer	93
5.2	Modélisation des opérations OCL sur le type Real	94
5.3	Modélisation des opérations OCL sur le type String	95
5.4	Transformation des formules OCL	96
5.5	Tansformation des opérations OCL retournant des collections	97

Introduction générale

Le développement des applications informatiques a commencé depuis l'invention de la machine de Von Neumann en 1945. Parallèlement aux évolutions technologiques, des dizaines de langages de programmation sont apparus. Nous trouvons des langages de programmation procédurale (de paradigme impératif) comme Fortran (depuis 1954), Pascal (depuis 1970) et C (depuis 1972). Pour éviter les effets de bord lors d'exécution de programmes et pour donner plus de rigueur aux produits logiciels, à la même époque, des langages de paradigmes fonctionnels comme Lisp (depuis 1960) et ML (depuis 1980) ont également été développés.

Puis (dans les années 80), le besoin de faciliter la conception, la maintenance et la stabilité de programmes a conduit à l'invention du paradigme orienté objet. Ce dernier considère les programmes en termes de classes d'objets regroupant des structures de données et des méthodes. Parmi les langages de paradigme objet citons C++ (depuis 1983) et Java (depuis 1995). Actuellement, la plupart des langages de programmation sont influencés par le paradigme objet.

Avec les développements industriels, les applications informatiques sont devenues très volumineuses (des millions des lignes de code), ce qui rend le contrôle et la maintenance des codes très difficile. La méthode classique d'écriture des codes (utilisée dans les années 70 et 80) est devenue insuffisante. En effet, construire une petite maison n'est pas comme la construction d'un grand bâtiment. Ce dernier a besoin d'une architecture et d'une conception détaillée avant d'aborder sa réalisation. Ainsi, plusieurs méthodes de planification et de conception des applications informatiques sont apparues au début des années 90. Les trois méthodes les plus répandues sont la méthode de Grady Booch [Boo91], la méthode OOSE [JCC94] de Ivar Jacobson, et la méthode OMT [RBL⁺90] de James Rumbaugh. En 1995, ces trois méthodes ont été unifiées dans un seul standard appelé UML (Unified Modeling Language).

Le langage UML [OMG15a] 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 servent à la modélisation de l'aspect dynamique. Comme successeur des notations antérieures, UML est rapidement devenu un standard pour l'OMG¹ (Object Management Group). Il jouit d'une grande popularité, à la fois dans l'industrie du logiciel et dans les milieux académiques. Cette popularité est renforcée par un nombre important d'outils, adoptant les notations UML dans les ateliers de génie logiciel.

L'OCL (Object Constraint Language) [OMG14] est un langage formel de la famille OMG. Il permet de spécifier les éléments des diagrammes UML par des contraintes (exigences). En effet, le langage OCL a remplacé le langage naturel (non formel) utilisé au début avec UML pour annoter les éléments des diagrammes.

¹<http://www.omg.org/>

L'implémentation d'un modèle UML/OCL nécessite sa traduction en un langage de programmation de paradigme objet. Ainsi, pour faciliter la génération des codes exécutables à partir des modèles UML/OCL, plusieurs techniques MDE (Model Driven Engineering) ont été développées. Les techniques MDE permettent le développement des applications par raffinements successifs des modèles, en partant des niveaux les plus abstraits jusqu'aux implantations concrètes. Actuellement, UML et OCL sont supportés par une large variété d'outils MDE dans les ateliers de génie logiciel.

Malgré les progrès enregistrés avec UML dans les outils MDE, il reste néanmoins un outil auquel il manque les bases formelles nécessaires pour faire de la vérification [LSC03, PL99]. En effet, les langages UML et OCL permettent uniquement la description d'un système et la spécification des contraintes. Ils ne disposent pas de moyens pour découvrir les défauts de modélisation ou pour vérifier la satisfaction des contraintes OCL. Ainsi, lorsqu'il s'agit d'applications critiques (nécessitant un degré élevé de sécurité), il est pertinent d'utiliser (conjointement avec UML/OCL) des techniques et des mécanismes formels permettant l'analyse et la vérification des modèles. Parmi les outils les plus utilisés dans la formalisation de modèles UML/OCL, figurent les méthodes formelles. Ces dernières fournissent des notations mathématiques permettant de spécifier très précisément et rigoureusement les propriétés du système construit. Le modèle est alors moins intuitif par rapport à UML, mais il peut être validé grâce à des techniques de preuve formelle. Le but des méthodes formelles consiste à guider le programmeur pour l'amener à donner, puis prouver les propriétés nécessaires à la cohérence de son modèle.

Dans ce contexte, plusieurs travaux ont opté pour la proposition d'outils MDE en combinant des modèles UML/OCL et des méthodes formelles. Les méthodes formelles les plus utilisées sont : la méthode B [Abr96], Alloy [Jac04], Maude [CDE⁺07], Isabelle/HOL [NPW02]. Des transformations puissantes et fiables ont été proposées, via ces méthodes formelles. Elles supportent plusieurs aspects statiques et dynamiques des diagrammes UML et des contraintes OCL. Cependant, plusieurs fonctionnalités UML/OCL clés sont ignorées dans les formalisations proposées. En particulier, Les fonctionnalités UML/OCL suivantes : l'héritage multiple, la redéfinition de méthodes, la liaison tardive, les templates d'UML (classes paramétrées) et la dérivation des classes liées à partir des templates (via la substitution de paramètres formels par des paramètres effectifs) et la propagation des contraintes OCL (à travers les fonctionnalités UML), ne sont pas supportées. En effet, ces fonctionnalités UML/OCL sont indispensables (au niveau des méthodes formelles) pour préserver les mêmes sémantiques et raisonnements d'UML/OCL dans les spécifications formelles dérivées.

Dans le cadre de cette thèse, nous envisageons une formalisation de modèles UML/OCL supportant un certain nombre des fonctionnalités UML/OCL ignorées dans les travaux de formalisation cités ci-dessus, en utilisant FoCaLiZe [HFPD16]. Ce dernier est une méthode formelle, initiée à la fin des années 90 par Thérèse Hardin et Renaud Rioboo au sein des laboratoires LIP6, CÉDRIC et INRIA. FoCaLiZe, issue de l'assistant d'aide à la preuve Coq [Coq16], est un environnement complet de développement comportant un langage de programmation (fonctionnel), un langage de spécification des besoins et un langage de preuve. L'environnement FoCaLiZe partage avec UML/OCL la plupart des fonctionnalités architecturales et conceptuelles et prend en charge la majorité des exigences imposées par les normes sur l'évaluation du cycle de vie de développement de logiciels [AHP09a, DÉDG08a, Fec05].

Bien que la correspondance sémantique entre les éléments des diagrammes de classes UML et FoCaLiZe ait déjà été analysée par David Delahaye et al. [DÉDG08b], la transformation d'UML/OCL vers FoCaLiZe n'a pas été encore traitée. Pour atteindre cet objectif, nous avons opté pour une approche directe (transformationnelle), en définissant un ensemble de diagrammes d'UML2 supportés par notre transformation et un sous ensemble des contraintes OCL. Les diagrammes UML supportés sont le diagramme de classes et le diagramme d'états-transitions, en considérant la plupart des fonctionnalités UML/OCL ignorées dans les travaux de formalisation précédents. Plus précisément, nous souhaitons supporter les fonctionnalités UML/OCL suivantes :

- Héritage (multiple) avec redéfinition d'opérations et liaison tardive ;
- Dépendance (multiple) ;
- Classes paramétrées par d'autres classes ;
- Substitutions de paramètres formels de classes (paramétrées) par des paramètres effectifs pour générer des modèles liés ("template binding") ;
- Associations binaires ;
- Les contraintes OCL de type invariants de classes et pré/post-conditions des opérations de classes ;
- L'héritage des contraintes OCL : les propriétés d'une super-classe sont aussi propriétés de ses héritières (sous-classes) ;
- La propagation des contraintes OCL à travers la spécialisation des classes paramétrées : les propriétés d'une classe paramétrée (UML template) sont aussi propriétés des classes dérivées par substitution des paramètres formels (de classes paramétrées) par des paramètres effectifs ;
- La propagation des contraintes OCL à travers le mécanisme de dépendance : les propriétés d'une classe fournisseur sont préservées au niveau des classes clientes ;
- L'utilisation du langage OCL au sein des diagrammes d'états-transitions pour spécifier des gardes de transitions ;
- La communication entre l'aspect statique (diagrammes de classes) et l'aspect dynamique (diagrammes d'états-transitions) dans un modèle UML.

Ainsi, nous allons étudier la sémantique des éléments du diagramme de classes UML, du diagramme d'états-transitions, des contraintes OCL et des spécifications FoCaLiZe ; puis nous définissons les règles de transformations d'UML/OCL vers FoCaLiZe.

La transformation proposée (d'UML/OCL vers FoCaLiZe) pourrait être exploitée comme un outil de vérification des modèles UML/OCL comme elle peut être utilisée pour la génération du code exécutable dans un contexte MDE. Côté vérification, la transformation d'UML/OCL vers FoCaLiZe nous permettra de déterminer les défauts de modélisation (dès les premières phases de développement), d'établir la preuve des contraintes OCL et de réaliser la vérification des contraintes des associations et diagrammes d'états-transitions. La génération du code exécutable est réalisée 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 atteindre l'étape ultime, la génération d'un exécutable (OCaml) certifié.

Plan du mémoire

Ce mémoire est composé de deux parties et d'une annexe :

- La première présente les constituants de base de cette thèse, à savoir, une introduction à l'environnement FoCaLiZe et une description syntaxique des éléments UML/OCL concernés par notre démarche de transformation. Un état de l'art sur les transformations d'UML/OC vers des techniques formelles est également présenté ;
- La seconde partie présente notre démarche de transformation d'UML/OCL vers FoCaLiZe. Les schémas de dérivation de diagrammes de classes et de diagrammes d'états-transitions vers FoCaLiZe ainsi que les schémas de transformation de contraintes OCL vers FoCaLiZe sont détaillés dans cette deuxième partie ;
- Au niveau de l'annexe, nous détaillons des codes FoCaLiZe et des règles de transformation brièvement présentés au cours de ce document.

Voici un bref survol des chapitres de ce document :

Dans le **chapitre 1**, 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. Pour plus de détails sur les concepts présentés dans ce chapitre, le lecteur peut faire référence au manuel de référence de FoCaLiZe disponible sur son site officiel².

Dans le **chapitre 2**, nous introduisons les diagrammes UML et les expressions OCL supportés par notre démarche de transformation. En particulier, nous détaillons la syntaxe des sous-ensembles des diagrammes de classes, diagrammes d'états-transitions et des contraintes OCL utilisés. Le chapitre se termine par la présentation d'études de cas (modèles UML/OCL) concrets.

Dans le **chapitre 3**, nous étudions les différentes techniques utilisées pour la formalisation et la vérification d'une modélisation UML/OCL. Plusieurs techniques de formalisation, comme la génération de tests, les réseaux de Petri, les model-checkers et les transformations en méthodes formelles sont discutées. Nous donnons plus de détails sur l'approche transformationnelle, adoptée dans ce travail de thèse.

Dans le **chapitre 4**, nous présentons notre 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 paramétrisation, la dérivation de classes liées à partir d'une classe paramétrée ("template binding"), le mécanisme de dépendance et les associations binaires. Nous concluons enfin sur la transformation des classes en général.

Le **chapitre 5** est consacré aux transformations des contraintes OCL (invariant et pré/post-conditions). Avant de traiter la transformation des contraintes OCL, nous détaillons la correspondance entre les opérations OCL et les opérations FoCaLiZe. Ceci nous a amenés à créer des bibliothèques FoCaLiZe définissant toutes les opérations OCL non fournies par FoCaLiZe.

²Manuel de référence de FoCaLiZe : <http://focalize.inria.fr>

Le **chapitre 6** présente un processus général pour la preuve des propriétés FoCaLiZe dérivées d'un modèle UML/OCL, la complétion du code FoCaLiZe, jusqu'à la génération d'un exécutable. Nous détaillons aussi les techniques utilisées pour détecter et corriger les erreurs d'un modèle UML/OCL. Le chapitre se termine par la description de la méthode et de l'environnement de programmation utilisés pour implémenter notre modèle de transformation d'UML/OCL en FoCaLiZe.

Dans le **chapitre 7**, nous exploitons toutes les techniques et les règles de transformations développées dans les chapitres précédents pour établir les règles de transformation d'un diagramme d'états-transitions. Le chapitre se termine par la présentation d'une démarche pour la détection d'éventuelles contradictions dans le diagramme d'états-transitions, vis-à-vis du diagramme de classes.

Première partie

Concepts et état de l'art

Chapitre 1

L’environnement FoCaLiZe

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.

Selon les mécanismes de vérifications utilisés, nous distinguons deux grandes familles de méthodes formelles : Les méthodes formelles utilisant la génération de tests et le Model-Checking et les méthodes formelles utilisant des assistants d’aide à la preuve (ou prouveurs). L’avantage de l’approche basée Model-Checking est qu’elle est rapide et entièrement automatique, surtout pour les cas de systèmes simples (possédant un nombre limité d’états). Par contre, pour les systèmes complexes, l’approche basée sur les assistants de preuves est plus efficace, mais nécessite l’interaction humaine. Côté programmation, les méthodes formelles sont basées sur des langages de paradigme fonctionnel et/ou impératif. Cependant, l’utilisation des mathématiques et de la logique imposent souvent des restrictions sur l’aspect conception et architecture de softwares.

Au cours des dernières décennies, plusieurs méthodes formelles ont vu le jour et sont utilisées avec succès pour le développement et la vérification d’applications critiques. Citons entre autres des méthodes formelles utilisant un assistant de preuve comme la méthode B [Abr96] (utilisée dans la protection automatique de système de trains en France [BBFM99]), Alloy [Jac04], CASL [Mos04] et HOL-OCL [BW07].

D’autres méthodes formelles utilisent le Model-Checking, comme SMV [CCGR00] et le système Maude [CDE⁺07] (utilisé dans l’analyse et la vérification de systèmes distribués et de protocoles de communications [DMT00]).

Contrairement aux méthodes formelles susmentionnées, l’environnement FoCaLiZe [HFPD16] se distingue tant au niveau preuve qu’au niveau conception et architecture. Côté preuve, en plus de l’assistant de preuve Coq, FoCaLiZe intègre le prouveur automatique de théorèmes Zenon [Dol15] pour minimiser l’interaction du développeur et utilise un générateur de tests pour permettre la détection rapide de contre-exemples (surtout dans les cas de systèmes simples). Côté conception, malgré son paradigme purement fonctionnel, FoCaLiZe supporte d’une manière naturelle la plupart

des traits orientés objets comme l'encapsulation de données, l'héritage multiple, la liaison tardive, la dépendance multiple etc.

L'environnement FoCaLiZe permet le développement des applications étape par étape (approche incrémentale), allant des spécifications abstraites, appelées **espèces**, jusqu'aux implémentations concrètes, appelées **collections**. Le développeur enrichit et affine progressivement ses spécifications jusqu'à atteindre un niveau qui peut être implémenté par le compilateur FoCaLiZe. Ce dernier produit du code OCaml pour la partie exécutable et produit du code Coq pour les aspects preuves.

Une première étude de cas par FoCaLiZe était le développement d'une librairie certifiée pour le calcul algébrique (délivrée avec le compilateur) par R. Rioboo [HR04]. Cette librairie a vu un grand succès par rapport aux systèmes de calcul formel existants. Par ailleurs, parmi les principales utilisations de l'environnement FoCaLiZe, on peut citer les contributions suivantes :

- La formalisation et l'implémentation d'une politique de sécurité (Bell-Lapadula) par M. Jaume et C. Morisset [JM05] ;
- La modélisation des règles de sécurité des aéroports par D. Delahaye et J.F. Etienne [DÉ08] ;
- Le développement d'un voteur générique par P. Ayrault, T. Hardin, et F. Pessaux [AHP09b] ;
- L'élaboration d'un cadre de développement de systèmes sécurisés en combinant programmes, spécifications et preuves dans un environnement de programmation orienté-objet [DJR12], par D. Damien, M. Jaume et R. Rioboo ;
- La proposition d'une approche pour la preuve des terminaisons des fonctions récursives en FoCaLiZe [DP15], par Catherine Dubois et François Pessaux.

Dans le présent chapitre, nous allons détailler les différents concepts de l'environnement FoCaLiZe. Nous commençons d'abord par l'aspect spécification, puis nous décrivons l'aspect implémentation et l'aspect preuve.

1.1 Niveau spécification : l'Espèce

Une spécification FoCaLiZe se présente comme une hiérarchie d'espèces. Une espèce est une structure qui regroupe un ensemble de méthodes, de trois types : la représentation, les fonctions calculatoires et les propriétés (contraintes à satisfaire). Chaque méthode peut être uniquement déclarée (statut abstrait) avant de lui fournir une définition complète (statut concret) au cours du développement, dans une espèce qui en héritera.

Dans ce qui suit, nous allons détailler les différents concepts liés à la spécification d'une espèce, et dont la syntaxe générale est présentée dans la table suivante (table 1.1) :

TAB. 1.1 – Syntaxe générale d'une espèce

<code>species <i>species_name</i> =</code>	
<code>[representation = <i>rep_type</i>];</code>	<code>(* représentation *)</code>
<code>signature <i>function_name</i> : <i>function_type</i>;</code>	<code>(* fonction déclarée *)</code>
<code>[local / logical] let [rec] <i>function_name</i> = <i>function_body</i>;</code>	<code>(* fonction définie *)</code>
<code>property <i>property_name</i> : <i>property_specification</i> ;</code>	<code>(* propriété déclarée *)</code>
<code>theorem <i>theorem_name</i> : <i>theorem_specification</i></code>	<code>(* propriété définie *)</code>
<code> proof = <i>theorem_proof</i> ;</code>	
<code>end ;;</code>	

Tout au long de ce chapitre, les concepts de FoCaLiZe seront illustrés à travers un exemple théorique modélisant les **points**, les **points colorés** et les **cercles** dans le plan. Un point est défini par ses coordonnées, abscisse et ordonnée (de types réels). Un point coloré est un point doté d'une couleur (rouge, vert ou bleu). Un cercle est défini par son centre (un point) et par son diamètre (dont la longueur est aussi de type réel).

1.1.1 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, aussi appelé type support, défini par une expression de type en FoCaLiZe : variables de types, types primitifs, types inductifs, enregistrements, etc.

Pour commencer avec notre 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 :

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

La représentation d'une espèce peut éventuellement 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.

Le fait que la représentation d'une espèce est définie par un type unique, ne représente pas une limitation. De même que dans les langages de programmation orientée objet, une classe peut avoir plusieurs variables d'instances de types différents, dans une espèce, la représentation peut être définie par un n-uplet regroupant plusieurs types différents, comme elle peut être définie par un type structuré ("record type") composé de plusieurs champs de types différents.

Entité d'espèce. Une entité d'espèce est un élément de type représentation de l'espèce, créé par une fonction (constructeur) de l'espèce. A titre d'exemple, la paire (2.5, 10.0) est une entité de l'espèce `Point`.

1.1.2 Fonctions

Les fonctions d'une espèce représentent les aspects calculatoire et dynamique de l'espèce. Elles permettent de manipuler les entités d'une espèce.

Le style de développement en FoCaLiZe permet deux niveaux de spécification de fonctions, le niveau déclaration (abstrait) et le niveau définition (concret). Une fonction déclarée (**signature**) sert à spécifier une fonction sans rentrer dans les détails de son implémentation. Une fonction définie (**let**) est une fonction complètement définie via son corps calculatoire.

Les paragraphes suivants donnent plus de précision sur les deux catégories de fonctions.

1.1.2.1 Signatures

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.

Ceci permet, d'un côté, une progression incrémentale lors de la conception des espèces, et d'un autre côté, plusieurs choix pour les futures implémentations d'une même signature. Ultimement, toute signature devra recevoir au moins un corps calculatoire qui doit correspondre à son type fonctionnel.

Nous complétons notre espèce `Point` par la signature de la fonction constructeur `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`) dénote un élément (entité) de type représentation de l'espèce en cours de spécification, même si sa représentation 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.

1.1.2.2 Fonctions définies

Une fonction définie est une fonction énoncée conjointement avec son corps calculatoire. 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`), les fonctions d'ordre supérieur, etc.

Une fonction définie est soit nouvellement énoncée, soit 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. La définition d'une fonction peut faire appel à n'importe quelle autre fonction reconnue dans le contexte de l'espèce courante. En particulier, elle peut faire appel à toutes les fonctions définies et signatures de l'espèce courante.

Le type d'une fonction définie peut être non spécifié, en effet, FoCaLiZe est capable d'inférer ce type dans le contexte de l'espèce, comme dans le cas de la fonction `different` de l'exemple.

FoCaLiZe permet aussi la définition des fonctions récursives en utilisant `let rec` (voir syntaxe, table 1.1). Dans ce cas, une preuve de terminaison devra être produite au cours du développement.

1.1.3 Propriétés

Les propriétés sont des contraintes à satisfaire par les entités de l'espèce. Elles précisent en général des restrictions sur les entités de l'espèce. Comme pour les fonctions d'une 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). Nous détaillons les deux types de propriétés dans les paragraphes suivants.

1.1.3.1 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 en l'absence 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 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 ($\forall$) et le symbole `->` désigne le connecteur d'implication.

Une propriété déclarée peut être utilisée ensuite pour définir d'autres propriétés comme elle peut être utilisée comme hypothèse pour prouver d'autres propriétés.

1.1.3.2 Théorèmes

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

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.

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, la preuve de théorème `egal_est_non_different` $((a = b) \Rightarrow \neg(a \neq b))$ 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;;
```

Une présentation plus détaillée sur les preuves en FoCaLiZe sera donnée en section 1.3.

1.1.4 Héritage et liaison tardive

Grâce aux concepts présentés ci-dessus (représentation, fonctions et propriétés), des espèces peuvent être créées à partir de rien (entièrement nouvelles), mais on peut aussi utiliser le mécanisme d'héritage pour créer de nouvelles espèces en utilisant des espèces existantes. La nouvelle espèce créée par héritage acquiert toutes les méthodes de ses super-espèces (y compris la représentation). En fournissant des corps calculatoires aux signatures héritées et des preuves formelles aux propriétés héritées, on procède à un raffinement progressif des espèces en concrétisant au fur et à mesure leurs méthodes. De plus, la nouvelle espèce peut introduire des fonctions et propriétés propres, comme elle peut redéfinir des fonctions et propriétés des super-espèces.

Le mécanisme d'héritage enrichit la syntaxe générale d'une espèce (voir table 1.1) comme suit :

```
species species_name = inherit species_name1, ... species_namek; ... end;;
```

où *species_name*_{*i*}, *i* : 1..*k* désignent les noms des espèces héritées.

Dans l'exemple proposé ci-dessous (voir table 1.2), les symboles **+f**, **-f**, ***f** et **/f** désignent respectivement les opérateurs d'addition, de soustraction, de multiplication et de division réelles (définies dans l'annexe B.3). Le **sqrt** est la fonction qui calcule la racine carrée d'un nombre réel positif. Le type **couleur** (défini au niveau top) modélise la couleur d'un point graphique.

TAB. 1.2 – Exemple de l'héritage en FoCaLiZe

```
type couleur = | Rouge | Vert | Bleu ;;

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

  let affiche_point (p:Self):string = (" X = " ^ string_of_float(get_x(p)) ^
                                       " Y = " ^ string_of_float(get_y(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) ) );

end;;

species PointConcret = inherit PointAbstrait;
representation = float * float;
let creerPointConcret(x:float, y:float):Self = (x , y);
let get_x(p) = fst(p);
let get_y(p) = snd(p);
let deplacer(p, dx, dy) = ( get_x(p) +f dx, get_y(p) +f dy);
end;;
```

```

species PointColore = inherit PointAbstrait;
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) =
  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;;

```

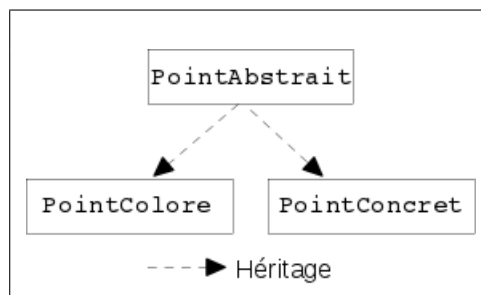


FIG. 1.1 – Exemple d'héritage en FoCaLiZe

Cet exemple présente deux cas d'utilisation de l'héritage :

- L'espèce **PointConcret** est créée par héritage de l'espèce **PointAbstrait**. Dans ce cas, l'héritage est uniquement utilisé pour affiner l'espèce **PointAbstrait**. L'espèce **PointConcret** fournit des définitions concrètes pour la représentation ($\text{float} * \text{float}$) et pour les fonctions **get_x**, **get_y** et **deplacer**, qui étaient uniquement déclarées. Elle introduit aussi la fonction **creerPointConcret** pour servir en tant que constructeur de l'espèce. Les fonctions **affiche_point** (affiche les coordonnées d'un point) et **distance** (calcule la distance entre deux points) sont héritées telles quelles.
- Pour manipuler des points colorés (points graphiques), une nouvelle espèce **PointColore** est aussi créée en héritant de l'espèce **PointAbstrait**. L'espèce **PointColore** a besoin de manipuler la couleur d'un point en plus de ses coordonnées. Par conséquent, elle hérite de toutes les méthodes de l'espèce **PointAbstrait** et apporte les enrichissements suivants : Elle définit la représentation par l'expression $(\text{float} * \text{float}) * \text{couleur}$, 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 redéfinit aussi la fonction **affiche_point**, (déjà définie dans l'espèce

`PointAbstrait`) pour permettre l'affichage de la couleur du point en plus de ses coordonnées.

Notre première vision sur l'espèce `PointAbstrait` est de modéliser les points du plan sans considérer d'autres arguments comme la couleur d'un point. Cependant, le mécanisme d'héritage en FoCaLiZe nous permet d'un côté d'utiliser notre espèce `PointAbstrait` telle quelle (en concrétisant ses méthodes dans l'espèce `PointConcret`), et d'un autre côté d'enrichir l'espèce `PointAbstrait` pour considérer la couleur d'un point en plus de ses coordonnées. Ceci nous amène à introduire de nouvelles fonctions et à en redéfinir certaines (dans l'espèce `PointColore`).

FoCaLiZe permet aussi l'héritage multiple. C'est seulement par souci de simplicité que les exemples ci-dessus utilisent uniquement l'héritage simple. Lors de l'héritage multiple, si une même méthode figure dans plus d'une espèce héritée, c'est l'ordre d'apparition des espèces dans la clause d'héritage (`inherit`) qui détermine la méthode choisie. En effet, seulement la dernière définition d'une méthode (apparaissant plusieurs fois dans la liste des espèces héritées) est conservée.

Deux contraintes doivent être respectées lors de l'héritage :

- Ne pas redéfinir la représentation (le type support). Elle peut demeurer abstraite pendant l'héritage, mais elle ne peut être définie qu'une et une seule fois au cours du développement.
- Les méthodes héritées, et/ou redéfinies ne doivent jamais changer de type. Cela est nécessaire pour assurer la cohérence du modèle.

1.1.5 Espèce complète et interface :

Après sa spécification, une espèce peut être utilisée comme une variable de type lors de la conception d'autres espèces du même modèle. Le type d'une espèce est obtenu en ignorant toutes ses définitions et preuves. C'est une sorte de type d'enregistrement composé des types de toutes les méthodes de l'espèce. L'interface d'une espèce est obtenue en faisant abstraction de la représentation dans le type de l'espèce.

Dans FoCaLiZe, il n'y a pas de construction spéciale pour dénoter l'interface d'une espèce. L'interface d'une espèce est simplement désignée par le nom de l'espèce.

En utilisant le mécanisme d'héritage, les espèces d'un modèle sont raffinées et enrichies, au fur et à mesure en descendant dans la hiérarchie. Ultimement, ce processus aboutira à un niveau où toutes les méthodes des espèces sont définies, les représentations explicitées par des expressions de types, les fonctions ont reçu leurs corps calculatoires et toutes les propriétés ont reçu leurs preuves formelles. De telles espèces sont appelées espèces **complètes** (ou terminales). Dans notre exemple (de la table 1.2), les espèces `PointConcret` et `PointColor` sont complètes.

1.2 Niveau Abstraction : la Collection

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**, comme nous allons l'expliquer dans la section suivante.

1.2.1 Collection

Une collection est obtenue à partir d'une espèce complète par abstraction de son type support. Elle définit un type abstrait uniquement manipulable via l'interface de l'espèce complète sous-jacente, avec la garantie que toutes les fonctions sont définies et que toutes les propriétés sont prouvées.

La syntaxe utilisée pour définir une collection (*collection_name*) à partir d'une espèce complète (*species_name*) est donnée comme suit :

```
collection collection_name = implement species_name ; end;;
```

Nous implémentons les espèces complètes **PointColore** et **PointConcret** présentées ci-dessus, puis créons une entité de l'espèce **PointColore** comme suit :

```
collection PointColoreCollection = implement PointColore; end;;
collection PointConcretCollection = implement PointConcret; end;;
let p = PointColoreCollection!creerPointColore(2.0, 5.0, Bleu);;
print_string(PointColoreCollection!affiche_point(p));;
```

Dans cet exemple, l'entité **p** est créée en appelant la fonction **creerPointColore** (constructeur) de la collection **PointColoreCollection**. La notation **" ! "** est équivalente à la notation **" . "** d'envoi de message en programmation orientée objet. La fonction **print_string** permet d'imprimer un texte à l'écran.

1.2.2 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 (la cliente) d'utiliser les méthodes et les entités d'autres espèces (les fournisseurs). En comparaison avec le paradigme objet, le paramétrage décrit un lien de dépendance entre les espèces.

Suivant ce principe, FoCaLiZe permet deux formes de paramétrage :

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

Ces deux formes de paramétrage sont développées dans les paragraphes suivants.

1.2.2.1 Paramètres de collections

Un paramètre de collection est une future collection d'une espèce (fournisseur) que les espèces d'hébergement (clientes) peuvent l'utiliser par l'intermédiaire de son interface pour définir leurs propres méthodes.

Un paramètre de collection est introduit par un nom (*parameter_name*) et une interface désignée 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) = fst(c);
  let get_rayon(c:Self) = 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;;
```

La preuve du théorème `appartient_spec` sera détaillée dans la section 1.3.

Pour créer une collection à partir d'une espèce paramétrée par paramètres de collections, chaque paramètre devra être substitué par le nom d'une collection effective du modèle. C'est pour cette raison qu'ils sont appelés paramètres de collections.

A titre d'exemple, nous pouvons créer les deux collections suivantes à partir de l'espèce complète **Cercle** :

```
collection CercleCollection1 = implement Cercle(PointConcretCollection); end;;
collection CercleCollection2 = implement Cercle(PointColoreCollection); end;;
```

Pour générer la collection `CercleCollection1`, nous avons substitué le paramètre de l'espèce **Cercle** par la collection `PointConcretCollection`, créée à partir de l'espèce **PointConcret** (voir section 1.2.1). Pour la deuxième collection, le paramètre de l'espèce **Cercle** est substitué par la collection `PointColoreCollection`, issue de l'espèce **PointColore**.

La figure suivante (figure 1.2), résume le schéma de l'héritage, de paramétrage et d'abstraction de nos espèces **PointAbstrait**, **PointColore**, **PointConcret** et **Cercle**.

Le code complet de l'exemple des espèces **PointAbstrait**, **PointConcret**, **PointColore** et **Cercle** est donné dans l'annexe A.

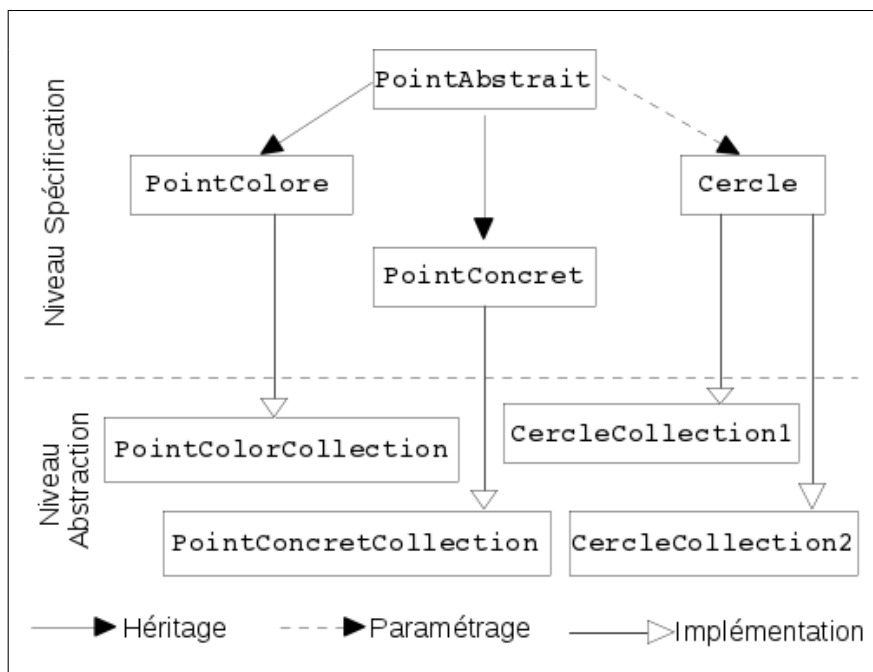


FIG. 1.2 – Exemple d’une hiérarchie de développement en FoCaLiZe

1.2.2.2 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, ...) ... end;;
```

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;;
```

Pour créer une collection `RepereOrthogonalZero` à partir de l'espèce `RepereOrthogonal`, nous avons substitué le paramètre d'entité `origine` par l'entité `o = (0.0, 0.0)` de la collection `PointConcretCollection`.

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.

1.3 Preuves en FoCaLiZe

Prouver une propriété consiste à s'assurer que son énoncé est satisfait dans le contexte de l'espèce courante. FoCaLiZe distingue entre les propriétés déclarées et les théorèmes. La preuve d'un théorème doit accompagner son énoncé. Par contre, la preuve d'une propriété déclarée (**property**) n'est pas intégrée à l'espèce.

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

```
theorem theorem_name :
  theorem_specification      (*      cas d'un théorème *)
  proof = proof ;

proof of property_name = proof ; (*      cas d'une propriété déjà déclarée      *)
```

Nous rappelons qu'une preuve en FoCaLiZe peut être introduite de trois manières différentes (voir section 1.1.3.2, page 19) :

- Soit en utilisant **assumed**, pour accepter une propriété sans fournir sa preuve ;
- Soit en utilisant un script de preuve directement écrit en Coq ;
- Soit en utilisant le langage de preuve de FoCaLiZe (FPL) pour écrire une preuve vérifiable par Zenon.

Dans la présente section, nous détaillons les trois méthodes de preuves.

1.3.1 Utilisation de **assumed**

Le mot clé **assumed** sert à accepter une propriété comme un axiome. La syntaxe utilisée pour accepter une propriété par **assumed** est :

```
theorem theorem_name :
  theorem_specification      (*      cas d'un théorème *)
  proof = assumed;

proof of property_name = assumed; (*      cas d'une propriété déjà déclarée      *)
```

La propriété `egal_symetrique` de l'espèce `Point` (voir section 1.1.3), est une propriété bien connue en mathématique. Pour l'admettre sans fournir sa preuve formelle, on procède comme suit :

```
species Point =
  proof of  egal_symetrique = assumed ;
  ...
end;;
```

Un développement certifié ne doit pas permettre une utilisation non contrôlée de **assumed**, car son utilisation ignore la vérification formelle du modèle. Cependant, son utilisation reste utile pour plusieurs raisons. Par exemple, dans le cas où le développeur ne sait pas encore comment prouver une propriété ou n'a pas suffisamment de temps pour écrire la preuve, ou bien s'il n'est pas intéressé par la preuve, et cherche à compiler son programme pour aboutir à un exécutable. Le mot clé **assumed** peut être utilisé aussi pour accepter une propriétés bien connue des mathématiques ou de la logique, et dont la preuve est irréalisable par machine. Par exemple, dans une machine, le fait qu'un entier est encodé par un mot mémoire de taille finie empêche une preuve formelle de la propriété :

$$\forall x \in \mathbb{N}, x + 1 > x.$$

Cependant, en mathématique, cette propriété est valide et peut être utile pour spécifier et prouver d'autres propriétés. Dans une application critique, il faut utiliser d'autres techniques telles que la génération de tests (fournie avec FoCaLiZe) pour faire confiance à une propriété acceptée par **assumed**.

1.3.2 Preuves en FPL

Une preuve écrite en FPL (FoCaLiZe Proof Language) est une suite d'indications de preuve ("facts") à utiliser par le prouveur automatique de théorèmes (Zenon) pour mener la preuve à son terme.

La syntaxe générale d'une preuve écrite en FPL est donnée comme suit :

$$\text{proof} = \{proof_step\}^* qed_step \mid \text{by } \{fact\}^+ \mid \text{conclude}$$

Une preuve écrite en FPL peut être simple (preuve feuille) ou composée de plusieurs étapes.

Une preuve feuille sert à prouver le but principal d'une propriété en utilisant une seule instruction **by** qui fournit des indications de preuve ou utilisant directement l'instruction **conclude** pour chercher la preuve sans fournir de faits.

Lors de la compilation, c'est Zenon qui s'occupe de la réalisation de la preuve en utilisant les indications fournies après le mot clé **by**.

Reprenons la propriété **egal_est_non_different** de l'espèce **Point** (voir section 1.1.3.2). La preuve de cette propriétés se présente comme suit :

```
species Point =
  ...
  let different(a:Self, b:Self):bool = ~(egal(a, b));
```

```

theorem egal_est_non_different : all x y:Self,
  egal(x, y) -> ~(different(x, y))
  proof = by definition of different;
  ...
end;;

```

Dans cet exemple, Zenon va déplier la définition de la fonction `different` dans la preuve, pour prouver le but.

D'autres faits peuvent être utilisés après le mot clé **by**, comme :

- **by theorem_name / property_name / hypothesis_name** : Zenon utilise l'hypothèse, le théorème, ou la propriété comme prémisses pour prouver le but.
- **by type_name** : Zenon utilise la définition d'un type pour prouver le but.

Le mot clé **conclude** sert à invoquer Zenon sans lui donner des faits ou des hypothèses. Dans ce cas, Zenon utilise ses propres connaissances pour prouver le but.

La propriété du premier ordre : $\forall a, b : \text{boolean}, a \Rightarrow b \Rightarrow a$.

peut être énoncée et prouvée en utilisant directement **conclude** comme suit :

```

theorem implications : all a b : bool, a -> (b -> a)
proof = conclude;;

```

Contrairement à une preuve feuille, une preuve composée contient plusieurs étapes. Lorsqu'il s'agit d'un théorème complexe, on peut décomposer le but du théorème en plusieurs sous-buts. Chaque sous-but est traité dans une étape particulière. La preuve est alors composée d'une séquence d'étapes qui se terminent par l'étape **qed** ou **conclude**. Le sous-but de chaque étape est énoncé dans l'étape même, à l'exception de l'étape **qed**. Le but de l'étape **qed** est celui du théorème.

Chaque étape de la preuve commence par un numéro qui indique son niveau, éventuellement suivi d'une hypothèse (en utilisant le mot clé **hypothesis** ou **assume**), du sous-but (qui suit le mot clé **prove**) et des indications de preuve (en utilisant **by**). Le but à prouver dans une étape peut à son tour être décomposé en plusieurs sous-buts, ce qui donne une structure arborescente aux étapes d'une preuve.

En plus des indications utilisées dans les preuves feuilles (**by property_name / hypothesis_name / type_name, ...**), une preuve composée peut aussi utiliser l'indication :

by step step_bulle,

où *step_bulle* désigne le numéro d'une étape précédente de la preuve. En particulier, l'indication **by step** permet à Zenon d'utiliser le but prouvé dans l'étape indiquée comme hypothèse pour prouver le sous-but de l'étape courante.

La preuve du théorème `appartient_spec` (voir section 1.2.2) de l'espèce `Cercle` peut être réalisée comme suit :

```

species Cercle (P is PointAbstrait) =
...
theorem appartient_spec : all c:Self, all p:P,
  appartient(p, c) <-> (P!distance(p, get_centre(c)) = get_rayon(c))
  proof =
    <1>1 assume c : Self, p : P,
      prove appartient(p, c) <-> (P!distance(p, get_centre(c)) = get_rayon(c))

    <2>1 hypothesis h1 : appartient(p, c),
      prove appartient(p, c) -> (P!distance(p, get_centre(c)) = get_rayon(c))
      by hypothesis h1 definition of appartient
    <2>2 hypothesis h2 : (P!distance(p, get_centre(c)) = get_rayon(c)),
      prove (P!distance(p, get_centre(c)) = get_rayon(c)) -> appartient(p, c)
      by hypothesis h2 definition of appartient property basics#beq_symm
    <2>3 qed by step <2>1, <2>2
    <1>2 conclude ;
  end;;

```

Le théorème `appartient_spec` spécifie que pour tout point `p` et cercle `c`, si `p` appartient au cercle `c`, alors la distance entre le point `p` et le centre du cercle `c` est égale au rayon du cercle et vice-versa. À l'étape <1>1, la preuve commence par la skolémisation du but de la propriété. Puis, <1>1 (<->) est décomposée en deux sous-buts <2>1 (->) et <2>2 (<-). Au niveau du sous-but <2>1, Zenon est invoqué pour utiliser l'hypothèse `h1` et déplier la définition de la fonction `appartient` pour chercher une preuve à ce sous but. Dans <2>2, Zenon est invoqué pour utiliser `h2` et la propriété `basics#beq_symm`¹ comme hypothèses et pour déplier la définition de la fonction `appartient` dans la preuve.

L'étape <2>3, sert à mettre fin au niveau <2> (preuve de la propriété énoncée dans <1>1), en utilisant les sous-buts prouvés dans les étapes <2>1 et <2>2 comme prémisses.

Enfin, `conclude` invoque Zenon pour utiliser toutes les preuves obtenues dans les étapes précédentes et ses connaissances internes pour conclure la preuve.

Il est à noter que l'utilisation de la définition d'une fonction pour prouver une propriété, implique que toute future redéfinition de cette fonction entraîne l'annulation de cette preuve. FoCaLiZe invite, à cet effet, le développeur à refaire la preuve.

1.3.3 Preuves par Coq

Une preuve Coq est une séquence d'appels de tactiques Coq (`unfold`, `intro`, `elim` ...) reconnues dans le contexte de la propriété en question.

La syntaxe générale pour introduire une preuve Coq dans une spécification FoCaLiZe est donnée comme suit :

```
proof = coq proof { * suite d'appels des tactiques Coq * }
```

¹exprime la symétrie de l'égalité de base, prédéfinie dans la bibliothèque `basics`

Soit la propriété mathématique suivante :

$$\forall x, y, z : \text{Entier}, (x - y = z) \Rightarrow (x = y + z).$$

La preuve de cette propriété en Coq introduit la définition des opérations d'addition et de soustraction des entiers (`int_plus` et `int_minus`) et la relation d'égalité (`syntactic_equal`) prédéfinies dans Coq, et utilise des propriétés sur ces opérations (`Zplus_eq_compat`, `Z_eq_dec`, ...) aussi prédéfinies dans Coq. La preuve Coq de cette propriété dans une spécification FoCaLiZe est donnée comme suit :

```
open "basics" ;;

theorem int_minus_plus: all x y z : int,
((x - y) = z) -> (x = (y + z))
proof =
coq proof {*
intros x y z; unfold int_plus, int_minus, (=),
syntactic_equal in |- *;
intros H;
unfold bi__int_minus;
apply EQ_base_eq; rewrite <- (Zplus_minus y x);
apply Zplus_eq_compat; trivial; apply decidable.
apply Z_eq_dec. assumption.
Qed.    *} ;;
```

L'introduction des preuves en Coq reste toujours indispensable dans le cas où Zenon échoue lors de la recherche d'une preuve. En particulier, lorsqu'il s'agit d'une propriété d'ordre supérieure, car Zenon ne prend pas en charge la preuve de cette dernière catégorie de propriétés.

1.4 Compilation et mise en œuvre

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 :

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 seront 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.

Soit à compiler notre exemple des espèces `PointAbstrait`, `PointConcret`, `PointColor` et `Cercle`, sauvegardées dans un fichier nommé *point.fcl*.

Après invocation de la commande `focalizec point.fcl`, nous obtiendrons les messages sans erreurs suivants :

```
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/focalize -c point.ml
Invoking zvtov...
>> zvtov -zenon /usr/local/bin/zenon -new point.zv
Invoking coqc...
>> coqc -I /usr/local/lib/focalize -I /usr/local/lib/zenon point.v
```

Ces messages montrent que la compilation, la génération de code exécutable et la vérification par Coq sont passées avec succès.

Lorsque Zenon échoue lors de la recherche d'une preuve, le vide qui correspond à cette preuve dans le code Coq intermédiaire (*point.zv*) sera remplacé par le texte "TO BE DONE MANUALLY" . Dans ce cas, le compilateur Coq (*coqc*) va à son tour échouer, et le développeur doit modifier le fichier source FoCaLiZe en reformulant le script FPL pour qu'il soit réalisable par Zenon, ou bien en insérant manuellement une preuve Coq.

1.5 Documentation

L'environnement FoCaLiZe dispose d'un outil qui permet la génération automatique de documentation (format XML, HTML, Latex ...) appelé **FoCaLiZeDoc**.

FoCaLiZeDoc utilise son propre format XML pour produire des informations extraites d'une spécification FoCaLiZe (espèces, collections, méthodes, définitions de types, commentaires, ...).

Les annotations (qui seront présentes dans la documentation générée par FoCaLiZe) sont introduites par les trois caractères **(**** et terminées par le deux caractères ***)**.

Soit l'exemple de l'espèce suivante :

```
(** In a setoid, we can test the equality (note for logicians:
                                     this is a congruence). *)

species Setoid =
inherit Basic_object;
(** @mathml <eq/> *)
signature equal : Self -> Self -> bool ;
property equal_symmetric : all x y : Self,
```

```

equal (x, y) -> equal (y, x) ;
...
end;;

```

La documentation générée à partir de cette spécification en appelant la commande : `focalizec -focalize-doc`, est présentée dans une page HTML comme suit :

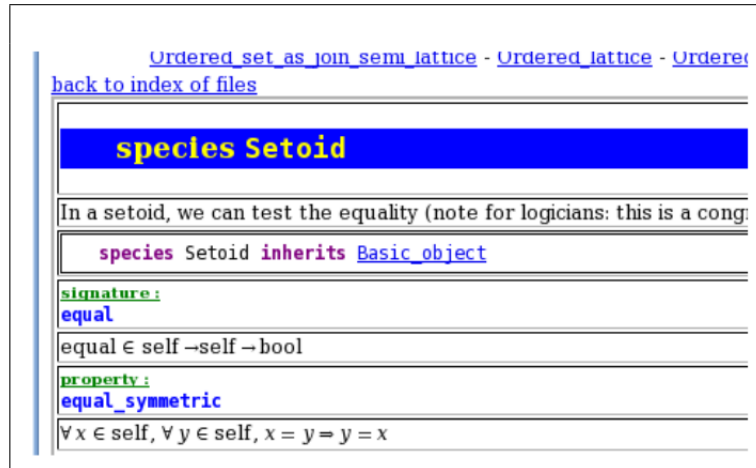


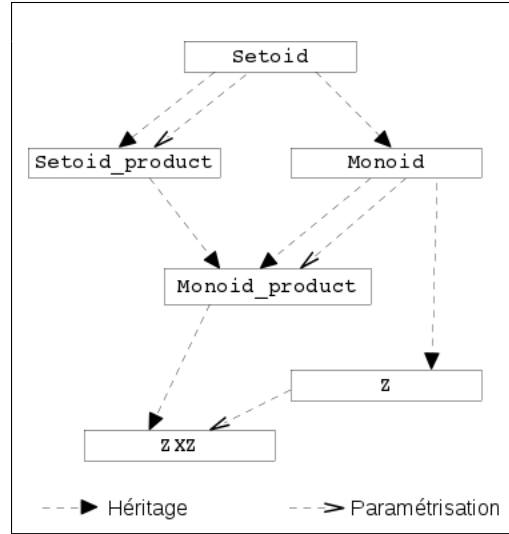
FIG. 1.3 – Exemple d'un document HTML dérivé d'une source FoCaLiZe

1.6 Méthode de développement en FoCaLiZe :

Les exemples des espèces `Point`, `PointColor`, `Cercle`, ..., traités au cours du présent chapitre ont été choisis pour des raisons pédagogiques. Pour mettre en évidence la puissance de spécification de FoCaLiZe, nous présentons, brièvement, les étapes de développement du produit cartésien $\mathbb{Z} \times \mathbb{Z}$, où $\mathbb{Z}$ est l'ensemble des entiers relatifs.

En mathématiques, le produit $\mathbb{Z} \times \mathbb{Z}$ est un monoïde qui résulte du produit de deux monoïdes ($\mathbb{Z}$ et $\mathbb{Z}$). Un monoïde est un setoïde doté d'une opération binaire (c'est l'opération de multiplication des entiers, dans ce cas) et un élément neutre (c'est l'entier 1). Un setoïde, à son tour, est un ensemble non vide doté d'une relation d'équivalence (sur la relation d'égalité, dans le cas de développement de $\mathbb{Z} \times \mathbb{Z}$).

Pour résumer, la figure suivante (figure 1.4) présente la hiérarchie des espèces qui correspond au développement de $\mathbb{Z} \times \mathbb{Z}$:

FIG. 1.4 – Hiérarchie de développement du produit $\mathbb{Z}x\mathbb{Z}$ en FoCaLiZe

Pour développer $\mathbb{Z}x\mathbb{Z}$, nous commençons par la spécification d'une espèce **Setoid** modélisant un ensemble non vide d'éléments, doté de la relation d'égalité **equal** (opération binaire à valeur booléenne). Les trois propriétés (réflexivité, symétrie et transitivité) de l'opération **equal**, permettent de satisfaire une relation d'équivalence sur les éléments de **Setoid** :

```

species Setoid =
  signature equal      : Self -> Self -> bool;
  signature element    : Self;
  property reflexive   : all x : Self, equal(x, x) ;
  property symmetric  : all x y : Self, equal(x, y) -> equal(y, x) ;
  property transitive  : all x y z : Self, equal(x, y) -> equal(y, x) -> equal(x, z) ;
  let different(x, y) = ~~(equal(x, y)) ;
end;;

```

Un monoïde est un setoïde muni d'une opération binaire (*) et d'un élément neutre (**one**) pour cette opération. Aussi, nous utilisons le mécanisme d'héritage en FoCaLiZe pour développer une espèce **Monoid** à partir de l'espèce **Setoid**, comme suit :

```

species Monoid = inherit Setoid ;
  signature ( * ) : Self -> Self -> Self ;
  signature one : Self ;
  let element = one * one ;
  property associative: all x y z : Self, equal((x * (y * z)), ((x * y) * z));
  property neutral: all x : Self, !equal((x * one), x) /\ !equal((one * x), x);
end ;;

```

Comme un monoïde est un setoïde, le produit de monoïdes (l'espèce `Monoid_product`) est aussi un produit de setoïdes (l'espèce `Setoid_product`). Cette dernière espèce est elle même un setoïde, obtenue par héritage de l'espèce `Setoid`, et paramétrée (c'est le mécanisme de paramétrisation en FoCaLiZe) par deux paramètres de la même espèce.

```
species Setoid_product (A is Setoid, B is Setoid) = inherit Setoid ;
representation = (A * B) ;
let equal(x, y) = (A!equal(fst(x), fst(y))) && (B!equal(snd(x), snd(y))) ;
let create(x, y):Self = (x, y) ;
let element = create (A!element, B!element) ;
proof of reflexive = by definition of equal property A!reflexive property B!reflexive;
end ;;
```

L'espèce `Setoid_product` construit ses méthodes en s'appuyant sur celles de ses paramètres de collection (A et B). Le type support est le produit cartésien des types supports des deux paramètres. Nous ajoutons une définition pour la signature `equal` (héritée de `Setoid`) en utilisant les fonctions `equal` de paramètres A et B, qui ne sont toujours que des signatures. Ensuite, nous prouvons que `equal` de `Setoid_product` est effectivement réflexive, sous les hypothèses faites sur `equal` de A et `equal` de B.

Le produit de monoïdes (l'espèce `Monoid_product`), en plus d'être un produit de setoïdes est aussi un monoïde. Ainsi, l'espèce `Monoid_product` hérite des deux espèces `Setoid_product` et `Monoid` d'un côté, et est paramétrée par deux paramètres de `Monoid` (modélisant deux ensembles de monoïd) pour réaliser le produit cartésien :

```
species Monoid_product (A is Monoid, B is Monoid)= inherit Monoid, Setoid_product(A, B);
let one = !create(A!one, B!one);
let ( * )(x, y) = (A!( * )(fst(x), fst(y)), B!( * )(snd(x), snd(y)));
end;;
```

Jusqu'à ce niveau, nos espèces `Setoid`, `Monoid`, `Setoid_product` et `Monoid_product` ne sont qu'abstraites, nous n'avons donné ni représentation concrète, ni corps calculatoire pour les signatures `equal`, `( * )` et `one`. En effet, c'est les choix d'implémentations de ces méthodes qui déterminent la nature des ensembles modélisés par les espèces `Monoid` et `Monoid_product`.

Ainsi, l'ensemble $\mathbb{Z}$ (modélisé par l'espèce `Z`) est dérivé par héritage de l'espèce `Monoid`, en définissant les méthodes `representation`, `equal` et `( * )`, respectivement par le type `int` de FoCaLiZe, l'égalité de base (prédéfinie dans `basics`) et le produit des entiers (aussi, prédéfinie dans `basics`), comme suit :

```
species Z = inherit Monoid;
representation = int ;
let one = 1;
let ( * ) = basics#( * );
```

```
let equal = basics#( = );  
end;;
```

Enfin, nous définissons l'ensemble $\mathbb{Z}x\mathbb{Z}$ par héritage de l'espèce `Monoid_product`, en substituant ses paramètres de collections par des paramètres effectifs de l'espèce `Z` :

```
species ZxZ (A is Z, B is Z)= inherit Monoid_product(A, B);  
end;;
```

1.7 Conclusion

FoCaLiZe est un atelier de développement et de la programmation orienté objet qui couvre toutes les phases du cycle de vie du logiciel, de la spécification de besoins jusqu'à la génération du code exécutable. Les premières étapes d'un développement FoCaLiZe consistent en la création des espèces avec des signatures et des propriétés. Les signatures servent à la spécification des exigences fonctionnelles et les propriétés pour la spécification des exigences de sûreté. Puis, la conception et l'architecture du software sont réalisées progressivement en plusieurs niveaux, en utilisant l'héritage, la paramétrisation, la redéfinition de méthodes, la substitution de paramètres et la liaison tardive. Chaque niveau de la spécification est un pas vers la concrétisation du modèle, en attribuant des corps calculatoires aux signatures héritées et des preuves formelles aux propriétés héritées. Les preuves de propriétés d'un niveau assurent une traçabilité formelle avec le niveau précédent. Ainsi, le passage d'un niveau à l'autre aboutit aux espèces complètement définies, donnant naissance à des collections utilisables en toute sécurité par l'utilisateur final.

Chapitre 2

Sous-ensembles UML/OCL

En raison de la taille et de la complexité des projets informatiques, le développement des applications informatiques nécessite souvent une modélisation abstraite préalable, avant leurs implémentations et mises en œuvres. Une modélisation consiste à concevoir une représentation informatique d'un système sans se préoccuper des détails de l'implémentation. Cependant, une modélisation reste toujours influencée par le paradigme du langage de programmation sous-jacent.

A partir des années 90, la plupart des langages de programmation intègrent des concepts orientés objets. En effet, une modélisation orientée objet permet de définir le problème en termes d'objets, sans rentrer dans les spécifications d'un langage de programmation orientée objet particulier. Les langages de programmation orientés objets utilisent chacun une démarche spécifique pour implémenter et mettre en œuvre une modélisation objet.

Au cours des dernières décennies, de nombreux outils et langages de modélisation orientée objet ont été mis au point. Le langage UML est le résultat de la fusion des trois langages de modélisation suivants :

- La méthode OMT (Object Modeling Technique) de Rumbaugh [RBL⁺90] : Les objets sont identifiés par leurs propriétés (variables d'instances), le comportement est définie par un système d'états et d'événements, et l'aspect fonctionnel (les opérations réalisées par les objets).
- La méthode BOOCH'93 de Booch [Boo91] : C'est une méthode graphique qui utilise quatre diagrammes pour la modélisation de systèmes : Les diagrammes de classes, d'états-transitions, d'objets et d'interactions. Le diagramme de classes décrit la vue logique d'un système par des classes et des relations de classes. Le diagramme d'états-transitions montre l'espace d'états d'une classe, les événements qui provoquent une transition d'un état à un autre, et les actions qui en résultent. Le diagramme d'objets représente les objets et leurs relations vis-à-vis de la vue logique d'un système. Le diagramme d'interaction décrit les étapes d'exécution d'un scénario du système étudié.
- la méthode OOSE (Object-Oriented Software Engineering) de Jacobson [JCC94] : Elle est basée sur la spécification de cas d'utilisation d'un système. En plus des quatre diagrammes de la méthode BOOCH'93, elle introduit trois nouveaux concepts : L'association pour représenter les relations de classes et d'objets, le modèle de cas d'utilisation pour modéliser les rôles (les acteurs) et les cas d'utilisations d'un système, et le modèle d'objet de domaine qui permet de modéliser un scénario de la vie réelle d'un système.

UML est un langage de modélisation orientée objet très complet [BM06], à base de notations graphique couvrant tous les aspects du développement de logiciels, comme la spécification de besoins, l'architecture, les structures et les comportements. Actuellement, UML est le standard de l'OMG pour la modélisation orientée objet. Toutefois, UML n'est pas une méthode dans la mesure où il ne présente aucune démarche, bien que supporté par une grande variété d'outils, allant de l'analyse, au test et à la génération du code.

Le langage **OCL** est un langage du standard OMG [OMG14] 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.

Dans ce chapitre, nous détaillons la syntaxe des ensembles UML/OCL supportés par notre étude. Plus particulièrement, nous présentons une syntaxe abstraite pour la description de l'aspect structurel (diagramme de classes), de l'aspect dynamique (diagramme d'états-transitions) et des contraintes OCL (invariants de classes et pré/post-conditions pour les opérations de classes).

2.1 Modèles UML/OCL

Un modèle UML/OCL est la description conceptuelle d'un système réel en utilisant les notations graphiques d'UML et OCL. UML permet la description d'un modèle à travers différentes vues, chacune étant représentée par un ou plusieurs diagrammes et des contraintes OCL. Un diagramme UML se représente par un graphe connexe où les sommets sont les éléments et les arcs des relations. Un diagramme seul ne représente en général qu'une facette du système construit. La version actuelle de UML (UML 2.5) utilise jusqu'à 23 types de diagrammes différents ([OMG15a]) pour décrire l'aspect structurel (décrivant les composants d'un système et leur relation sans tenir compte le facteur temps) et l'aspect comportemental (montre le comportement du système, les interactions des objets et leur évolution dans le temps) d'un système. Les neuf types de diagrammes UML les plus utilisés [MG00] (ceux de la version UML1.4) sont :

Diagrammes de structure

- Diagramme de classes : Il décrit la structure d'un système par un ensemble de classes et de relations (héritage, association, dépendance, ...). C'est le diagramme le plus fréquent dans une modélisation UML. C'est à la fois l'axe essentiel de la modélisation objet et la notation la plus riche de tous les diagrammes UML.
- Diagramme d'objets : Il représente une vue statique des instances des éléments apparaissant dans un diagramme de classes. Il est composé d'un ensemble d'objets et de leurs relations à un instant donné.
- Diagramme de cas d'utilisation : Il présente, sous forme d'un ensemble de scénarios et d'acteurs, le comportement d'un système en face d'un utilisateur. Ce diagramme consiste à répondre à la question par qui et dans quel cas un système est utilisé.
- Diagramme de composants : Un composant correspond généralement à une ou plusieurs classes ou interfaces. Le diagramme de composants consiste à regrouper sous forme de paquetage les

<i>Umlm</i>	::=	<i>decl*</i>
<i>decl</i>	::=	<i>class</i> <i>association</i> <i>oclConstraint</i> <i>stateMachine</i>
<i>class</i>	::=	<i>option</i> [« <i>class_stereotype</i> »] class <i>class_name</i> [<i>param_signature</i>] [binds <i>bind</i> {, <i>bind</i> }*] [inherits <i>class_name</i> {, <i>class_name</i> }*] [depends <i>class_name</i> {, <i>class_name</i> }*] = <i>attr*</i> <i>opr*</i> end
<i>param_signature</i>	::=	(<i>formal_param</i> {, <i>formal_param</i> }*)
<i>formal_param</i>	::=	<i>formal_param_name</i> [: <i>parameter_kind</i>] [= <i>default</i>]
<i>parameter_kind</i>	::=	<i>type_exp</i> Class
<i>default</i>	::=	<i>class_name</i>
<i>bind</i>	::=	<i>class_template_name</i> < <i>subs</i> [, <i>subs</i> *] >
<i>subs</i>	::=	<i>formal_param_name</i> -> <i>actual_param</i>
<i>actual_param</i>	::=	<i>class_name</i> <i>value_specification</i>
<i>option</i>	::=	[<i>class_visibility</i>] [final abstract]
<i>class_visibility</i>	::=	public private protected package
<i>class_stereotype</i>	::=	enumeration
<i>opr</i>	::=	operation [<i>visibility</i>] [« <i>op_stereotype</i> »] <i>op_name</i> ([<i>op_param</i> {, <i>op_param</i> }*]) [: <i>return_type</i>] [[<i>multiplicity</i>]] [redefines <i>op_name</i>]
<i>op_param</i>	::=	[<i>direction</i>] <i>param_name</i> : <i>type_exp</i> [[<i>multiplicity</i>]] [= <i>default</i>]
<i>direction</i>	::=	in out
<i>op_stereotype</i>	::=	create
<i>attr</i>	::=	attribute [<i>visibility</i>] <i>attr_name</i> [: <i>type_exp</i>] [[<i>multiplicity</i>]] [= <i>default</i>] [redefines <i>attr_name</i>] [<i>attr_modifiers</i>]
<i>visibility</i>	::=	+ ~ # -
<i>attr_modifiers</i>	::=	{ <i>attr_modifier</i> [, <i>attr_modifier</i>] * }
<i>attr_modifier</i>	::=	readOnly union ordered unique nonunique <i>attr_constraint</i> <i>subsets attr_name</i>
<i>type_exp</i> <i>return_type</i>	::=	Integer Boolean String Real UnlimitedNatural <i>class_name</i>
<i>association</i>	::=	association <i>ass_name</i> <i>ass_direction</i> = [<i>ass_type</i>] <i>extrem_left</i> , <i>extrem_right</i> end
<i>ass_type</i>	::=	aggregate composite
<i>extrem_left</i> <i>extrem_right</i>	::=	[<i>role_ident</i>] <i>class_name</i> [[<i>multiplicity</i>]]
<i>multiplicity</i>	::=	<i>lower</i> .. <i>upper</i>
<i>lower</i>	::=	<i>NaturalExpression</i>
<i>upper</i>	::=	<i>UnlimitedNaturalExpression</i>
<i>ass_direction</i>	::=	LeftToRight RightToLeft Both
<i>stateMachine</i>	::=	stateMachine of <i>class_name</i> <i>transition</i> { <i>transition</i> }* end
<i>transition</i>	::=	transition <i>transitionIdent</i> (<i>sourceState</i> , <i>targetState</i>) [<i>event</i> !][[<i>guard</i>]] [<i>behavior-expression</i>]
<i>sourceState</i> / <i>targetState</i>	::=	<i>stateIdent</i> [<i>entry</i>] [<i>exit</i>]
<i>entry</i>	::=	entry <i>behavior-expression</i>
<i>exit</i>	::=	exit <i>behavior-expression</i>
<i>behavior-expression</i>	::=	/ <i>action</i> {, <i>action</i> }*
<i>event</i>	::=	<i>eventIdent</i> ([<i>parameter</i> : <i>type</i> { ; <i>parameter</i> : <i>type</i> }*])
<i>gard</i>	::=	<i>OclExpression</i>

FIG. 2.1 – Sous-ensemble UML

différents composants d'un système. Il s'intéresse généralement à l'implémentation statique et logique du système.

- Diagramme de déploiements : Il montre la disposition physique des différentes 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 différents objets qui constituent le système et les messages qu'ils échangent entre eux.
- Diagramme de séquences : Il décrit l'échange de messages entre les objets intervenant en tenant compte du séquençement chronologique de messages.
- Diagramme d'états-transitions : C'est un automate d'états fini, composé d'états, de transitions et d'événements d'activités. Ce diagramme décrit les états et le comportement des instances d'une classe en réponse à des événements extérieurs.
- Diagrammes d'activités : Il décrit l'enchaînement des activités (en sequence, en parallèle ou selectif) d'un système en fonction de ses états et ses événements. 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 et les diagrammes d'états-transitions pour décrire l'aspect comportemental. Les contraintes OCL considérées sont les invariants de classes et les pré/post-conditions pour les opérations de classes. Nous utilisons aussi des expressions OCL pour décrire les gardes de transitions dans un diagramme d'états-transitions.

Afin de fournir un cadre formel à notre démarche de formalisation de modèles UML/OCL, nous avons défini une syntaxe abstraite pour l'ensemble des fonctionnalités UML supportées par notre étude (voir figure 2.1) et une syntaxe abstraite pour l'ensemble des contraintes OCL (voir figure 2.2) composée des invariants de classes et des pré/post-conditions des opérations de classes. La syntaxe des ensembles UML/OCL suit les méta-modèles d'UML/OCL [OMG15a, OMG14] adoptés par l'OMG. Pour faciliter l'échange de modèles UML/OCL avec d'autres environnements, l'OMG définit aussi le standard XMI (XML Metadata Interchange) [OMG15b] qui permet une description textuelle de modèles UML/OCL en utilisant le format XML. Par souci de lisibilité, nous écrivons la syntaxe des ensembles UML/OCL en notation EBNF ("Extended Backus-Naur Form").

Notations

Tout au long des sections et chapitres suivants, nous utilisons les notations suivantes pour les concepts de direction, stéréotype et visibilité :

Pour un élément UML e ,

- e_n est le nom de l'élément e ,
- e_dir est la direction de e ,
- e_st est le stéréotype de e et
- e_vis est la visibilité de e .

2.2 Diagramme de classes

Un diagramme de classes fournit une vue globale d'un système en spécifiant ses classes et les relations entre elles. Un diagramme de classes est représenté par un graphe, les nœuds sont les classes et les arcs modélisent les relations entre les classes (associations, héritages et dépendances). UML permet aussi la description de classes paramétrées ("templates") et la substitution des paramètres génériques formels d'une classe paramétrée par des paramètres génériques effectifs ("template binding"), pour générer des modèles liés ("bound models").

2.2.1 Classe

Une classe définit la structure commune d'un ensemble d'objets qui partagent les mêmes caractéristiques en termes de fonctionnalités, contraintes et sémantique [BRJ03]. La définition générale d'une classe c_n est exprimée comme suit :

$$[\mathbf{public}][\mathbf{final} \mid \mathbf{abstract}][\langle \text{class-stereotype} \rangle] \mathbf{class} \ c_n \\ (\mathbb{P}_{c_n}) \ \mathbf{binds} \ \mathbb{T}_{c_n} \ \mathbf{depends} \ \mathbb{D}_{c_n} \ \mathbf{inherits} \ \mathbb{H}_{c_n} = \mathbb{A}_{c_n} \ \mathbb{O}_{c_n} \ \mathbf{end} \quad (2.1)$$

avec

- $\mathbb{P}_{c_n}$ une liste de déclarations des paramètres formels,
- $\mathbb{T}_{c_n}$ une liste de substitutions des paramètres formels par des paramètres effectifs,
- $\mathbb{H}_{c_n}$ une liste de noms des classes dont la classe c_n hérite,
- $\mathbb{D}_{c_n}$ une liste de noms des classes dont la classe c_n dépend,
- $\mathbb{A}_{c_n}$ une liste d'attributs et
- $\mathbb{O}_{c_n}$ une liste d'opérations.

Une classe c_n est composée d'une liste d'attributs ($\mathbb{A}_{c_n}$) qui représentent ses variables d'instances et une liste d'opérations ($\mathbb{O}_{c_n}$) pour manipuler ses instances. Une classe peut être spécifiée par la visibilité **public** qui exprime la portée de la classe vis-à-vis des autres classes. Les stéréotypes ($\langle \text{class-stereotype} \rangle$) sont des mécanismes d'extensibilité en UML, ils permettent aux concepteurs d'étendre le vocabulaire d'UML, pour créer des éléments spécifiques. Deux stéréotypes sont supportées : **enumeration** pour créer des types d'énumérations et **intervalType** pour créer des types d'intervalles.

Une classe peut être racine (créée à partir de rien) ou hériter des classes existantes ($\mathbb{H}_{c_n}$) en utilisant la clause **inherits**, comme elle peut dépendre d'autres classes ($\mathbb{D}_{c_n}$) en utilisant la clause **depend**.

Une classe paramétrée ("template") est spécifiée par une liste de paramètres formels ($\mathbb{P}_{c_n}$). Une nouvelle classe peut être créée à partir d'une classe paramétrée via la substitution de ses paramètres formels ($\mathbb{T}_{c_n}$) en utilisant la clause **binds**.

Le mot clé **final** indique qu'une classe ne peut pas être spécialisée (héritée) par une autre classe et le mot clé **abstract** spécifie qu'une classe est partiellement définie et ne peut pas être instanciée.

2.2.1.1 Attributs (variables d'instances)

La liste des attributs $\mathbb{A}_{c_n}$ ($attr_1_n, \dots, attr_k_n$) caractérise les états des instances de la classe c_n . La syntaxe générale de la définition d'un attribut est :

$$\textbf{attribute } attr_vis \ attr_n : typeExp \ [mult] [=default] \quad (2.2)$$

Un attribut est identifié par un nom ($attr_n$) et spécifié par un type ($typeExp$). 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 ($mult$), 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 ($attr_vis$) 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 visibilité :

- **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 visibilité **public** et **private**.

2.2.1.2 Opérations

La liste des opérations de la classe $\mathbb{O}_{c_n}$ ($op_1, \dots, op_k$) permet de manipuler les attributs de la classe c_n et de changer les états de ses instances. Une opération est introduite par sa signature, comme suit :

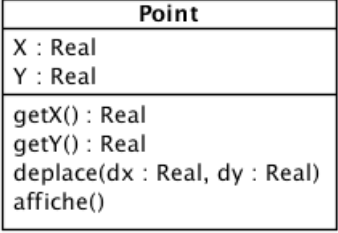
$$\textbf{operation } op_vis \ op_st \ op_n \left(\begin{array}{l} dir_1 \ p_1_n : typeExp_1 \ [mult_1], \\ \dots, \\ dir_k \ p_k_n : typeExp_k \ [mult_k] \end{array} \right) : returnType \ [mult] \quad (2.3)$$

Chaque opération est identifiée par son nom (op_n) et possède une visibilité (op_vis , similaire aux visibilité des attributs), un stéréotype (op_st), une liste de paramètres formels ($p_1_n, \dots, p_k_n$) et un type de retour ($returnType$). Nous considérons la visibilité par défaut (**public**), qui signifie que l'opération est accessible par toutes les classes d'un modèle. Le stéréotype d'une opération vaut **create**, lorsqu'il s'agit d'une opération constructeur (opération retournant une instance de la classe). Tout paramètre formel (p_i_n) de l'opération est spécifié par un type ($typeExp_i$), une multiplicité ($mult_i$, similaire à la multiplicité d'un attribut) et une direction (dir_i). 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 *Point* (voir sa représentation graphique et sa description textuelle, table 2.1) modélise les points d'un plan doté d'un repère orthogonal. Chaque point est défini par ses cordon-

nées (abscisse X et ordonnée Y). L'opération *deplace* permet de déplacer un point et l'opération *affiche* pour afficher les coordonnées d'un point.

TAB. 2.1 – Description de la classe *Point*

Notation UML	Description textuelle
	<pre> public class <i>Point</i> = attribute X : Real attribute Y : Real operation <i>getX</i>() : Real operation <i>getY</i>() : Real operation <i>deplace</i>(dx : Real, dy : Real) operation <i>affiche</i>() end </pre>

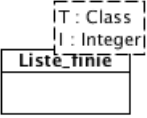
2.2.1.3 Classe paramétrée ("template")

Une classe paramétrée c_n est une classe spécifiée par une liste de paramètres formels (fp_1_n , ..., fp_k_n). Elle est utilisée pour modéliser une classe générique qui pourrait être spécialisée en substituant ses paramètres formels par des paramètres effectifs. Chaque paramètre formel d'une classe paramétrée est spécifié par une expression de type :

$$\textbf{public class } c_n (fp_1_n : typeExp_1, \dots, fp_k_n : typeExp_k) = \dots \textbf{end} \quad (2.4)$$

Seuls les éléments paramétrables [OMG15a] (dans le méta-modèle d'UML) peuvent être utilisés comme paramètres formels d'une classe paramétrée. Dans le cadre de cette étude, nous utilisons des paramètres formels de type **Class** (un type générique modélisant toutes les classes UML) et des paramètres formels de types primitifs (voir exemple, table 2.2).

TAB. 2.2 – La classe paramétrée *Liste_finie*

Notation UML	Description textuelle
	<pre> public class <i>Liste_finie</i>(T : Class, I : Integer) = ... end </pre>

En UML, une classe paramétrée ne peut pas être utilisée de la même manière qu'une classe non paramétrée. Elle est uniquement utilisée pour créer des nouvelles classes par substitution de ses paramètres formels, ou pour être héritée par une nouvelle classe paramétrée du modèle. Une classe paramétrée ne peut pas être utilisée pour créer des instances, ou bien pour typer un attribut ou une opération d'une autre classe.

TAB. 2.3 – Le type d'énumération *Day*

Notation UML	Description textuelle
	<pre>public class «enumeration» Day = Monday Tuesday Wednesday Thursday Friday Saturday Sunday end</pre>

2.2.1.4 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 ($value_1 \dots value_k$) :

$$\text{public class «enumeration» } enum_n = value_1 \dots value_k \text{ end} \quad (2.5)$$

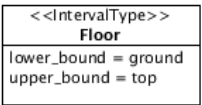
Exemple. Voir la spécification de l'énumération *Day* (en table 2.3) qui modélise les jours de la semaine.

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

$$\begin{aligned} \text{public class «intervalType» } interval_n = \\ \text{lowerBound} = lowerValue \quad \text{upperBound} = uperValue \text{ end} \end{aligned} \quad (2.6)$$

Exemple. Voir la spécification de l'intervalle *Floor* (en table 2.4) qui modélise les étages d'un bâtiment.

TAB. 2.4 – Le type d'intervalle *Floor*

Notation UML	Description textuelle
	<pre>public class «intervalType» Floor = lower_bound = ground upper_bound = top end</pre>

2.2.2 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.

2.2.2.1 Héritage

L'héritage est le mécanisme qui permet à une nouvelle sous-classe c_n d'acquérir tous les attributs, les opérations et les propriétés de ses super-classes $c_1_n, \dots, c_k_n$. 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 :

public class c_n **inherits** $c_1_n, \dots, c_k_n = \dots$ **end** (2.7)

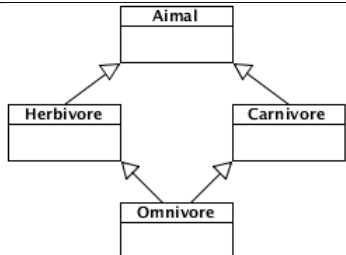
Exemple. Pour manipuler des points colorés (graphiques), la classe *PointColor* (voir table 2.5) est définie par héritage de la classe *Point* (donnée en table 2.1). En plus des attributs et des opérations de la classe *Point*, la classe *PointColor* définit l'attribut *C* qui modélise la couleur d'un point, et l'opération *getC* pour récupérer la valeur de cet attribut. De plus, *PointColor* redéfinit l'opération *affiche* (déjà définie dans la classe *Point*) pour prendre en considération l'affichage de la couleur *C* d'un point en plus de ses coordonnées.

TAB. 2.5 – Exemple de l'héritage simple

Notation UML	Description textuelle
<pre> classDiagram class Point { X : Integer Y : Integer getX() : Integer getY() : Integer deplace(dx : Integer, dy : Integer) affiche() } class PointColor { C : Integer getC() : Integer affiche() } Point < -- PointColor </pre>	<pre> public class <i>PointColor</i> inherits <i>Point</i> = attribute <i>C</i> : Integer ; operation <i>getC</i>() : Integer ; operation <i>affiche</i>() ; end </pre>

Notons que l'héritage multiple (voir exemple, table 2.6) est considéré comme une extension du modèle d'héritage simple, où l'on autorise une classe à posséder plusieurs super-classes.

TAB. 2.6 – Exemple de l'héritage multiple

Notation UML	Description textuelle
 <pre> classDiagram Aimal < -- Herbivore Aimal < -- Carnivore Herbivore < -- Omnivore Carnivore < -- Omnivore </pre>	<pre> public class <i>Herbivore</i> inherits <i>Animal</i> = ... end public class <i>Carnivore</i> inherits <i>Animal</i> = ... end public class <i>Omnivore</i> inherits <i>Herbivore</i> , <i>Carnivore</i> = ... end </pre>

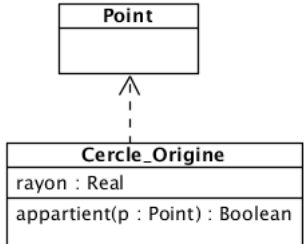
2.2.2.2 Dépendance

Une relation de dépendance spécifie qu'une classe cliente (c_n) a besoin des classes fournisseurs ($c_{1_n}, \dots, c_{k_n}$) 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 :

$$\text{public class } c_n \text{ depends } c_{1_n}, \dots, c_{k_n} = \dots \text{ end} \quad (2.8)$$

Exemple. La classe *Cercle_Origine* (voir table 2.7), modélise les cercles centrés sur l'origine (le point de coordonnées (0.0, 0.0)). Pour décider de l'appartenance d'un point quelconque à un cercle centré sur l'origine, nous utilisons l'opération *appartient*($p : \text{Point}$) paramétrée par une instance de la classe *Point*. Ceci exprime une relation de dépendance entre la classe *Cercle_Origine* (cliente) et la classe *Point* (fournisseur).

TAB. 2.7 – Exemple d'une relation de dépendance

Notation UML	Description textuelle
 <pre> classDiagram Cercle_Origine ..> Point class Cercle_Origine { rayon : Real appartient(p : Point) : Boolean } </pre>	<pre> public class <i>Cercle_Origine</i> depend <i>Point</i> = attribute <i>rayon</i> : Real operation <i>appartient</i>($p : \text{Point}$) : Boolean end </pre>

2.2.2.3 Génération des classes liées ("template binding")

Une classe liée ("bound class") c'_n peut être dérivée d'une classe paramétrée c_n en substituant ses paramètres formels par des paramètres effectifs. Ces substitutions ($\mathbb{T}_{c_n}$) sont spécifiées dans une relation particulière qui relie les paramètres effectifs de la classe liée aux paramètres formels de la classe paramétrée :

$$\text{public class } c'_n \text{ bind } c_n \langle fp_{1_n} \rightarrow ap_{1_n}, \dots, fp_{k_n} \rightarrow ap_{k_n} \rangle = \dots \text{ end} \quad (2.9)$$

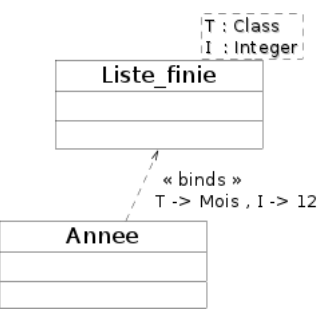
où $fp_i_n \rightarrow ap_i_n$ représente la substitution du paramètre formel fp_i_n par le paramètre effectif ap_i_n .

Notons que dans le cas d’une substitution partielle de paramètres formels (uniquement certains paramètres sont substitués), le modèle dérivé est aussi une classe paramétrée. Dans le cas où tous les paramètres formels sont substitués par des paramètres effectifs, la classe dérivée est une classe feuille qui peut être utilisée et instanciée.

Le standard UML impose que le type de chaque paramètre effectif (dans le modèle lié) doit être un sous-type du paramètre formel correspondant. Dans le cas particulier d’un paramètre formel de type **Class** (voir ensemble UML, figure 2.1), il peut être substitué par n’importe quelle classe du modèle. Le type **Class** est une super-classe de toutes les classes.

Exemple. Une classe paramétrée *Liste_finie* peut servir comme classe générique pour définir toute liste avec un nombre fini d’éléments. Par exemple, la classe *Annee* (voir table 2.8) est un modèle lié dérivé de la classe paramétrée *Liste_finie*, en substituant ses paramètres formels par des paramètres effectifs ($T \rightarrow Mois$, $I \rightarrow 12$).

TAB. 2.8 – Exemple d’une classe paramétrée et d’une relation de “binding” en UML

Notation UML	Description textuelle
	<pre> public class Liste_finie(<i>T</i> : Class, <i>I</i> : Integer) = ... end public class Année binds Liste_finie <<i>T</i> -> <i>Mois</i>, <i>I</i> -> 12> = ... end </pre>

2.2.2.4 Associations

Une association entre deux classes (association binaire) ou plus (association n-aire), décrit des connexions structurelles entre leurs instances.

Une association binaire entre deux classes c_{left_n} et c_{right_n} est définie par son identificateur (ass_n), sa direction de navigation (ass_dir) et les multiplicités associées aux extrémités gauche (MU_{left}) et droite (MU_{right}) de l’association :

$$\textbf{association } ass_n \textbf{ } ass_dir = c_{left_n} MU_{left}, c_{right_n} MU_{right} \textbf{ end} \quad (2.10)$$

La Multiplicité d’une association binaire est définie par un intervalle d’entiers naturels de la forme :

$$MU_{left/right} = lower..upper \quad (2.11)$$

où *lower* est un nombre entier et *upper* est un nombre entier qui peut être infini (de type `UnlimitedNatural`).

Les multiplicités (MU_{left} et MU_{right}) des deux extrémités d'une association expriment des contraintes à satisfaire par les instances de classes reliées par l'association. La multiplicité côté gauche (MU_{left}) décrit le nombre d'instances de la classe côté gauche pouvant être reliées avec une instance de la classe sur le côté droit, et vice-versa.

La Navigabilité d'une association binaire définit dans quel sens l'association peut être parcourue. Pour une association navigable de gauche à droite (spécifiée par **LeftToRight** dans notre ensemble d'UML), à partir d'une instance de la classe côté gauche (c_{left_n}) on peut accéder aux instances (0, 1 ou plusieurs selon la multiplicité) de la classe côté droit (c_{right_n}). La navigabilité de droite à gauche (**RightToLeft**) est définie d'une manière symétrique. Dans le cas d'une navigabilité bidirectionnelle (**Both**) l'association est parcourue dans les deux sens.

Notons qu'en UML, la navigabilité d'une association est annotée par une flèche sur l'extrémité pouvant être atteinte par navigation. L'absence de flèche sur les deux extrémités signifie que l'association est bidirectionnelle.

Selon la navigabilité d'une association, une ou deux fonctions implicites (f_{left} et f_{right}) sont induites pour relier les objets des deux classes c_{left_n} et c_{right_n} :

$$f_{\text{left}} : \{\text{instances de } c_{\text{left}_n}\} \rightarrow P(\{\text{instances de } c_{\text{right}_n}\}) \quad (2.12)$$

Cette fonction retourne une famille d'instances de c_{right_n} avec une cardinalité MU_{right} . La deuxième fonction est définie d'une manière symétrique.

Exemple. Le processus de notification des papiers scientifiques dans le cadre d'une conférence ou d'un journal peut être modélisé par une association binaire (*Review*, voir table 2.9) entre les classes *Person* et *Paper*.

TAB. 2.9 – L'association <i>Review</i> entre les classes <i>Person</i> et <i>Paper</i>	
Notation UML	Description textuelle
	association <i>Review</i> Both <i>Person</i> [3..3] <i>Paper</i> [0..Infini] end

Les multiplicités de l'association *Review* sont :

- $\text{MU}_{\text{left}} = 3$, c'est l'ensemble $\{3\}$ ($\text{lower}=3$ et $\text{upper}=3$).
- $\text{MU}_{\text{right}} = *$, c'est l'ensemble des entiers naturels ($\text{lower}=0$ et $\text{upper}=\text{Infini}$).

La navigabilité de l'association *Review* est bidirectionnelle (**Both**). Par conséquent, elle induit les deux fonctions f_{left} et f_{right} suivantes :

- $f_{\text{left}} : \{\text{instances de } \textit{Person}\} \rightarrow P(\{\text{instances de } \textit{Paper}\})$. Cette fonction permet de récupérer toutes les instances de la classe *Paper* en relation (via l'association *Review*) avec une instance de la classe *Person*, autrement dit, tous les papiers rapportés par une personne donnée.
- $f_{\text{right}} : \{\text{instances de } \textit{Paper}\} \rightarrow P(\{\text{instances de } \textit{Person}\})$. Cette fonction permet de récupérer pour chaque papier les instances de la classe *Person* qui ont rapporté.

2.3 Diagrammes d'états-transitions

Un diagramme d'états-transitions SM_{c_n} décrit le comportement interne des instances d'une classe c_n à l'aide d'un automate à états finis. Il modélise les séquences possibles de transitions (TR_{c_n}) et d'états (ST_{c_n}) qu'une instance de la classe peut prendre au cours de son cycle de vie :

$$\text{SM}_{c_n} = \langle \text{ST}_{c_n}, \text{TR}_{c_n} \rangle \quad (2.13)$$

2.3.1 État

Un état st_i d'une instance de la classe c_n ($st_i \in \text{ST}_{c_n}$) modélise une configuration particulière de valeurs des attributs de la classe. Les états sont définies par leurs noms :

$$\text{ST}_{c_n} = \{ st_{1_n}, \dots, st_{k_n} \}$$

Exemple. Un objet de la classe *Person* (voir figure 2.3, page 52) est à l'état *Minor* lorsque la valeur de son attribut *age* est inférieure à 18 ans. Il est à l'état *Major* dans le cas contraire (la valeur de l'attribut *age* est supérieure ou égale à 18 ans) :

$$\text{ST}_{Person} = \{ Minor, Major \}$$

2.3.2 Transition

Une transition représente le passage d'un état source vers un état cible. Il se produit suite au déclenchement d'un **événement** et seulement si sa **condition de garde** est vérifiée. Une transition peut entraîner l'exécution des **actions**, suite à son franchissement.

Les transitions d'une classe c_n sont définies par $\text{TR} = \{ tr_1, \dots, tr_k \}$, où chaque transition tr_i est définie par :

$$tr_i = \textbf{transition } tr_{i_n} (source_{i_n}, target_{i_n}) \text{ ev}_i [g_i] / ac_{i1}, \dots, ac_{im} \quad (2.14)$$

où

- $source_{i_n}$ et $target_{i_n}$ sont les noms des états source et cible de la transition tr_i ,
- ev_i est l'événement déclencheur de la transition tr_i ,
- g_i est la garde qui conditionne la transition tr_i et
- $ac_{i1}, \dots, ac_{im}$ sont les actions qui s'exécutent suite au déclenchement de la transition tr_i .

Un événement ev_i peut être la réception d'un signal, l'appel d'une opération, la modification de la valeur d'un attribut, l'écoulement d'une durée du temps, etc. Nous considérons, à titre d'exemple, le cas le plus fréquent, l'appel d'une opération de classe :

$$ev_i = ev_n(p_{1_n} : typeExp_1 [mult_1], \dots, p_{k_n} : typeExp_k [mult_k])$$

Une condition de garde g_i est une formule logique définie par une expression OCL (en utilisant la syntaxe présentée en figure 2.2) sur les attributs de la classe c_n et/ou sur les paramètres de l'événement déclencheur (ev_i). La condition de garde est évaluée seulement si l'événement déclencheur se produit.

Les actions sont aussi des appels aux opérations de la classe c_n . Chaque action ac_j est de la forme suivante :

$$ac_j = ac_{j_n}(p_{1_n} : typeExp_1 [mult_1], \dots, p_{k_n} : typeExp_k [mult_k])$$

Exemple. Selon l'âge d'une instance de la classe *Person* (voir table 2.3, page 52), elle peut être à l'état *Minor* ($age < 18$) ou à l'état *Major* ($age \geq 18$). La transition de l'état *Minor* vers l'état *Major* se franchit suite à l'exécution de l'opération *setAge(a : Integer)* (joue le rôle d'événement) de la classe *Person* et uniquement si sa condition de garde ($age \geq 18$) est satisfaite. Les autres transitions (voir table 2.10) sont définies d'une manière similaire.

TAB. 2.10 – Diagramme d'états-transitions de la classe *Person*

Notation UML	Description textuelle
<pre> stateDiagram-v2 state Minor state Major Minor --> Minor : setAge(a: Integer) [age < 18] Minor --> Major : setAge(a: Integer) [age >= 18] Major --> Major : setAge(a: Integer) [age >= 18] </pre>	<pre> stateMachine of Person transition tr1 (Minor, Minor) setAge(a : Integer) [age < 18] transition tr2 (Minor, Major) setAge(a : Integer) [age >= 18] transition tr3 (Major, Major) setAge(a : Integer) [age >= 18] end </pre>

Notons que les diagrammes d'états-transitions d'un système (de toutes les classes du modèle) s'exécutent concurremment et peuvent changer d'états de façon indépendante.

Le diagramme d'états-transitions est le seul diagramme, de la norme UML, qui offre une vision complète et non ambiguë des comportements d'une classe [Aud09].

2.4 Contraintes OCL

OCL [OMG14] est un langage formel basé sur la logique du premier ordre, qui permet la spécification des contraintes sur les éléments d'un modèle UML.

Une contrainte OCL est définie par une expression typée du langage OCL (voir syntaxe d'ensemble OCL, figure 2.2). Un type OCL décrit un domaine de valeurs et un ensemble d'opérations applicables sur ces valeurs. Dans le cadre de notre étude, nous supportons les types OCL suivants :

- Les types primitifs : Integer, Real, Boolean, String et UnlimitedNatural ;
- Les types d'énumérations ;
- Les types de collections : Collection(T), où T est un type OCL ;
- Les types classes (les classes d'un modèle UML).

Un type de collections est un type générique (comme le type tableau en Ocaml) paramétré par un autre type du langage OCL. Soit T un type OCL, le type de collections `Collection(T)` représente une famille de collections d'éléments de type T.

L'appel d'une opération de collections (voir annexe B.3) est dénotée par le symbole `->`. Par exemple, l'appel de l'opération `isEmpty()`, qui permet de tester si une collection est vide, sur la collection C, s'écrit :

`C -> isEmpty().`

2.4.1 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 `c_n`, un invariant $\mathbb{E}_{inv}$ attaché à la classe `c_n`, est défini comme suit :

`context c_n inv :  $\mathbb{E}_{inv}$` (2.15)

où $\mathbb{E}_{inv}$ est une formule OCL (voir règle *formula*, figure 2.2).

Exemple. Soit la propriété de la classe *Person* (voir figure 2.3) suivante : "la valeur de l'attribut *age* est supérieur à 0, quelque soit l'instance de la classe". Cette propriété s'exprime par un invariant sur les instances de la classe *Person*, énoncée comme suit :

`context Person inv : self.age > 0`

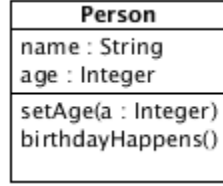
Le mot clé `self` désigne une instance quelconque de la classe en contexte. Ce même invariant peut être exprimé en utilisant l'opération `allInstances()` :

`context Person inv : Person.allInstances()->forAll(p | p.age > 0)`

Les opérations `allInstances()` et `forAll()` sont des opérations de collections (voir annexe B.3). L'opération `allInstances()`, lorsqu'elle s'applique sur une classe, retourne la collection composée de toutes les instances de cette classe. L'opération `forAll()`, permet d'évaluer l'expression (`p.age > 0`) sur tous les éléments de la collection retournée par l'opération `allInstances()`.

<i>oclConstraint</i>	::=	<i>context</i> { <i>oclStereotype</i> : <i>formula</i> } ⁺
<i>context</i>	::=	context { <i>classContext</i> <i>opContext</i> }
<i>oclStereotype</i>	::=	inv pre post
<i>classContext</i>	::=	<i>class_name</i>
<i>opContext</i>	::=	<i>class_name</i> : : <i>op-name</i> ([<i>op-param</i> { , <i>op-param</i> } [*]])
<i>op-param</i>	::=	<i>paramId</i> : <i>typeId</i>
<i>formula</i>	::=	true false not <i>formula</i> <i>formula</i> or <i>formula</i> <i>formula</i> and <i>formula</i> <i>formula</i> implies <i>formula</i> <i>formula</i> xor <i>formula</i> if <i>formula</i> then <i>formula</i> else <i>formula</i> endif let <i>varIdent</i> : <i>type</i> = <i>OCLExp</i> in <i>formula</i> <i>OCLExp</i> <i>compOp</i> <i>OCLExp</i> <i>collection</i> -> forAll (<i>varIdents</i> <i>formula</i>) <i>collection</i> -> exists (<i>varIdents</i> <i>formula</i>) <i>collection</i> -> isEmpty () <i>collection</i> -> notEmpty () <i>collection</i> -> includes (<i>object</i>) <i>collection</i> -> includesAll (<i>collection</i>) <i>collection</i> -> excludes (<i>object</i>) <i>collection</i> -> excludesAll (<i>collection</i>) <i>object</i> . <i>attributeId</i> [@pre] <i>object</i> . <i>queryIdent</i> ([<i>parameters</i>])[@pre]
<i>compOp</i>	::=	= > < <= >= <>
<i>collection</i>	::=	<i>object</i> <i>collectionExp</i>
<i>collectionExp</i>	::=	<i>object</i> . allInstances () <i>collection</i> -> intersection (<i>collectionExp</i>) <i>collection</i> -> union (<i>collectionExp</i>) <i>collection</i> -> select (<i>varIdents</i> <i>formula</i>) <i>collection</i> -> reject (<i>varIdents</i> <i>formula</i>) <i>object</i> . <i>queryIdent</i> [@pre]([<i>parameters</i>])
<i>OCLExp</i>		<i>object</i> <i>collectionExp</i> <i>intExp</i> <i>strExp</i> <i>realExp</i>
<i>intExp</i>	::=	<i>intExp</i> + <i>intExp</i> <i>intExp</i> - <i>intExp</i> <i>intExp</i> * <i>intExp</i> <i>intExp</i> / <i>intExp</i> - <i>intExp</i> <i>intExp</i> . abs () <i>intExp</i> . mod (<i>intExp</i>) <i>intExp</i> . div (<i>intExp</i>) <i>intExp</i> . max (<i>intExp</i>) <i>intExp</i> . min (<i>intExp</i>) <i>collection</i> -> count (<i>object</i>) <i>collection</i> -> size ()
<i>strExp</i>	::=	<i>strExp</i> . concat (<i>strExp</i>) <i>strExp</i> . substring (<i>intExp</i> , <i>intExp</i>) max (<i>intExp</i>) min (<i>intExp</i>) <i>strExp</i> . toUpperCase () <i>strExp</i> . toLowerCase () <i>strExp</i> . size ()
<i>realExp</i>	::=	<i>realExp</i> + <i>realExp</i> <i>realExp</i> - <i>realExp</i> <i>realExp</i> * <i>realExp</i> <i>realExp</i> / <i>realExp</i> - <i>realExp</i> round (<i>realExp</i>) floor (<i>realExp</i>) <i>realExp</i> . abs () <i>realExp</i> . max (<i>realExp</i>) <i>realExp</i> . min (<i>realExp</i>)
<i>object</i>	::=	<i>class_name</i> <i>object_id</i> self

FIG. 2.2 – Sous-ensemble OCL

FIG. 2.3 – La classe *Person*

2.4.2 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 l'exécution d'une opération.

Une pré-condition ($\mathbb{E}_{pre}$) et une post-condition ($\mathbb{E}_{post}$) spécifiées sur l'opération op_n de la classe c_n sont exprimées comme suit :

$$\text{context } c_n : : op_n(p_1_n : typeExp_1, \dots, p_k_n : typeExp_k) \text{ pre : } \mathbb{E}_{pre} \text{ post : } \mathbb{E}_{post} \quad (2.16)$$

où $p_1_n, \dots, p_k_n$ sont les paramètres de l'opération op_n et $typeExp_1, \dots, typeExp_k$ leurs types.

Exemple. Avant d'affecter la valeur de l'attribut *age* d'une personne (par l'opération *setAge(a)*, voir figure 2.3), il faudrait s'assurer que le nouvel âge (donné en paramètre à l'opération *setAge*) soit supérieur à l'âge actuel. Après affectation, la valeur de l'attribut *age* doit être égale à la valeur donnée en paramètre à l'opération *setAge*. Ces contraintes expriment des pré/post-conditions sur l'opération *setAge*, définies comme suit :

```
context Person::setAge(a:Integer) pre : (a > self.age) post: (self.age = a)
```

Quand dans une post-condition il est nécessaire de faire référence à la valeur d'un attribut avant l'exécution de l'opération en contexte, il faut suffixer le nom de l'attribut par le mot clé **@pre**. Par exemple, nous pouvons spécifier la post-condition suivante sur l'opération *birthdayHappens()* :

```
context Person::birthdayHappens() post : self.age = self.age@pre + 1
```

Cette contrainte exprime que la valeur de l'attribut *age* s'incrémente les jours anniversaires.

2.5 Étude de cas

Après la description des ensembles UML/OCL supportés par notre étude, nous présentons deux exemples de modèles, en faisant apparaître leurs représentations graphiques et leurs descriptions textuelles.

Le premier exemple modélise les tableaux et piles finis. Il porte sur les fonctionnalités de diagrammes de classes UML : héritage, classe paramétrée ("template"), génération de classes liées à partir d'une classe paramétrée ("template binding") et propagation des contraintes OCL à travers ces fonctionnalités.

Le deuxième cas est axé sur la modélisation de l'aspect dynamique (diagrammes d'états-transitions) d'un système de contrôle d'un passage à niveau.

2.5.1 Tableaux et piles finis

Parmi les principales applications de classes paramétrées en UML on peut citer la modélisation de classes génériques (comme les classes templates en C++), qui permet aux développeurs d'éviter la répétition et d'améliorer la réutilisation de codes. Nous présentons ici l'exemple (voir figure 6.2) d'une classe paramétrée (**FArray**) modélisant les tableaux finis et ses spécialisations (les classes **FStack** et **PersonStack**) par héritage et par substitution des paramètres.

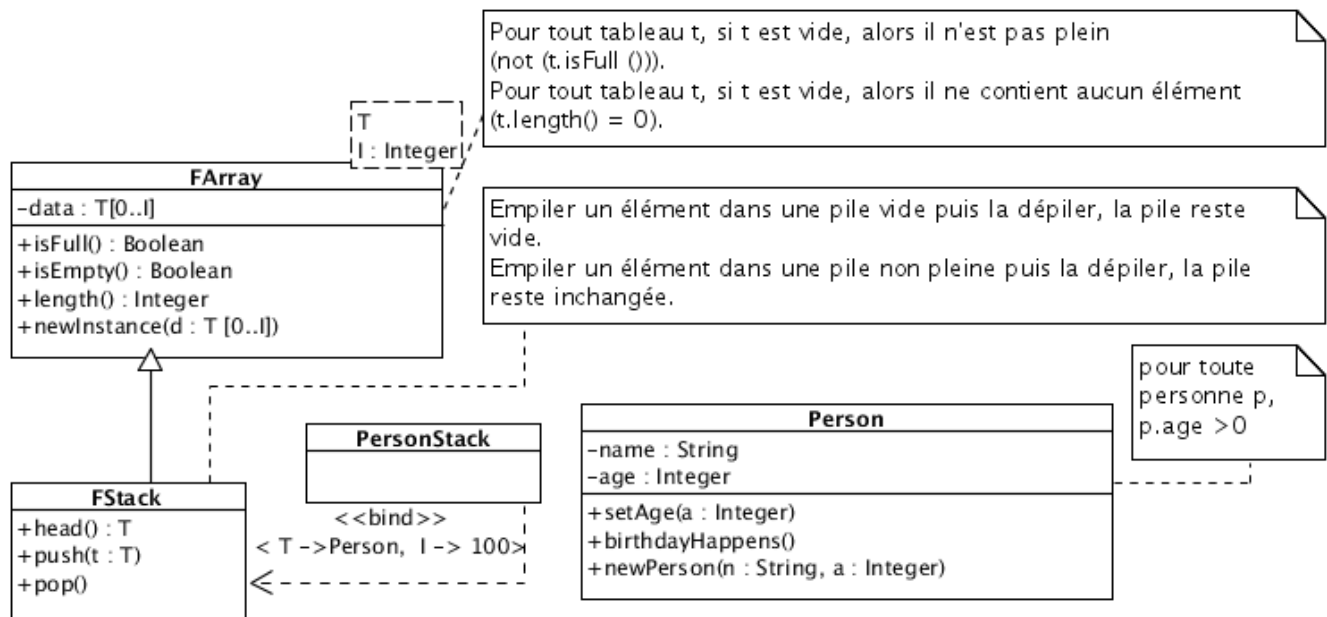


FIG. 2.4 – Diagramme de classes - Tableaux et piles finis -

La spécification textuelle de ce modèle, en utilisant notre ensemble d'UML est donnée comme suit :

Spécification de diagrammes de classes

```

public class Person =
  attribute - name:String
  attribute - age:Integer
  operation + setAge(a:Integer)
  operation + birthdayHappens()
  operation <<create>> + newPerson(n:String, a:Integer)

```

```

end

public class FArray (T:Class, I:Integer) =
  attribute - data: T[0..I]
  operation + isFull():Boolean
  operation + isEmpty():Boolean
  operation + length():Integer
  operation <<create>> + newInstance(d:T[0..I])
end

public class FStack inherits FArray =
  operation + head():T
  operation + push(t:T)
  operation + pop()
end

public class PersonStack binds FStack < T-> Person, I->100 > = end

```

Spécification de contraintes OCL

Les annotations attachées aux classes `FArray`, `FStack` et `Person` sont exprimées par des contraintes OCL comme suit :

```

context FArray
  inv : self.allInstances() -> forAll(s|s.isEmpty() implies s.length()=0)
  inv : self.isEmpty() implies not(self.isFull())

context FStack :: push(t:T)
  pre : self.isEmpty()
  post: let s = self.pop() in (s.isEmpty())

context FStack :: push(t:T)
  pre : not(self.isFull())
  post: self.pop() = self@pre

context Person   inv : age > 0

```

2.5.2 Contrôle d'un passage à niveau

L'exemple présenté ici modélise le contrôle des barrières et des feux de signalisation d'un passage à niveau (voir le diagramme de classes en figure 2.5). Le système se compose d'un contrôleur (la classe `Control`), d'un feu de signalisation (la classe `Light`) et d'une barrière (la classe `Barrier`). Par souci de simplicité, nous ignorons les objets de type véhicule et train dans le modèle.

Le système fonctionne comme suit : A l'état normal, le feu est vert, la barrière est ouverte. Quand le train arrive, afin d'arrêter la circulation, le contrôleur lance une commande pour faire passer le feu au rouge et fermer la barrière. Quand le train est passé, afin de remettre en route la circulation, le contrôleur lance une commande pour ouvrir la barrière et faire passer le feu au vert. Les diagrammes d'états-transitions décrivant le comportements des instances de classes `Control`, `Barrier` et `Light` sont présentés en figure 2.6.

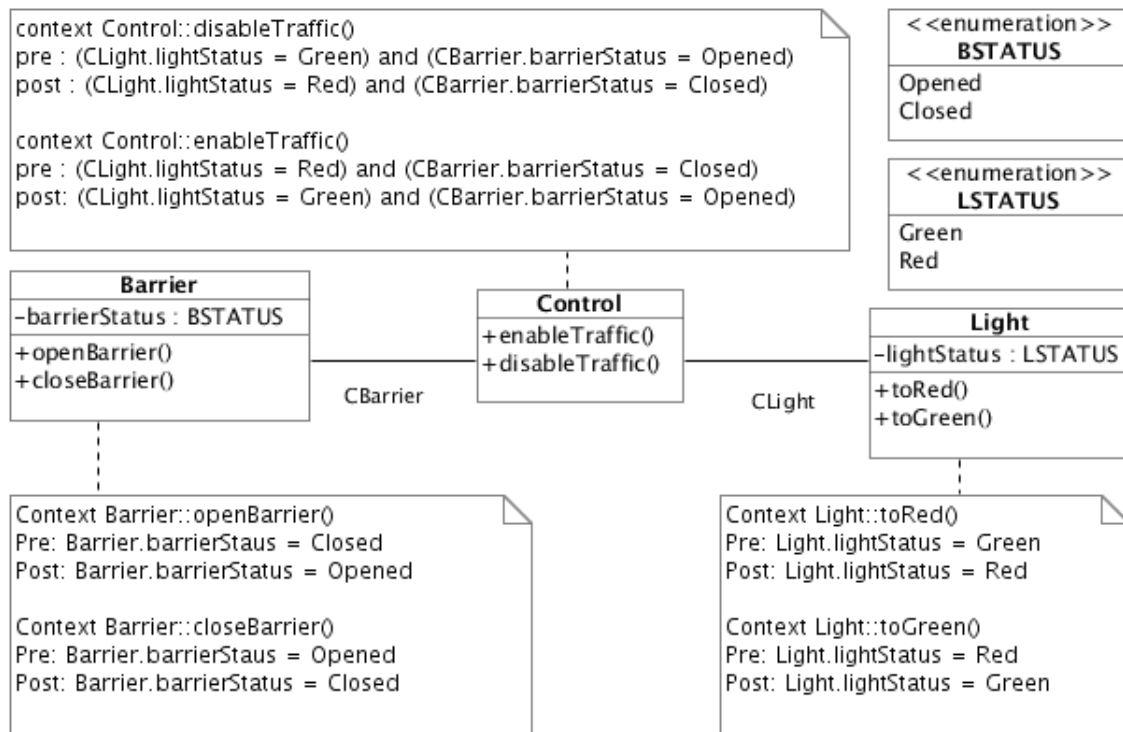
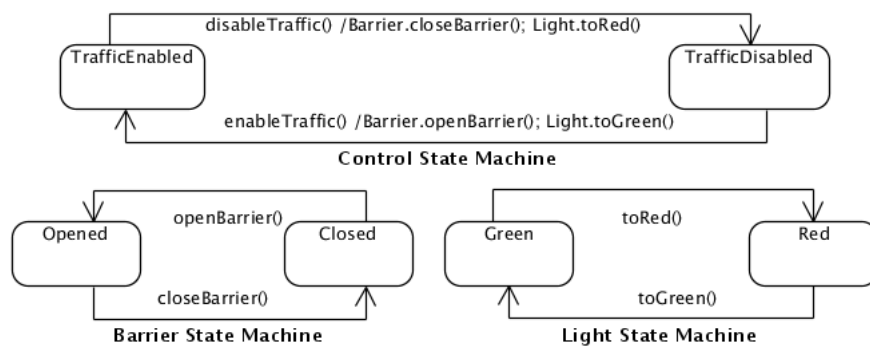


FIG. 2.5 – Diagramme de classes - Contrôle d'un passage à niveau

FIG. 2.6 – Diagrammes d'états-transitions de classes **Control**, **Barrier** et **Light**

La spécification textuelle de ce modèle, en utilisant la syntaxe proposée est donnée comme suit :

Spécification de diagrammes de classes

```
public class «enumeration» BSTATUS  Opened  Closed end
public class «enumeration» LSTATUS   Green   Red   end
```

```
public class Barrier =
  attribute - barrierStatus : BSTATUS
```

```

operation + openBarrier()
operation + closeBarrier()
end

public class Light =
  attribute - lightStatus : LSTATUS
  operation + toRed()
  operation + toGreen()
end

public class Control =
  operation + enableTraffic()
  operation + disableTraffic()
end

```

Spécification des associations

```

association CBarrier LeftToRight Control [1..1] Barrier [1..1] end

association CLight LeftToRight Control [1..1] Light [1..1] end

```

Spécification de contraintes OCL

```

context Control::disableTraffic()
pre : (CLight.lightStatus = Green) and (CBarrier.barrierStatus = Opened)
post : (CLight.lightStatus = Red) and (CBarrier.barrierStatus = Closed)

context Control::enableTraffic()
pre : (CLight.lightStatus = Red) and (CBarrier.barrierStatus = Closed)
post: (CLight.lightStatus = Green) and (CBarrier.barrierStatus = Opened)

Context Barrier::openBarrier()
pre: Barrier.barrierStaus = Closed
post: Barrier.barrierStatus = Opened

Context Barrier::closeBarrier()
pre: Barrier.barrierStaus = Opened
post: Barrier.barrierStatus = Closed

Context Light::toRed()
pre: Light.lightStatus = Green
post: Light.lightStatus = Red

Context Light::toGreen()
pre: Light.lightStatus = Red
post: Light.lightStatus = Green

```

Spécification de diagrammes d'états-transitions

```
stateMachine of Control
transition tr1 (TrafficEnabled, TrafficDisabled) disableTraffic() /
Barrier.closeBarrier(), Light.toRed()
transition tr2 (TrafficDisabled, TrafficEnabled) enableTraffic() /
Barrier.openBarrier(), Light.toGreen()
end
```

```
stateMachine of Barrier
transition tr1 (Opened, Closed) openBarrier()
transition tr2 (Closed, Opened) closeBarrier()
end
```

```
stateMachine of Light
transition tr1 (Green, Red) toRed()
transition tr2 (Red, Green) toGreen()
end
```

2.6 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 aspect statique et dynamique de systèmes, en utilisant des diagrammes de classes, des diagrammes d'états-transitions 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. Nous avons aussi utilisé des expressions OCL pour spécifier les conditions de gardes des transistons, ce qui permet une spécification rigoureuse de diagrammes d'états-transitions.

Le méta modèle proposé, qui se veut plus lisible que le standard XML, est exprimé en notation EBNF afin de permettre une meilleure description des modèles UML. Ce méta modèle sera aussi utilisé dans la description des correspondances syntaxiques (règles de transformation) entre UML/OCL et FoCaLiZe dans les chapitres suivants.

Enfin, le méta modèle présenté a été utilisé pour décrire deux études de cas : Les tableaux et piles finis et le contrôle d'un passage à niveau, qui seront aussi utilisées tout au long des chapitres suivants.

Chapitre 3

Formalisation et vérification de modèles UML/OCL : Etat de l'art

Actuellement, UML est un des langages les plus répandus pour la conception des applications informatiques. UML étant un langage semi-formel, le principal reproche qui peut lui être fait réside dans l'absence de bases formelles permettant l'application des techniques de vérifications formelles. Quant au langage OCL, bien que permettant une description formelle des propriétés d'un modèle UML, il ne dispose pas d'outils pour la vérification et la preuve de ces propriétés.

Dans ce contexte, de nombreux travaux ont été réalisés afin de doter les modèles UML/OCL de bases formelles et d'outils de vérifications. La technique la plus anciennement utilisée pour la vérification des propriétés de modèles UML/OCL est la génération de tests [FL00]. Les réseaux de Petri [GV13] sont aussi utilisés pour le même objectif. Une autre approche consiste à utiliser des environnements formels pour la description d'une sémantique formelle pour UML, comme CASL [RCA01] ou UML lui-même dans pUML [CEK01]. Cependant, l'approche la plus largement adoptée est la transformation d'un modèle UML/OCL vers une spécification formelle, en utilisant des méthodes formelles comme B [Abr96], Alloy [Jac11], Isabelle/HOL [NPW02], Maude [CDE⁺07], LOTOS [Lot89], etc. Cette dernière approche rentre dans le cadre de l'ingénierie de modèles (MDE, "Model Driven Engineering"), qui vise la production de softwares par raffinements automatiques de modèles, depuis les spécifications abstraites jusqu'aux implémentations concrètes.

Nous trouvons aussi des travaux qui s'intéressent à la génération des modèles UML/OCL à partir des spécifications formelles ("reverse engineering"). Plus particulièrement, la transformation de spécifications B, de spécifications Alloy et de spécifications FoCaLiZe vers UML/OCL. Cette approche vise à produire une vue UML/OCL à des utilisateurs qui ne sont pas familiarisés avec les notations formelles utilisées.

Cependant, l'utilisation des mathématiques et de la logique réduit souvent la capacité architecturale et conceptuelle des méthodes formelles. Ces limitations entraînent que certaines fonctionnalités majeures d'UML/OCL ne sont pas toujours supportées par les formalisations existantes. C'est le cas de l'héritage multiple, la redéfinition de méthodes, la liaison tardive, les templates (classes paramétrées), la dérivation de modèles liés à partir des templates (substitution de paramètres formelles par des paramètres effectifs) et la propagation des contraintes OCL dans les modèles UML/OCL.

Dans ce chapitre, nous présentons un état détaillé des différentes approches de formalisation de

modèles UML/OCL. Nous commençons par les travaux visant à l'attribution d'une sémantique formelle pour UML, c'est la base de toute formalisation d'UML. Puis, nous discutons les travaux de formalisation utilisant la transformation vers des méthodes formelles et celles utilisant la génération de tests et les réseaux de Petri. Nous nous concentrerons sur l'approche transformationnelle (en particulier la transformation vers la méthode B), approche retenue dans le cadre de ce travail de thèse. Enfin, nous terminons le chapitre par la discussion des transformations inverses (du formel vers UML) et par une étude comparative des principales approches de formalisations.

3.1 Attribution d'une sémantique formelle pour UML

L'attribution d'une sémantique formelle pour UML consiste à donner un sens (une signification) précis à ses notations, en utilisant une technique formelle (mathématiques et logique, langage formel, ...). Une telle sémantique est indispensable [EK99], pour des raisons de clarté, de cohérence et de précision (absence d'ambiguïté). Les principales contributions de cette approche sont :

- L'attribution d'une sémantique dénotationnelle aux diagrammes de classes UML [CE97], basée sur l'utilisation du langage Z [DRS95]. La démarche proposée permet de préciser la sémantique d'un sous-ensemble d'UML en utilisant directement un langage de programmation formelle (au lieu d'utiliser les mathématiques et la logique).
Nous trouvons aussi une approche récente pour l'intégration des notations UML et Z [SSS16]. L'approche proposée permet la transformation des diagrammes UML : de cas d'utilisation, de classes et de séquence en Z. Le modèle formel résultant est analysé et vérifié à l'aide de l'outil Z/Eves ¹.
Dans ce même contexte, la syntaxe et la sémantique du diagramme d'activité de UML 2.5 ont été aussi formalisées par des notations Z [JZ16]. Tous les blocs de construction de base de diagramme d'activité et leurs sémantiques structurelles ont été formalisés en utilisant des schémas Z.
- L'attribution d'une sémantique très précise aux notations OMT ("Object Modeling Technique") [BC95]. La sémantique attribuée est basée sur le langage LSL (Larch Shared Language), un langage modulaire basé sur des spécifications algébriques. La démarche proposée permet d'utiliser les notations statiques d'OMT (classe, attribut, association, ...) pour la construction de spécifications formelles. Par contre, le modèle d'exigences d'OMT n'est pas formalisé.
- Une sémantique dénotationnelle est aussi attribuée aux notations d'UML, dans le cadre du projet Sémantique d'UML2 [BCD⁺07]. Bien que la sémantique proposée couvre des aspects structurels et dynamiques d'UML2, elle ignore l'utilisation des contraintes OCL.
- L'attribution d'une sémantique d'action ("action semantics") [YMP08] aux notations d'UML, en utilisant un langage d'action ("action language") comme Executable UML [MBFBJ02]. Ce dernier joue le rôle d'interface entre les diagrammes UML et leurs sémantiques d'actions. La sémantique proposée est suggérée pour être utilisée dans des approches MDA (Model Driven Architecture) [MM⁺03]. Cependant, la synchronisation des objets par des actions de communication n'est pas supportée.

¹Z/Eves : <http://czt.sourceforge.net/eclipse/zeves/>

- Les travaux du groupe precise UML (pUML) [CEK01] visent à fournir une sémantique à UML en utilisant UML lui-même et le langage OCL. L'objectif est de rendre l'analyse formelle directement accessible via les diagrammes UML (même pour les non-spécialistes) et sans utiliser d'autres formalismes. Seuls les diagrammes de classes sont considérés par pUML.
- D'autres propositions utilisent CASL (Common Algebraic Specification Language) [RCA01, Fav10] pour donner une sémantique formelle aux diagrammes UML, à base de descriptions algébriques. Les sémantiques proposées sont très complètes et couvrent les aspects statique (diagramme de classes et diagramme d'objets) et dynamique (diagramme d'état-transitions et diagramme d'activités). Par contre, elles ignorent aussi l'utilisation des contraintes OCL et ne suggèrent aucune démarche de vérification.
- Enfin, les travaux de l'OMG pour l'élaboration d'une sémantique officiellement adoptée pour UML. Cette sémantique est complètement spécifiée par des contraintes OCL et a donné naissance au méta-modèle d'UML [OMG15a] et son standard XMI (XML Metadata Interchange). Actuellement, toute transformation d'UML vers d'autres formalismes passe par le standard XMI.

Avantages et inconvénients

L'attribution d'une sémantique formelle pour UML permet de lever l'ambiguïté de modèles UML et de spécifier rigoureusement ses propriétés. Ceci a permis d'implémenter les diagrammes d'UML2 par plusieurs outils de modélisation comme Visual Paradigm², Papyrus³, UML2 Eclipse plug-in⁴, etc. Cependant, une sémantique formelle pour UML ne permet pas (tout seule) de faire la preuve et la vérification de propriétés sur les modèles UML, elle a besoin d'être accompagnée d'une technique de preuve et vérification pour atteindre cet objectif.

3.2 Transformation en méthodes formelles

La transformation d'un modèle UML/OCL vers une méthode formelle permet dans une première étape de produire une spécification formelle, puis d'utiliser les techniques de vérifications fournies par la méthode cible pour vérifier et prouver les propriétés du modèle UML/OCL. Généralement, les travaux qui ont opté pour cette approche, s'attachent plus particulièrement à la transformation des diagrammes de classes, diagrammes d'états-transitions et des contraintes OCL.

On distingue deux approches [FL95] différentes de dérivation de spécifications formelles à partir de spécifications non formelles : l'approche interprétée et l'approche compilée. L'approche compilée (par traduction), consiste à produire des règles permettant de traduire un modèle non formel directement dans un langage formel.

L'approche interprétée (par méta-modélisation) consiste à décrire les deux modèles (semi-formel et formel) par des méta-modèles, puis de donner une fois pour toute la formalisation du modèle semi-formel. La formalisation d'une application particulière s'effectue par projection au niveau de

²Visual Paradigm : <http://www.visual-paradigm.com/>

³Papyrus : <http://www.eclipse.org/papyrus/>

⁴UML2 Eclipse plug-in : <http://www.eclipse.org/modeling/mdt/downloads/?project=uml2>

la formalisation du méta-modèle.

L'intérêt de l'approche par traduction est d'être réalisable par une traduction directe d'un modèle semi-formel en une spécification formelle. Cependant, son principal inconvénient réside dans le fait que la sémantique de traduction entre les deux formalismes est cachée derrière les règles de traduction [Akr06]. A l'inverse, l'approche interprétée permet de préciser les éléments des méta-modèles des deux formalismes et de clarifier la sémantique de leurs correspondances. Bien que non directe, elle est adaptable aux besoins propres des systèmes.

Dans ce qui suit, nous détaillons la transformation d'UML/OCL vers les méthodes formelles. Nous commençons par la transformation vers la méthode B [Abr96]. Puis, nous étudions les techniques de transformation d'UML/OCL vers d'autres méthodes formelles, à savoir Alloy [Jac11], Isabelle/HOL [NPW02], Maude [CDE⁺07], SMV [CCGR00], Spin [Hol04] et LOTOS [ISO85].

3.2.1 Transformations vers B

La méthode B est un langage de programmation basé sur la théorie des ensembles, elle couvre toutes les étapes de développement de logiciels, de la spécification abstraite jusqu'à la génération de code exécutable.

La brique de base dans un développement B est la machine abstraite, une structure caractérisée par :

- Un état désigné par les données de la machine,
- des opérations permettant d'échanger l'état de la machine et
- un invariant exprimant une exigence que la future implémentation de la machine doit satisfaire.

Une opération d'une machine abstraite peut aussi être spécifiée d'une pré-condition précisant une condition sous laquelle l'invocation de l'opération aboutira à un état correct de la machine.

Un développement B commence par la spécification des machines abstraites. Puis, on en réduit au fur et à mesure le niveau d'abstraction, en utilisant des raffinements successifs. Le raffinement d'une machine abstraite est un composant qui conserve les mêmes caractéristiques de la machine et reformule les données et les opérations de la machine à l'aide des données et des opérations plus concrètes. Il représente un pas vers l'implémentation concrète de la machine abstraite. Le générateur d'obligations de preuves de la méthode B analyse le modèle à chaque niveau de raffinement et génère les différentes obligations à prouver pour assurer la cohérence du modèle. L'étape ultime des raffinements est l'implantation de la machine, un modèle concret garantissant que les exigences de la machine sont satisfaites.

Plusieurs travaux ont été réalisés en termes de transformations de modèles UML/OCL vers la méthode B. Nous trouvons d'abord la proposition d'une dérivation de spécifications formelles B à partir de spécifications semi-formelles UML [Ngu98]. Puis, dans le cadre de l'élaboration d'un développement formel par objets, nous trouvons une proposition pour l'utilisation conjointe de B et d'UML. [Mey01]. Ensuite, les travaux précédents sont exploités pour la conception et le développement formels d'applications de bases de données, en utilisant UML et B [Lal02]. Nous trouvons aussi une démarche générale pour la conception et la modélisation formelle des applications informatiques assistée UML [SB06]. L'objectif de tels travaux est de préciser la sémantique d'UML et de traduire les spécifications semi-formelles en B pour profiter des outils de preuve et de vérifica-

tion de la méthode B. La plupart de ces travaux adoptent une approche compilée, basée sur un ensemble de règles de traduction.

La transformation d'un modèle UML/OCL vers B se concentre sur la transformation des diagrammes de classes, diagrammes d'états-transitions et des contraintes OCL. Généralement, Les classes sont modélisées par des machines abstraites [TF15], les attributs de classes sont formalisés par des données de machines (dérivées de classes) et les opérations de classe sont reportées vers des opérations de ces machines. La relation d'héritage est modélisée par un invariant spécifiant une inclusion entre les instances de machines abstraites modélisant la super-classe et la sous-classe.

Les diagrammes d'états-transitions sont formalisés par des énumérations B modélisant les états et par des opérations B permettant le changement d'état.

Les principales contraintes OCL supportées dans les transformations d'OCL vers B sont les invariants de classes et les pré/post-conditions des opérations de classes [LS02]. Les pré-conditions OCL sont reportées vers des pré-conditions des opérations B, les post-conditions donnent lieu aux substitutions B (affectations des variables) et les invariants de classes sont formalisés par des invariants des machines (dérivées de classes).

Ainsi, La transformation d'un modèle UML/OCL produit une spécification B (ensemble des machines abstraites). Puis, des obligations de preuve sont générées à partir de ces machines abstraites. Ensuite, le prouveur de la méthode B utilise ces obligations de preuves pour analyser et prouver les propriétés du modèle. En cas d'échec, un expert analyse l'échec et distingue deux cas : soit il modifie la spécification UML/OCL d'origine, soit il interagit avec le prouveur en enrichissant la base de ses règles de déduction, et recommence le cycle.

Avantages et inconvénients

Les transformations d'UML/OCL vers B proposent des formalisations directes et puissantes qui ont donné naissance à plusieurs outils de transformation comme UML2B [HLM04], U2B [SB04] et ArgoUML+B [LSC03]. Ces outils permettent d'un côté d'utiliser UML comme point de départ dans un processus de développement en B, et de l'autre côté, ils permettent d'utiliser les outils de vérification de B pour analyser et vérifier la correction des spécifications formelles dérivées à partir des modèles UML/OCL.

Cependant, les divergences entre l'approche UML/OCL et l'approche B ne permettent pas une formalisation cohérente de certaines fonctionnalités fondamentales d'UML/OCL. Ces divergences proviennent du fait qu'une spécification B est exprimée en termes de machines abstraites dans une approche formelle guidée par les preuves, et une modélisation UML/OCL est exprimée en termes de classes et contraintes OCL dans une approche guidée par les programmes. A titre d'exemple, les raffinements dans B ne sont pas suffisants pour modéliser le mécanisme d'héritage en UML, en effet une machine abstraite ne peut être raffinée qu'une seule fois lors de son cycle de vie [Abr96]. Le mécanisme d'héritage est indirectement traité par une relation d'inclusion en B, comme nous l'avons indiqué plus haut.

Le paramètre formel d'une machine abstraite ne peut pas être une autre machine abstraite du modèle, un raffinement ou une implantation, seuls les types scalaire et ensemble [Abr96] sont autorisés. Cette limitation empêche une modélisation cohérente des classes paramétrées ("templates") d'UML.

Les limitations dans la transformation des mécanismes d'héritage et de dérivation de modèles liés aux classes paramétrées ("templates bindings") ne permettent pas la propagation des contraintes

OCL à travers ces fonctionnalités dans les transformations d'UML/OCL vers B.

Finalement, Les transformations d'UML/OCL vers B ignorent aussi la navigation des contraintes OCL via les associations, en effet une machine abstraite ne peut pas accéder aux propriétés d'une autre machine abstraite. Aussi, les contraintes OCL ne sont pas exploitées pour contrôler les transitions au sein de diagrammes d'états-transitions.

3.2.2 Transformation vers Alloy

Alloy [Jac11] est un langage de modélisation basé sur la logique relationnelle, utilisant un style de conception modulaire inspiré du paradigme orienté objet. Un module Alloy regroupant des spécifications (**signatures**) permet d'exprimer des contraintes structurelles et dynamiques sur les modèles. En plus, Alloy est supporté par une infrastructure logicielle (l'Analyseur d'Alloy⁵), qui permet d'analyser et de vérifier d'une manière systématique les propriétés d'un modèle Alloy.

De nombreux travaux sont axés sur la transformation de l'aspect structurel d'UML vers Alloy. Nous trouvons d'abord, une étude exhaustive sur la transformation d'UML/OCL vers Alloy [ABGR07]. Cette étude a donné naissance à un prototype (appelé **UML2Alloy**) de transformation automatique des modèles UML/OCL en spécifications Alloy. Nous trouvons aussi une transformation d'UML/OCL vers Alloy [MGB05] qui considère (en plus) la transformation des interfaces UML, mais elle ne traite pas les pré/post-conditions OCL. Une autre proposition [MC13] vise la transformation des fonctionnalités UML qui ne sont pas naturellement formalisables par Alloy, comme l'héritage multiple. Cependant, elle génère des modèles Alloy très difficile à lire et ignore les contraintes OCL.

L'aspect dynamique (diagrammes d'états-transitions d'UML) est aussi formalisé par Alloy [GPCR12] pour simuler et vérifier la consistance entre les aspects statique et dynamique, et les contraintes OCL.

La vérification des propriétés d'un modèle Alloy est assurée par l'analyseur d'Alloy, un outil très efficace qui permet de détecter la plupart des défauts et des contre-exemples. L'analyseur d'Alloy permet la génération automatique des instances et la vérification des assertions. Compte tenu de l'indécidabilité de la logique d'Alloy, l'analyse doit être aidée par des interactions utilisateurs.

Avantages et inconvénients

Alloy est un langage de modélisation avec un ensemble limité des constructions, utilisé avec succès dans la modélisation et la vérification des projets industriels. A titre d'exemple, Alloy a été utilisé dans la vérification de la modélisation UML du fonctionnement d'une machine de radiothérapie [DSRJ04] et dans la modélisation et l'analyse d'un système distribué [GBF01].

Contrairement à la méthode B, Alloy ne couvre pas toutes les phases de développement d'un logiciel, il ne dispose pas du niveau implémentation des modèles. Ainsi, les modèles UML sont transformés d'abord vers Alloy pour la vérification, puis vers un langage de programmation (comme java, C, ...) pour l'implémentation.

Côté fonctionnalités UML, l'héritage multiple, la redéfinition de méthodes et la liaison tardive ne sont pas supportés dans les transformations d'UML vers Alloy. Dans un développement Alloy, un module peut être paramétré par des signatures. Puis, ces paramètres seront instanciés lors de l'importation des modules. Cependant, ce type de paramétrage ne permet pas le même niveau

⁵Analyseur d'Alloy : <http://alloy.mit.edu/>

d'abstraction que le mécanisme de substitutions de paramètres formels (d'une classe paramétrée) par des paramètres effectifs en UML.

3.2.3 Traduction vers HOL

Isabelle/HOL ⁶ est un assistant de preuve basé sur la logique d'ordre supérieur ("Higher Order Logic"). HOL-OCL ⁷ [BW07] est un environnement interactif très complet pour la transformation et l'évaluation de contraintes OCL, basé sur l'assistant de preuve Isabelle/HOL. HOL-OCL utilise un prototype UML/OCL appelé su4sml [BDW07] qui permet une description textuelle des diagrammes de classes, des diagrammes d'états-transitions et des contraintes OCL.

Pour la spécification et la vérification d'un modèle UML/OCL, HOL-OCL suit les étapes suivantes : Au début, le modèle UML/OCL est conçu par un outil graphique (ArgoUML ⁸), puis importé par su4sml en utilisant le standard XMI ("XML Metadata Interchange"). Ensuite, l'encodeur de HOL-OCL transforme les contraintes OCL vers des expressions de la logique d'ordre supérieur (HOL) et procède à la preuve et la vérification du modèle en utilisant la bibliothèque de théorèmes de HOL-OCL et des techniques basées sur la méthode de tableaux.

Avantages et inconvénients

HOL-OCL propose une formalisation très complète de modèles UML/OCL, supportant la plupart des fonctionnalités (statiques et dynamiques) d'UML et des contraintes OCL. Cependant, HOL-OCL est beaucoup plus orienté vers l'évaluation des contraintes OCL, plutôt que la génération des logiciels sûrs. D'un autre côté, le prototype UML support (su4sml) de HOL-OCL ne permet pas certaines fonctionnalités UML comme l'héritage multiple et la substitution de paramètres formels de classes paramétrées ("templates") par des paramètres effectifs. Le mécanisme d'héritage simple est formalisé par une relation d'inclusion et la liaison tardive est partiellement supportée par HOL-OCL [BW08].

3.2.4 Transformation vers des méthodes basées model-checker

Plusieurs méthodes formelles utilisent des model-checkers pour la vérification des propriétés des modèles. Utilisant une méthode de cette catégorie, les modèles UML/OCL sont formalisés par le langage de spécification du model-checker, puis le model-checker sera déclenché pour la détection des défauts et des éventuelles contradictions. Parmi les principales méthodes de cette catégorie, nous citons le système Maude [CDE⁺07], Promela [Ios98], SMV [McM93] et LOTOS [ISO85]. Dans ce qui suit, nous décrivons les contributions basées sur ces méthodes pour la formalisation de modèles UML/OCL.

Maude est un environnement basé sur la logique de réécriture [MOM02] et la logique équationnelle. Il permet de spécifier formellement la dynamique d'un système à l'aide de règles de réécriture, puis d'analyser et de vérifier ses propriétés en utilisant son model-checker. Maude dispose aussi d'un outil de preuve (de théorèmes) interactif pour vérifier les propriétés des systèmes complexes.

⁶Isabelle : <http://isabelle.in.tum.de/>

⁷HOL-OCL : <https://www.brucker.ch/projects/hol-ocl/>

⁸ArgoUML : <http://argouml.tigris.org/>

Un prototype Maude pour les modèles UML [DGR11] a été proposé pour permettre la preuve et la vérification des contraintes OCL d'une manière dynamique, en cours de leurs simulations. Cette contribution s'attache plus spécialement aux transformations de diagrammes d'états-transitions spécifiés par des contraintes OCL. Une autre contribution consiste dans la transformation et la vérification d'un système d'agents modélisé par UML vers le système Maude [MSBK10].

Maude est une méthode complète qui couvre toutes les phases de développement des logiciels. Le système Maude autorise l'héritage simple, et permet de spécifier un module paramétré par des paramètres formels, ensuite un nouveau module peut être créé en substituant les paramètres formels par des paramètres effectifs. Cependant, des fonctionnalités comme l'héritage multiple, la redéfinition de méthodes, la liaison tardive et la substitution des paramètres formels de classes par des paramètres effectifs sont ignorées dans la transformation d'UML/OCL vers Maude.

Promela est le langage de spécification de l'outil SPIN [Hol97] (model-checker). Le travail présenté dans [PL99] montre que les diagrammes d'états-transitions d'UML peuvent être traduits vers le langage Promela. Ce langage permet de représenter des automates communiquant entre eux, puis d'utiliser l'outil SPIN pour vérifier des propriétés de la logique temporelle à partir de ces automates. L'utilisation de SPIN et PROMELA est limitée à la vérification de diagrammes d'états-transitions isolés, ce qui limite ces approches à des systèmes relativement statiques.

LOTOS est un langage de spécification basé sur l'algèbre de processus et l'utilisation de types abstraits. Une modélisation LOTOS est une spécification algébrique exprimant les propriétés que doit vérifier toute réalisation du modèle sans imposer des contraintes d'implémentations. RT-LOTOS étend LOTOS avec des opérateurs temporels : délai, latence et offre limitée dans le temps. La vérification de programmes LOTOS est réalisée par le model-checker CADP (Construction and Analysis of Distributed Processes) [FGK⁺96].

LOTOS a été utilisé pour la formalisation et la vérification automatique d'un système de chaudière à vapeur [CC00], modélisé par un sous ensemble d'UML.

Les notions de références, de liens dynamiques entre objets, et de création d'objets ne sont pas traitées, ce qui limite ces transformations à des spécifications statiques. Les contraintes OCL ne sont pas supportées dans les transformations d'UML vers LOTOS. Les propriétés de classes sont spécifiées par le langage Object-Z.

SMV est un autre model-checker équipé de deux langages de modulation (Extended SMV et synchronous verilog). L'outil SMV a été utilisé pour la vérification [Kwo00] de propriétés sur des diagrammes d'états-transitions d'UML en utilisant des spécifications CTL et LTL. Aussi avec SMV, les contraintes OCL ne sont pas considérées dans la vérification des diagrammes d'états-transitions.

UPPAAL-SMC est une nouvelle variante de la famille UPPAAL qui permet de raisonner sur des réseaux de systèmes complexes en temps réel avec une sémantique stochastique. Une contribution récente [GZC⁺16] vise à étendre les éléments du diagramme d'activité UML pour permettre une modélisation stochastique des entrées des utilisateurs et des exécutions d'actions, basée sur le vérificateur du model-checker UPPAAL-SMC [DLL⁺15].

Avantages et inconvénients

En général, les méthodes basées model-checker permettent une vérification rapide et entièrement automatique des propriétés des modèles UML/OCL. Cependant, l'utilisation des model-checkers reste limitée pour la vérification de systèmes simples (disposant d'un nombre limité d'états), car ils cherchent à explorer tous les états possibles d'un système. Pour les systèmes complexes, les model-checkers ne sont aussi efficaces que les prouveurs de théorèmes. A l'exception des transformations vers Maude, la plupart des travaux de cette catégorie sont limités à des systèmes statiques et ignorant l'utilisation des contraintes OCL.

3.3 Vérification par génération de tests

Un test (sur un système donné) est défini par l'exécution (manuelle ou automatique) d'un scénario particulier du système, en comparant les résultats attendus avec les résultats obtenus [BD04].

Les tests manuels sont largement utilisés pour construire des scénarios et les exécuter, leurs inconvénients résident dans les erreurs produites lors de la construction des tests, surtout dans le cas d'un grand nombre de tests. C'est ce qui a conduit à la construction automatique des scénarios de tests. Plusieurs études ont montré que la génération automatique des scénarios de tests permet la vérification rapide des propriétés et la détection des contre-exemples :

- Tests à partir des diagrammes UML [Fle99] : Un modèle est représenté par un ensemble de cas d'utilisation et de diagrammes de séquences. Les diagrammes décrivant le modèle sont compilés en un modèle de simulation. Puis, pour un scénario donné, on teste le modèle de simulation avec les événements d'entrée et on compare la trace obtenue avec le scénario concret.
- Tests à partir de gardes des transitions [OA99] : Une technique permettant de tester les différentes conditions apparaissant dans les gardes des diagrammes d'états-transitions.
- Tests pour des systèmes concurrents et non déterministes [Gue01] : Une proposition pour la génération automatique de tests permettant de vérifier la conformité d'une implantation vis-à-vis de sa spécification. L'algorithme de synthèse des cas de tests a été spécialement conçu pour traiter des systèmes fortement concurrents et non déterministes.

Avantages et inconvénients

La génération de tests est un moyen efficace pour la vérification des propriétés, mais elle reste limitée aux systèmes ayant des réactions complètement déterministes et séquentielles, et disposant d'un nombre limité d'états.

Dans le cadre de la formalisation de modèles UML/OCL critiques, la vérification par génération de tests n'est pas suffisante, car elle ne permet pas de balayer tous les cas théoriques. Cependant, la vérification par génération de tests reste indispensable pour accepter des propriétés dont les preuves ne sont pas réalisables par machine (voir exemple, section 1.3.1 page 27).

3.4 Vérification par réseau de Petri

Un réseau de Petri [Mur89] est un graphe biparti orienté qui permet la modélisation du comportement des systèmes dynamiques. Les nœuds de graphe sont des places (marquées par un nombre de jetons) et des transitions reliées par des arcs étiquetés. Le marquage de toutes les places d'un réseau de Petri à un instant donné représente l'état du système, qui se modifie dynamiquement suite aux franchissements des transitions du réseau de Petri.

Ainsi, pour vérifier un système dynamique par un réseau de Petri, les propriétés du système sont exprimées en termes des propriétés du réseau de Petri (comme la bornitude de places, la vivacité d'un réseau de Petri, invariant de places, ...) correspondant, puis sont vérifiées et prouvées en utilisant des mécanismes comme la réduction des réseaux, l'algèbre linéaire ou des model-checkers.

Les aspects dynamiques d'UML (diagrammes d'états-transitions) sont naturellement modélisés par des réseaux de Petri [PL99, ACK12]. L'idée consiste à traduire les états d'un diagramme d'états-transition en places et les actions comportementales (qui se produisent suite au déclenchement des transitions d'un diagramme d'états-transitions) sont traduites en transitions. La vérification du système consiste alors à analyser les états (les marquages de places) du réseau de Petri généré, en particulier les invariants sur les marquages de ses places.

Avantages et inconvénients

Le réseau de Petri est un outil assez efficace et largement adopté pour la vérification des propriétés dynamiques des systèmes industriels. Cependant, il reste limité à des systèmes ayant un comportement déterministe et sans explosion combinatoire d'états. Dans la transformation d'UML vers les réseaux de Petri, la notion d'identité d'objet et le transfert de messages inter-objets (collaborations d'objets) ne sont pas représentés, ce qui limite l'approche à des spécifications relativement statiques.

3.5 Transformation inverse : du formel vers UML

La transformation des spécifications formelles vers UML s'intègre dans le cadre de l'ingénierie inverse ("reverse engineering"). Ceci permet aussi de rendre une spécification formelle compréhensible par les utilisateurs UML qui ne sont pas familiarisés avec les méthodes formelles. En effet, une partie de la difficulté des méthodes formelles réside dans la compréhension des notations sur lesquelles elles s'appuient [BH95].

De B vers UML

C'est principalement cette difficulté que les travaux de dérivation de B vers UML [HTVW02, Akr06] ont cherché à combler. Une autre proposition de dérivation d'une spécification B vers UML a été aussi proposé [Voi04] pour permettre une traduction partielle et systématique de spécifications B en diagrammes de classes et d'états-transitions.

Cependant, les règles proposées pour la transformation d'une spécification B en UML restent partielles, seul un sous-ensemble assez réduit des constructions B est considéré. Ceci est dû au fait que la méthode B couvre toutes les étapes de développement, par contre UML se limite au niveau

modélisation. De plus, les transformations de B vers UML ignorent l'utilisation des contraintes OCL, ce qui limite l'intérêt de telle transformation.

D'Alloy vers UML/OCL

Une démarche de transformation d'Alloy vers UML (Alloy-to-UML) [SAB09] a été aussi proposée pour la traduction des instances (générées par l'Analyseur d'Alloy) d'un modèle Alloy en diagrammes d'objets UML. Cette transformation complète l'outil UML2Alloy [ABGR10], et permet à partir d'une spécification Alloy générée par l'outil UML2Alloy de renvoyer des contre-exemples (trouvés par l'analyseur d'Alloy) exprimés en UML/OCL. Ainsi, la combinaison de deux outils (UML2Alloy et Alloy-to-UML) permet aux développeurs UML/OCL de vérifier les modèles sans connaître Alloy et son analyseur. Ceci est assez pratique pour les équipes uniquement intéressés à l'utilisation d'UML pour la modélisation des logiciels.

Nous trouvons aussi une transformation dans les deux sens entre UML/OCL et Alloy [CGR13] qui donne plus de flexibilité pour la génération d'un diagramme de classes UML et de contraintes OCL à partir d'une spécification Alloy.

De FoCaLiZe vers UML/OCL

Enfin, Une transformation automatique des spécifications FoCaLiZe vers des diagrammes de classes UML a aussi été proposée dans le cadre du projet EDEMOI [DÉDG08a]. Pour atteindre cet objectif, les auteurs considèrent un sous-ensemble de diagrammes de classes d'UML et proposent un profil UML pour FoCaLiZe qui représente tous les aspects (spécification et implémentation) d'un développement FoCaLiZe. En particulier, tous les éléments (représentation, signatures, fonctions définies et propriétés) et les fonctionnalités (héritage, paramétrisation et substitution des paramètres) d'une espèce sont supportées. Les propriétés d'espaces sont transformées en contraintes OCL (invariants) de classes.

L'objectif de cette transformation est de permettre l'intégration et l'application de plusieurs techniques d'ingénieries et de méthodes formelles pour l'analyse et la vérification des règlements de la sécurité des aéroports (au niveau terrestre).

Le modèle de transformation de FoCaLiZe vers UML/OCL [DÉDG08a] a été implémenté via le développement d'une feuille de style ("stylesheet") XSLT spécifiant les règles de transformation d'une spécification FoCaLiZe au format FoCaLiZeDoc (voir section 1.5, page 31) vers un modèle UML exprimé en XMI (le standard XML de modèles UML).

Discussion

Considérant les travaux de transformation dans les deux sens (UML/OCL $\leftrightarrow$ formel), une sérieuse question est posée concernant l'équivalence de deux modèles (formel et non formel). En effet, les deux approches n'ont pas le même raisonnement ni le même objectif, il faudrait plutôt parler des correspondances entre les propriétés des deux approches. Cependant, jusqu'à présent, aucune tentative n'a été proposée pour la vérification formelle (via la preuve des théorèmes) des propriétés des deux approches. Toutes les propositions de transformation sont énoncées à partir d'expérimentations.

Dans le cas particulier de la méthode Alloy, une transformation bidirectionnelle est réalisable, car Alloy (comme UML) est un outil qui se limite au niveau modélisation (ne génère pas de code exécutable). En utilisant les outils UML2Alloy et Alloy-to-UML, une spécification Alloy est générée à partir d'un modèle UML/OCL, puis le modèle UML/OCL est reproduit en utilisant la transformation inverse. Le but consiste à générer un code exécutable à partir d'UML/OCL, en utilisant d'autres outils d'ingénierie de logiciels.

Par contre, avec la méthode B, la transformation bidirectionnelle n'est pas toujours claire. Ceci est dû aux divergences dans le raisonnement de spécification entre les formalismes B et UML/OCL d'un côté, et des détails de constructions B que UML n'autorise pas, de l'autre côté.

3.6 Comparaison et Conclusion

La formalisation et la vérification des modèles UML/OCL dépendent de la capacité et de la puissance de la méthode formelle utilisée. Trois aspects fondamentaux caractérisent une méthode formelle : les fonctionnalités orientés objets (architecturales et conceptuelles) supportées, le paradigme de son langage de programmation et les techniques de vérification et de preuves utilisées.

D'abord, sur l'axe des fonctionnalités orientés objets, les méthodes formelles s'appuient sur l'utilisation des mathématiques et de la logique. Ceci impose souvent, des restrictions fortes sur les constructions que UML/OCL permet [Gue01] tels que l'héritage multiple, la liaison tardive, la génération des modèles liés aux classes paramétrées et la propagation des contraintes OCL via ces fonctionnalités. Certaines fonctionnalités UML sont directement supportées par des constructeurs propres aux méthodes formelles et d'autres fonctionnalités sont indirectement formalisées via des spécifications mathématiques et logiques.

Concernant le paradigme de programmation, la plupart des méthodes formelles utilisent un paradigme impératif. Cependant, le paradigme fonctionnel est beaucoup fiable dans un contexte formel. Dans la programmation impérative, il y a toujours des éventuels effets de bords lors d'exécution des programmes.

Côté preuve et vérification, certaines méthodes formelles utilisent la génération de tests et le model-checker, d'autres méthodes utilisent la preuve de théorèmes. Cependant, il est souvent plus efficace de combiner les deux approches pour permettre une vérification idéale, selon la taille du système modélisé.

Une récapitulation sur les trois axes discutés ci-dessus, vis-à-vis les principales méthodes formelles utilisées dans la formalisation de modèles UML/OCL, référencées au cours du présent chapitre, est présentée dans la table suivante.

Méthodes Formelles :
Fonctionnalités Orientées Objets, Techniques de preuves et Paradigmes de programmations.

	Fonctionnalités architecturales et conceptuelles								Vérification		Paradigme de programmation	
	Encapsulation	Héritage simple	Héritage multiple	Redéfinition de méthodes	Liaison tardive	Spécification des modèles paramétrés	Substitution des paramètres formels par des paramètres effectifs en cours de spécification	Propagation de propriétés via l'héritage ou la paramétrisation	Modèle checker ou génération de tests	Prouveur de théorème	Impératif	Fonctionnel
B	X	X				X				X	X	
Alloy	X	X				X			X			
Maude	X	X				X	X		X	X	X	
HOL-OCL	X	X		X	X	X				X		X
FoCaLiZe	X	X	X	X	X	X	X	X	X	X		X

Deuxième partie

De UML/OCL vers FoCaLiZe

Introduction

Dans la première partie de cette thèse, nous avons présenté les concepts sous-jacents de l'environnement FoCaLiZe, les ensembles d'UML/OCL supportés par notre étude et les différentes approches de formalisations des modèles UML/OCL. Dans cette partie, nous aborderons notre propre démarche de transformation d'UML/OCL vers FoCaLiZe. Pour ceci, nous adoptons une approche interprétée (voir section 3.2, page 60), basée sur la définition des règles de transformations. Nous commençons (chapitre 4) par la transformation de diagrammes de classes, c'est l'axe de tous les diagrammes UML et la structure la plus riche de tous les diagrammes UML. Notons que la transformation de diagrammes de classes vers FoCAL (l'ancienne dénomination de l'environnement FoCaLiZe) a été initiée dans mon travail de magister en 2007 [Abb07]. Puis (chapitre 5), nous détaillons la transformation des contraintes OCL attachées aux éléments d'un diagramme de classes. Ensuite (chapitre 6), nous exploitons les règles de transformation de diagrammes de classes et de des contraintes OCL pour l'élaboration d'une démarche générale pour la génération et la vérification d'un code FoCaLiZe à partir d'un modèle UML/OCL. Enfin (chapitre 7), nous décrivons la transformation et la vérification d'un diagramme d'états-transitions en utilisant les résultats obtenus dans les trois premiers chapitres de cette partie.

Tout au long de cette partie, nous utiliserons les exemples de modèles UML/OCL présentés en chapitre 2 (voir section 2.5, page 52) pour illustrer nos règles de transformation d'UML/OCL vers FoCaLiZe. Enfin, pour générer des expressions FoCaLiZe qui correspondent aux expressions UML/OCL, nous utiliserons aussi des bibliothèques FoCaLiZe fournies avec le compilateur (comme `basics.fcl`, `orders.fcl`, ...) et des bibliothèques spécifiques (que nous avons développé dans le cadre de ce travail de thèse) pour traiter les expressions UML/OCL de types `Real`, `String` et `Collection`.

Chapitre 4

Transformation de diagrammes de classes

Dans ce chapitre, nous allons détailler les règles de transformation d'un diagramme de classes vers FoCaLiZe [ABYR14b, ABYR14a]. Ainsi, nous suivons une approche progressive, en commençant par la description des règles de transformation des éléments (d'un diagramme de classes) jusqu'aux règles de transformation d'un diagramme complet. Nous présentons d'abord les règles de transformation des membres d'une classe, les attributs et les opérations. Puis, nous concluons avec la règle de transformation d'une classe simple, sans relations avec d'autres classes. Ensuite nous entameons la transformation de mécanismes UML qui invoquent plusieurs classes, à savoir les mécanismes d'héritage, de dépendance, de paramétrage (UML template), de substitution des paramètres formels par des paramètres effectifs et d'association binaire. Enfin, nous utilisons toutes les règles définies pour établir la règle générale de transformation des classes (diagramme de classes).

Avant de commencer, nous rappelons la structure générale d'une classe telle qu'elle est définie par notre ensemble d'UML (voir section 2.2, page 40) :

```
[public][final | abstract][«class – stereotype»] class  $c\_n$ 
    ( $\mathbb{P}_{c\_n}$ ) binds  $\mathbb{T}_{c\_n}$  depends  $\mathbb{D}_{c\_n}$  inherits  $\mathbb{H}_{c\_n} = \mathbb{A}_{c\_n} \mathbb{O}_{c\_n}$  end
```

avec

- $\mathbb{P}_{c_n}$ une liste de déclarations des paramètres formels,
- $\mathbb{T}_{c_n}$ une liste de substitutions des paramètres formels par des paramètres effectifs,
- $\mathbb{H}_{c_n}$ une liste de noms des classes dont la classe c_n hérite,
- $\mathbb{D}_{c_n}$ une liste de noms des classes dont la classe c_n dépend,
- $\mathbb{A}_{c_n}$ une liste d'attributs et
- $\mathbb{O}_{c_n}$ une liste d'opérations.

Notations et conventions

Tout au long des chapitres suivants, nous utilisons les notations et conventions suivantes :

- Pour tout élément e du modèle UML, nous dénotons $\llbracket e \rrbracket$ sa transformation en FoCaLiZe ;
- Pour toute classe nommée c_n , on lui fait correspondre l'espèce nommée s_n ;
- Pour un élément UML nommé e_n , nous maintenons le même nom pour sa transformation en FoCaLiZe.

4.1 Attributs et opérations

Les convergences entre les concepts UML et FoCaLiZe nous ont amené à utiliser la similitude entre classe et espèce. La transformation proposée découle naturellement de cette similitude.

Avant d'aborder la transformation d'attributs et d'opérations, nous présentons la transformation des expressions de types UML, utilisées pour typer les attributs et les opérations d'une classe.

4.1.1 Expressions de types

Nous distinguons l'utilisation des expressions de types suivants : types primitifs, types d'énumérations, types d'intervalles et les classes des modèles UML. Certains types UML sont directement modélisés par des types équivalents prédéfinis en FoCaLiZe, et d'autres types sont transformés via la création de nouveaux types en FoCaLiZe.

4.1.1.1 Types primitifs

Les types primitifs d'UML (Integer, Real, Boolean, String et UnlimitedNatural) sont directement formalisés par des types correspondants en FoCaLiZe :

<code>[[Integer]]</code>	=	<code>int</code>
<code>[[Real]]</code>	=	<code>float</code>
<code>[[Boolean]]</code>	=	<code>bool</code>
<code>[[String]]</code>	=	<code>string</code>
<code>[[UnlimitedNatural]]</code>	=	<code>Unlimited_Nat</code>

Les types `int`, `float`, `bool` et `string` sont prédéfinis dans FoCaLiZe (dans `basics.fcl`). Par contre, Le type `Unlimited_Nat` est défini dans une bibliothèque support, que nous dédions aux transformations de modèles UML (voir annexe C), comme suit :

```
type nat =
  internal (* Internal#nat *)
  external
  | caml -> {* Nat *}
  | coq -> {* nat *}
;;

type unlimited_Nat = | Infini | I(nat) ;;

species Unlimited_Nat_spec = inherit Ordered_set;
representation = unlimited_Nat;

let make (x:unlimited_Nat):Self = x ;
let to_unlimited_Nat(x:Self): unlimited_Nat = x ;

let min : Self = I(0);
```

```

let max : Self = Infini;

theorem unlimited_Nat_spec : all x : Self,
  !geq(x, min) /\ !leq(x, max)
proof = assumed ;

let leq (u1 : Self, u2: Self): bool = match (u1, u2) with
| ( _ , Infini ) -> true
| ( I(x), I(y) ) -> (x <=0x y)
| ( _ , _ ) -> false ;

let element = I(0);
end;;

collection Unlimited_Nat = implement Unlimited_Nat_spec ; end;;

```

Les deux fonctions `geq` (supérieur ou égal) et `leq` (inférieur ou égal) sont héritées de l'espèce `Ordered_set`, prédéfinie dans FoCaLiZe. Ainsi, nous définissons le type d'énumération `unlimited_Nat` qui n'est rien d'autre que le type naturel ($I(0)$, $I(1)$, ...) augmenté d'une valeur particulière (`Infini`) modélisant l'infini. Puis, nous introduisons l'espèce `Unlimited_Nat_spec` qui utilise le type `unlimited_Nat` comme représentation et définissons une valeur minimale (`min`) et une valeur maximale (`max`). Le théorème `unlimited_Nat_spec` spécifie que toutes les entités de l'espèce `Unlimited_Nat_spec` sont comprises entre les valeurs `min` et `max`. Enfin, C'est l'implémentation de l'espèce `Unlimited_Nat_spec` (la collection `Unlimited_Nat`) qui correspond au type `UnlimitedNatural` d'UML.

4.1.1.2 Type d'énumérations

Un type d'énumération UML `enum_n` de valeurs $value_1 \dots value_k$ est transformé en un type d'énumération FoCaLiZe (au niveau top). En FoCaLiZe, 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 (voir section 1.2.2, page 23). Aussi, nous avons été amené à transformer une énumération `enum_n` en une espèce `Enum_n` pour prendre en charge des éventuelles utilisations comme paramètres d'autres espèces :

UML :

«enumeration» `enum_n` $value_1 \dots value_k$

FoCaLiZe :

type `enum_n` = | $\llbracket value_1 \rrbracket$... | $\llbracket value_k \rrbracket$;;

```

species Enum_n =
  representation = enum_n ;
  let to_enum (x:Self):enum_n = x ;
  let from_enum (x: enum_n ):Self = x ;

```

```
end ;;
```

Les fonctions `to_enum` et `from_enum` permettent de faire la conversion entre les entités de l'espèce *Enum_n* et les valeurs de l'énumération *enum_n*.

Exemple. La transformation de l'énumération *Day* (voir table 2.3, page 43) qui modélise les jours de la semaine donne lieu à une énumération (*day*) et une espèce (*Day*), comme suit :

<pre>public class «enumeration» Day = Monday Tuesday Wednesday Thursday Friday Saturday Sunday end</pre>	<pre>type day = Monday Tuesday Wednesday Thursday Friday Saturday Saturday ;; species Day = representation = day; let to_enum (x:Self):day = x ; let from_enum (x:day):Self = x ; end ;;</pre>
--	---

L'énumération FoCaLiZe *day* peut être utilisée que ce soit au niveau top ou à l'intérieur des espèces pour typer des signatures. Quant à l'espèce *Day*, elle sera utile pour la spécification des paramètres des nouvelles espèces.

4.1.1.3 Type d'intervalles

Un type d'intervalle *interval_n* (voir formule 2.6, page 43) est modélisé par une espèce, en spécifiant ses bornes inférieure (*lowerBound*) et supérieure (*upperBound*) par les valeurs correspondantes en FoCaLiZe :

UML :

```
«intervalType» interval_n =
  lowerBound = lowerValue
  upperBound = upperValue
```

FoCaLiZe :

```
species interval_n = inherit Ordered_set;
let lowerBound = [[lowerValue]] ;
let upperBound = [[upperValue]] ;
property interval_spec: all x: Self,
!geq(x, lowerBound) /\ !leq(x, upperBound) ;
end ;;
```

La propriété *interval_spec* garantit que tous les éléments de l'espèce *interval_n* sont inclus entre ses deux bornes (*lowerBound* et *upperBound*).

Exemple. L'intervalle *Floor* (voir table 2.4, page 43) modélise les étages d'un bâtiment (0..*top*). L'application de la règle de transformation des types intervalles sur l'intervalle *Floor* produit une espèce de même nom, comme suit :

<pre>«intervalType» Floor = lowerBound = 0 upperBound = top</pre>	<pre>= species Floor = inherit Ordered_set; let lowerBound = 0; let upperBound = top; property interval_spec: all x : Self, !geq(x, 0) /\ !leq(x, top) ; end;;</pre>
---	--

L'espèce **Floor** hérite de l'espace **Ordered_set**, ce qui garantit que ses entités sont soumises à une relation d'ordre. L'espèce **Floor** peut être utilisé (au niveau spécification) pour spécifier des nouvelles espèces, comme elle peut être utilisée comme un type FoCaLiZe via la création d'une collection.

4.1.1.4 Expressions de types en général

Une expression de type UML peut être suivie d'un intervalle (*typeExp* [*mult*]) spécifiant sa multiplicité (voir formules 2.2 et 2.3). Lorsque la multiplicité ([*mult*]) d'un type vaut [1..1] (la valeur par défaut), ses éléments sont des valeurs uniques, comme les éléments de type entier. Dans l'autre cas, ses éléments sont des valeurs multiples, une collection de valeurs, comme les éléments de type tableau d'entier. En général, une expression de type UML (suivie d'une multiplicité) est transformée en un type FoCaLiZe comme suit :

$$\llbracket typeExp\ [mult] \rrbracket = \begin{cases} \llbracket typeExp \rrbracket & \text{si } mult = 1..1 \\ \text{list}(\llbracket typeExp \rrbracket) & \text{si } mult \neq 1..1 \end{cases} \quad (4.1)$$

Nous utilisons le type FoCaLiZe **list()** pour transformer les types UML multivalués (de multiplicité différente de 1..1).

4.1.2 Attributs

UML est un langage de modélisation orienté objet inspiré de langages de programmation impératifs, le concept d'état mémoire est une notion fondamentale dans UML. Ainsi, la liste d'attributs $\mathbb{A}_{c_n}$ d'une classe c_n (voir formule 2.2, page 41) caractérise l'état mémoire de ses objets. Par contre, FoCaLiZe, un langage purement fonctionnel, n'utilise pas le concept d'état mémoire. En prenant en considération cette divergence entre UML et FoCaLiZe, nous distinguons deux approches pour la transformation des variables d'instances d'une classe.

La première consiste à fournir deux définitions (implémentations) dans l'espèce dérivée de la classe :

- L'implémentation de la représentation (type support) de l'espèce.
- L'implémentation de la fonction getter de chaque variable d'instance.

La seconde approche consiste à ne pas fournir la définition de la représentation de l'espèce dérivée, elle sera abstraite dans ce cas. Chaque variable d'instance sera uniquement modélisée par la

signature de sa fonction `getter`.

Bien que la première approche soit plus concrète, la seconde approche donne plus de flexibilité pour la transformation de l'héritage (multiple ou simple) entre les classes, car FoCaLiZe ne permet pas la redéfinition de représentations (types supports) d'espèces à travers l'héritage (voir section 1.1.1, page 16).

En utilisant la seconde approche, un attribut `attr_n` est formalisé par une signature dans l'espèce correspondante, comme suit :

UML :

`attr_n : typeExp [mult];`

FoCaLiZe :

`signature get_attr_n : Self -> [[ typeExp [mult]]];`

Exemple. Les attributs `name` et `age` de la classe `Person` (voir figure 6.2, page 105) sont transformés en signatures de l'espèce `Person` comme suit :

<code>attribute - name: String</code>	<code>signature get_name : Self -> string;</code>
<code>attribute - age: Integer</code>	<code>signature get_age : Self -> int;</code>

Un objet `o` de la classe `Person` est formalisé par une entité `e` de l'espèce correspondante. L'accès à la valeur d'un attribut de l'objet `o` (`o.name` ou `o.age`) correspond à l'invocation de la fonction `getter` correspondante dans l'espèce (`get_name(e)` ou `get_age(e)`).

4.1.3 Opérations

La liste d'opérations $\mathbb{O}_{c_n}$ déclarées dans une classe `c_n` (voir formule 2.3, page 41) définit des services à invoquer par les instances de la classe pour affecter leurs comportements. Ainsi, une opération de classe (spécifiée par son type de retour et par les types de ses paramètres) est transformée en signature (une fonction déclarée) de l'espèce correspondante. Nous distinguons plusieurs cas, selon la nature des paramètres et stéréotypes de l'opération :

UML :

$$op_st\ op_n \left(\begin{array}{l} dir_1\ p1_n : typeExp_1 [mult_1] \\ \dots \\ dir_k\ p_k_n : typeExp_k [mult_k] \end{array} \right) : returnType [mult]$$

FoCaLiZe :

`signature op_n :`

$$\left\{ \begin{array}{l}
\llbracket \text{TypeExp}_1[\text{mult}_1] \rrbracket \rightarrow \dots \rightarrow \llbracket \text{TypeExp}_k[\text{mult}_k] \rrbracket \rightarrow \text{Self}; \\
\quad \text{si } (op_st = \text{create}) \\
\text{Self} \rightarrow \llbracket \text{TypeExp}_1[\text{mult}_1] \rrbracket \rightarrow \dots \rightarrow \llbracket \text{TypeExp}_k[\text{mult}_k] \rrbracket \rightarrow \llbracket \text{returnType}[\text{mult}] \rrbracket; \\
\quad \text{si } (\text{returnType} \text{ est donné}) \\
\text{Self} \rightarrow \llbracket \text{TypeExp}_1[\text{mult}_1] \rrbracket \rightarrow \dots \rightarrow \llbracket \text{TypeExp}_m[\text{mult}_m] \rrbracket \rightarrow \\
\quad \llbracket \text{TypeExp}_{(m+1)}[\text{mult}_{(m+1)}] \rrbracket * \llbracket \text{TypeExp}_{(m+2)}[\text{mult}_{(m+2)}] \rrbracket * \dots * \llbracket \text{TypeExp}_k[\text{mult}_k] \rrbracket ; \\
\quad \text{si } (dir_j = \text{out}, j : (m+1)..k) \\
\text{Self} \rightarrow \llbracket \text{TypeExp}_1[\text{mult}_1] \rrbracket \rightarrow \dots \rightarrow \llbracket \text{TypeExp}_k[\text{mult}_k] \rrbracket \rightarrow \text{Self}; \\
\quad \text{sinon}
\end{array} \right.$$

Le premier cas désigne la transformation d'un constructeur de la classe, spécifiée par le stéréotype **create**. Puis, le deuxième cas représente la transformation d'une opération avec type de retour. Le troisième cas décrit la transformation d'une opération spécifiée d'une liste de paramètres de sortie (de type **out**). Enfin, le quatrième cas montre la transformation d'une opération qui change l'état de l'instance (sans valeur de retour).

Exemple. Les opérations **setAge** et **birthdayHappens** de la classe **Person** (voir figure 6.2, page 105) sont transformées en signatures de l'espèce correspondante comme suit :

<pre> operation + setAge(a:Integer) operation + birthdayHappens() operation <<create>> + newPerson(n:String, a:Integer) </pre>	<pre> signature setAge : Self -> int -> Self; signature birthdayHappens : Self -> Self; signature newPerson : string -> int -> Self; </pre>
--	--

Les opérations **setAge** et **birthdayHappens** correspondent au dernier cas dans la règle de transformation des opérations, donnée ci-dessus. Elles changent l'état de l'objet, et ne disposent pas de retours. Par contre, l'opération **newPerson** correspond au constructeur de la classe, sa transformation génère la signature d'un constructeur de l'espèce dérivée de la classe.

4.2 Classes sans relations (cas particulier)

Dans le cadre de notre étude, nous nous limitons au niveau spécification. En effet, nous souhaitons transformer une classe UML qui ne contient que des spécifications (sans aucune implémentation concrète). Pour cette raison, le résultat de la transformation d'une classe UML en FoCaLiZe sera aussi une espèce abstraite, uniquement spécifiée par des signatures.

En utilisant les règles de transformation d'attributs et d'opérations données ci-dessus, une classe c_n (sans relations avec d'autres classes) spécifiée par une liste d'attributs $\mathbb{A}_{c_n}$ et une liste d'opérations $\mathbb{O}_{c_n}$ est transformée en une espèce s_n . Les attributs sont modélisés par les signatures de leurs getters et les opérations sont reportées en fonctions de l'espèce, spécifiées aussi par leurs signatures :

UML :

```

[public |private |protected] [final |abstract]
[«class-stereotype»] class c_n =
  A_c_n
  O_c_n

```

```

    end
FoCaLiZe :
    species s_n =
        [A_c_n]
        [O_c_n]
    end ;;

```

Exemple. Pour illustrer les règles de transformations de classes (avec attributs et opérations), nous présentons la transformation de la classe **Person** (voir figure 6.2, page 105) comme suit :

<pre> public class Person = attribute - name: String attribute - age: Integer operation + setAge(a: Integer) operation + birthdayHappens() operation <<create>> + newPerson(n: String, a: Integer) end </pre>	<pre> species Person = signature get_name : Self -> string; signature get_age : Self -> int; signature setAge : Self -> int -> Self; signature birthdayHappens : Self -> Self; signature newPerson : string -> int -> Self; end;; </pre>
---	--

Les attributs **age** et **name** sont spécifiés par la visibilité "private" (-). Ceci signifie que l'utilisateur final n'aura pas d'accès direct aux valeurs de ces attributs. Cette contrainte est bien assurée au niveau de FoCaLiZe, l'utilisateur final de l'espèce **Person** ne peut manipuler les éléments de sa représentation qu'à travers les fonctions getters (**get_age** et **get_name**).

Dans l'exemple, le constructeur de la classe **Person** est explicitement déclaré (une opération spécifiée par le stéréotype «create»). Même dans le cas où le constructeur d'une classe n'est pas donné, nous introduisons (d'une manière systématique) la fonction correspondante au niveau de l'espèce.

4.3 Relations de classes

Dans cette section, nous allons traiter la transformation des fonctionnalités conceptuelles et architecturales d'UML qui mettent en relation plusieurs classes. Nous rappelons les fonctionnalités supportées par notre étude, qui sont :

- Classes paramétrées par d'autres classes ;
- Héritage (multiple) avec redéfinition d'opérations et liaison tardive ;
- Dépendance (multiple) ;
- Substitutions de paramètres formels de classes (paramétrées) par des paramètres effectifs pour générer des modèles liés ("template binding") ;
- Associations binaires.

Comme dans la transformation d'attributs et d'opérations d'une classe, nous nous inspirons des similitudes entre UML et FoCaLiZe pour traiter la transformation de ces fonctionnalités. Certaines fonctionnalités (comme l'héritage, la dépendance et la substitution de paramètres de classes) découlent naturellement à travers les similitudes entre les deux approches (UML et FoCaLiZe). Par contre, d'autres fonctionnalités (comme les associations binaires) nécessitent une modélisation formelle via le développement des bibliothèques ("frameworks") spéciales.

4.3.1 Paramètres de classes

Une classe paramétrée c_n (template, voir formule 2.4, page 42) spécifiée par une liste de paramètres formels $(\mathbb{P}_{c_n}) : fp_1_n : typeExp_1, \dots, fp_k_n : typeExp_k$ est transformée en une espèce paramétrée, où chaque paramètre du template c_n est transformé en paramètre de l'espèce correspondante :

UML :

```
public class c_n (fp_1_n : typeExp_1, ..., fp_k_n : typeExp_k) = ... end
```

FoCaLiZe :

```
species s_n(fp_1_n is/in  $\llbracket typeExp_1 \rrbracket, \dots, fp_k_n$  is/in  $\llbracket typeExp_k \rrbracket$ )=...end;;
```

Nous distinguons deux traitements, selon qu'un paramètre formel est de type primitif ou de type **Class**. Dans le premier cas (paramètre de type primitif), il est transformé en paramètre d'entité en utilisant le mot clé **in** et l'espèce modélisant le type primitif correspondant en FoCaLiZe (**UnlimitedNaturalCollection**, **IntCollection**, **BoolCollection**, **StringtCollection** et **FloatCollection**). Les paramètres formels de type **Class** sont transformés en paramètres de collections de type **Setoid**, en utilisant le mot clé **is** :

UML :

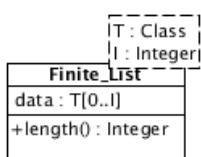
$fp_n : typeExp$

FoCaLiZe :

$$\left\{ \begin{array}{ll} fp_n \text{ is Setoid} & \text{si } typeExp = \mathbf{Class} \\ fp_n \text{ in IntCollection} & \text{si } typeExp = \mathbf{Integer} \\ fp_n \text{ in BoolCollection} & \text{si } typeExp = \mathbf{Boolean} \\ fp_n \text{ in StringtCollection} & \text{si } typeExp = \mathbf{String} \\ fp_n \text{ in RealCollection} & \text{si } typeExp = \mathbf{Real} \\ fp_n \text{ in UnlimitedNaturalCollection} & \text{si } typeExp = \mathbf{UnlimitedNatural} \end{array} \right.$$

Notons que dans notre démarche de transformation, l'espèce **Setoid** est une super-espèce (au sens de l'héritage en FoCaLiZe) de toutes les espèces, comme la classe **Class** (voir section 2.2.2.3, page 45) est une super-classe de toutes les classes en UML.

Exemple. La classe paramétrée **Finite_List** (voir la table ci-après) définit un template général qui modélise les listes finis d'éléments. La transformation de la classe **Finite_List** donne naissance à une espèce paramétrée (du même nom), comme suit :



```
public class Finite_List
  (T:Class, I:Integer)=
  attribute - data: T[0..I]
  operation + length():Integer
end
```

```
species Finite_List(T is Setoid,
  i in IntCollection)=
  signature get_data : Self -> list(T);
  signature length : Self -> int;
end;;
```

Comme le montre l'exemple, les espèces paramétrées en FoCaLiZe reflètent parfaitement les classes paramétrées en UML. En effet, les deux constructions (classe paramétrée et espèce paramétrée) sont des spécifications abstraites, et ne sont utilisables qu'à travers la substitution de leurs paramètres formels par des paramètres effectifs.

4.3.2 Substitution de paramètres de classes

Pour transformer une relation de dérivation d'une classe paramétrée c'_n à partir d'une classe paramétrée c_n (voir formule 2.9), nous utilisons le mécanisme d'héritage et le mécanisme de substitution de paramètres en FoCaLiZe. En utilisant le mécanisme d'héritage, FoCaLiZe permet la substitution de paramètres (formels) d'une espèce par des paramètres effectifs. Comme dans UML, FoCaLiZe impose que le type de chaque paramètre effectif (dans l'espèce liée) soit une sous-espèce du paramètre formel correspondant. Si le paramètre formel est de type **Setoid**, le paramètre effectif correspondant sera une espèce plus affinée, descendant de **Setoid**. Dans le cas d'un paramètre formel d'entité, le paramètre effectif correspondant est une valeur concrète.

Chaque paramètre formel fp_i_n du template c_n est substitué par un paramètre effectif ap_i_n dans c'_n . Si le paramètre formel est de type primitif, on lui associe un paramètre effectif d'entité en utilisant l'espèce modélisant le type primitif correspondant en FoCaLiZe (**IntCollection**, **BoolCollection**, **StringtCollection**, **FloatCollection** et **UnlimitedNaturalCollection**). Si le paramètre formel est de type **Class**, le paramètre effectif sera un paramètre de collection, dans l'espèce correspondante :

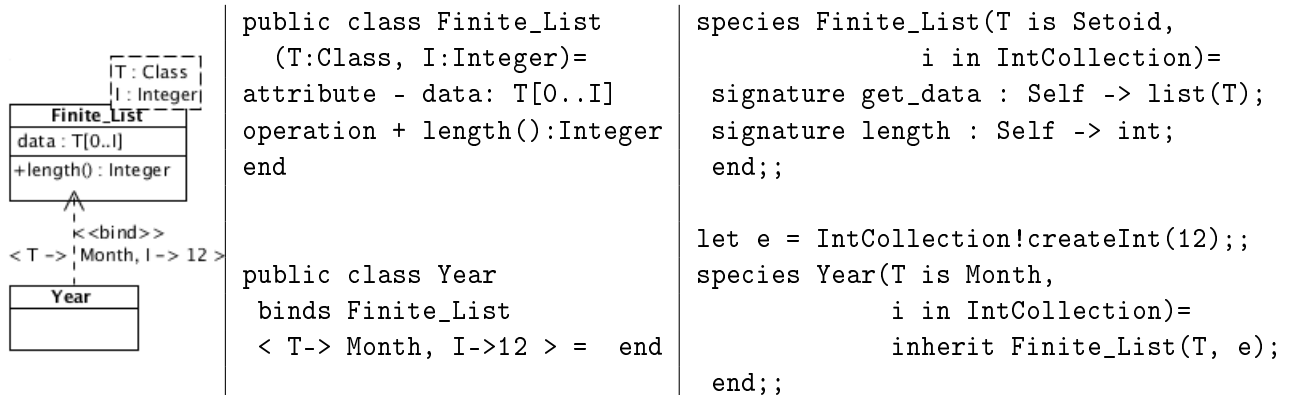
UML :

```
public class c_n (fp_1_n : typeExp_1, ..., fp_k_n : typeExp_k) = ... end
public class c'_n bind c_n <fp_1_n -> ap_1_n, ..., fp_k_n -> ap_k_n> = ... end
```

FoCaLiZe :

```
species s_n(fp_1_n is/in [[typeExp_1]], ..., fp_k_n is/in [[typeExp_k]]) = ... end;;
species s'_n(ap_1_n is/in [[typeExp_1]], ..., ap_k_n is/in [[typeExp_k]]) =
  inherit s_n(ap_1_n, ..., ap_k_n) ... end;;
```

Example. L'exemple suivant présente la transformation de la classe **Year** (une liste de 12 mois), dérivée de la classe paramétrée **Finite_List** (modélisant les listes finies) par substitution de ses paramètres formels par des paramètres effectifs ($T \rightarrow \text{Month}$ et $I \rightarrow 12$) :



Notons que la transformation proposée supporte aussi la substitution partielle des paramètres. Dans cet exemple, la liste de substitution des paramètres (**T** -> **Month** et **I** -> **12**) provoque la substitution de tous les paramètres formels par des paramètres effectifs. Ainsi, la classe résultante (**Year**) est une classe concrète qui peut être implémentée et instanciée. Dans le cas d'une substitution partielle des paramètres, la classe résultante sera aussi une classe paramétrée. Côté FoCaLiZe, la substitution (partielle ou complète) des paramètres d'une espèce renvoie toujours une espèce paramétrée.

4.3.3 Héritage

En UML, l'héritage (multiple) permet à une sous-classe c_n d'acquérir tous les attributs, les opérations et les propriétés de ses super-classes $c_1_n, \dots, c_k_n$ (voir formule 2.1, page 40). FoCaLiZe autorise ces mêmes fonctionnalités entre les espèces (voir section 1.1.4, page 20). Ainsi, nous transformons l'héritage (multiple) entre les classes en relation d'héritage (multiple) entre les espèces, en utilisant la clause **inherit** de FoCaLiZe :

UML :

```
public class c_n inherits c_1_n, ..., c_k_n = ...end
```

FoCaLiZe :

```
species s_n( $\mathbb{P}_{s\_n}$ ) = inherit s_1_n( $\mathbb{P}_{s_1\_n}$ ), ..., s_k_n( $\mathbb{P}_{s_k\_n}$ ); ...end ; ;
```

où, $\mathbb{P}_{s_n}$ est la liste des paramètres de l'espèce s_n (dérivée de la classe c_n) et $\mathbb{P}_{s_1_n}, \dots, \mathbb{P}_{s_k_n}$ sont les listes de paramètres des espèces $s_1_n, \dots, s_k_n$ (dérivées de classes $c_1_n, \dots, c_k_n$).

Calcul de la liste des paramètres ($\mathbb{P}_{s_n}$)

La liste des paramètres de l'espèce s_n ($\mathbb{P}_{s_n}$) est l'union de la liste des paramètres propres à l'espèce s_n et les listes des paramètres des espèces $s_1_n, \dots, s_k_n$. Les paramètres propres à s_n sont obtenus par la transformation des paramètres de la classe c_n (P_{c_n}), en utilisant la règle de transformation des classes paramétrées (voir section 4.3.1). Les paramètres des espèces $s_1_n, \dots, s_k_n$ ($\mathbb{P}_{s_1_n}, \dots, \mathbb{P}_{s_k_n}$) sont disponibles au moment de l'héritage, et qui sont calculés de la même manière, en utilisant la règle de transformation des classes paramétrées. Ainsi, les paramètres de l'espèce s_n (P_{s_n}) sont obtenus par la formule :

$$\mathbb{P}_{s_n} = \bigcup_{i=1}^k \{\mathbb{P}_{s_i_n}\} \cup \{P_{c_n}\} \quad (4.2)$$

Notons que dans le cas où les classes $c_1_n, \dots, c_k_n$ ne sont pas paramétrées, la transformation est simplifiée comme suit :

UML :

```
public class c_n inherits c_1_n, ..., c_k_n = ...end
```

FoCaLiZe :

```
species s_n = inherit s_n1, ..., s_nk; ...end ; ;
```

Exemple. La classe **FStack** est définie par héritage de la classe **FArray** (voir figure 6.2, page 105). La transformation de ce modèle en une spécification FoCaLiZe est donnée comme suit :

<pre> public class FArray (T:Class, I:Integer) = attribute - data: T[0..I] operation + isFull():Boolean operation + isEmpty():Boolean operation + length():Integer operation <<create>> + newInstance(d:T[0..I]) end public class FStack inherits FArray = operation + head():T operation + push(t:T) operation + pop() end </pre>	<pre> species FArray (Obj is Setoid, i in IntCollection) = inherit Setoid; signature get_data : Self -> list(Obj); signature isFull : Self -> bool ; signature isEmpty : Self -> bool ; signature length: Self -> int ; signature newInstance : list(Obj) -> Self; end;; species FStack (Obj is Setoid, i in IntCollection) = inherit FArray(Obj, i); signature head : Self -> Obj ; signature push : Obj -> Self -> Self; signature pop : Self -> Self; end;; </pre>
--	---

Les paramètres de la classe **FStack** (hérités de la classe **FArray**) sont cachés, ils sont déduits du contexte. Par contre, au niveau FoCaLiZe, les paramètres de l'espèce correspondante (**FStack**) sont explicites. L'espèce **FStack** n'a pas de paramètres propres, tout comme la classe n'a pas des paramètres propres. Ainsi, les paramètres de l'espèce (**FStack**) sont les paramètres de sa super-espèce (**FArray**), déterminés à partir des paramètres la classe **FArray**.

4.3.4 Dépendances

Une relation de dépendance spécifie qu'une classe cliente c_n a besoin de classes fournisseurs $c_{1_n}, \dots, c_{k_n}$ pour spécifier ses propres opérations (voir formule 2.8, page 45). Le mécanisme de paramétrisation d'espèces en FoCaLiZe (voir section 1.2.2, page 23) joue le même rôle que la relation de dépendance en UML. En effet, une espèce est paramétrée d'une autre espèce afin de l'utiliser pour développer ses propres méthodes.

Ainsi, la dépendance entre la classe cliente c_n et les classes fournisseurs $c_{1_n}, \dots, c_{k_n}$ est formalisée par la paramétrisation de l'espèce s_n (dérivée de la classe c_n) par les espèces $s_{1_n}, \dots, s_{k_n}$ (dérivées des classes $c_{1_n}, \dots, c_{k_n}$) :

UML :

```
public class c_n depends c_{1_n}, ..., c_{k_n} = ...end
```

FoCaLiZe :

```
species s_n( $\mathbb{P}_{s\_n}, \mathbb{P}_{s_{1\_n}}, \dots, \mathbb{P}_{s_{k\_n}}, fp_{1\_n}$  is  $s_{1\_n}(\mathbb{P}_{s_{1\_n}}), \dots, fp_{k\_n}$  is  $s_{k\_n}(\mathbb{P}_{s_{k\_n}})$ ) = ...end;;
```

où, $\mathbb{P}_{s_{1_n}}, \dots, \mathbb{P}_{s_{k_n}}$ sont les listes de paramètres des espèces $s_{1_n}, \dots, s_{k_n}$ (dérivées des classes $c_{1_n}, \dots, c_{k_n}$).

Notons que les paramètres de l'espèce s_n sont déterminés en utilisant des techniques similaires à celles utilisées dans le traitement de l'héritage (voir formule 4.2).

Dans le cas où les classes $c_n, c_1_n, \dots, c_k_n$ ne sont pas paramétrées, la transformation est simplifiée comme suit :

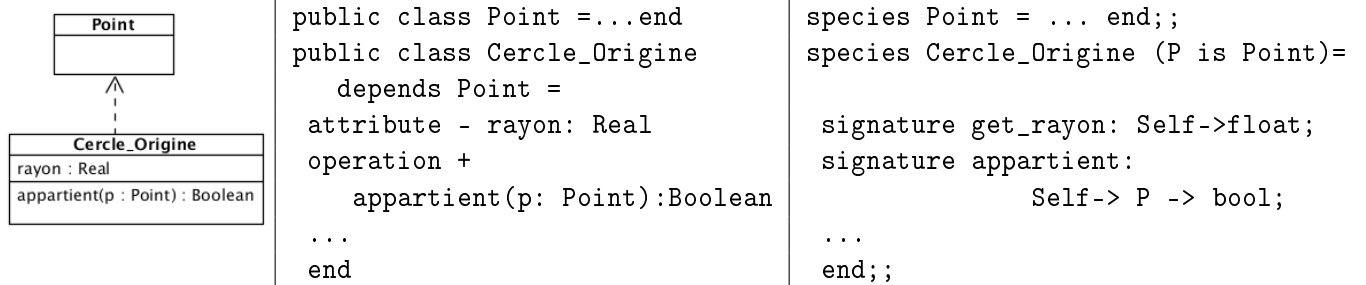
UML :

```
public class  $c_n$  depends  $c_1_n, \dots, c_k_n = \dots$ .end
```

FoCaLiZe :

```
species  $s_n$  ( $fp_1_n$  is  $s_1_n, \dots, fp_k_n$  is  $s_k_n$ ) = ...end;;
```

Exemple. Soit l'exemple de la relation de dépendance entre la classe **Cercle_Origine** et la classe **Point** présentée en table 2.7 (page 45). La transformation de ce modèle donne naissance à l'espèce **Cercle_Origine** paramétrée par l'espèce **Point** (dérivée de la classe **Point**), comme suit :



L'espèce **Cercle_Origine** peut utiliser toutes les méthodes (y compris la représentation et les propriétés) de l'espèce **Point**, pour développer et spécifier ses propres méthodes.

4.3.5 Associations binaires

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.

Ces espèces sont prédéfinies dans notre bibliothèque de développement FoCaLiZe comme suit :

```

(* L'espèce Association *)
species Association
    (R is Range ,
     Left is Setoid , Left_Set is List(Left), multl in R ,
     Right is Setoid, Right_Set is List(Right), multr in R,
     Dir is Direction, d in Dir ) =

signature left_of_right : Right -> Left_Set ;
    
```

```

signature right_of_left : Left -> Right_Set ;

property multl_prop : all ri :
  Right, R!mult_leq( Left_Set!size(left_of_right(ri)), R!upper(multl) )/\
  R!mult_geq ( Left_Set!size(left_of_right(ri)), R!lower(multl) ) ;
property multtr_prop : all le :
  Left, R!mult_leq( Right_Set!size(right_of_left(le)), R!upper(multtr) )/\
  R!mult_geq ( Right_Set!size(right_of_left(le)), R!lower(multtr) ) ;
end;;

(* L'espèce Range : modélise les multiplicités *)
species Range =
  representation = Unlimited_Nat * Unlimited_Nat ;
  let make_range(l :Unlimited_Nat , u : Unlimited_Nat ):Self = (l, u) ;
  let lower ( r : Self):Unlimited_Nat = fst(r) ;
  let upper ( r : Self):Unlimited_Nat = snd(r) ;
  end;;

collection Range_collection = implement Range; end;;

(* L'espèce Direction : modélise la direction de navigabilité *)

type ass_direction = | LeftToRight| RightToLeft | Both ;;

species Direction =
  representation = ass_direction;
  let make_direction (d:ass_direction): Self = d ;
  let to_direction (d : Self):ass_direction = d ;
  end;;

collection Direction_collection = implement Direction; end;;

```

L'espèce **Association** est paramétrée par les deux espèces dérivées des deux classes en association (**Left** et **Right**), les multiplicités des deux extrémités de l'association (**multl** et **multtr**), la direction de navigabilité de l'association (**d**) et par des ensembles effectifs d'entités (**Left_Set** et **Right_Set**) de deux espèces **Left** et **Right**.

Une association binaire est modélisée par deux fonctions permettant de naviguer dans l'un ou les deux sens de l'association, selon la direction de navigabilité (voir section 2.2.2.4, page 46). Ces deux fonctions sont modélisées par les fonctions **left_of_right** et **right_of_left** de l'espèce **Association**. Les deux propriétés **multl_prop** et **multtr_prop** sont déclarées dans l'espèce **Association** pour satisfaire les contraintes induites par les multiplicités d'une association. Les collections **Range_collection** et **Direction_collection** implémentent les espèces **Range** et **Direction**. Elles sont utilisées pour créer des entités d'espèces sous-jacentes.

Ainsi, l'association binaire ass_n entre les classes c_{left_n} et c_{right_n} , caractérisée par :

- une direction de navigabilité ass_dir ,
 - une multiplicité gauche MIU_{left} ($lower_{left}..upper_{left}$) et
 - une multiplicité droite MIU_{right} ($lower_{right}..upper_{right}$),
- est modélisée par une espèce ass_n qui hérite de l'espèce générale **Association**, comme suit :

UML :

```
association ass_n ass_dir = c_left_n [lower_left..upper_left],
                        c_right_n [lower_right..upper_right] end
```

FoCaLiZe :

```
(** Au niveau top **)
```

```
(* Transformation de la direction de navigabilité *)
```

$$\llbracket ass_dir \rrbracket = \begin{cases} \text{LeftToRight} & \text{si } ass_dir = \text{LeftToRight} \\ \text{RightToLeft} & \text{si } ass_dir = \text{RightToLeft} \\ \text{Both} & \text{si } ass_dir = \text{Both} \end{cases}$$

```
let d = Direction_collection!make_direction( $\llbracket ass\_dir \rrbracket$ );;
```

```
(* Transformation des multiplicités *)
```

$$\llbracket \{lower|upper\}_{\{left|right\}} \rrbracket = \begin{cases} \text{valeur naturelle} & \text{si } \{lower|upper\}_{\{left|right\}} \text{ est une valeur naturelle} \\ \text{Infini} & \text{si } \{lower|upper\}_{\{left|right\}} = \text{Infini} \end{cases}$$

```
(* Transformation de la multiplicité gauche *)
```

```
let multl = Range_collection!make_range
(Unlimited_Nat!make( $\llbracket lower_{left} \rrbracket$ ), Unlimited_Nat!make( $\llbracket upper_{left} \rrbracket$ ));;
```

```
(* Transformation de la multiplicité droite *)
```

```
let multtr = Range_collection!make_range
(Unlimited_Nat!make( $\llbracket lower_{right} \rrbracket$ ), Unlimited_Nat!make( $\llbracket upper_{right} \rrbracket$ ));;
```

```
(** Génération de l'espèce qui modélise l'association binaire **)
```

```
species ass_n (  $\mathbb{P}_{s_{left\_n}}$  , Left is  $s_{left\_n}(\mathbb{P}_{s_{left\_n}})$ , Left_Set is List(Left),
 $\mathbb{P}_{s_{right\_n}}$  , Right is  $s_{right\_n}(\mathbb{P}_{s_{right\_n}})$ , Right_Set is List(Right) )=
inherit Association (Range_collection, Left, Left_Set, multl ,
Right, Right_Set, multtr , Direction_collection , d );
end;;
```

telle que,

- s_{left_n} et s_{right_n} sont les espèces dérivées des classes c_{left_n} et c_{right_n} .
- $\mathbb{P}_{s_{right_n}}$ et $\mathbb{P}_{s_{left_n}}$ sont les listes des paramètres des espèces s_{left_n} et s_{right_n} , obtenues via l'application de la règle de traitement des classes paramétrées (voir section 4.3.2).

Exemple. L'association **Review** (voir table 2.9, page 47) entre les classes **Person** et **Paper** modélise le processus de notification de papiers scientifiques (dans le cadre d'une conférence ou d'un journal). La spécification FoCaLiZe correspondante est donnée comme suit :

```

association Review Both =
  Person [3..3]

  Paper [0..Infini]
end

let d = Direction_collection!make_direction(Both);

let multl = Range_collection!make_range
(Unlimited_Nat!make(I(3)), Unlimited_Nat!make(I(3)));

let multr = Range_collection!make_range
(Unlimited_Nat!make(I(0)), Unlimited_Nat!make(Infini));

species Review(Left is Person , Left_Set is List(Left),
               Right is Paper, Right_Set is List(Right) )=
  inherit Association (Range_collection, Left,
                      Left_Set, multl,
                      Right, Right_Set, multr,
                      Direction_collection, d );

end;;

```

La direction de cette association est bidirectionnelle (**Both**). Ainsi, les deux fonctions modélisant une association (**left_of_right** et **right_of_left**) sont indispensables. Ce qui n'est pas le cas pour une association monodirectionnel (sa direction soit **LeftToRight** soit **RightToLeft**). Dans ce dernier cas, nous utilisons une variante de l'espèce générale **Association** disposant seulement de l'une des deux fonctions (**left_of_right** ou **right_of_left**).

4.4 Classes avec relations (cas général)

Après avoir étudié la transformation d'attributs, d'opérations et les relations de classes, nous décrivons maintenant la règle générale de transformation de classes.

Tout au long de ce chapitre, nous avons exploité les convergences (voir table 4.1) entre l'approche UML et l'approche FoCaLiZe pour définir les règles de transformation des éléments d'un diagramme de classe UML en FoCaLiZe.

TAB. 4.1 – Convergences entre les concepts UML et les concepts FoCaLiZe

UML	FoCaLiZe
Classe abstraite	Espèce
Paramètres formelles de classe	Paramètres d'espèce
Attributs de classe	Représentation (type support) d'espèce
Opérations de classe	Signatures d'espèce
Héritage (multiple)	Héritage (multiple)
Redéfinition d'opérations de classes	Redéfinition des fonctions d'espèce
Liaison tardive	Liaison tardive
Substitution de paramètres formels de classe par des paramètres effectifs ("template binding")	Substitution de paramètres d'espèce
Encapsulation d'objets de classe	Encapsulation d'entités d'espèce
Dépendance	Paramétrisation
Classe finale (spécifiée par le mot clé final)	Collection

En réalité, une classe c_n (spécifiée par une liste d'attributs $\mathbb{A}_{c_n}$ et une liste d'opérations $\mathbb{O}_{c_n}$) peut au même niveau, être paramétrée par liste des paramètres formels ($\mathbb{P}_{c_n}$), hériter d'autres

classes ($\mathbb{H}_{c_n}$), dépendre d'autres classes ($\mathbb{D}_{c_n}$) et être soi-même est une classe liée, dérivée par la substitution ($\mathbb{T}_{c_n}$) des paramètres formels (d'une autre classe paramétrée) par des paramètres effectifs. Le fait qu'une classe c_n est en relation (de différents types) avec plusieurs autres classes c_{i_n} , pose le problème d'ordre de transformation de ces classes. Par exemple, si une classe A hérite d'une classe B, cela implique que la transformation de B doit précéder la transformation de A. Pour traiter la transformation de ces relations dans un ordre correct, nous commençons par la génération d'une spécification UML textuelle qui doit être compatible avec la syntaxe proposée pour UML (voir figure 2.1) et structurée suivant les dépendances syntaxiques et sémantiques entre les classes. La classe c_n est syntaxiquement et sémantiquement dépendante d'une classe c_{i_n} dans les cas suivants :

- La classe c_n hérite de la classe c_{i_n} ($c_{i_n} \in \mathbb{H}_{c_n}$) ;
- La classe c_n est paramétrée par la classe c_{i_n} ($c_{i_n} \in \mathbb{P}_{c_n}$) ;
- La classe c_n est dépendante de la classe c_{i_n} ($c_{i_n} \in \mathbb{D}_{c_{i_n}}$) ;
- La classe c_n est une classe liée obtenue par la substitution des paramètres formels de la classe c_{i_n} par des paramètres effectifs ($c_{i_n} \in \mathbb{T}_{c_n}$) .

Ces dépendances définissent un **graphe de dépendances** qui va déterminer l'ordre de traitement des classes.

Ainsi, nous utilisons le graphe de dépendances, les règles de transformations élémentaires présentées dans les paragraphes précédents pour définir la règle générale de transformation de classes, comme suit :

UML :

```
[public |private |protected] [final |abstract]
[«class-stereotype»] class c_n (P_c_n) binds T_c_n depends D_c_n inherits H_c_n=
A_c_n
O_c_n
end
```

FoCaLiZe :

```
species s_n ([P_c_n] , [T_c_n] , [D_c_n]) = inherit Setoid, [H_c_n] ;
[A_c_n]
[O_c_n]
end ;;
```

La transformation d'un diagramme de classes UML en FoCaLiZe consiste à appliquer cette règle sur toutes les classes de diagramme, en suivant l'ordre de traitement induit par le graphe de dépendances.

Exemple. Comme exemple récapitulatif, nous présentons la transformation du modèle UML des tableaux et piles finis (voir figure 6.2, page 105). L'exemple est composé de classes paramétrées et utilise la majorité de relations de classes. Voici la spécification FoCaLiZe générée :

<pre> public class Person = attribute - name:String attribute - age:Integer operation + setAge(a:Integer) operation + birthdayHappens() operation <<create>> + newPerson(n:String, a:Integer) end public class FArray (T:Class, I:Integer) = attribute - data: T[0..I] operation + isFull():Boolean operation + isEmpty():Boolean operation + length():Integer operation <<create>> + newInstance(d:T[0..I]) end public class FStack inherits FArray = operation + head():T operation + push(t:T) operation + pop() end public class PersonStack binds FStack < T-> Person, I->100 > = end </pre>	<pre> species Person = inherit Setoid; signature get_name : Self -> string; signature get_age : Self -> int; signature setAge : Self -> int -> Self; signature birthdayHappens : Self -> Self; signature newPerson : string -> int -> Self; end;; species FArray (Obj is Setoid, i in IntCollection) = inherit Setoid; signature get_data : Self -> list(Obj); signature isFull : Self -> bool ; signature isEmpty : Self -> bool ; signature length: Self -> int ; signature newInstance : list(Obj) -> Self; end;; species FStack (Obj is Setoid, i in IntCollection) = inherit FArray(Obj, i); signature head : Self -> Obj ; signature push : Obj -> Self -> Self; signature pop : Self -> Self; end;; let e = IntCollection!newInstance(100); species PersonStack (T is Person, i in IntCollection)= inherit FStack(T, e); end;; </pre>
---	---

Dans cet exemple, les classes sont correctement ordonnées suivant leurs dépendances. Les classes **Person** et **FArray** apparaissent en premier, car elles n'ont pas de dépendances avec d'autres classes. Puis, vient la classe **FStack**, dépendant (via une relation d'héritage) de la classe **FArray**. Enfin, la classe **PersonStack**, dépendant de deux classes **FStack** et **Person** (via une relation de substitution des paramètres). La spécification FoCaLiZe générée est une hiérarchie d'espèces, aussi ordonnée suivant l'ordre d'apparition de classes dans le modèle UML d'origine. La transformation d'une spécification UML incorrectement ordonnée génère une spécification FoCaLiZe incorrectement ordonnée. Ceci conduit à une erreur lors de compilation de code FoCaLiZe correspondant.

4.5 Conclusion

Dans ce chapitre, nous avons proposé une transformation formelle des diagrammes de classes UML en FoCaLiZe. A partir d'un diagramme de classes UML, nous générons une spécification textuelle qui est conforme à la syntaxe proposée pour UML (voir figure 2.1) et structurée suivant les dépendances syntaxiques et sémantiques entre les éléments du modèle. Ensuite, nous produisons une spécification FoCaLiZe, où chaque classe correspond à une espèce, les attributs de classe

sont 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.

Plus précisément, la transformation proposée préserve de façon naturelle les traits suivants (de diagrammes de classes UML) :

- **L'héritage multiple avec redéfinition de méthodes et liaison tardive** : Nous dérivons une hiérarchie d'espèces qui reflète les relations de généralisation/spécialisation entre les classes.
- **Les classes paramétrées, dérivation des modèles liés aux classes paramétrées et dépendances en utilisant la paramétrisation d'espèces et la substitution de paramètres d'espèces** : Une espèce dérivée d'une classe peut être utilisée comme paramètre d'une autre espèce (dérivée d'une autre classe), même au niveau spécification (avant l'implémentation finale des espèces). Les mécanismes de collections et de liaison tardive en FoCaLiZe garantissent que toutes les méthodes apparaissant dans une espèce (utilisée comme paramètre formel) sont effectivement mises en œuvre.
- **L'association binaire, avec multiplicités aux extrémités de l'association, et direction de navigation** : Elle est modélisée par une espèce générale paramétrée par les espèces dérivées des deux classes aux extrémités de l'association et par des paramètres formels spécifiant les multiplicités et la direction de navigation de l'association.

La démarche proposée permet de faire cohabiter toutes ces fonctionnalités pour la spécification et la transformation d'une classe dans un modèle UML, en respectant les dépendances structurelles et syntaxiques entre les classes. Elle permet aussi de détecter toute incohérence due à une spécification incorrecte des dépendances entre les classes.

Chapitre 5

Transformation des contraintes OCL

Une contrainte OCL (voir section 2.4, 50) est une expression faisant intervenir des types et opérations du langage OCL.

Rappelons que les principaux types du langage OCL sont :

- Les types primitifs : `Integer`, `Boolean`, `Real` et `String` ;
- Les types d'énumérations : les énumérations définies dans un modèle UML ;
- Les types objets : issus de classes UML ;
- Les types de collections : `Collection(T)`, tel que `T` est un type OCL.

Les types d'énumérations et les types objets (classes) ont déjà été traités dans le chapitre précédent (voir section 4, page 73). En effet, toute énumération UML est aussi un type d'énumération OCL et toute classe UML est un type objet OCL.

Toutes les opérations OCL sur les types `Integer` et `boolean` sont prises en compte dans FoCaLiZe. Par contre, les opérations OCL sur les types `Real`, `String` et `Collection` ne sont pas fournies avec FoCaLiZe.

Ainsi, pour transformer des expressions OCL, nous avons dû construire des bibliothèques formalisant les expressions OCL de base qui ne sont pas fournies avec FoCaLiZe. En particulier, nous distinguons la construction des bibliothèques FoCaLiZe suivantes :

- La bibliothèque `real_operations` (voir annexe B.1) définit les opérations OCL sur le type `Real` en FoCaLiZe ;
- La bibliothèque `string_operations` (voir annexe B.2) définit les opérations OCL sur le type `String` en FoCaLiZe ;
- La bibliothèque `collection_operations` (voir annexe B.3) définit les opérations OCL sur le type `Collection` en FoCaLiZe.

Rappelons aussi que les contraintes OCL supportées par notre étude sont les invariants de classes et les pré/post-conditions des opérations de classes.

Ainsi, les contraintes OCL spécifiées dans le contexte d'une classe UML sont alors transformées en propriétés (`property`) de l'espèce correspondante.

Dans ce chapitre, nous détaillons la transformation des opérations OCL sur les types primitifs et les opérations OCL de collections, puis nous abordons la transformation des contraintes OCL [Abb14].

5.1 Expressions OCL sur les types primitifs

5.1.1 Expressions OCL sur le type Integer

On utilise les correspondances entre le type `Integer` du langage OCL et le type `int` de FoCaLiZe, toutes les opérations OCL sur le type `Integer` sont exprimées par des expressions FoCaLiZe sur le type `int`. La table 5.1 définit la correspondance entre les opérations des deux types :

TAB. 5.1 – Transformation des expressions OCL sur le type `Integer`

OCL	FoCaLiZe	Commentaire
n	n	n est une valeur entière.
$\alpha + \beta$	$\llbracket \alpha \rrbracket + \llbracket \beta \rrbracket$	α et β sont des expressions entières.
$\alpha - \beta$	$\llbracket \alpha \rrbracket - \llbracket \beta \rrbracket$	
$-\alpha$	$-\llbracket \alpha \rrbracket$	
$\alpha * \beta$	$\llbracket \alpha \rrbracket * \llbracket \beta \rrbracket$	
$\alpha \text{ div } \beta$	$\llbracket \alpha \rrbracket / \llbracket \beta \rrbracket$	
$\alpha \text{ mod } \beta$	$\llbracket \alpha \rrbracket \% \llbracket \beta \rrbracket$	
$\alpha.\text{min}(\beta)$	$\text{min}(\llbracket \alpha \rrbracket, \llbracket \beta \rrbracket)$	
$\alpha.\text{max}(\beta)$	$\text{max}(\llbracket \alpha \rrbracket, \llbracket \beta \rrbracket)$	
$\alpha.\text{abs}$	$\text{abs}(\llbracket \alpha \rrbracket)$	valeur absolue.
$\text{coll} \rightarrow \text{size}()$	$\text{size}(\llbracket \text{coll} \rrbracket)$	nombre d'éléments d'une collection.
$\text{coll} \rightarrow \text{count}(o)$	$\text{count}(\llbracket o \rrbracket, \llbracket \text{coll} \rrbracket)$	nombre d'occurrences d'un objet (o) dans une collection (coll).

Les opérations `size` et `count` sont des opérations OCL de collections (voir section 5.3, ci-dessous).

5.1.2 Expressions OCL sur le type Real

Le type réel est défini dans les deux langages : OCL (`Real`) et FoCaLiZe (`float`). Ainsi, les expressions OCL sur le type `Real` sont modélisées par des expressions équivalentes, en utilisant le type `float` de FoCaLiZe. Contrairement aux opérations OCL de type `Integer`, toutes les opérations OCL sur les type `Real` ne sont pas prédéfinies dans FoCaLiZe. Ainsi, pour toute opération OCL sur le type `Real`, nous définissons une opération correspondante dans la bibliothèque `real_operations`.

L'opérateur OCL d'addition réelle (+) est modélisé par la définition de l'opérateur `+f` dans la bibliothèque `real_operations`, comme suit :

```
let ( +f ) =
  internal float -> float -> float
  external
  | caml -> {*(+.) *}
  | coq -> { * Rplus * } ;;
```

La définition de l'opération FoCaLiZe `+f` est donnée au niveau top, en utilisant des opérations `+` du langage Ocaml et `Rplus` de Coq.

En raison du paradigme purement fonctionnel du langage FoCaLiZe, nous distinguons entre l'opération d'addition d'entiers (dénotée `+` : `int -> int -> int`) et l'opération d'addition de réels (dénotée `+f` : `float -> float -> float`).

Les autres opérations OCL sur les réels (`*`, `-`, `/`, ...) sont modélisées d'une manière similaire (voir annexe B.1).

La correspondance entre les opérations OCL sur le type `Real` et les opérations FoCaLiZe sur le type `float` (table 5.2), est donnée comme suit :

TAB. 5.2 – Modélisation des opérations OCL sur le type `Real`

OCL	FoCaLiZe
$x + y$	<code>x +f y</code>
$x - y$	<code>x -f y</code>
$x * y$	<code>x *f y</code>
a / b	<code>a /f b</code>
$a > b$	<code>a >f b</code>
$a \geq b$	<code>a >=f b</code>
$a < b$	<code>a <f b</code>
$a \leq b$	<code>a <=f b</code>
<code>a.max(b)</code>	<code>max_f(a , b)</code>
<code>a.min(b)</code>	<code>min_f(a , b)</code>
<code>a.abs</code>	<code>abs_f(a)</code>
<code>a.round</code>	<code>round_f(a)</code>
<code>a.floor</code>	<code>floor_f(a)</code>

5.1.3 Expressions OCL sur le type `String`

Comme le type réel, le type chaîne de caractères (`String` dans OCL / `string` dans FoCaLiZe) est défini dans les deux langages. Ainsi, les expressions OCL de type `String` sont formalisées par des expressions de type `string` de FoCaLiZe. La plupart des opérations OCL sur les types `String` (à l'exception de l'opération de concaténation de chaînes, `^`) ne sont pas fournies avec FoCaLiZe. Ainsi, pour toute opération OCL sur les type `String`, nous définissons une opération correspondante dans la bibliothèque `string_operations`.

L'opération `size()` (retournant la taille d'une chaîne de caractères) d'OCL est modélisée par la définition de l'opération `length()` dans la bibliothèque `string_operations`, comme suit :

```
let length =
  internal string -> int
  external
  | caml -> { * String.length *}
  | coq -> { * length *}
;;
```

La définition de l'opération FoCaLiZe `length()` est basée sur l'opération `length` du module OCaml `String` (`String.length`) et l'opération `length` de Coq.

L'opération OCaml `length` permettra de générer l'exécutable de la nouvelle opération (`length` de FoCaLiZe) et l'opération Coq permettra de prouver des propriétés sur cette opération.

Les autres opérations OCL sur le type `String` (extraction d'une sous-chaine, conversion en minuscule et conversion en majuscule) sont modélisées d'une manière similaire (voir annexe B.1).

En utilisant les opérations FoCaLiZe définies dans la bibliothèque `string_operations`, les expressions OCL sur le type `String` sont alors formalisées par des expressions correspondantes, comme suit (table 5.3) :

TAB. 5.3 – Modélisation des opérations OCL sur le type `String`

OCL	FoCaLiZe	Commentaire
<code>a.concat(b)</code>	<code>([a] ^ [b])</code>	concaténation de chaînes
<code>a.size()</code>	<code>length([a])</code>	longueur d'une chaîne
<code>a.subString(lower, upper)</code>	<code>subStr([a], lower, upper)</code>	l'extraction d'une sous chaîne
<code>a.toUpper()</code>	<code>toUpper([a])</code>	conversion en majuscule
<code>a.toLower()</code>	<code>toLower([a])</code>	conversion en minuscule

5.1.4 Expressions OCL sur le type Boolean

Les formules OCL (les expressions OCL de type `Boolean`) sont modélisées par des expressions FoCaLiZe de type `bool`. Comme dans le cas du type `Integer`, toutes les opérations OCL sur le type `Boolean` sont prédéfinies dans la bibliothèque `basics` de FoCaLiZe. Ainsi, nous détaillons la correspondance entre les expressions booléennes d'OCL et de FoCaLiZe dans la table suivante (table 5.4) :

TAB. 5.4 – Transformation des formules OCL

OCL	FoCaLiZe
true	true
false	false
not (ϕ)	$\sim\llbracket\phi\rrbracket$
ϕ and ψ	$\llbracket\phi\rrbracket \wedge \llbracket\psi\rrbracket$
ϕ or ψ	$\llbracket\phi\rrbracket \vee \llbracket\psi\rrbracket$
ϕ xor ψ	$\llbracket\phi\rrbracket \mid \langle \rangle \mid \llbracket\psi\rrbracket$
ϕ implies ψ	$\llbracket\phi\rrbracket \rightarrow \llbracket\psi\rrbracket$
if ϕ then ψ else φ	if $\llbracket\phi\rrbracket$ then $\llbracket\psi\rrbracket$ else $\llbracket\varphi\rrbracket$
let $x : \text{type} = \text{Exp}$ in ϕ	let $x = \llbracket\text{Exp}\rrbracket$ in $\llbracket\phi\rrbracket$
$\alpha = \beta$	$\llbracket\alpha\rrbracket = \llbracket\beta\rrbracket$
$\alpha < > \beta$	$\sim\llbracket\llbracket\alpha\rrbracket = \llbracket\beta\rrbracket\rrbracket$
$\alpha > \beta$	$\llbracket\alpha\rrbracket > \llbracket\beta\rrbracket$
$\alpha < \beta$	$\llbracket\alpha\rrbracket < \llbracket\beta\rrbracket$
$\alpha \geq \beta$	$\llbracket\alpha\rrbracket \geq \llbracket\beta\rrbracket$
$\alpha \leq \beta$	$\llbracket\alpha\rrbracket \leq \llbracket\beta\rrbracket$
allInstances $\rightarrow$ forAll ($x \mid \phi$)	all $x : \text{Self}, \llbracket\phi\rrbracket$
allInstances $\rightarrow$ exists ($x \mid \phi$)	ex $x : \text{Self}, \llbracket\phi\rrbracket$
$\text{coll} \rightarrow$ forAll ($x \mid \phi$)	forAll ($\llbracket\text{coll}\rrbracket, \llbracket\phi\rrbracket$)
$\text{coll} \rightarrow$ exists ($x \mid \phi$)	exists ($\llbracket\text{coll}\rrbracket, \llbracket\phi\rrbracket$)
$\text{coll} \rightarrow$ isEmpty ()	isEmpty ($\llbracket\text{coll}\rrbracket$)
$\text{coll} \rightarrow$ notEmpty ()	notEmpty ($\llbracket\text{coll}\rrbracket$)
$\text{coll} \rightarrow$ includes (o)	includes ($\llbracket o \rrbracket$)
$\text{coll} \rightarrow$ excludes (o)	excludes ($\llbracket o \rrbracket$)
$\text{coll}_1 \rightarrow$ includesAll (coll_2)	includesAll ($\llbracket\text{coll}_1\rrbracket, \llbracket\text{coll}_2\rrbracket$)
$\text{coll}_1 \rightarrow$ excludesAll (coll_2)	excludesAll ($\llbracket\text{coll}_1\rrbracket, \llbracket\text{coll}_2\rrbracket$)

où ϕ et ψ sont des formules OCL, α et β sont des expressions numériques (entières ou réelles), o est un objet du modèle et $\text{coll}, \text{coll}_1, \text{coll}_2$ sont des collections OCL.

La description et la transformation des opérations OCL sur le type **Collection** (**allInstances**, **forAll**, **exists**, **isEmpty**, **includes**, **excludes**, **includesAll** et **excludesAll**) seront détaillées dans la section 5.3.

5.2 Expressions OCL de type Collection

Pour un type OCL T (**Integer**, **Real**, UML class, ...), le type **Collection**(T) définit une famille de collections d'éléments de type T (voir section 2.4, page 50).

Le type **Collection**(T) est modélisé par la définition de l'espèce **OCL_Collection**(**obj is Setoid**) dans la bibliothèque **collection_operations**. Les opérations OCL de collections sont alors modélisées par des fonctions (signatures) dans l'espèce **OCL_Collection**. Toute signature de l'espèce **OCL_Collection** (modélisant une opération OCL de collections) est aussi spécifiée grâce à une propriété garantissant une sémantique correcte, par rapport à l'opération OCL d'origine.

La signature **includesAll** de l'espèce **OCL_Collection** modélise l'opération OCL de même nom.

Cette opération OCL possède deux paramètres de type `Collection(T)` et renvoie vraie si et seulement si tous les éléments de la deuxième collection sont inclus dans la première collection :

OCL :

```
(Collection(T) -> includesAll( Collection(T)) : Boolean
```

FoCaLiZe :

```
signature includesAll : Self -> Self -> bool;

property includesAll_specif: all c1 c2 : Self,
  (all x : Obj, (includes(x, c1) -> includes(x, c2))) <->
    includesAll(c1, c2);
```

La propriété `includesAll_specif` spécifie la signature `includesAll`. Elle permet de s'assurer que la sémantique de la signature `includesAll` correspond parfaitement à la sémantique de l'opération OCL d'origine.

Les autres opérations OCL de collections sont modélisées d'une manière similaire (voir annexe B.3).

Les opérations OCL (de collections) `forAll` et `exists`, lorsqu'elles s'appliquent sur la collection `C.allInstances` (où `C` est une classe du modèle), sont transformées via le quantificateur universel (`all`) et le quantificateur existentiel (`ex`) de FoCaLiZe, comme suit :

OCL :

```
(C.allInstances -> forAll(x | Exp : Boolean)) : Boolean
(C.allInstances -> exists(x | Exp : Boolean)) : Boolean
```

FoCaLiZe :

```
all x : [C], [Exp] ;
ex x : [C], [Exp] ;
```

En effet, dans ces cas, l'itération des opérations OCL (`forAll` et `exists`) s'effectue sur toutes les instances de la classe `C`.

La correspondance entre les opérations OCL de collections retournant des collections et celles de FoCaLiZe est donnée dans la table 5.5, suivante :

TAB. 5.5 – Transformation des opérations OCL retournant des collections

OCL	FoCaLiZe	Commentaire
<code>c_n.allInstances</code>	<code>all x : Self</code>	dans le contexte de l'espèce <code>s_n</code> dérivée de la classe <code>c_n</code>
<code>coll₁ -> intersection(coll₂)</code>	<code>intersection([coll₁], [coll₂])</code>	<code>coll₁</code> et <code>coll₂</code> sont des collections OCL <code>φ</code> est une formule OCL
<code>coll₁ -> union(coll₂)</code>	<code>union([coll₁], [coll₂])</code>	
<code>coll -> select(x φ)</code>	<code>select([coll], [φ])</code>	
<code>coll -> reject(x φ)</code>	<code>reject([coll], [φ])</code>	

Les opérations OCL de collections retournant des expressions entières ou booléennes sont traitées dans les tables 5.1 et 5.4, données ci-dessus.

En exploitant toutes les règles de transformation des opérations OCL de collections, la modélisation d'un type de collection OCL particulier est obtenue par héritage de l'espèce générale `OCL_Collection`, et en substituant son paramètre formel (`Obj is Setoid`) par une espèce du modèle :

OCL :

```
Collection(c_n)
```

FoCaLiZe :

```
species s_n_Collection(Obj is c_n) = inherit OCL_Collection(Obj) end;;
```

Exemple. Le type `Collection(Person)` d'objets de la classe `Person` est transformé en une espèce (`Person_Collection`) paramétrée par l'espèce `Person` (dérivée de la classe `Person`) et hérite de l'espèce générale `OCL_Collection`, comme suit :

```
Collection(Person) | species Person_Collection(Obj is Person) =
                      | inherit OCL_Collection(Obj) end;;
```

Rappelons qu'en FoCaLiZe, l'implémentation de l'espèce `Person_Collection` nécessite une implémentation préalable de son paramètre de collection (l'espèce `Person`).

5.3 Contraintes OCL

Toutes les contraintes OCL (invariants de classes et pré/post-conditions des opérations de classes) sont traduites en propriétés FoCaLiZe (**property**). Les contraintes OCL d'une même classe sont traduites en propriétés de l'espèce dérivée (de la classe). Puis, comme dans UML, les propriétés d'une espèce sont automatiquement propagées à travers l'héritage et la paramétrisation en FoCaLiZe.

5.3.1 Invariant

Un invariant OCL $\mathbb{E}_{inv}$ de la classe c_n (voir formule 2.15, page 50) est transformé en propriété FoCaLiZe dans l'espèce correspondante (s_n), comme suit :

OCL :

```
context c_n inv :  $\mathbb{E}_{inv}$ 
```

FoCaLiZe :

```
property inv_i : all e : Self,  $\llbracket \mathbb{E}_{inv} \rrbracket$  ;
```

La transformation de l'expression OCL décrivant l'invariant ($\mathbb{E}_{inv}$) vers FoCaLiZe s'inspire des règles de transformations d'expressions de base, présentées ci-dessus. Dans une expression OCL, l'accès à un attribut de la classe (en contexte) se traduit par l'invocation de la fonction `getter` modélisant l'attribut dans l'espèce correspondante. L'invocation d'une opération de classe dans une expression OCL se traduit par l'invocation de la fonction correspondante de l'espèce dérivée de la classe.

Au niveau OCL, les invariants de classes sont généralement énoncés sans identificateurs. Par contre, au niveau FoCaLiZe, les identificateurs des propriétés sont obligatoires. Ainsi, nous adoptons un mécanisme de génération automatique d'un identificateur unique (`inv_1`, `inv_2`, ...) à chaque propriété dérivée d'un invariant.

Exemple. Nous complétons, maintenant, la transformation de la classe **Person** (voir section 4.2, page 79) par la transformation de l'invariant de classe ($\text{age} > 0$, voir figure 6.2, page 105), comme suit :

<pre>public class Person = ... attribute - age:Integer end context Person inv : age > 0</pre>	<pre>species Person = ... signature get_age : Self -> int; property inv_1: all p:Self, (get_age(p) > 0); end;;</pre>
--	---

Notons que si une classe UML a plusieurs invariants OCL, la transformation produira plusieurs propriétés dans l'espèce correspondante.

5.3.2 Pré/post-conditions

Une pre-condition ($\mathbb{E}_{pre}$) et post-condition ($\mathbb{E}_{post}$) OCL d'une opération op_n d'une classe c_n (voir formule 2.16, page 52) spécifient que la post-condition doit être satisfaite après l'exécution de l'opération, lorsque la pre-condition est satisfaite juste avant l'exécution de l'opération. Ainsi, nous transformons une pré/post-condition (identifiée par pre_post_ident) en une implication ($pre_condition \Rightarrow post_condition$) dans l'espèce correspondante, comme suit :

OCL :

```
context c_n : : op_n(p1_n : typeExp1 ... p_k_n : typeExp_k) pre :  $\mathbb{E}_{pre}$  post :  $\mathbb{E}_{post}$ 
```

FoCaLiZe :

```
property pre_post_op_n_i :
all e : Self,
all p1_n :  $\llbracket typeExp_1 \rrbracket$  , ... , all p_k_n :  $\llbracket typeExp_k \rrbracket$  ,
 $\llbracket \mathbb{E}_{pre} \rrbracket \rightarrow \llbracket \mathbb{E}_{post} \rrbracket$  ;
```

Nous intégrons toujours la quantification $\text{all } e : \text{Self}$ afin d'utiliser l'entité e pour exprimer la transformation des formules $\mathbb{E}_{pre}$ et $\mathbb{E}_{post}$. Ces dernières sont exprimées par des opérateurs de différents types OCL, leurs transformations en FoCaLiZe ($\llbracket \mathbb{E}_{pre} \rrbracket$ et $\llbracket \mathbb{E}_{post} \rrbracket$) utilisent toutes les règles de transformation de classes, d'énumérations et de types primitifs (**Integer**, **Real**, **Boolean**, **String** et **Collection**), présentées ci-dessus.

Comme dans la transformation des invariants, un identificateur unique ($pre_post_op_n_1$, $pre_post_op_n_2$, ...) est attribué à toute propriété FoCaLiZe dérivée d'une pré/post-condition.

Exemple. La pré/post-condition de l'opération **push**($t : T$) de la classe **FStack** (voir figure 6.2, page 105) est transformée en propriété dans l'espèce correspondante comme suit :

<pre> public class FStack inherits FArray = operation + head():T operation + push(t:T) operation + pop() end context FStack :: push(t:T) pre : self.isEmpty() Post: let s = self.pop() in (s.isEmpty()) </pre>	<pre> species FStack (Obj is Setoid, i in IntCollection) = inherit FArray(Obj, i); signature head : Self -> Obj ; signature push : Obj -> Self -> Self; signature pop : Self -> Self; property pre_post_push_1: all e : Self, all t : Obj, isEmpty (e) -> let s = (push(t, e)) in isEmpty(pop(s)); end;; </pre>
--	---

Notons qu'une opération de classe peut avoir plusieurs pré/post-conditions, il en résulte plusieurs propriétés spécifiant la fonction correspondante dans l'espèce dérivée de la classe en contexte.

5.3.3 Post-condition utilisant @pre.

Dans une post-condition, il est parfois utile de désigner la valeur d'un attribut avant l'exécution de l'opération. Ceci est réalisé en postfixant le nom de l'attribut avec @pre (voir section 2.4.2, page 52).

Soit un objet o d'une classe c_n , et une opération op_n de cette classe. Suite à l'invocation de l'opération op_n sur l'objet o , nous distinguons deux états mémoires différents de l'objet o : l'état o_E avant l'invocation de op_n et l'état $o_{E'}$ après l'invocation de op_n .

Après l'invocation de l'opération op_n (l'objet o est à l'état $o_{E'}$), l'expression $o.attr_n@pre$ indique la valeur de l'attribut $attr_n$ (de la classe c_n) tel qu'il était à l'état o_E (avant le passage à l'état $o_{E'}$).

Au niveau de FoCaLiZe, les états o_E et $o_{E'}$ sont transformés en deux entités totalement différentes (e_1 et e_2), car FoCaLiZe (langage fonctionnel) distingue entre l'entité (e_1) qui invoque l'opération et l'entité (e_2) retournée par l'opération. Ainsi, la transformation de l'expression $o.attr_n@pre$ revient à invoquer le getter de l'attribut sur l'entité e_1 ($get_attr_n(e_1)$), comme détaillé dans la règle suivante :

OCL :

```

 $o_E.attr\_n$ 
 $o_{E'}.attr\_n$ 
 $o_{E'}.attr\_n@pre$ 

```

FoCaLiZe :

```

 $get\_attr\_n(\llbracket o_E \rrbracket)$ 
 $get\_attr\_n(\llbracket o_{E'} \rrbracket)$ 
 $get\_attr\_n(\llbracket o_E \rrbracket)$ 

```

Exemple. Lorsque l'anniversaire d'une personne est atteint, son âge s'incrmente. Cette contrainte (voir section 2.4.2, page 52) exprime une post-condition de l'opération `birthdayHappens()` de la classe `Person`, transformée comme suit :

<pre> context Person : :birthdayHappens() post : age = age@pre + 1 </pre>	<pre> property post_birthdayHappens: all x : Self, get_age(birthdayHappens(x)) = get_age(x) + 1; </pre>
---	---

La transformation de cette contrainte est réalisée dans le contexte de l'espèce **Person** (dérivée de la classe en **Person**).

5.4 Conclusion

Dans ce chapitre, nous avons proposé une démarche de transformation des contraintes OCL vers FoCaLiZe. Les contraintes OCL supportées sont les invariants de classes et les pré/post-conditions des opérations de classes.

A partir d'une spécification OCL conforme à la syntaxe de l'ensemble du langage OCL supporté (voir figure 2.2), nous produisons une spécification FoCaLiZe équivalente. Les expressions OCL de types primitifs sont transformées en des expressions de types correspondantes en FoCaLiZe et les expressions OCL de collections sont transformées via la création de l'espèce **OCL_Collection** qui modélise la plupart de opérations OCL de collections.

Plus précisément, la transformation proposée préserve les propriétés suivantes sur les contraintes OCL, vis-à-vis les fonctionnalités de diagrammes de classes :

- Toutes les propriétés (d'une espèce) dérivées de contraintes OCL sont propagées à travers le mécanisme d'héritage : Les propriétés d'une super-espèce sont aussi propriétés de ses héritiers (sous-espèces) ;
- Toutes les propriétés (d'une espèce) dérivées de contraintes OCL sont propagées à travers le mécanisme de paramétrage en FoCaLiZe : Les propriétés d'une espèce fournisseur (utilisée comme paramètre formel) peuvent être utilisées en toute sécurité par des espèces clientes (espèces paramétrées) ;
- Les propriétés d'une espèce dérivée d'une classe paramétrée (UML template) deviennent aussi propriétés des espèces dérivées des modèles liés ("bound models") correspondants.

Chapitre 6

Génération et vérification de code FoCaLiZe

La transformation d'un modèle UML/OCL, produit un code FoCaLiZe composé d'une hiérarchie d'espèces spécifiées par des signatures et des propriétés. Les propriétés sont dérivées des associations, de diagrammes d'états-transitions et des contraintes OCL. Ces propriétés génèrent des obligations de preuve lors de la compilation de la spécification FoCaLiZe générée. Pour décharger ces obligations, le développeur a besoin de fournir des définitions (implémentations) aux signatures dérivées. Nous rappelons, qu'une même signature peut avoir plusieurs définitions différentes correspondant à son type fonctionnel. La résolution des obligations de preuves garantit que les définitions (des signatures) proposées par le développeur implémentent correctement la spécification FoCaLiZe générée. La production du code exécutable OCaml n'aura lieu que lorsque le code soumis est complet et les obligations de preuve remplies.

Dans ce chapitre, nous présentons une démarche générale décrivant les étapes de génération et de vérification d'une spécification FoCaLiZe à partir d'un modèle UML/OCL, basée sur la démarche de transformation présentée dans les chapitres précédents.

6.1 Processus de vérification

Après la description des règles de transformation d'UML/OCL vers FoCaLiZe, nous pouvons maintenant définir un cadre ("framework") qui intègre un outil UML/OCL, les règles de transformation et l'environnement FoCaLiZe. Dans ce cadre, le développeur FoCaLiZe sera capable de compléter la spécification générée et de vérifier les propriétés dérivées, jusqu'à obtenir un code exécutable. Pour atteindre cet objectif, nous adoptons le processus de preuve suivant (voir figure 6.1) :

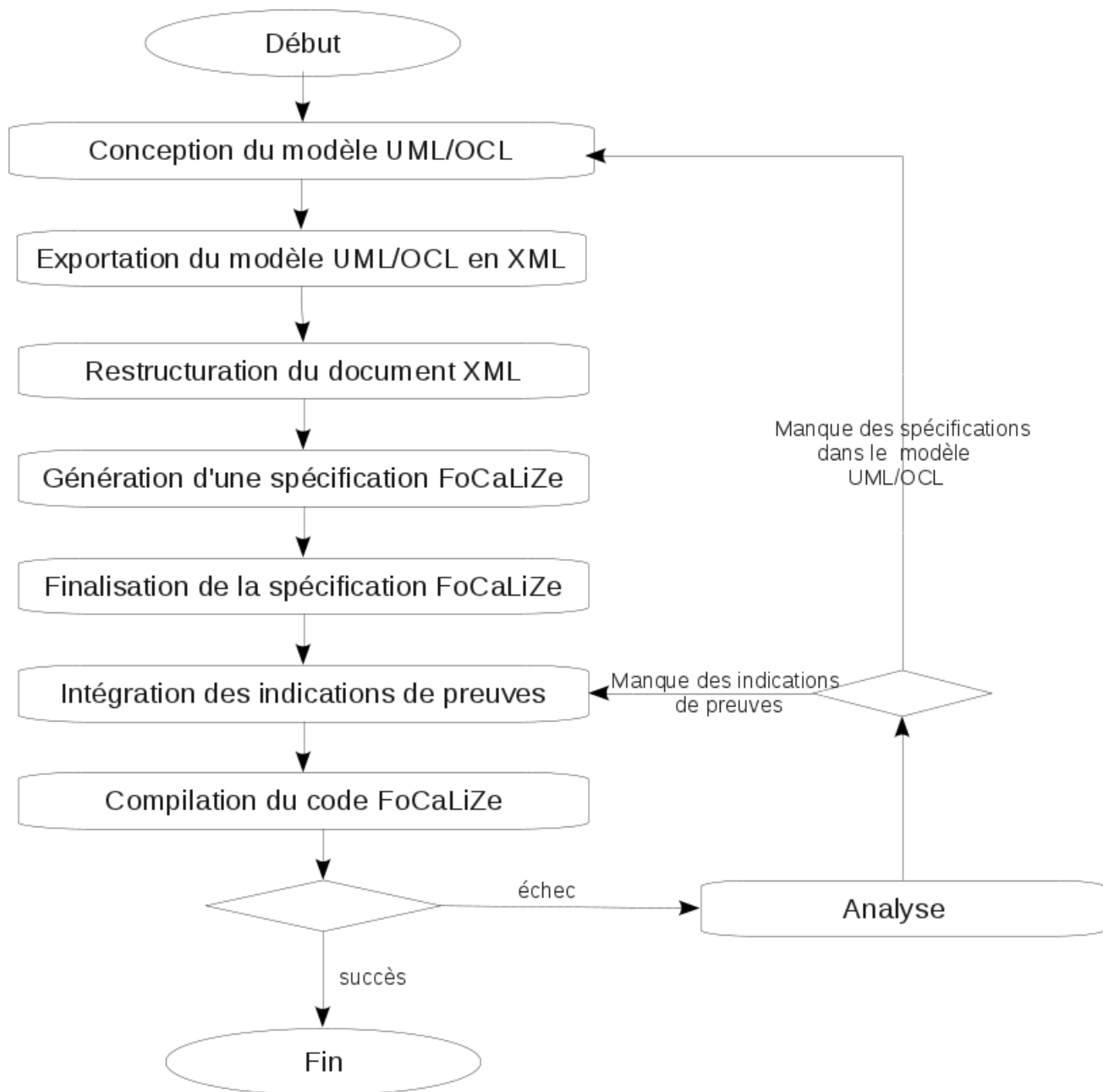


FIG. 6.1 – Processus de vérification des modèles UML/OCL par FoCaLiZe

Ainsi, pour produire, compléter et vérifier une spécification FoCaLiZe dérivée d'un modèle UML/OCL, on suit les étapes suivantes :

1. Conception du modèle UML/OCL en utilisant un outil UML2 qui supporte la spécification des contraintes OCL, comme Papyrus¹ ou UML2 Eclipse plug-in². Ces outils UML/OCL permettent l'exportation des modèles UML/OCL en spécifications textuelles au format XML, en utilisant le standard XMI [OMG15b].

¹Papyrus : <http://www.eclipse.org/papyrus/>

²UML2 Eclipse plug-in : <http://www.eclipse.org/modeling/mdt/downloads/?project=uml2>

2. Restructuration de la spécification textuelle obtenue en modifiant l'ordre d'apparition des éléments (XML), suivant les dépendances syntaxiques et sémantiques entre les éléments du modèle.
3. Génération d'une spécification FoCaLiZe à partir de la spécification obtenue dans l'étape précédente, en utilisant notre démarche de transformation.
4. Finalisation de la spécification FoCaLiZe obtenue en implémentant toutes ses méthodes abstraites (représentations et signatures).
5. Intégration des indications de preuves aux propriétés dérivées, en utilisant le langage de preuve de FoCaLiZe (FPL).
6. Compilation du code FoCaLiZe en utilisant la commande *focalizec*. Cette dernière invoque à son tour la commande *zvtov* pour lancer la recherche de preuves par Zenon.

Notons qu'en cas d'échec, le compilateur FoCaLiZe indique la ligne de code responsable de l'erreur. Dans ce cas, le développeur interagit (c'est la phase d'analyse dans le diagramme de la figure 6.1) selon la cause de l'échec indiqué au niveau du message d'erreur. S'il s'agit d'une incohérence ou manque de spécification au niveau du modèle UML/OCL, le développeur doit corriger et/ou compléter le modèle UML/OCL, puis recommencer le cycle de transformation et de vérification. Un échec peut également se produire si l'utilisateur introduit des indications de preuves inappropriées ou insuffisantes pour prouver une propriété. Dans ce cas, le développeur réintroduit des indications qui permettent à Zenon de trouver un chemin vers la preuve.

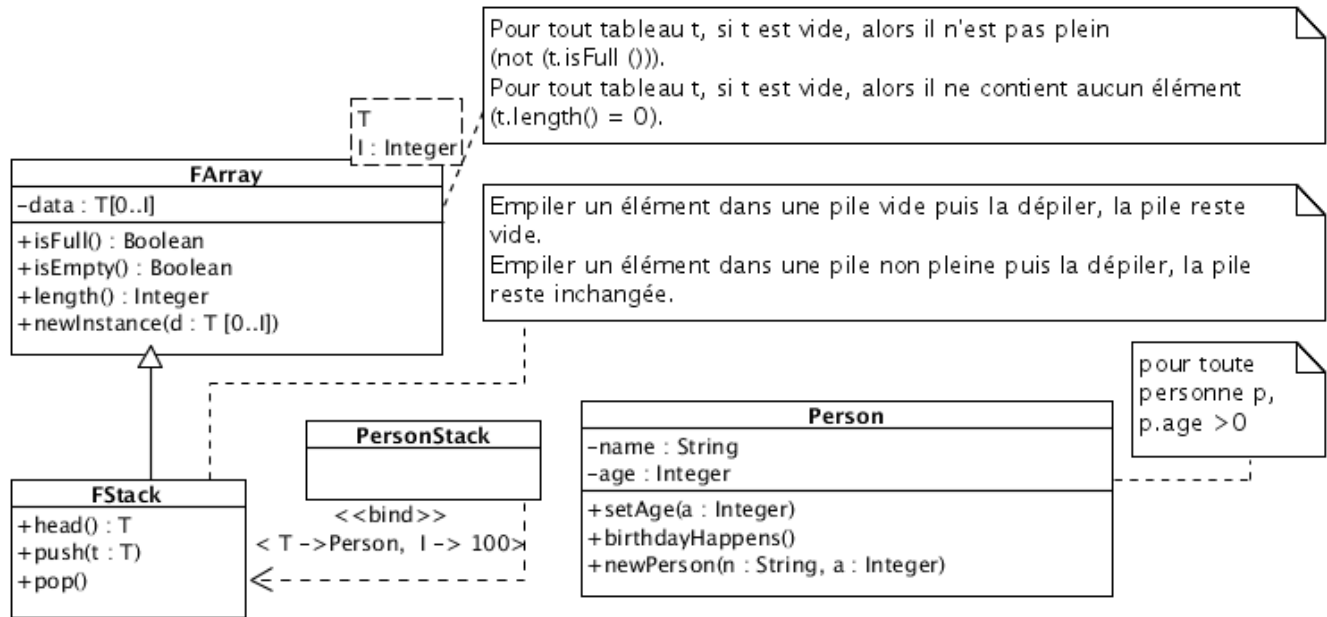
En plus des propriétés explicites dérivées du modèle UML/OCL, le compilateur FoCaLiZe rajoute des obligations de preuve implicites lors de l'analyse des espèces dérivées. En particulier, la redéfinition d'une fonction (à travers l'héritage) annule toutes les preuves liées à son ancienne définition et toute définition récursive doit être accompagnée par la preuve de sa terminaison [HFPD16].

6.2 Etude de cas

Pour mieux illustrer cette démarche de preuve, nous présentons la preuve d'une propriété de notre exemple de tableaux et piles finis présenté en section 2.5, page 52.

Etapes 1 & 2 :

Nous rappelons la modélisation graphique de l'exemple de tableaux et piles finis (voir figure 6.2) ainsi que sa spécification textuelle :



Diagrammes de classes

```

public class Person =
  attribute - name:String
  attribute - age:Integer
  operation + setAge(a:Integer)
  operation + birthdayHappens()
  operation <<create>> + newPerson(n:String, a:Integer)
end

public class FArray (T:Class, I:Integer) =
  attribute - data: T[0..I]
  operation + isFull():Boolean
  operation + isEmpty():Boolean
  operation + length():Integer
  operation <<create>> + newInstance(d:T[0..I])
end

public class FStack inherits FArray =
  operation + head():T
  operation + push(t:T)
  operation + pop()
end

public class PersonStack binds FStack < T-> Person, I->100 > = end
  
```

Contraintes OCL

```
context FArray
```

```

inv : self.allInstances() -> forAll(s|s.isEmpty() implies s.length()=0)
inv : self.isEmpty() implies not(self.isFull())

context FStack :: push(t:T)
pre : self.isEmpty()
post: let s = self.pop() in (s.isEmpty())

context FStack :: push(t:T)
pre : not(self.isFull())
post: self.pop() = self@pre

context Person   inv : age > 0

```

Etape 3 :

L'application de notre démarche de transformation (d'UML/OCL vers FoCaLiZe) sur la description textuelle du modèle de tableaux et piles finies génère la spécification FoCaLiZe suivante :

```

(* Transformation of the class Person *)
species Person = inherit Setoid;
signature get_name : Self -> string;
signature get_age : Self -> int;
signature setAge : Self -> int -> Self;
signature birthdayHappens : Self -> Self;
signature newPerson : string -> int -> Self;

    (* mapping of OCL constraints on Person *)
property inv_1: all p:Self, (get_age(p) > 0);
end;;

(* Transformation of the class FArray *)
species FArray (Obj is Setoid, i in IntCollection) = inherit Setoid;
signature get_data : Self -> list(Obj);
signature isFull : Self -> bool ;
signature isEmpty : Self -> bool ;
signature length: Self -> int ;
signature newInstance : list(Obj) -> Self;

    (* mapping of OCL constraints on FArray *)
property inv_1 : all s : Self, isEmpty(s) -> (length(s) = 0);
property inv_2 : all s : Self, isEmpty(s) -> ~~(isFull(s));
end;;

(* Transformation of the class FStack *)
species FStack (Obj is Setoid, i in IntCollection) = inherit FArray(Obj, i);
signature head : Self -> Obj ;

```

```

signature push : Obj -> Self -> Self;
signature pop : Self -> Self;

(* mapping of OCL constraints on FStack *)
property pre_post_push_1: all e : Obj, all s : Self,
  isEmpty (s) -> isEmpty(pop(push(e, s)));
property pre_post_push_2 : all e : Obj , all s : Self ,
  ~~( isEmpty(s)) -> equal(pop (push(e, s)), s);
end;;

(* creation of the entity modeling the integer value 100 *)
let e = IntCollection!newInstance(100);;

(* Transformation of the class PersonStack *)
species PersonStack (T is Person, i in IntCollection)=inherit FStack(T, e);
(* Definitions of undefined methods: next step *)
end;;

```

Etape 4 :

Un développeur FoCaLiZe complète la spécification abstraite de l'étape précédente en fournissant des définitions (implémentations concrètes) aux représentations et signatures non définies :

```

(* The species Person_Def provides definitions for the species Person methods *)
species Person_Def = inherit Person;
representation = string * (int) ;

let newPerson(s :string, a:int ):Self = (s, a);
let get_name(p:Self):string = fst(p);
let get_age(p:Self):int = snd(p);
let setAge (p:Self, a:int):Self = newPerson(get_name(p), a);
let birthdayHappens (p:Self):Self = newPerson(get_name(p), (get_age(p)+ 1));

(* The functions element and equal are inherited from Setoid *)
let equal (x: Self, y : Self): bool = (get_name(x) = get_name(y)) ;
let element:Self = ("toto", 1);
end;;

species PersonStack (T is Person, i in IntCollection)= inherit FStack(T, e) ;
representation = list(T);
let newInstance(x: list(T))Self = x ;
let get_data (x: Self):list(T)= x;

let isEmpty(x:Self):bool = (length(x) = (IntCollection!to_int(i))) ;

```

```

let isEmpty(x:Self):bool = (length(x) = 0) ;
let head(x: Self):T = match get_data(x) with
| [] -> focalize_error ("The stack is empty")
| y :: z -> y;
let push (o: T, y: Self):Self =
  if ~~(isFull(y)) then newInstance(o::get_data(y))
  else focalize_error ("The stack is full");
let pop(x: Self):Self = match get_data(x) with
| [] -> focalize_error ("The stack is empty")
| y :: z -> newInstance(z);

let rec length(x: Self):int = match (x) with
| [] -> 0
| y :: z -> 1 + length(z);

(* The functions element and equal are inherited from Setoid *)
let equal(x:Self, y:Self): bool = (x = y);
let element:Self = [] ;
(* Proofs of inherited properties: next step *)
end;;

```

Etape 5 :

Afin de prouver les propriétés du modèle, le développeur complète le code par l'introduction des indications de preuves ("proof hints"). Par souci de simplicité, nous présentons uniquement la preuve d'une pré/post-condition (`pre_post_push_1`) et d'un invariant (`inv_1`) :

```

species PersonStack (T is Person, i in IntCollection)= inherit FStack(T, e);
(* Proof of the property inv_1 *)
proof of inv_1 = by definition of isEmpty;

(* Proof of the property pre_post_push_1 *)
proof of pre_post_push_1 =
  <1>1 assume e : T, s : Self,
    hypothesis H1 : isEmpty ( s ),
    prove isEmpty (pop(push(e, s)))
  <2>1 prove equal( pop ( push ( e , s ) ) , s )
    by hypothesis H1 property inv_2 , pre_post_push_2
  <2>2 prove isEmpty( pop ( push ( e , s ) ) )
    by hypothesis H1 step <2>1 definition of equal
    property equal_symmetric
  <2>3 qed by step <2>2
  <1>2 conclude ;
...

```

```
end;;
```

La preuve de l'invariant (`inv_1`) est une preuve feuille, simplement achevée par le dépliage de la définition de la méthode `isEmpty` dans la preuve. Par contre, la preuve de la pré/post condition (`pre_post_push_1`) est composée de plusieurs étapes et fait référence à plusieurs hypothèses et propriétés du modèle.

Etape 6 :

Enfin, cet étape consiste à lancer la compilation de spécifications FoCaLiZe (enregistré dans un fichier *arrayStack.fcl*). Une fois la commande *focalizec* (*focalizec arrayStack.fcl*) appelée, le compilateur FoCaLiZe affiche les messages :

```
focalizec arrayStack.fcl
Invoking ocamlc...
>> ocamlc -I /usr/local/lib/focalize -c arrayStack.ml
Invoking zvtov...
>> zvtov -zenon /usr/local/bin/zenon -new arrayStack.zv
Invoking coqc...
>> coqc -I /usr/local/lib/focalize -I /usr/local/lib/zenon arrayStack.v
```

Aucune erreur ne se produit, ceci montre que la compilation, la génération du code et la vérification par Coq sont réalisées avec succès.

6.3 Déploiement

Pour implémenter notre démarche de transformation d'UML/OCL vers FoCaLiZe, nous avons utilisé le langage de transformation XSLT (Extensible Stylesheet Language Transformations) [W3C17]. Le XSLT est un standard W3C (World Wide Web Consortium) pour la transformation des documents XML vers différents formats comme HTML, XML, texte, et PDF.

Le choix de XSLT est motivé par sa puissance et par sa simplicité dans la déclaration des règles de transformation. Un autre avantage de l'XSLT est le fait d'être intégré dans la plupart des langages de programmation comme Java, C++, C#, PHP, Python, etc. En effet, tous ces langages implémentent les processeurs XSLT et XPath.

Ainsi, nous avons développé une feuille de style XSLT spécifiant les règles de transformation d'un document UML/OCL exporté au format XMI (généré par l'outil graphique **Papyrus**) vers FoCaLiZe. Le résultat de la transformation est un document texte qui contient le code FoCaLiZe correspondant au modèle UML/OCL. La spécification FoCaLiZe générée peut être directement lue et compilée par FoCaLiZe (voir figure 6.2).

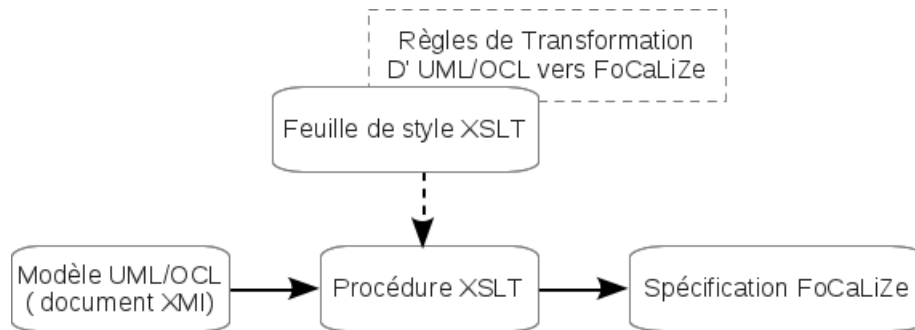


FIG. 6.2 – Transformation systématique d’UML/OCL vers FoCaLiZe

Nous avons utilisé l’environnement **Eclipse**³ pour incorporer les quatre modules de la figure 6.2, (l’outil UML/OCL, la feuille de style XSLT, la procédure XSLT et FoCaLiZe) dans le même environnement. Ainsi, sous le même environnement Eclipse, le développeur construit un modèle graphique UML/OCL, procède à sa transformation et lance la commande de compilation de FoCaLiZe. Suivant les messages retournés par le compilateur FoCaLiZe, le développeur complète et vérifie le code toujours dans le même environnement.

Des informations supplémentaires sur notre outil de transformation (**UML2FOC**) sont maintenant disponibles sur le site <http://www.univ-eloued.dz/uml2foc/>.

6.4 Conclusion

Dans ce chapitre nous avons présenté un processus pour l’exploitation de notre démarche de transformation d’UML/OCL vers FoCaLiZe. Ce processus est réalisé en plusieurs étapes dans un cadre (framework) regroupant un outil UML/OCL graphique, l’environnement FoCaLiZe et les règles de transformation d’UML/OCL vers FoCaLiZe. Le processus proposé permet une transformation systématique d’un modèle UML/OCL vers FoCaLiZe. Puis, un développeur FoCaLiZe pourra compléter la spécification abstraite obtenue par la définition de ses méthodes et la preuve de ses propriétés, jusqu’à la génération du code exécutable. Ce processus de définitions et de preuves aboutit à des espèces complètes qui donnent naissances à des collections exploitables par l’utilisateur final en toute sécurité. Ainsi, le modèle de développement et de vérification présenté dans ce chapitre décrit un framework MDE qui produit un code exécutable cohérent à partir d’un modèle UML/OCL. Une implémentation de la démarche de transformation est réalisée par le développement d’une feuille de style XSLT décrivant les règles de transformation.

³<http://www.eclipse.org/>

Chapitre 7

Transformation de diagrammes d'états-transitions

Dans le cadre de cette thèse, nous avons choisi les diagrammes d'états-transitions pour modéliser l'aspect dynamique d'un modèle, en faisant apparaître la collaboration d'objets et les modifications apportées à leurs états en réaction à des événements (voir section 2.3, page 48). La transformation d'un diagramme d'états-transitions vers FoCaLiZe [ABYR18] consiste à formaliser les états, les transitions et les conditions de garde. Nous adoptons les correspondances suivantes entre le diagramme d'états-transitions et FoCaLiZe :

- Les états d'un diagramme d'états-transitions sont modélisés par un type d'énumération ;
- La condition de garde d'une transition correspond à une formule de la logique du premier ordre, en FoCaLiZe ;
- Les transitions sont formalisées par des fonctions constructeurs dans l'espèce dérivée de la classe en question.

La condition de garde d'une transition est une expression du langage OCL (voir section 2.3). Elle est formalisée par une expression FoCaLiZe, en utilisant les règles de transformation des expressions OCL, présentées dans le chapitre précédent. Dans ce qui suit, nous détaillons la transformation des états et des transitions. Puis, nous présentons une démarche pour la vérification des diagrammes d'états-transitions.

7.1 Transformation d'états

Les états de diagramme d'états-transitions (voir section 2.3.1, page 48) d'une classe c_n sont définis par leurs noms, comme suit :

$$ST_{c_n} = \{st_1_n, \dots, st_k_n\}$$

L'ensemble d'états est modélisé par un type d'énumération en FoCaLiZe (au niveau top) :

UML :

$$\{st_1_n, \dots, st_k_n\}$$

FoCaLiZe :

$$\text{type } states_sum_type = \mid \llbracket st_1_n \rrbracket \ . \ . \ . \ \mid \llbracket st_k_n \rrbracket \ ;;$$

Chaque état du diagramme d'états-transitions correspond à une valeur de l'énumération générée.

Exemple. Le diagramme d'états-transitions de la classe **Control** de l'exemple de contrôle d'un passage à niveau (voir figure 2.5 et figure 2.6, page 55) possède deux états (**TrafficEnabled** et **TrafficDisabled**), qui sont modélisés par une énumération FoCaLiZe, comme suit :

```
{TrafficEnabled, TrafficDisabled} | type controlStatus = | TrafficEnabled
                                     | TrafficDisabled ;;
```

7.2 Transformation de transitions

Nous rappelons qu'un diagramme d'états-transitions d'une classe c_n est composé d'une suite de transition $\mathbb{T}\mathbb{R} = \{ tr_1, \dots, tr_k \}$ (voir formule 2.14, page 48), où chaque transition tr_i est définie par :

$$tr_i = \mathbf{transition} \ tr_i_n \ (source_i_n, target_i_n) \ ev_i \ [g_i] \ / \ ac_{i1}, \dots, ac_{im}$$

Soit $\mathbb{I}$ la liste des instances de la classe c_n , nous dénotons par $state(o)$ l'état de l'instance o ($o \in \mathbb{I}$) de la classe c_n , à un moment donné.

Une première interprétation, consiste à considérer une transition tr_i comme une fonction qui prend l'état source ($source_i_n$) et la garde (g_i) de transition comme paramètres, et retourne l'état cible ($target_i_n$). Cependant, cette interprétation isole le diagramme d'état-transition de l'aspect structurel. Autrement dit, il n'y a aucun lien entre les variables d'instances de l'objet et son état. Ceci conduit aussi à ignorer totalement les communications (l'envoi de messages) entre les diagrammes d'états-transitions de classes d'un modèle UML.

Pour éviter ces inconvénients, nous considérons plutôt la transition tr_i comme une fonction paramétrée par une instance $o \in \mathbb{I}$, avec $state(o) = source_i_n$, et retournant une autre instance o' , tel que $state(o') = target_i_n$. En effet, dans un contexte orienté objet, les instances o et o' représentent le même objet mais avec deux états mémoires différents. Dans un langage de programmation fonctionnel (sans états mémoires) comme FoCaLiZe, o et o' sont deux entités distinctes. En utilisant cette dernière interprétation, la transition tr_i est modélisée par une fonction paramétrée par une instance o de la classe c_n et est conditionnée par une nouvelle garde g'_i , définie par :

$$g'_i = g_i \ \text{and} \ (state(o) = source_i_n)$$

La fonction tr_i (qui formalise la transition tr_i) satisfaisant toujours la propriété :

$$\forall o \in \mathbb{I}, (g_i = true) \wedge (state(o) = source_i_n) \implies state(tr_i(o)) = target_i_n \quad (7.1)$$

Les actions ($ac_{i1}, \dots, ac_{im}$) qui s'exécutent suite au déclenchement de la transition tr_i sont directement formalisées par des appels aux fonctions correspondantes (spécifiées dans l'espèce dérivée de la classe) :

UML :

$$ac_{i_n} \ (ap_1_n, \dots, ap_k_n)$$

FoCaLiZe :

$$ac_{i_n} \ (\llbracket ap_1_n \rrbracket, \dots, \llbracket ap_k_n \rrbracket)$$

où $ap_1_n, \dots, ap_k_n$ sont des paramètres effectifs passés à l'opération de classe identifiée par ac_i_n , lors de son invocation.

Pour résumer, la formalisation d'une transitions (tr_i_n) de diagramme d'états-transitions de la classe c_n correspond à la création d'une fonction et d'une propriété spécifiées dans l'espèce s_n dérivée de la classe c_n , comme suit :

UML :

```
tr_i = transition tr_i_n (source_i_n, target_i_n) ev_i [g_i] / ac_i1 ... ac_im
```

FoCaLiZe :

```
species s_n =
...
signature state : Self -> states_sum_type ;

let tr_ident_i(o:Self, g:bool):Self =
  if (g) && (state(o) = [source_i_n]) then
    let exp_1 = [ac_i1] and ... and exp_m = [ac_im] in ([ev_i])
  else focalize_error ("ERROR");

property specif_tr_ident_i : all o : Self,
  (([g_i]) /\ (state(o) = [source_i_n])) -> state(tr_ident_i(o, [g_i])) = [target_i_n]
end;;
```

Dans le cadre de ce travail de thèse, les diagrammes d'états-transitions sont supposés déterministes [DJR12], sinon nous aurions dû prouver cette propriété dans la spécification. Nous considérons donc qu'il existe au plus une transition partant d'un état ($source_n$) et portant sur la même étiquette ($ev[g]$). Ceci nous a conduit à regrouper la transformation de toutes les transitions d'un diagramme d'états-transitions dans une fonction unique, en utilisant le mécanisme de filtrage ("pattern matching") dans Focalize.

Exemple. En utilisant les règles de transformation données ci-dessus, le diagramme d'états-transitions de la classe **Control** de l'exemple du contrôle d'un passage à niveau (voir figure 2.5 et figure 2.6, page 55) est formalisé en FoCaLiZe comme suit :

UML

```
stateMachine of Control
transition tr1 (TrafficEnabled, TrafficDisabled) disableTraffic() /
  Barrier.closeBarrier(), Light.toRed()
transition tr2 (TrafficDisabled, TrafficEnabled) enableTraffic() /
  Barrier.openBarrier(), Light.toGreen()
end
```

FCaLiZe

```
type controlStatus = | TrafficEnabled | TrafficDisabled ;;
species Control(L is Light, B is Barrier) =

signature enableTraffic : Self -> Self;
```

```

signature disableTraffic : Self -> Self;

signature state : controlStatus -> Self;

let controlTransitions (c:Self, guard:bool):Self =
  match (state(c), guard) with
  |(TrafficEnabled, true) -> let exp1 = L!toRed(cLight(c)) and
                             exp2 = B!closeBarrier(cBarrier(c))
                             in disableTraffic(c)
  |(TrafficDisabled, true) -> let exp1 = L!toGreen(cLight(c)) and
                              exp2 = B!openBarrier(cBarrier(c))
                              in enableTraffic(c)
  |(_, _) -> focalize_error ("ERROR");

property specif_controlTransitions_1 : all c :Self,
(state(c)= TrafficEnabled) ->
state(controlTransitions(c, true))= TrafficDisabled ;

property specif_controlTransitions_2 : all c :Self,
(state(c)= TrafficDisabled) ->
state(controlTransitions(c, true)) = TrafficEnabled;

end;;

```

7.3 Détection d'erreurs de diagrammes d'états-transitions

L'objectif de transformation des diagrammes d'états-transitions est de générer une spécification FoCaLiZe qui nous permettra de détecter des éventuelles contradictions entre l'aspect statique (classes et contraintes OCL) et l'aspect dynamique (diagramme d'états-transitions).

En utilisant notre démarche de transformation (de diagrammes de classes, de contraintes OCL et de diagrammes d'états-transitions), nous obtenons deux types de propriétés dans la spécification FoCaLiZe générée :

- Les propriétés dérivées de contraintes OCL (invariant et pré/post-conditions) et
- et les propriétés de spécification des transitions.

Si l'événement déclencheur d'une transition est une opération spécifiée d'une pré/post-conditions, nous obtenons (dans la spécification FoCaLiZe générée) deux propriétés portant sur la même fonction : Une propriété dérivée de pré/post-conditions et une propriété dérivée de la transformation de la transition déclenchée par l'événement.

Pour détecter des éventuelles contradictions, nous supposons que la propriété dérivée de pré/post-conditions (*pre-post_property*) est satisfaite, comme axiome. Puis, nous tentons de fournir une preuve (en utilisant le langage FPL) de la propriété dérivée de la spécification de la transition (*transition_property*) comme suit :

```
proof of transition_property = by property pre-post_property ...;
```

S'il n'y a aucune contradiction entre les deux propriétés, Zenon va trouver la preuve. Lorsque Zenon ne peut pas trouver automatiquement une preuve, le développeur doit ajouter des faits ("facts") pour prouver la propriété. Autrement, cela signifie qu'il y a une contradiction dans le modèle UML/OCL d'origine, et Zenon indique la ligne de code responsable de l'erreur. Ainsi, le développeur va réexaminer et corriger le modèle UML/OCL d'origine et refaire le cycle.

Exemple. Pour mettre en évidence ce processus de vérification, nous présentons la preuve de la propriété `specif_lightTransition1` spécifiant la transition de l'état `Green` vers l'état `Red` dans l'espèce `Light` (voir l'exemple du contrôle d'un passage à niveau, page 54) :

```
type lSTATUS = | Green | Red ;;

species Light =

  (* Transformation d'attributs et d'opérations de la classe *)
  signature get_lightStatus : Self -> lSTATUS;
  signature toRed : Self -> Self;
  signature toGreen : Self -> Self;

  (* Transformation de contraintes OCL *)
  property prepost_toRed : all l : Self,
    (get_lightStatus(l)= Green) -> (get_lightStatus(toRed(l)) = Red) ;

  property prepost_toGreen : all l : Self,
    (get_lightStatus(l)= Red) -> (get_lightStatus(toGreen(l)) = Green) ;

  (* Transformation de diagramme d'états-transitions *)

  let lightTransitions(l:Self, guard:bool):Self =
    match (get_lightStatus(l), guard) with
    | (Green, true) -> toRed(l)
    | (Red, true) -> toGreen(l)
    | (_, _) -> focalize_error ("ERROR");

  property specif_lightTransitions1 : all l :Self,
    (get_lightStatus(l)= Green) ->
    equal(lightTransitions(l, true), newLight(Red)) ;

  (* Preuves et détection d'erreurs *)

  proof of prepost_toRed = assumed;
  proof of prepost_toGreen = assumed;
  proof of specif_lightTransition1 = by definition of lightTransitions
    property prepost_toRed, prepost_toGreen;

end;;
```

La compilation de la spécification FoCaLiZe précédente montre la cohérence du modèle. Aucune erreur n'a eu lieu, ce qui signifie que la compilation, la génération de code et la vérification Coq ont réussi.

7.4 Conclusion

Dans ce chapitre, nous avons proposé une transformation des diagrammes d'états-transitions avec des contraintes OCL qui fonctionne naturellement avec la transformation des diagrammes de classes, où nous supportons des fonctionnalités UML comme l'héritage multiple, les classes paramétrées et la substitution de ses paramètres formelles par des paramètres effectifs ("template binding"). De cette manière, il est possible d'obtenir une transformation rigoureuse de diagramme d'états-transitions d'une classe, même si la classe est créée par héritage multiple ou créée par la substitution des paramètres formels d'un template UML. Ce résultat est une amélioration significative par rapport aux transformations proposées par les autres méthodes formelles.

De plus, la disponibilité du prouveur automatique de théorèmes Zenon, permet de détecter automatiquement des éventuelles contradictions entre la spécification de classes et la spécification des diagrammes d'états-transitions.

La transformation des diagrammes d'états-transitions proposée peut être naturellement étendue pour traiter des fonctionnalités de diagrammes d'états-transitions d'UML2 comme les états composites, les sous-diagrammes d'états-transitions et les transitions composées.

Conclusion générale

Dans ce travail de thèse, nous avons proposé une alternative pour la formalisation des modèles UML/OCL, en utilisant l'environnement FoCaLiZe. Dans cette formalisation, nous avons pris en compte les aspects statique et dynamique d'UML et les contraintes OCL. En particulier, nous avons traité des fonctionnalités UML/OCL qui n'étaient pas prises en compte dans les travaux similaires. Ainsi, à partir d'un modèle UML composé des diagrammes des classes, des diagrammes d'états-transitions et des contraintes OCL, nous avons pu générer une spécification UML/OCL textuelle. Les éléments de la spécification textuelle obtenue sont ensuite organisés suivant les dépendances syntaxique et sémantique reliant ces éléments. Nous appliquons ensuite les règles de transformation proposées pour produire une spécification FoCaLiZe.

Dans la formalisation introduite, les classes sont transformées en espèces, les attributs de classes en signatures modélisant leurs getters, les opérations de classes en signatures spécifiant leurs types fonctionnels. Les expressions du langage OCL sont transformées vers des expressions FoCaLiZe équivalentes. Ainsi, des bibliothèques FoCaLiZe ont été créées pour le traitement des expressions de types **Real**, **String** et **Collection**. Les contraintes OCL d'une classe sont alors traduites en propriétés dans l'espèce correspondante.

Le diagramme d'états-transitions d'une classe est formalisé dans l'espèce dérivée de la classe, où les états sont modélisés par une énumération FoCaLiZe, les transitions sont modélisées par des fonctions et chaque transition est spécifiée par une propriété dans la même espèce.

Synthèse des contributions

Les contributions de ce travail de thèse se situent dans les fonctionnalités UML/OCL que nous avons pris en compte vis-à-vis des travaux similaires. Dans notre démarche de transformation, nous avons supporté de façon naturelle et systématique les fonctionnalités UML/OCL suivantes :

- **Héritage multiple** : nous dérivons une hiérarchie d'espèces qui reflète les relations d'héritage entre les classes. Les mécanismes de redéfinition de méthodes et de liaison tardive sont aussi préservés entre les espèces dérivées ;
- **Classes paramétrées** : une classe paramétrée est formalisée par une espèce paramétrée. Les paramètres de type **Class** sont formalisés par des paramètres de collection (en utilisant le mot clé **is**) et les paramètres de types primitifs sont formalisés par des paramètres d'entités (en utilisant le mot clé **in**) ;
- **Dérivation des classes liées aux classes paramétrées** : cette relation est formalisée via la substitution des paramètres formels d'une espèce par des paramètres effectifs. Au niveau spécification, une espèce dérivée d'une classe peut être utilisée comme paramètre d'une autre espèce (dérivée d'une autre classe), même avant l'implémentation finale des espèces. Ultérieurement, les

mécanismes de collection et de liaison tardive en FoCaLiZe garantissent que toutes les méthodes apparaissant dans une espèce (utilisée comme paramètre formel) sont effectivement définies ;

- **Dépendance** : la dépendance entre deux classes (cliente et fournisseur) est formalisée par la paramétrisation de l'espèce cliente (dérivée de la classe cliente) par l'espèce fournisseur (dérivée de la classe fournisseur) ;
- **Association binaire** : elle est modélisée par une espèce paramétrée par les espèces dérivées des deux classes aux extrémités de l'association. Les multiplicités aux extrémités de l'association et la direction de navigation de l'association sont aussi formalisées par des paramètres formels de l'espèce dérivée de l'association. Les contraintes imposées par les multiplicités d'une association sont spécifiées par des propriétés FoCaLiZe ;
- **Héritage des contraintes OCL** : les propriétés d'une super-espèce sont aussi propriétés de ses héritiers (sous-espèces) ;
- **Propagation des contraintes OCL à travers le paramétrage et la dépendance** : Les propriétés (d'une espèce) dérivées de contraintes OCL sont propagées à travers le mécanisme de paramétrage en FoCaLiZe. Les propriétés d'une espèce fournisseur (utilisée comme paramètre formel) peuvent être utilisées en toute sécurité par des espèces clientes (espèces paramétrées) ;
- **Propagation des contraintes OCL via la relation de dérivation de classes liées aux classes paramétrées** : les propriétés d'une espèce dérivée d'une classe paramétrée (UML template) sont aussi propriétés des espèces dérivées des classes liées ("bound models") correspondantes ;
- **Diagramme d'états-transitions** : la démarche de transformation proposée permet une formalisation rigoureuse du diagramme d'états-transitions d'une classe, même si cette dernière est créée par héritage multiple ou par la substitution des paramètres formels d'un template UML par des paramètres effectifs. Nous avons aussi pris en compte la communication des diagrammes d'états-transitions avec les diagrammes de classes et l'utilisation des expressions OCL pour la spécification des conditions de garde des transitions.

Utilisation et expérimentation de la transformation

Une application directe de l'approche proposée consiste à combiner UML/OCL, FoCaLiZe et les règles de transformations proposées dans un cadre (framework) MDE pour permettre la génération de codes certifiés. Dans ce but, nous avons abouti à une implémentation de la démarche de transformation proposée en utilisant le langage XSLT sous l'environnement Eclipse : **UML2FOC**¹.

Ainsi, sous le même environnement (Eclipse), il est possible de construire un modèle UML/OCL en utilisant un outil UML/OCL graphique (Papyrus), puis en appelant les règles de transformation de générer le code FoCaLiZe correspondant. Toujours sous le même environnement, on peut procéder à la compilation du code FoCaLiZe produit pour la vérification et la détection d'éventuelles incohérences. En cas d'erreur, l'outil MDE proposé assiste l'utilisateur en indiquant la ligne de code FoCaLiZe responsable de l'erreur.

Enfin, pour produire un exécutable, un développeur FoCaLiZe complète le code généré en définis-

¹<http://www.univ-eloued.dz/uml2foc/>

sant toutes ses signatures et en prouvant toutes ses propriétés.

Perspectives

L'objectif de ce travail de thèse était de proposer une approche de formalisation des modèles UML/OCL en considérant des constructions UML/OCL de base, comme le diagramme de classes, les invariants et les pré/post-conditions. Parallèlement à cette approche, nous avons travaillé sur la transformation des diagrammes UML décrivant l'aspect comportemental d'un système. Ainsi, nous avons d'abord traité la transformation du diagramme d'états-transitions (voir chapitre 7). Nous travaillons actuellement sur la transformation du diagramme d'activité. L'idée consiste à modéliser un diagramme d'activité UML par la définition d'une fonction récursive, en utilisant le mécanisme de filtrage (pattern matching) en FoCaLiZe. Chaque pattern correspondra au traitement d'une action du diagramme d'activité.

Côté OCL, nous avons traité des contraintes OCL exprimées au moyen de types primitifs et de types objet. Pour les expressions OCL de types collections, nous avons uniquement traité le type OCL `Collection`. Les sous-types de `Collection` (`Set`, `OrderedSet`, `Bag` et `Sequence`) ne sont pas pris en compte. Comme extension de ce travail, nous souhaitons étendre notre transformation pour supporter un sous-ensemble plus riche d'OCL, en particulier la contrainte OCL **derive** et le sous-types de type OCL `Collection`. La contrainte OCL **derive** permet de spécifier la valeur d'un attribut ou le rôle d'une association. L'idée que nous envisageons pour transformer les sous-types de `Collection`, consiste à créer une espèce FoCaLiZe qui formalise les opérations OCL spécifiques à chaque sous-type en héritant de l'espèce `OCL_Collection` (qui modélise le type OCL `Collection`).

Bibliographie

- [Abb07] Messaoud Abbas. Transcription des spécifications UML vers le système FoCAL. Juillet 2007. Mémoire de Magister, USTHB (Alger).
- [Abb14] Messaoud Abbas. Using FoCaLiZe to check OCL constraints on UML classes. In *International Conference on Information Technology for Organization Development, IT4OD 2014*, pages 31–38. Conference proceedings, 2014. available at https://www.researchgate.net/publication/269094681_IT4OD_2014_Proceedings_Tebessa, accessed January 2018.
- [ABGR07] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. Uml2alloy : A challenging model transformation. In *Model Driven Engineering Languages and Systems*, pages 436–450. Springer, 2007.
- [ABGR10] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On challenges of model transformation from UML to Alloy. *Software & Systems Modeling*, 9(1) :69–86, 2010.
- [Abr96] J.R. Abrial. The B book. 1996. Cambridge University Press.
- [ABYR14a] Messaoud Abbas, Choukri-Bey Ben-Yelles, and Renaud Rioboo. Generating FoCaLiZe Specifications from UML Models. In *International Conference on Advanced Aspects of Software Engineering, ICAASE, Conference proceedings*, pages 157–164, 2014.
- [ABYR14b] Messaoud Abbas, Choukri-Bey Ben-Yelles, and Renaud Rioboo. Modeling UML Template Classes with FoCaLiZe. In *The International Conference on Integrated Formal Methods, IFM2014*, volume 8739, pages 87–102. Springer, LNCS, 2014.
- [ABYR18] Messaoud Abbas, Choukri-Bey Ben-Yelles, and Renaud Rioboo. Modelling UML state machines with FoCaLiZe. *Int. J. Information and Communication Technology*, 13 :34–54, 2018.
- [ACK12] Étienne André, Christine Choppy, and Kais Klai. Formalizing non-concurrent uml state machines using colored petri nets. *ACM SIGSOFT Software Engineering Notes*, 37(4) :1–8, 2012.
- [AHP09a] Philippe Ayrault, Thérèse Hardin, and François Pessaux. Development life-cycle of critical software under Focal. *Electronic Notes in Theoretical Computer Science*, 243 :15–31, 2009.
- [AHP09b] Philippe Ayrault, Thérèse Hardin, and François Pessaux. Development of a generic voter under focal. In *Tests and Proofs*, pages 10–26. Springer, 2009.
- [Akr06] I. Akram. *B/UML : Mise en relation de spécifications B et de descriptions UML pour l’aide à la validation externe de développements formels en B*. PhD thesis, Grenoble 1, Novembre 2006.

- [Aud09] Laurent Audibert. *UML 2*. Ellipses, 2009.
- [BBFM99] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Meteor : A successful application of B in a large project. In *FM'99-Formal Methods*, pages 369–387. Springer, 1999.
- [BC95] Robert H Bourdeau and Betty HC Cheng. A formal semantics for object model diagrams. *Software Engineering, IEEE Transactions on*, 21(10) :799–821, 1995.
- [BCD⁺07] M. Broy, M. Crane, J. Dingel, A. Hartman, B. Rumpe, and B. Selic. Formal semantics for UML. *Models in Software Engineering*, pages 318–323, 2007.
- [BD04] Pierre Bourque and Robert Dupuis. Software engineering body of knowledge (swebok). *IEEE Computer Society, EUA*, 2004.
- [BDW07] Achim D Brucker, Jürgen Doser, and Burkhart Wolff. An MDA framework supporting OCL. *Electronic Communications of the EASST*, 5, 2007.
- [BH95] Jonathan P Bowen and Michael G Hinchey. Seven more myths of formal methods. *IEEE software*, 12(4) :34–41, 1995.
- [BM06] Xavier Blanc and Isabelle Mounier. *UML 2 pour les développeurs : cours avec exercices corrigés*. Editions Eyrolles, 2006.
- [Boo91] Grady Booch. Object-oriented design with applications benjamin. *Cummings, Redwood City (CA)*, 1991.
- [BRJ03] G. Booch, J. Rumbaugh, and I. Jacobson. Le guide de l'utilisateur UML. 2003. Edition Eyrolle.
- [BW07] A.D. Brucker and B. Wolff. *The HOL-OCL tool*. 2007. <http://www.brucker.ch/> accessed January 2018.
- [BW08] Achim D Brucker and Burkhart Wolff. HOL-OCL : A Formal Proof Environment for UML/OCL. In *Fundamental Approaches to Software Engineering*, pages 97–100. Springer, 2008.
- [CC00] P. J. F. Carreira and M. E. F. Costa. Automatically verifying an object-oriented specification of the steam-boiler system. *Proceedings of the 5th International ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS'2000)*., pages 345–360, 2000.
- [CCGR00] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv : a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4) :410–425, 2000.
- [CDE⁺07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All about maude-a high-performance logical framework : how to specify, program and verify systems in rewriting logic*. Springer-Verlag, 2007.
- [CE97] Tony Clark and Andy Evans. Foundations of the unified modeling language. 1997.
- [CEK01] Tony Clark, Andy Evans, and Stuart Kent. The metamodeling language calculus : foundation semantics for UML. In *Fundamental Approaches to Software Engineering*, pages 17–31. Springer, 2001.
- [CGR13] Alcino Cunha, Ana Garis, and Daniel Riesco. Translating between alloy specifications and uml class diagrams annotated with ocl. *Software & Systems Modeling*, pages 1–21, 2013.

- [Coq16] Coq. *The Coq Proof Assistant, Tutorial and Reference Manual, Version 8.5*. INRIA à LIP à LRI à LIX à PPS, 2016. Distribution available at : <http://coq.inria.fr/>.
- [DÉ08] D. Delahaye and J.F. Étienne. *modélisation de la réglementation de l'aviation civile en FoCAL*. PhD thesis, CNAM, Paris, 2008.
- [DÉDG08a] David Delahaye, J-F Étienne, and Véronique Viguié Donzeau-Gouge. Producing uml models from focal specifications : an application to airport security regulations. In *Theoretical Aspects of Software Engineering, 2008. TASE'08. 2nd IFIP/IEEE International Symposium on*, pages 121–124. IEEE, 2008.
- [DÉDG08b] David Delahaye, Jean-Frédéric Étienne, and Véronique Viguié Donzeau-Gouge. A formal and sound transformation from Focal to UML : an application to airport security regulations. *Innovations in Systems and Software Engineering*, 4(3) :267–274, 2008.
- [DGR11] F. Durán, M. Gogolla, and M. Roldán. Tracing properties of UML and OCL models with Maude. *arXiv preprint arXiv :1107.0068*, 2011.
- [DJR12] Damien Doligez, Mathieu Jaume, and Renaud Rioboo. Development of secured systems by mixing programs, specifications and proofs in an object-oriented programming environment. In *ACM SIGPLAN Seventh Workshop on Programming Languages and Analysis for Security (PLAS 2012) Proceedings. To appear*, 2012.
- [DLL⁺15] Alexandre David, Kim G Larsen, Axel Legay, Marius Mikučionis, and Danny Bøgsted Poulsen. Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4) :397–415, 2015.
- [DMT00] Grit Denker, Jose Meseguer, and Carolyn Talcott. Formal specification and analysis of active networks and communication protocols : The maude experience. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, volume 1, pages 251–265. IEEE, 2000.
- [Dol15] Damien Doligez. "the Zenon tool". 2015. Software and documentations freely available at <http://focal.inria.fr/zenon/>.
- [DP15] Catherine Dubois and François Pessaux. Termination proofs for recursive functions in FoCaLiZe. In *International Symposium on Trends in Functional Programming*, pages 136–156. Springer, 2015.
- [DRS95] Roger Duke, Gordon Rose, and Graeme Smith. Object-z : A specification language advocated for the description of standards. *Computer Standards & Interfaces*, 17(5) :511–533, 1995.
- [DSRJ04] Greg Dennis, Robert Seater, Derek Rayside, and Daniel Jackson. Automating commutativity analysis at the design level. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 165–174. ACM, 2004.
- [EK99] Andy Evans and Stuart Kent. Core meta-modelling semantics of UML : the pUML approach. In *«UML» 99 à The Unified Modeling Language*, pages 140–155. Springer, 1999.
- [Fav10] Liliana María Favre. Formalization of mof-based metamodels. *Model Driven Architecture for Reverse Engineering Technologies. Information Resources Management Association*, 2010.

- [Fec05] Stéphane Fechter. *Sémantique des traits orientés objet de Focal*. PhD thesis, Université PARIS 6, Juillet 2005.
- [FGK⁺96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu. Cadp a protocol validation and verification toolbox. In *Computer Aided Verification*, pages 437–440. Springer, 1996.
- [FL95] P. Facon and R. Laleau. Des spécifications informelles aux spécifications formelles : compilation ou interprétation ? *Actes du 13ème congrès INFORSID*, 1995.
- [FL00] P. Fröhlich and J. Link. Automated test case generation from dynamic models. *Proceedings of ECOOP 2000*, volume 1850 of LNCS :472–491, 2000.
- [Fle99] W. Fleisch. Applying use cases for the requirements validation of component-based real-time software. *Proceedings of the 2nd International Symposium on Object-Oriented Real-Time Distributed Computing, ISORC'99*, May 1999.
- [GBF01] Geri Georg, Jores Bieman, and Robert B France. Using Alloy and UML/OCL to Specify Run-Time Configuration Management : A case study. In *pUML*, pages 128–141. Citeseer, 2001.
- [GPCR12] Ana Garis, Ana CR Paiva, Alcino Cunha, and Daniel Riesco. Specifying UML protocol state machines in Alloy. In *Integrated Formal Methods*, pages 312–326. Springer, 2012.
- [Gue01] A. L. Guennec. Génie logiciel et méthodes formelles avec UML spécification, validation et génération de tests. *Thèse doctorat de l'université de Rennes I*, juin 2001.
- [GV13] Claude Girault and Rüdiger Valk. *Petri nets for systems engineering : A guide to modeling, verification, and applications*. Springer Science & Business Media, 2013.
- [GZC⁺16] Fan Gu, Xinqian Zhang, Mingsong Chen, Daniel Große, and Rolf Drechsler. Quantitative timing analysis of uml activity diagrams using statistical model checking. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, pages 780–785. EDA Consortium, 2016.
- [HFDP16] Thérèse Hardin, Pessaux Francois, Weis Pierre, and Doligez Damien. FoCaLiZe : Tutorial and Reference Manual, version 0.9.1. *CNAM/INRIA/LIP6*, 2016. available at : <http://focalize.inria.fr>.
- [HLM04] L. Hazem, N. Levy, and R. Marcano-Kamenoff. UML2B : un outil pour la génération de modèles formels. *AFDL*, 2004.
- [Hol97] J. G. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering.*, 23(5) :279–295, May 1997.
- [Hol04] Gerard J Holzmann. *The SPIN model checker : Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.
- [HR04] Thérèse Hardin and Renaud Rioboo. Les objets des mathématiques. *RSTI - L'objet*, Octobre 2004.
- [HTVW02] Ahmed Hammad, Bruno Tatibouët, Jean-Christophe Voisinet, and Wu Weiping. From a b specification to uml statechart diagrams. In *Formal Methods and Software Engineering*, pages 511–522. Springer, 2002.
- [Ios98] Radu Iosif. The PROMELA language. <http://www.dai-arc.polito.it/dai-arc/manual/tools/jcat/main/node168.html>, 1998.

- [ISO85] . ISO. A formal description technique based on the temporal ordering of observational behaviour. *ISO/ DP 8807*, March 1985.
- [Jac04] Daniel Jackson. Alloy 3.0 reference manual. *Software Design Group*, 2004.
- [Jac11] Daniel Jackson. *Software Abstractions : Logic, Language, and Analysis*. The MIT Press, 2011.
- [JCC94] Ivar Jacobson, Magnus Christerson, and Larry L Constantine. The oose method : a useâcase-driven approach. In *Object development methods*, pages 247–270. SIGS Publications, Inc., 1994.
- [JM05] Mathieu Jaume and Charles Morisset. Formalisation and implementation of access control models. In *Information Technology : Coding and Computing, 2005. ITCC 2005. International Conference on*, volume 1, pages 703–708. IEEE, 2005.
- [JZ16] Maryam Jamal and Nazir Ahmad Zafar. Formalizing structural semantics of UML 2.5 activity diagram in Z notation. In *Open Source Systems & Technologies (ICOSST), 2016 International Conference on*, pages 66–71. IEEE, 2016.
- [Kwo00] G. Kwon. Rewrite rules and operational semantics for model checking UML state-charts. *Third International Conference, York, UK*, October 2000.
- [Lal02] R. Laleau. *Conception et développement formels d’applications bases de données*. PhD thesis, Université d’Evry, 2002.
- [Lot89] ISO Lotos. A formal description technique based on the temporal ordering of observational behaviour. *ISO8807, 1XS989*, 1989.
- [LS02] Hung Ledang and Jeanine Souquière. Integration of UML and B specification techniques : Systematic transformation from OCL expressions into B. In *Asia-Pacific Software Engineering Conference*, pages 495–495. IEEE Computer Society, 2002.
- [LSC03] H. Ledang, J. Souquière, and S. Charles. Un outil de transformation systématique de spécification UML en B. *AFDL*, 2003.
- [MBFBJ02] Stephen J Mellor, Marc Balcer, and Ivar Foreword By-Jacobson. *Executable UML : A foundation for model-driven architectures*. Addison-Wesley Longman Publishing Co., Inc., 2002.
- [MC13] Nuno Macedo and Alcino Cunha. Implementing qvt-r bidirectional model transformations using Alloy. In *Fundamental Approaches to Software Engineering*, pages 297–311. Springer, 2013.
- [McM93] Kenneth L McMillan. The SMV system. In *Symbolic Model Checking*, pages 61–85. Springer, 1993.
- [Mey01] E. Meyer. *Développements formels par objets : utilisation conjointe de B et d’UML*. PhD thesis, Nancy 2, Mars 2001.
- [MG00] Pierre-Alain Muller and Nathalie Gaertner. *Modélisation objet avec UML*, volume 514. Eyrolles Paris, 2000.
- [MGB05] Tiago Massoni, Rohit Gheyi, and Paulo Borba. Formal refactoring for uml class diagrams. In *19th Brazilian Symposium on Software Engineering (SBES), Uberlândia, Brazil*, pages 152–167, 2005.
- [MM+03] Joaquin Miller, Jishnu Mukerji, et al. MDA guide. *Object Management Group*, 2003.

- [MOM02] Narciso Martí-Oliet and José Meseguer. Rewriting logic : roadmap and bibliography. *Theoretical Computer Science*, 285(2) :121–154, 2002.
- [Mos04] Peter Mosses. *CASL reference manual : The complete documentation of the common algebraic specification language*, volume 1. Springer, 2004.
- [MSBK10] Farid Mokhati, Brahim Sahraoui, Soufiane Bouzaher, and Mohamed Tahar Kimour. A tool for specifying and validating agentsâ interaction protocols : From agent uml to maude. *Journal of object technology*, 9(3), 2010.
- [Mur89] Tadao Murata. Petri nets : Properties, analysis and applications. *Proceedings of the IEEE*, 77(4) :541–580, 1989.
- [Ngu98] H. P. Nguyen. *Dérivation de spécifications formelles B à partir de spécifications semi-formelles*. PhD thesis, CNAM-Paris, December 1998.
- [NPW02] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL : A Proof Assistant for Higher-order Logic*, volume 2283. Springer, 2002.
- [OA99] J. Offutt and A. Generating Abdurazik. Generating tests from uml specifications. *Proceedings of Second International Conference, Fort Collins, CO, USA*, volume 1723 of LNCS :416–429, Octobre 1999.
- [OMG14] OMG. OCL : Object Constraint Language 2.4. January 2014. available at : <http://www.omg.org/spec/OCL/2.4/PDF>.
- [OMG15a] OMG. UML : Unified Modeling Language, version 2.5. 2015. available at : <http://www.omg.org/spec/UML/2.5/PDF>.
- [OMG15b] OMG. XML Metadata Interchange (XMI) Specification, version 2.5.1. 2015. available at : <http://www.omg.org/spec/XMI/2.5.1/PDF>.
- [PL99] I. Paltor and J. Lilius. A tool for verifying UML models. *Proceedings of the 14th IEEE International Conference on Automated Software Engineering.*, 1999.
- [RBL⁺90] James R Rumbaugh, Michael R Blaha, William Lorensen, Frederick Eddy, and William Premerlani. Object-oriented modeling and design. 1990.
- [RCA01] Gianna Reggio, Maura Cerioli, and Egidio Astesiano. Towards a rigorous semantics of UML supporting its multiview approach. In *Fundamental Approaches to Software Engineering*, pages 171–186. Springer, 2001.
- [SAB09] Seyyed MA Shah, Kyriakos Anastasakis, and Behzad Bordbar. From uml to alloy and back again. In *Models in Software Engineering*, pages 158–171. Springer, 2009.
- [SB04] C. Snook and M. Butler. U2b-a tool for translating UML-B models into B. in *J. Mermet (ed.), UML-B Specification for Proven Embedded Systems Design.*, 2004.
- [SB06] C. Snook and M. Butler. UML-B : Formal modeling and design aided by UML. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(1) :92–122, 2006.
- [SSS16] Monika Singh, AK Sharma, and Ruhi Saxena. Formal transformation of UML Diagram : Use case, class, sequence diagram with z notation for representing the static and dynamic perspectives of system. In *Proceedings of International Conference on ICT for Sustainable Development*, pages 25–38. Springer, 2016.

- [TF15] Li Tao and Jia Fengsheng. B Formal Modeling Based on UML Class. In *IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC), 2015*, pages 1–6. IEEE Conference Publications, 2015. DOI : 10.1109/ICSPCC.2015.7338854.
- [Voi04] Jean-Christophe Voisinet. *Contribution aux processus de développement d'applications spécifiées à l'aide de la méthode B par validation utilisant des vues UML et traduction vers des langages objets*. PhD thesis, 2004.
- [W3C17] W3C. XSL transformations (XSLT) version 3.0, W3C recommendation, june 2017. 2017. available at : <https://www.w3.org/TR/2017/REC-xslt-30-20170608/>, accessed January 2018.
- [YMP08] M. Yang, G.J. Michaelson, and R.J. Pooley. Formal action semantics for a UML action language. *Journal of Universal Computer Science*, 14(21) :3608–3624, 2008.

Annexe A

Sources FoCaLiZe

Le code complet des espèces `PointAbstrait`, `PointColore`, `PointConcret` et `Cercle` (utilisées dans le chapitre 1 de cette thèse) est donné comme suit :

```
open "basics" ;;

(* l'énumération color *)

type couleur = | Rouge | Vert | Bleu ;;

species Point =

  representation = float * float;

  signature deplacer_point : Self -> float-> float -> Self;
  signature egal : Self -> Self -> bool;

  let get_x(p:Self) = fst(p);
  let get_y(p:Self) = snd(p);
  let different(a, b) = ~(egal(a, b));

  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);

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

```

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

  let affiche_point (p:Self):string = (" X = " ^ string_of_float(get_x(p)) ^
                                         " Y = " ^ string_of_float(get_y(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) ) );
  end;;

species PointConcret = inherit PointAbstrait;
representation = float * float;
let creerPointConcret(x:float, y:float):Self = (x , y);
let get_x(p) = fst(p);
let get_y(p) = snd(p);
let deplacer(p, dx, dy) = ( get_x(p) +f dx, get_y(p) +f dy);
end;;

species PointCouleur = inherit PointAbstrait;
representation = (float * float) * couleur;
let get_color(p:Self):couleur = snd(p);
let creerPointCouleur(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) =
  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;;

collection PointCouleurCollection = implement PointCouleur; end;;
collection PointConcretCollection = implement PointConcret; end;;
let p = PointCouleurCollection!creerPointCouleur(2.0, 5.0, Bleu);;

basics#print_string(PointCouleurCollection!affiche_point(p));;

```

```
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;;

collection CercleCollection1 = implement Cercle(PointConcretCollection); end;;
collection CercleCollection2 = implement Cercle(PointColoreCollection); end;;

species RepereOrthogonal (origine in PointConcretCollection) =
  representation = PointConcretCollection;
end;;

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

Annexe B

Transformation des expressions OCL

B.1 Définition des opérations OCL sur le type Real

(* addition réelle *)

```
let ( +f ) =  
  internal float -> float -> float  
  external  
  | caml -> {*(+.)*}  
  | coq -> {* Rplus *}  
;;
```

(* soustraction réelle *)

```
let ( -f ) =  
  internal float -> float -> float  
  external  
  | caml -> {*(-.)*}  
  | coq -> {* Rminus *}  
;;
```

(* multiplication réelle *)

```
let ( *f ) =  
  internal float -> float -> float  
  external  
  | caml -> {* ( *. )*}  
  | coq -> {* Rmult *}  
;;
```

(* division réelle *)

```
let ( /f ) =  
  internal float -> float -> float  
  external
```

```

| caml -> {*( /. )*}
| coq -> {* Rdiv *}
;;

(* valeur absolue d'un nombre réel *)
let abs_f =
  internal float -> float
  external
  | caml -> {* abs_float *}
  | coq -> {* Rbasic_fun.Rabs *}
;;

(* comparaison des réels : la fonction supérieure *)
let ( >f ) =
  internal float -> float -> bool
  external
  | caml -> {* Float_externals.sup_float *}
  | coq -> {* operation_float.float_gt *}
;;

(* comparaison des réels : la fonction inférieure *)
let ( <f ) =
  internal float -> float -> bool
  external
  | caml -> {* Float_externals.inf_float *}
  | coq -> {* operation_float.float_lt *}
;;

(* comparaison des réels : la fonction supérieure ou égale *)
let ( >=f ) =
  internal float -> float -> bool
  external
  | caml -> {* Float_externals.sup_eg_float *}
  | coq -> {* operation_float.float_ge *}
;;

(* comparaison des réels : la fonction inférieure ou égale *)
let ( <=f ) =
  internal float -> float -> bool
  external
  | caml -> {* Float_externals.inf_eg_float *}
  | coq -> {* operation_float.float_le *}
;;

(* le maximum de deux réels *)
let max_f =
  internal float -> float -> float
  external

```

```

| caml -> {* Float_externals.max_f *}
| coq -> {* Rbasic_fun.Rmax *}
;;

(* le minimum de deux réels *)
let min_f =
  internal float -> float -> float
  external
  | caml -> {* Float_externals.min_f *}
  | coq -> {* Rbasic_fun.Rmin *}
;;

(* floor_f : retourne l'entier inférieur ou égal à la
               valeur donnée en paramètre *)
let floor_f =
  internal float -> int
  external
  | caml -> {* Float_externals.floor_f *}
  | coq -> {* R_Ifp.Int_part *}
;;

(* round_f : retourne le plus petit entier supérieur ou
               égal à la valeur donnée en paramètre *)
let round_f =
  internal float -> int
  external
  | caml -> {* Float_externals.round_f *}
  | coq -> {* operation_float.Rround *}
;;

(* intTofloat : conversion d'un entier en réel *)
let intTofloat =
  internal int -> float
  external
  | caml -> {* float_of_int *}
  | coq -> {* IZR *}
;;

(* floatToint : conversion d'un réel en entier *)
let floatToint =
  internal float -> int
  external
  | caml -> {* int_of_float *}
  | coq -> {* R_Ifp.Int_part *}
;;

```

B.2 Définition des opérations OCL sur le type String

(* Longueur d'une chaine *)

```
let length =
  internal string -> int
  external
  | caml -> { * String.length *}
  | coq -> { * length *}
;;
```

(* Soustraction d'une sous chaine *)

```
let substring =
  internal string -> int -> int -> string
  external
  | caml -> { * function (s : string) ->
                function (lower : int) ->
                function (upper : int) ->
                String.sub s (lower-1) (upper-lower+1) *}
  | coq -> { * operation_string.substring *}
;;
```

(* Transformation d'une chaine en majuscule *)

```
let upper_case =
  internal string -> string
  external
  | caml -> { * String.uppercase *}
  | coq -> { * operation_string.upper_case *}
;;
```

(* Transformation d'une chaine en minuscule *)

```
let lower_case =
  internal string -> string
  external
  | caml -> { * String.lowercase *}
  | coq -> { * operation_string.lower_case *}
;;
```

```
;;

(* Conversion d'une chaine en réel *)
let float_of_string =
  internal string -> float
  external
  | caml -> { * float_of_string * }
  | coq -> { * fun (x : string) => 42%R * }
;;
```

B.3 Définition des opérations OCL de collection

- `isEmpty()` :Boolean , `notEmpty()` :Boolean : Permettent de tester si la collection est vide ou non.
- `size()` :Integer : Donne la taille (nombre d'éléments) d'une collection.
- `count(Objet :T)` :Integer : Donne le nombre d'occurrences d'un objet dans la collection.
- `includes(o :T)` :Boolean , `excludes(o :T)` :Boolean : Retourne vrai (resp. faux) si et seulement si o est (resp. n'est pas) un élément de la collection.
- `includesAll(c :Collection(T))` :Boolean , `excludesAll(c :Collection(T))` :Boolean : Retourne vrai (resp. faux) si et seulement si la collection contient tous les (resp. ne contient aucun des) éléments de la collection c.
- `forall(Expression :Boolean)` :Boolean : Vaut vrai si et seulement si l'expression `Expression` est vraie pour tous les éléments de la collection.
- `exists(Expression :Boolean)` :Boolean : Vrai si et seulement si au moins un élément de la collection satisfait l'expression `Expression`.
- `isUnique(Expression :Boolean)` :Boolean : Vrai si et seulement si l'expression `Expression` s'évalue avec une valeur distincte pour chaque élément de la collection.
- `any(Expression :Boolean)` :T : Retourne un élément arbitraire de la collection vérifiant l'expression `Expression`.
- `one(Expression :Boolean)` :Boolean : Vrai si et seulement si un et un seul élément de la collection vérifie l'expression `Expression`.
- `select(Expression :Boolean)` :Collection(T) : Retourne une collection du même type construite par sélection des éléments vérifiant l'expression `Expression`.
- `reject(Expression :Boolean)` :Collection(T) : Retourne une collection du même type construite par sélection des éléments ne vérifiant pas l'expression `Expression`.
- `including(Objet :T)` :Collection(T) , `excluding(Objet :T)` :Collection(T) : Retourne une collection résultant de l'ajout (resp. du retrait) de `Objet` à la collection.
- `union(c :Collection)` :Collection : Retourne l'union de deux collections.

L'appel d'une opération sur une collection est dénoté par le symbole `->`. Par exemple l'appel de l'opération `isEmpty()` sur la collection `c`, s'écrit : `c -> isEmpty()`.

```
species OCL_Collection(Obj is Basic_object) =
```

```

signature empty_collection : Self ;

(* C -> size():Integer : returns the number of elements of the collection C *)
signature size : Self -> int;

(* the empty collection has 0 element *)
property empty_collection_spec : size( empty_collection ) = 0;

(* C -> includes(obj:T):Boolean: returns true if obj belongs to
the collection C *)
let rec includes (e:Obj , c : Self ) = match c with
| [] -> false
| el::r -> if (e = el) then true else includes(e , r) ;

(* C -> includesAll(C'):Boolean: returns true if all elements of
the collection C' are contained in the collection C *)
signature includesAll : Self -> Self -> bool;
property includesAll_specif: all c1 c2 : Self,
  (all x : Obj,(includes(x, c1) -> includes(x, c2))) <->
  includesAll(c1, c2);

(* C -> excludes(obj:T):Boolean:
returns true if obj is not contained in the collection C *).
signature excludes : Obj -> Self -> bool;
property excludes_specif : all c : Self, all x : Obj,
  excludes( x , c) <-> ~(includes(x , c));

(* C -> excludesAll(C'):Boolean: returns true if no elements of
the collection C' is contained in the collection C *).
signature excludesAll : Self -> Self -> bool;
property excludesAll_specif: all c1 c2 : Self,
  (all x : Obj,(includes(x, c1) ->  ~~ (includes(x, c2))) ) <->
  excludesAll(c1, c2);

(* C -> count(obj:T):Boolean: returns how many times obj
appears in the collection C *)
signature count : Obj -> Self -> int;

(* C -> isEmpty():Boolean: returns true if the collection C is empty *)
let isEmpty ( c : Self): bool = (c = empty_collection);

(* C -> notEmpty():Boolean: returns true if the collection C is not empty *)
let notEmpty ( c : Self): bool = ~(isEmpty(c));

(* C -> forAll(t:T | expr: Boolean):Boolean: returns true if
all elements of C verify expr *)
signature forAll : Self -> ( Obj -> bool) -> bool ;
property forAll_specif : all c : Self, all p : Obj -> bool,
  (all x : Obj, includes(x, c) -> p(x) ) <-> forAll(c, p);

```

```

(* C -> exists(expr:Boolean):Boolean: returns true if exists
    an element of C verifying expr *)
signature exists : Self -> ( Obj -> bool) -> bool ;
property exists_specif : all c : Self, all p : Obj -> bool,
    ( ex x : Obj, includes(x, c) -> p(x) ) <-> exists(c, p);

(* C -> intersection(C'):Collection(T): Returns a collection
    containing all elements of the collection C that are also
    contained by C' *)

signature intersection : Self -> Self -> Self ;
property intersect_specif: all c1 c2 : Self, all x : Obj,
    includes( x , intersection(c1 , c2)) <->
    (includes(x , c1)) /\ (includes(x , c2));

(* C -> union(C'):Collection(T): Returns a collection containing
    all elements of the collection C and all elements of C' *)
signature union : Self -> Self -> Self ;
property union_specif: all c1 c2 : Self, all x : Obj,
    includes( x , union(c1 , c2)) <->
    (includes(x , c1)) \/ (includes(x , c2));

(* C -> select(t:T | expr: Boolean):Collection(T)}: returns a
    collection with all elements of C that validate the expression expr *)
signature select : Self -> ( Obj -> bool) -> Self ;
property select_specif: all c: Self, all p : Obj -> bool, all x : Obj,
    includes( x , select(c , p)) <->
    (includes(x , c)) /\ (p(x));

(* C -> reject(t:T | expr: Boolean):Collection(T)}: returns a collection
    with all elements of C except for those who validate the expression expr *)
signature reject : Self -> ( Obj -> bool) -> Self ;
property reject_specif : all c: Self, all p : Obj -> bool, all x : Obj,
    includes( x , reject(c , p)) <->
    (is_membre(x , c)) /\ ~~(p(x));
end;;

```

Annexe C

Espèces supports pour la transformation des associations

```
type nat =
  internal (* Internal#nat *)
  external
  | caml -> {* Nat *}
  | coq -> {* nat *}
;;

type unlimited_Nat = | Infini | I(nat) ;;

species Unlimited_Nat_spec = inherit Ordered_set;
representation = unlimited_Nat;

let make (x:unlimited_Nat):Self = x ;
let to_unlimited_Nat(x:Self): unlimited_Nat = x ;

let min : Self = I(0);

let element = min;

let max : Self = Infini;
  theorem unlimited_Nat_spec : all x : Self,
    !geq(x, min) /\ !leq(x, max)
  proof = assumed ;

proof of leq_reflexive = assumed ;
proof of leq_antisymmetric = assumed ;
proof of total_order = assumed ;
proof of leq_transitive = assumed ;

let leq (u1 : Self, u2: Self): bool = match (u1, u2) with
| ( I(x) , Infini) -> true
| ( Infini, I(x)) -> false
| ( Infini , Infini) -> true
```

```
| ( I(x), I(y) ) -> (x <=0x y) ;

end;;

collection Unlimited_Nat = implement Unlimited_Nat_spec ; end;;
species Range =
  representation = Unlimited_Nat * Unlimited_Nat ;
  let make_range(l :Unlimited_Nat , u : Unlimited_Nat ):Self = (l , u);
  let lower ( r : Self):Unlimited_Nat = fst(r) ;
  let upper ( r : Self):Unlimited_Nat = snd(r) ;
  end;;

type ass_direction = | LeftToRight| RightToLeft | Both ;;

species Direction =
  representation = ass_direction;
  let make_direction (d:ass_direction): Self = d ;
  let to_direction (d : Self):ass_direction = d ;
  end;;

collection Direction_collection = implement Direction; end;;

species Association
  (R is Range ,
   Left is Setoid , Left_Set is List(Left), multl in R ,
   Right is Setoid, Right_Set is List(Right), multtr in R,
   Dir is Direction, d in Dir ) =
  signature all_left_of_right : Right -> Left_Set ;
  signature all_right_of_left : Left -> Right_Set ;
  property multl_prop : all ri : Right,
    R!mult_leq( Left_Set!size(all_left_of_right(ri)), R!upper(multl) ) /\
    R!mult_geq ( Left_Set!size(all_left_of_right(ri)), R!lower(multl) ) ;
  property multtr_prop : all le : Left,
    R!mult_leq( all_Right_Set!size(right_of_left(le)), R!upper(multtr) ) /\
    R!mult_geq ( all_Right_Set!size(right_of_left(le)), R!lower(multtr) ) ;
  end;;
```