ROYAUME DU MAROC
Université Mohammed V - RABAT -
جامعة محمد الخامس – الرباط –
Faculté des sciences
كلية العلوم
CENTRE D'ETUDES DOCTORALES - SCIENCES ET TECHNOLOGIES

N° d'ordre : 3283

THESE

En vue de l'obtention du : ***DOCTORAT***

Structure de Recherche : Laboratoire Conception et Systèmes
Discipline : Informatique
Spécialité : Génie Logiciel

Présentée et soutenue le 10/01/2020 par :

Khadija EL MILOUDI

Approche Formelle pour la Spécification et la Vérification des Modèles UML en Utilisant la Notation Z

JURY

Mourad ELBELKACEMI	PES	Faculté des Sciences, Université Mohammed V - Rabat	Président
Aziz ETTOUHAMI	PES	Faculté des Sciences, Université Mohammed V - Rabat	Directeur de Thèse
Youssef ELBENANI	PES	Faculté des Sciences, Université Mohammed V - Rabat	Rapporteur / Examinateur
Noussaima ELKHATTABI	PH	Faculté des Sciences, Université Mohammed V - Rabat	Rapporteur / Examinateur
Rochdi MESSOUSSI	PES	Faculté des Sciences, Université Ibn Tofail - Kénitra	Rapporteur / Examinateur

Année Universitaire : 2019/2020

DEDICACE

A mes très chers parents ;
Pour leurs prières et leur soutien infaillible tout au long de ces années, avec toute ma reconnaissance et mon amour.
Puisse Dieu tout puissant vous protéger du mal, vous procurer longue vie, santé et bonheur afin que je puisse vous rendre un minimum de ce que je vous dois ;

A ma chère sœur Fatima Zohra et son mari Mohammed ; Ma chère sœur Khaoula et son mari Rachid ; A mon cher frère Mehdi ;
Merci pour votre support, encouragements et d'avoir toujours été là pour moi ;

A mon cher mari Karim qui m'a comblé de son amour, ses encouragements et son soutien quotidien indéfectible et qui a toujours répondu présent dans les moments les plus difficiles ;

A mes adorables filles Rita et Nada ;

A mes chers neveux Hafssa et Fahd ;

A la mémoire de mon oncle Hassan EL AZZOUZI EL IDRISSI qui aurait été fier de moi ;

A ma grand mère, mes tantes, mes oncles, mes cousins et cousines ;

A toute ma famille.

Je vous aime.

REMERCIEMENTS

Ce travail a été réalisé au sein du Laboratoire Conception et Systèmes à la Faculté des Sciences de Rabat sous la direction du professeur Aziz ETTOUHAMI.

C'est avec une immense gratitude que je tiens à exprimer ma reconnaissance envers les personnes qui m'ont encouragées et soutenues tout au long de ma thèse.

C'est à double titre que j'adresse mes remerciements à Monsieur **Aziz ETTOUHAMI**, Responsable du Laboratoire Conception et Systèmes à la Faculté des Sciences de Rabat et mon Directeur de thèse, d'une part pour m'avoir fait confiance dès le début de la thèse et avoir accepté de m'accueillir au sein du laboratoire et d'autre part pour sa disponibilité et ses remarques constructives qu'il m'a fournies ainsi que ses précieux conseils.

Je tiens également à remercier Monsieur **Mourad ELBELKACEMI**, Professeur d'Enseignement Supérieur et Doyen de la Faculté des Sciences de Rabat d'avoir accepté de présider le jury de thèse.

Je suis très reconnaissante à Monsieur **Youssef ELBENANI**, Professeur d'Enseignement Supérieur à la Faculté des Sciences de Rabat pour avoir pris le temps de rapporter cette thèse et d'avoir accepté de faire partie du jury.

Mes vifs remerciements vont également à Monsieur **Rochdi MESSOUSSI**, Professeur d'Enseignement Supérieur à la Faculté des Sciences de Kénitra d'avoir accepté de rapporter cette thèse et de participer au jury.

Un grand merci à Madame **Noussaima ELKHATTABI**, Professeur à la Faculté des Sciences de Rabat, pour l'intérêt qu'elle a porté pour ma thèse en acceptant d'être rapporteur et membre du jury.

Si j'en suis arrivée ici, c'est également grâce à ma famille qui ont cru en moi et m'ont soutenue durant ces longues années d'études. Je les remercie pour tous les sacrifices qu'ils ont faits, l'amour et le soutien qu'ils m'ont apportés.

RESUME

L'objectif de nos travaux de recherche consiste à proposer une approche formelle pour la validation et la vérification du standard UML en utilisant la notation Z. Nous avons étudié UML dans le cadre d'une modélisation multi vues à travers ses diagrammes les plus fréquemment utilisés. La modélisation multi vues présente plusieurs défis, notamment ceux liés à la cohérence, qui pourraient potentiellement mettre en péril l'intégrité du système. Nous avons exploité le langage Z qui, d'un côté, dispose d'une sémantique rigoureuse basée sur la logique mathématique et, d'un autre côté, offre suffisamment d'expressivité pour spécifier les différents diagrammes UML autant au niveau statique qu'au niveau dynamique. Nous avons proposé une modélisation formelle d'un ensemble de diagrammes UML en utilisant la notation Z. Une étude de la consistance est proposée à l'aide d'un ensemble de règles et de théorèmes en se basant sur le modèle formel proposé pour chaque diagramme.

Ainsi, les logiciels obtenus sont de qualité suffisante pour assurer une réelle crédibilité et fiabilité à leur contenu. Les solutions proposées dans cette thèse contribuent de manière significative à l'amélioration de la qualité des logiciels et de leurs consistances et à la réduction des coûts lors du processus de développement.

Mots clés : Méthodes formelles, UML, Z, Vérification, Inconsistances, Modélisation multi vues.

ABSTRACT

The objective of our research is to propose a formal approach for the validation and verification of the UML standard using Z notation. We studied UML as part of a multi-view modeling through its more frequently used diagrams. Multi-view modeling presents several challenges, including those related to consistency, which could potentially jeopardize the integrity of the system. We showed the interest of using the Z language which, on the one hand, has a rigorous semantics based on mathematical logic and, on the other hand, offers enough expressiveness to specify the different UML diagrams both static and dynamic. We proposed formal modeling of a set of UML diagrams using Z notation. A consistency study is proposed using a set of rules and theorems based on the formal model proposed for each diagram.

Thus, the software obtained is of sufficient quality to ensure real credibility and reliability to their content. The solutions proposed in this thesis contribute significantly to improving the quality of software and their consistencies and reducing costs during the development process.

Keywords: Formal Methods, UML, Z, Consistency Checking, Multi-view Modeling.

LISTE DE PUBLICATIONS

Journaux internationaux :

i. **Khadija El Miloudi** and Aziz Ettouhami, "A Multiview Formal Model of Use Case Diagrams Using Z Notation: Towards Improving Functional Requirements Quality", Journal of Engineering, vol. 2018, Article ID 6854920, 9 pages, 2018. https://doi.org/10.1155/2018/6854920.

ii. **Khadija El Miloudi** and Aziz Ettouhami, "A Multi-view Approach for Formalizing UML State Machine Diagrams Using Z Notation", WSEAS Transactions on Computers, vol. 14, pp. 72-78, 2015.

iii. **Khadija El Miloudi**, Younes El Amrani and Aziz Ettouhami, "Using Z Formal Specification for Ensuring Consistency in Multi-View Modeling", Journal of Theoretical & Applied Information Technology, vol. 57, no 3, 2013.

Conférences internationales :

iv. **Khadija El Miloudi**, Younes El Amrani and Aziz Ettouhami, "An Automated Translation of UML Class Diagrams Into a Formal Specification to Detect UML Inconsistencies". In Proceeding of 6th International Conference on Software Engineering Advances (ICSEA), Barcelona, Spain, pp. 432-438, 2011.

.

Journée doctorale :

v. **Khadija El Miloudi** and Younes El Amrani, "Approche Formelle pour la Détection des Inconsistances UML", Journées Doctorales en Systèmes d'information, Réseaux et Télécommunication, May 2012.

TABLE DES MATIERES

INTRODUCTION GENERALE

1. Introduction

Les systèmes automatiques développés aujourd'hui contiennent une proportion logicielle de plus en plus significative. Le défi actuel est de développer le meilleur logiciel qui convient le mieux aux clients. Au cours du développement des systèmes informatiques ou des logiciels, il existe différents ensembles de processus ou d'activités allant de la spécification et la conception à la phase de validation en passant par le codage et les tests. Compte tenu de la complexité des systèmes informatiques qui ne cesse d'évoluer au cours de ces dernières décennies, la phase de conception représente une phase primordiale durant le développement des logiciels qui les gèrent. Les parties logicielles d'un système sont particulièrement vulnérables aux erreurs de conception. Ainsi, les logiciels doivent être conçus de manière à répondre à des exigences de fiabilité et de consistance.

Dans un système orienté objet, le langage UML (Unified Modeling Language) est utilisé pour visualiser, capturer et documenter les exigences du logiciel. Actuellement, UML est représenté par quatorze diagrammes utilisés pour décrire les différentes vues d'un système que ce soit de point de vue structurel ou comportemental. Chacun des diagrammes UML est représenté par ses propres éléments graphiques. La syntaxe abstraite est utilisée pour montrer les attributs des éléments UML et leur relation avec d'autres éléments ainsi que leurs contraintes ou règles de bonne forme. Il ne suffit pas de montrer la description de chaque élément UML par la syntaxe abstraite uniquement, donc dans le standard UML, leur sémantique, leur notation et les contraintes ajoutées sont décrites en langage naturel et une partie des contraintes est prise en charge par le langage OCL (Object Constraint Language)(OMG OCL 2.2, 2010). Cependant, les contraintes UML manquent de syntaxe formelle conduisant ainsi à des modèles erronés ou incohérents. D'où la nécessité d'étudier les règles de consistances des modèles UML. Un modèle cohérent aide à l'implémentation des modèles, en veillant à ce que les modèles ne rencontrent pas de problèmes telles que les incohérences.

2. Problématique

La modélisation des systèmes en utilisant les notations graphiques réduit la complexité dans la phase de conception et permet une vue plus agréable du système en cours de développement. Cependant, les notations graphiques de modélisation les plus populaires souffrent d'une sémantique vague et ambigüe. En revanche, les méthodes formelles disposent d'une sémantique rigoureuse basée sur la logique mathématique. D'où l'intérêt de l'utilisation des méthodes

formelles qui permettent d'être plus précis dans le développement des logiciels et d'éliminer les erreurs de spécification et de conception.

Etant donné que le langage de modélisation unifié (UML) devient le langage standard de modélisation des systèmes informatiques, un nombre croissant de travaux de recherche se sont intéressés à la validation formelle des modèles utilisant la notation UML en vue d'identifier et résoudre les incohérences. Bien que plusieurs travaux ont été effectués afin d'améliorer le langage UML, sa sémantique reste toujours ambigüe conduisant ainsi à une interprétation erronée. Ces ambiguïtés peuvent être couteûses non seulement dans des domaines tels que les systèmes critiques pour la sécurité où des vies peuvent être perdues mais aussi dans les systèmes non critiques où les entreprises peuvent dépenser du temps et de l'argent à développer des correctifs.

3. Intérêts et objectifs de l'étude

Tandis que les modèles en général aident à réduire les erreurs de conception dans le processus de développement des logiciels et par conséquent améliorent la qualité des logiciels, la qualité des modèles eux-mêmes est rarement examinée. Alors en tenant compte de la cohérence des modèles UML à un stade précoce du processus de développement des logiciels, la qualité du logiciel qui en résultera sera indirectement améliorée. Puisque la cohérence est l'un des facteurs principaux des problèmes rencontrés au moment de l'implémentation des modèles, la formalisation des règles syntaxiques et de consistance est la solution pour pallier à ce problème.

Ainsi, l'objectif de nos travaux de recherche consiste à proposer une approche formelle pour la validation et la vérification du standard UML à travers ses diagrammes les plus fréquemment utilisés. Ce travail de recherche est consacré essentiellement à l'étude de la consistance de quatre diagrammes UML notamment le diagramme de classes, le diagramme de séquences, le diagramme d'états et le diagramme des cas d'utilisation. Une approche formelle a été présentée pour améliorer la qualité des modèles de logiciels construits en utilisant le langage de modélisation UML (OMG UML 2.5.1, 2017). Nous abordons ce sujet en explorant profondément la vérification et la validation des modèles UML en utilisant la notation Z (Spivey, 1992). A partir du modèle UML, nous construisons une spécification formelle en utilisant la notation Z afin de valider le modèle. L'intérêt d'aborder la modélisation multi-vue en utilisant un langage de spécification formelle tel que Z réside dans sa sémantique rigoureuse

fondée sur des bases mathématiques. En outre, le langage Z offre suffisamment d'expressivité pour spécifier les différents diagrammes UML autant au niveau statique qu'au niveau dynamique. De plus, des outils de preuves automatiques supportant le langage Z tel que Z/EVES (Meisels, 2004; Saaltink, 1997), sont disponibles pour la vérification des propriétés et la validation des spécifications. Ainsi, les logiciels obtenus sont de qualité suffisante pour assurer une réelle crédibilité et fiabilité à leur contenu.

4. Contributions et organisation de la thèse

Les synthèses réalisées et les réflexions développées dans le cadre de ce travail de recherche ont fait l'objet de trois chapitres répartis comme suit :

Dans le premier chapitre, nous présentons brièvement les concepts et les notions de base des langages étudiés à travers cette thèse et qui représentent le cadre théorique de nos contributions notamment le langage de modélisation unifié UML et la notation Z. Ce premier chapitre est consacré également à une revue de la littérature des principaux travaux effectués sur la modélisation formelle des diagrammes UML.

Nos contributions à travers cette thèse sont organisées en deux chapitres. Le deuxième chapitre présente une sémantique formelle du diagramme de classes en utilisant la notation Z. Un prototype de transformation automatique d'UML vers le langage Z a été développé afin de valider l'approche. Pour pallier aux problèmes d'ambiguïté au niveau du diagramme de classes, le modèle formel proposé a été utilisé afin de détecter la plupart des inconsistances publiées à ce jour.

Le troisième chapitre est dédié à la consistance des modèles à vues multiples. Une sémantique formelle des diagrammes dynamiques les plus fréquemment utilisés est proposée en utilisant la notation Z. Nous nous sommes intéressés aux diagrammes suivants : le diagramme de séquences, le diagramme d'états et le diagramme des cas d'utilisation. Ce chapitre propose également une formalisation des différentes règles de cohérence définies par le standard UML.

Enfin, ce document s'achève par une conclusion générale ainsi que les perspectives que nous avons tracées pour poursuivre ce travail.

Chapitre I : CONTEXTE DE L'ETUDE ET REVUE DE LITTERATURE

I.1. Avantages et inconvénients du langage de modélisation UML

UML (OMG UML 2.5.1, 2017) est un langage de modélisation largement utilisé en industrie du logiciel qui permet de visualiser, spécifier, construire et documenter les objets d'un système logiciel. UML est décrit par une notation graphique qui définit la manière dont le langage est présenté aux concepteurs. La syntaxe est décrite elle-même avec la notation UML constituant ainsi le méta-modèle UML. Le méta-modèle UML présente les concepts du langage ainsi que les relations entre les différents objets du langage. L'utilisation des concepts UML est restreinte par des règles de bonne formation écrites en OCL (OMG OCL 2.2, 2010) ou en langage naturel explicitant l'usage et la signification des concepts du méta-modèle. Les différents types de diagrammes UML offrent une vue complète des aspects statiques et dynamiques d'un système. La notation UML est décrite par un ensemble de diagrammes regroupés dans deux classes principales : statiques et dynamiques.

- Les diagrammes statiques regroupent les diagrammes de classes, les diagrammes d'objets, les diagrammes de structure composite, les diagrammes de composants, les diagrammes de déploiement, les diagrammes de profils et les diagrammes de paquetages.
- Les diagrammes dynamiques regroupent les diagrammes de séquence, les diagrammes d'états, les diagrammes de communication, les diagrammes d'activités, les diagrammes d'interaction, les diagrammes de temps et les diagrammes des cas d'utilisation.

Le langage de modélisation unifié UML s'est rapidement établi comme étant le standard de modélisation et de spécification de logiciels grâce à sa notation graphique simple d'utilisation et facile à comprendre. Cependant, les concepts et les notations de modélisation de la notation UML restent ambigus, et souffrent d'un manque de précision sémantique pour l'analyse rigoureuse des modèles.

I.2. Spécification formelle : langage et outils de preuves

La complexité qui caractérise les systèmes développés récemment entraine une difficulté à comprendre et à concevoir correctement les logiciels qui les gèrent. La définition d'un modèle formel s'avère alors nécessaire pour remédier à ce problème en utilisant des langages formels tels que Z (Spivey, 1992), B (Abrial, 2005) ou VDM (Jones, 1991) qui se caractérisent par une sémantique précise. Nous avons adopté pour le reste de ce travail la notation Z pour la

spécification formelle des modèles orientés objets. Nous présentons dans ce qui suit les concepts de base de la notation Z ainsi que son système de preuves Z/EVES (Meisels, 2004; Saaltink, 1997).

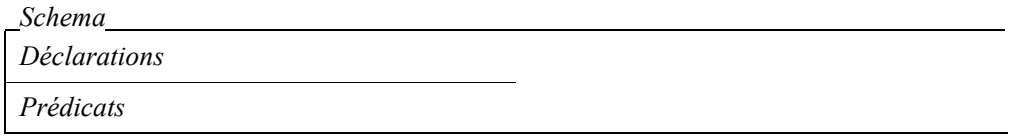
I.2.1. Introduction à la notation Z

Contrairement à UML, Z est un langage de spécification formelle créé par J.R Abrial basé sur la théorie des ensembles et la logique mathématique. En Z, une spécification utilise la notion de schéma qui permet de structurer les mathématiques sous-jacentes et de réutiliser les différentes parties d'une spécification. Comme présenté dans (Woodcock and Davies, 1996), la notation Z est composée de deux langages : le langage mathématique et le langage de schémas. Le langage mathématique est utilisé pour décrire les différents aspects de la conception : les objets, et les relations entre eux. Le langage de schémas est utilisé pour structurer et composer les spécifications : rassembler des éléments d'information, les encapsuler, et les nommer pour une réutilisation

Selon (Woodcock and Davies, 1996), un schéma est une structure décrivant un ensemble de variables dont les valeurs sont soumises à des contraintes. Un schéma est composé de deux parties :

- La partie déclarative contient la déclaration des variables d'état.
- La partie prédicative définit les contraintes sur les valeurs des variables d'état. Ces prédicats expriment des propriétés sur les variables d'état et introduisent les relations entre elles.

Un schéma peut être utilisé ou réutilisé comme une déclaration, un type ou un prédicat. Un schéma est représenté comme suit :



Les nouveaux types sont introduits comme des ensembles donnés pour servir comme type de base dans la spécification. Un ensemble donné est introduit entre crochets. Par exemple, pour introduire un ensemble donné nommé *Objet*, nous écrivons : [*Objet*].

Une spécification Z regroupe deux aspects principaux : statique et dynamique. L'aspect statique est représenté par les schémas qui définissent les différents états du système ainsi que

les propriétés invariantes de chaque état. L'aspect dynamique est décrit par les opérations qui modifient ou non l'état ainsi que la relation entre les entrées et les sorties.

Le schéma Δ*Schema* spécifie les relations entre l'état avant l'opération (*Schema*) et l'état après l'opération (*Schema'*). Ce schéma doit être inclus chaque fois que nous voulons décrire une opération qui peut changer l'état. De même, le schéma Ξ*Schema* doit être inclus dans la description de toute opération qui ne modifie pas l'état du système. Les entrées et les sorties d'une opération sont représentées par le nom de la variable suivi respectivement par un point d'interrogation ou un point d'exclamation. Les paramètres d'une opération sont insérés dans la partie déclarative comme le montre l'exemple ci-dessous.

Opération

Δ*Schema*
e? : *Entrée*
s! : *Sortie*
Prédicats

Notre choix a porté sur la notation Z car elle est basée sur la logique mathématique et la théorie des ensembles qui permet par la suite une vérification rigoureuse des spécifications. En plus, le langage de schémas utilisé par Z a été parfaitement adapté pour représenter l'ensemble des constructions UML étudiées. Et afin de composer, vérifier et analyser une spécification formelle en Z, la notation Z est soutenue par plusieurs outils de preuves tels que Z/EVES (Meisels, 2004; Saaltink, 1997), ZTC ou Fuzz.

Et pour nos travaux de recherche, nous avons choisi le système Z/EVES afin de valider les spécifications car il permet une vérification syntaxique et sémantique aussi bien que la preuve de théorèmes. Contrairement aux autres outils, il dispose également d'une interface interactive qui facilite la composition et la vérification des spécifications. Ainsi, nous présentons dans ce qui suit une brève description du système Z/EVES que nous avons utilisé pour valider notre approche tout au long de notre travail de recherche.

I.2.2. Outil de preuve formelle : Z/EVES

Z/EVES (Meisels, 2004; Saaltink, 1997) est un système interactif pour la composition, la vérification, l'analyse des spécifications Z et la preuve de théorèmes. Z/EVES fonctionne en deux modes indépendants : mode textuel ou mode graphique. L'utilisation de l'interface

graphique est plus ergonomique pour la construction des spécifications. Les spécifications sont organisées par paragraphes. Les paragraphes vérifiés peuvent être modifiés, revérifiés ou supprimés.

Z/EVES supporte la preuve de théorème de différentes manières. Le démonstrateur de preuves de Z/EVES combine des stratégies automatiques et l'intervention détaillée de l'utilisateur, ce qui permet un effort de collaboration dans l'achèvement des preuves. Diverses commandes du prouveur qui sont offertes par Z/EVES peuvent être utilisées pour explorer l'espace preuve, et, avec un peu de chance, le but peut être prouvé. Dans ce cas, l'utilisateur souhaite utiliser au maximum les commandes automatiques du démonstrateur (par exemple : *prove*, ou *reduce*). En appliquant une commande de preuve, le but est transformé en un nouveau but équivalent. Une fois le but est transformé au prédicat *true*, la preuve est alors terminée.

I.3. Revue de littérature

Cette partie présente une synthèse bibliographique sur les approches et techniques liées à la formalisation des diagrammes UML ainsi que l'étude de la consistance dans le cadre d'une modélisation orientée objet à vues multiples. Nous nous sommes intéressés aux travaux liés à la formalisation des diagrammes UML les plus utilisés dans la modélisation des logiciels orientés objet notamment le diagramme de classes, de séquences, d'états et le diagramme des cas d'utilisations. Nous exposons, à travers cette revue de littérature, leurs limites comparées à leurs avantages.

I.3.1. Formalisation du diagramme de classes

Il existe différentes approches pour la formalisation du diagramme de classes dans le but de détecter les incohérences dans les modèles UML. Une méthode de détection automatique de contradictions dans les diagrammes de classes UML a été introduite par Satoh *et al.*(Satoh et al., 2006). Deux types d'inconsistances ont été détectés : les multiplicités contradictoires et la violation de la contrainte de disjonction. Une sémantique de UML en termes de logique de premier ordre a été utilisée afin de traduire le diagramme de classes en un programme logique. Cependant, la compréhension de cette approche nécessite des connaissances approfondies en mathématiques et en logique du premier ordre.

Falah et *al.* (Falah et al., 2018) ont proposé une formalisation des diagrammes de classes en utilisant la théorie des ensembles. Leur transformation a été basée sur un ensemble universel qui comprend tous les éléments utilisés par le diagramme tel que les classes, les objets et les messages. Ils ont abordé à travers leur étude les différents aspects du diagramme de classes sauf qu'aucune étude de la consistance n'a été abordée et leur formalisation n'a pas été vérifiée ou validée à l'aide d'un outil.

Singh et *al.* (Singh et al., 2016) ont présenté une formalisation des diagrammes de classes, de séquences et de cas d'utilisation en utilisant la notation Z sauf que leur approche s'est contentée uniquement des notions basiques et leur formalisation est restée vague sans spécifier chaque élément en détails et sans aucune étude de la consistance.

Liu *et al.* ont proposé dans (Liu et al., 2002) une définition d'un langage et des règles de système de production spécifiques aux modèles basés sur UML. Ce système vise à détecter les inconsistances, informer les utilisateurs et corriger automatiquement les inconsistances durant le processus de modélisation. Ce système de production utilise le moteur de règles Jess (Liu et al., 2002).

Dans le même contexte, l'outil RoZ (Dupuy et al., 2000) a été développé afin de générer automatiquement des spécifications formelles à partir de diagrammes de classes UML annotés en Z. Par contre, il ne propose pas la détection des incohérences structurelles. Ainsi que l'outil RoZ nécessite des connaissances en Z à l'étape de conception UML.

Amalio, Stepney et Polack (Amálio et al., 2004, 2010) proposent une représentation formelle des diagrammes de classes UML. Bien que la spécification formelle ainsi obtenue permet d'exprimer et de vérifier certaines propriétés sur le modèle, elle ne traite que les incohérences structurelles sur une étude de cas spécifique et non du modèle en général.

I.3.2. Formalisation du diagramme de séquences

Une grande variété d'approches pour la formalisation des diagrammes comportementaux a été proposée dans la littérature afin de vérifier la cohérence des modèles.

Dubauskaite et Vasilecas (Dubauskaite and Vasilecas, 2013) ont choisi d'utiliser UML pour exprimer des règles de cohérence entre les différentes vues d'un système basé sur UML. Les règles sont définies au niveau du méta-modèle. Cependant, UML en soi est un langage semi formel qui dispose d'une sémantique encore ambigüe.

Zafar (Zafar, 2016) a proposé une formalisation des diagrammes de séquences en utilisant la notation Z. L'approche qu'il a adoptée a couvert l'ensemble des concepts liés au diagramme de séquences. Par contre, aucune vérification de la consistance n'a été proposée.

Dans (Ledang and Souquières, 2001a), un cadre destiné à dériver des spécifications B à partir de diagrammes structurels et comportementaux a été proposé. La conformité entre deux aspects de spécification UML peut être formellement vérifiée par l'analyse de la spécification B correspondante. Leur proposition a été appliquée pour obtenir automatiquement des spécifications B à partir de diagrammes de classes et d'interaction. Cette approche est similaire à la nôtre en termes d'utilisation des méthodes formelles.

Dans (Litvak et al., 2003), les auteurs ont utilisé une approche algorithmique afin de vérifier la cohérence entre le diagramme de séquence et le diagramme d'état. L'outil BVUML a été mis en œuvre pour automatiser le processus de validation. Cependant, la vérification de la cohérence entre la vue statique et dynamique du système n'a pas été proposée.

Une sémantique formelle du diagramme de séquences UML a été présentée par Li *et al.* (Li et al., 2004). La sémantique capture la cohérence entre le diagramme de séquences, le diagramme de classes et le diagramme d'états. Le diagramme de séquences est représenté par une structure hiérarchique arborescente ordonnée.

Lopez-Herrejon et Egyed (Lopez-Herrejon and Egyed, 2010) ont proposé la composition sûre comme technique de vérification de la cohérence des modèles à vues multiples avec variabilité. Un ensemble représentatif de règles de cohérence UML et une technique de composition de fonctions sont utilisés.

Xia et Kane (Xia and S Kane, 2003) ont défini la sémantique des diagrammes de classes et de séquences UML en utilisant un ensemble de notations de base. Leur approche est basée sur une grammaire d'attributs reflétant les propriétés sémantiques des programmes. Un ensemble d'axiomes qui représente les principes de la programmation orientée objet a été défini.

Treize règles de cohérence ont été données par Liu (Liu, 2013) pour repérer les incohérences les plus fréquentes entre les six types de diagrammes UML les plus utilisés dans la modélisation des systèmes d'information. Quatre méthodes ont été prévues pour vérifier les incohérences entre les diagrammes UML. Les quatre méthodes sont : contrôle manuel, la restriction obligatoire, maintenance automatique et contrôle dynamique. Ce travail a contribué de manière significative à la définition des règles de cohérence, cependant il ne propose pas une solution au problème de consistance qui soit précise incluant à la fois la formalisation et la détection des incohérences.

I.3.3. Formalisation du diagramme d'états

Plusieurs études ont été menées pour formaliser les diagrammes UML et en particulier le diagramme d'états afin d'améliorer ses lacunes.

Tian et Gu (Tian and Gu, 2013) ont présenté une approche de modélisation formelle et de validation des processus de développement logiciel, qui transforme les modèles UML basés sur Rational Unified Process (RUP) en réseau de Petri coloré (CPN) en utilisant des outils CPN pour étudier le comportement du système modélisé.

Une approche pour la modélisation des diagrammes d'états en B est proposée par Ledang et Souquières (Ledang and Souquières, 2001b). Seule la modélisation des concepts UML en B est considérée. L'analyse de la spécification provenant de B n'est pas traitée.

Meng et al. (Meng et al., 2005) ont fourni une formalisation pour le diagramme d'états UML en utilisant le langage de spécification RAISE.

McUmber et Cheng (McUmber and Cheng, 2001) ont introduit un cadre général pour la formalisation d'un sous-ensemble de diagrammes UML en termes de différents langages formels. Cette formalisation est fondée sur une correspondance entre des méta-modèles décrivant UML et un langage formel. UML est formalisé en termes de Promela.

Latella et al. (Latella et al., 1999) ont défini la base pour le développement d'une sémantique formelle pour les diagrammes d'états UML basée sur des structures de Kripke (Latella et al., 1999). Des règles de transformation des diagrammes d'états au format intermédiaire sous forme d'automates hiérarchiques ont été proposées et une sémantique opérationnelle pour ces automates a été définie.

Une sémantique formelle pour un sous-ensemble du diagramme d'états est proposée par David et al. (David et al., 2003). Mostafa et al. (Mostafa et al., 2007) ont présenté une formalisation des différents diagrammes UML en utilisant la notation Z. Par contre, la cohérence entre eux n'a pas été traitée.

Le travail effectué par Santos et al. (Santos et al., 2014) a présenté une approche pour transformer jusqu'à trois différents diagrammes comportementaux (diagramme de séquences, d'états et d'activités) en un système de transition unique pour supporter la vérification de modèles de logiciels basés sur UML, néanmoins les incohérences entre les différents diagrammes ne sont pas détectées.

Une traduction automatique des diagrammes comportementaux de UML vers des modèles formels est proposée par Fernandes et Song (Fernandes and Song, 2014) afin d'être vérifiés par

un vérificateur symbolique de modèle. En outre, l'étude de la consistance multi-vues n'a pas été abordée par ce travail.

Une étude réalisée par Liu et al. (Liu et al., 2013) utilise les systèmes de transitions étiquetées comme modèle sémantique pour fournir une sémantique formelle aux diagrammes d'états UML. Snook et al. (Snook et al., 2012) ont proposé une vérification des diagrammes d'états en les traduisant vers UML-B en utilisant la plate-forme Rodin et ses outils de preuve automatique.

I.3.4. Formalisation du diagramme de cas d'utilisation

La formalisation des diagrammes de cas d'utilisation UML a beaucoup attiré l'attention de la communauté des chercheurs car elle offre un potentiel considérable pour aider les développeurs à analyser si la spécification correspond aux exigences souhaitées. Dans cette section, nous présentons une revue de littérature montrant les approches qui définissent la sémantique des diagrammes de cas d'utilisation UML. Deux grandes écoles de recherche travaillent sur ce sujet.

La première école consiste à proposer des techniques et des outils pour transformer les diagrammes de cas d'utilisation en différents formalismes. Citons les principaux auteurs de cette école.

Par exemple, Junior et al. (Oliveira et al., 2015) proposent une formalisation des cas d'utilisation à l'aide de modèles de transformation de graphes dans le but d'exécuter des analyses basées sur les outils et de révéler les erreurs possibles.

Sinnig et al. (Sinnig et al., 2013) proposent une méthodologie de développement intégrée basée sur un formalisme de machine à états finis non déterministe, qui définit une sémantique commune pour les cas d'utilisation et les modèles de tâches.

Butler et al. (Butler et al., 1997) définit la notion de cas d'utilisation et la terminologie associée au moyen d'une spécification écrite en Z. L'accent est mis sur la formalisation et la clarification des notions de base du cas d'utilisation et de ses concepts associés. Par contre il ne propose pas la spécification des différents composants et contraintes imposées aux cas d'utilisation. La formalisation des acteurs et la relation d'héritage entre eux n'ont pas été abordées.

Barajas (Barajas, 2006) présente la spécification formelle d'un outil modélisant les exigences fonctionnelles d'un système basé sur des diagrammes de cas d'utilisation utilisant Alloy.

Scandurra et al. (Scandurra et al., 2012) proposent une structure nommée asmetaRE qui transforme automatiquement les diagrammes de cas d'utilisation en spécifications exécutables

ASM, puis valide les exigences du système par le biais de simulations par scénarios de l'ASM généré.

Shen et Liu (Shen and Liu, 2003) proposent une technique de révision, comprenant l'exécution et les tests, qui peut être appliquée aux modèles de configuration logicielle utilisant un nouveau langage HCL (langage de contrainte de haut niveau).

Murali et al. (Murali et al., 2015) décrivent une approche permettant d'incorporer une méthode formelle dans la modélisation de cas d'utilisation UML à l'aide d'event-B. Ils élargissent la modélisation des cas d'utilisation pour permettre la représentation explicite des problèmes de sécurité.

Mostafa et al.(Mostafa et al., 2007) proposent de formaliser la syntaxe des diagrammes UML courants (diagramme de cas d'utilisation, diagramme de classes et diagramme d'états) en utilisant la spécification Z. Ils formalisent chaque notion de base de diagramme de cas d'utilisation de façon isolée sans la spécifier dans son ensemble ainsi que par rapport aux différentes vues du système.

Muhamad et ses collègues (Muhamad et al., 2019) ont proposé une formalisation d'un système de gestion d'une librairie en utilisant la notation Z. Ce système est représenté par un diagramme des cas d'utilisation et un diagramme d'activité. Leur formalisation est destinée à un cas spécifique et ne traite pas la sémantique du diagramme des cas d'utilisation.

Alors que la première école n'a ciblé que les problèmes de formalisation, la seconde se concentre sur la cohérence multi-vues entre le diagramme de cas d'utilisation et les autres diagrammes UML.

Par exemple, Yue et al.(Yue et al., 2010, 2015) proposent une méthode et un outil appelé aToucan pour générer automatiquement, à partir d'un modèle de cas d'utilisation, un modèle d'analyse UML comprenant un diagramme de classe, de séquence et d'activité. Cependant, un diagramme de cas d'utilisation ne dispose pas d'informations suffisantes pour générer un modèle d'analyse UML complet.

Giese et Heldal (Giese and Heldal, 2004) présentent un moyen de relier les exigences informelles sous forme de cas d'utilisation à des spécifications plus formelles écrites dans OCL.

Ibrahim et al. (Ibrahim et al., 2012) proposent un ensemble de règles de cohérence entre le diagramme de cas d'utilisation, le diagramme de séquence et le diagramme de classe. La syntaxe abstraite des règles de cohérence proposées est formellement définie à l'aide d'une approche logique.

I.3.5. Synthèse et conclusion

Dans la plupart des travaux réalisés dans le domaine de validation des diagrammes UML, beaucoup d'efforts ont été déployés dans la définition des diagrammes en termes de logique de premier ordre, de systèmes de transitions étiquetées, avec un langage intermédiaire ou sous forme d'automates en se concentrant exclusivement sur la sémantique. Dans l'ensemble, ces études soulignent la nécessité de formaliser les diagrammes UML. Cependant, ces études restent peu ciblées et seulement peu de travaux qui ont abordé les deux aspects à savoir la formalisation et l'étude de la consistance multi vues. Nous contribuons à travers cette thèse à l'étude de la sémantique de différents diagrammes UML en utilisant la notation Z tout en vérifiant leurs consistances par rapport aux différentes vues du système modélisé. Notre approche consiste à tirer profit des avantages des méthodes formelles notamment la notation Z en les appliquant à la conception des logiciels. L'objectif est de pouvoir offrir aux développeurs la possibilité de raisonner, analyser et vérifier l'architecture d'un système logiciel de façon rigoureuse. Il va sans dire que, grâce à notre approche, clients et développeurs peuvent bénéficier des avantages des langages formels pour améliorer la qualité des exigences et, par conséquent, la qualité du logiciel obtenu. Il est à noter que notre approche est le premier travail sur la validation des diagrammes UML dans le cadre d'une modélisation à vues multiples en utilisant la notation Z.

Chapitre II : SPECIFICATION FORMELLE DE LA VUE STATIQUE DES SYSTEMES ORIENTES OBJET

II.1. Présentation du diagramme de classes et ses principaux composants

Le diagramme de classes est l'un des diagrammes fondamentaux les plus utilisés en UML. Il fournit presque toutes les fonctionnalités nécessaires à la modélisation structurelle d'un système orienté objet. Nous introduisons dans ce qui suit les concepts de base liés aux diagrammes de classes UML.

Dans la modélisation orientée objet, une classe décrit l'état et le comportement des objets de la classe. Les objets d'une classe sont aussi appelés les instances de classe. Une classe est définie en plus de son nom par des attributs et des méthodes. Les attributs sont des propriétés définies par un nom et un type et éventuellement une multiplicité. Quant aux méthodes, elles spécifient le comportement des instances de la classe. Considérons le diagramme de classes d'un système d'information des cavaliers, proposé par (Hall, 1994), représenté sur la Figure II-1. Le diagramme de classes se compose de plusieurs classes. La classe *Rider* est définie par un ensemble d'attributs et une méthode appelée *changeName*.

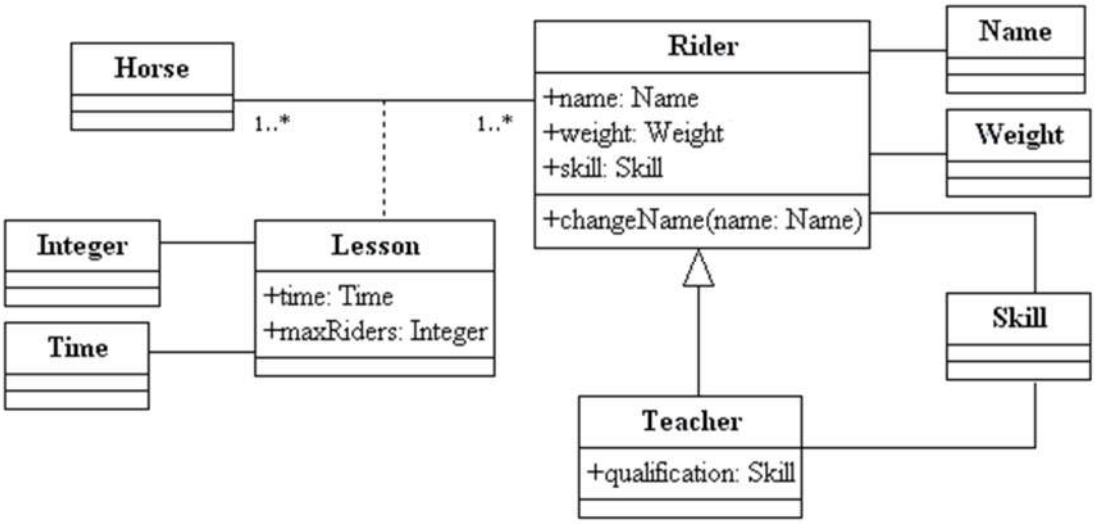


Figure II-1: Diagramme de classes d'un système de gestion des cavaliers

Les classes sont liées entre elles par des relations qui entrainent des connexions entre leurs instances. Ces relations sont représentées en UML par des associations. Les associations peuvent être nommées par un texte unique décrivant la sémantique de l'association. Une

association UML peut contenir des informations supplémentaires telles que les rôles et les multiplicités. La multiplicité précise le nombre d'instances pouvant être liées par une extrémité d'association à une instance pour chaque autre extrémité d'association. Dans le cas où une association possède des propriétés ou des opérations, il est possible de la qualifier à l'aide d'une classe-association. La classe association *Lesson* définie dans la Figure II-1 illustre la relation entre les classes *Rider* et *Horse*.

UML offre aussi l'un des mécanismes de réutilisation les plus importants de la conception orientée objet à travers la notion de l'héritage. Le but de l'héritage ou la généralisation est de permettre aux classes possédant des attributs communs de factoriser ces propriétés dans une super-classe pour conserver au niveau de la classe de base ses propriétés spécifiques. L'exemple illustré dans la Figure II-1 montre une relation d'héritage entre les classes *Teacher* et *Rider*. Un objet *Rider* peut désigner une instance des classes *Rider* et *Teacher*.

Nous présentons dans la partie suivante un modèle formel d'un sous ensemble des diagrammes de classes en Z. Le passage du diagramme UML vers une spécification formelle en Z s'effectue suivant des règles de traduction que nous présentons dans ce qui suit à l'aide d'exemples pour valider notre approche (El Miloudi et al., 2011). Nous reprenons l'exemple du système des cavaliers traité par Anthony Hall (Hall, 1994) dont le diagramme de classes est défini précédemment. Nous présentons également une étude des cas d'inconsistances les plus soulevées dans la littérature ainsi que leurs formalisations en Z. Nous exposons ensuite le prototype que nous avons développé à l'aide de la technologie XSLT afin de traduire automatiquement un diagramme de classes vers une spécification Z.

II.2.Modèle formel

Nous présentons dans ce qui suit notre approche pour la formalisation d'un sous ensemble des diagrammes de classes. Il est important de noter que notre étude ne traite pas les diagrammes de classes de façon exhaustive. La formalisation a été limitée aux aspects les plus pertinents.

Dans la littérature, de nombreux formalismes ont été appliqués pour la modélisation des concepts orientés objets en Z (Amálio and Polack, 2003). Notre première contribution est basée sur le modèle d'Anthony Hall de spécification et d'interprétation des concepts orientés objets en Z (Hall, 1990, 1994). Nous appliquons ce modèle au diagramme de classes UML dans le but d'étudier les différents cas d'inconsistances. Le modèle d'Anthony Hall a été choisi car il est référencé par la plupart des études traitant la modélisation des systèmes orientés objets ainsi

qu'il permet de modéliser tous les concepts nécessaires pour l'étude des inconsistances liées aux diagrammes de classes. La clarté et la simplicité du modèle d'Anthony Hall nous a permis par la suite d'introduire les différentes modifications que nous avons apportées. Le passage du diagramme UML vers une spécification formelle en Z s'effectue suivant des règles de traduction que nous présentons dans cette partie à travers l'exemple du système d'information des cavaliers introduit précédemment dans la Figure II-1.

II.2.1. Représentation des Classes

Lors du passage du diagramme de classes UML vers une spécification formelle en Z, nous commençons par déclarer en majuscule tous les noms de classes comme des variables de type *OBJECT*. Ces variables représentent l'ensemble d'identifiants des objets de chaque classe. Nous supposons que toutes les identités d'objets sont du même type *OBJECT* déclaré sous forme de type donné (given set).

$$[OBJECT]$$

$$RIDER : \mathbb{P} OBJECT$$

Ensuite, toute classe donne lieu à deux schémas. Le premier schéma contient, en plus des attributs et leurs types, un attribut *self* qui représente l'identifiant de chaque instance. Ce schéma porte le nom de la classe suivi de '*CoreClass*'.

RiderCoreClass

$$self : RIDER$$
$$name : Name$$
$$weight : Weight$$
$$skill : Skill$$

Avant de spécifier le deuxième schéma, nous déclarons un type énuméré qui prend le nom de la classe contenant soit la valeur nulle, soit un objet de la classe.

$$Rider ::= nilRider \mid ridercoreclasstorider \langle\langle RiderCoreClass \rangle\rangle$$

Le deuxième schéma représente l'ensemble des instances de la classe.

$\mathbb{S}Rider$

$riders: \mathbb{P}\ Rider$
$idRider: RIDER \pfun Rider$
$riderIds: \mathbb{P}\ RIDER$

$idRider$
$= \{\ ridercoreclass: RiderCoreClass$
$\bullet\ ridercoreclass.self \mapsto ridercoreclasstorider\ ridercoreclass\ \}$
$riderIds = \mathrm{dom}\ idRider$

Dans l'exemple ci-dessus, *riders* représente l'ensemble des cavaliers reconnus par le système et *ridersIds* indique l'ensemble de leurs identifiants. Ce schéma introduit aussi une fonction *idRider* qui lie chaque identifiant au cavalier correspondant.

II.2.2. Initialisation

Pour chaque schéma de classe, une étape d'initialisation est nécessaire pour définir l'état initial du système. Nous proposons deux types d'initialisation.

Le premier type considéré comme une initialisation par défaut, initialise les attributs avec des valeurs nulles.

InitRiderCoreClassByDefault

$RiderCoreClass'$

$name' = nilName$
$weight' = nilWeight$
$skill' = nilSkill$

Le deuxième type d'initialisation prend des valeurs en entrée correspondant aux types des attributs.

InitRiderCoreClassWithValues

$RiderCoreClass'$
$name?: Name$
$weight?: Weight$
$skill?: Skill$

$name' = name?$
$weight' = weight?$
$skill' = skill?$

Une fois que la classe est initialisée, le schéma ci-dessous nous permettra d'ajouter cette instance au système.

AddRiderToSystem

InitRiderCoreClassByDefault
$\Delta SRider$

$\theta\ RiderCoreClass'.self \notin \text{dom}\ idRider$
$idRider'$
$= idRider$
$\cup \{(\theta\ RiderCoreClass'.self \mapsto ridercoreclasstorider\ \theta\ RiderCoreClass')\}$

Nous introduisons le schéma $\Delta SRider$ afin d'exprimer l'effet de la création de cet objet sur l'ensemble du système. Les identités étant uniques, il est nécessaire de vérifier que l'identité choisie pour le nouvel objet est distincte de toutes les autres identités dans le système. Le nouvel état de l'ensemble du système dispose désormais du nouvel objet ajouté à *idRider*.

II.2.3. Représentation des opérations

Formellement, les opérations sont représentées par un schéma qui prend le nom de l'opération suivi du nom de la classe. Chaque opération inclut un schéma qui indique si l'état du système sera modifié ou restera inchangé. Ce schéma nous garantit aussi que l'identificateur de l'objet (*self*) reste inchangé.

Si l'opération ne change pas l'état du système, le schéma *RiderGet* ci-dessous est inclut dans la partie déclarative du schéma de l'opération.

RiderGet

Ξ*RiderCoreClass*

---

self′ = self

Dans le cas où l'opération change l'état du système, le schéma *RiderOp* est inclut dans le schéma de l'opération.

RiderOp

Δ*RiderCoreClass*

---

self′ = self

Selon le modèle d'Anthony Hall, une opération donne lieu à plusieurs schémas :

Lors de la génération automatique de la spécification formelle à partir du diagramme de classes UML, le corps de la fonction n'étant pas défini en UML, seule la signature de l'opération est déclarée.

changeName

RiderOp

name?:Name

Nous définissons, pour chaque opération, une fonction qui relie les paramètres en entrée de l'opération à l'état du système avant et après l'opération.

R*changeName: Name* → *Rider* → *Rider*

R*changeName*

= { *name : Name*

• *name*

↦ { *changeName* | *name? = name*

• (*ridercoreclasstorider* θ *RiderCoreClass*

↦ *ridercoreclasstorider* θ *RiderCoreClass′*) } }

Ensuite, nous décrivons à l'aide d'un troisième schéma l'effet de l'opération sur l'ensemble du système en fournissant en entrée l'objet modifié.

changeNameSystem

$\Delta SRider$
rider? : RIDER
name? : Name

---

idRider' = idRider ⊕({rider?}◁idRider ⨾ RchangeName name?)

La nouvelle valeur de *idRider* après l'opération est le résultat de la composition de la fonction *RchangeName* appliquée à *name?* avec un mapping sélectionné parmi *idRider* correspondant à *rider?* fourni en entrée.

II.2.4. Représentation des méthodes setters/getters

En programmation orientée objet, les accesseurs connus sous le nom des méthodes setters / getters sont souvent utilisées. Ils permettent de modifier ou récupérer les attributs d'une classe. Dans notre prototype, les accesseurs sont générés automatiquement pour chaque classe selon les règles de traduction comme suit.

La méthode setter prend une nouvelle valeur comme paramètre d'entrée pour modifier l'attribut privé. La procédure est similaire à la formalisation des opérations expliquée précédemment.

setskillRider

RiderOp
skill?: Skill

R*setskillRider: Skill → Rider → Rider*

R*setskillRider*
= { *skill: Skill*
• *skill*
↦ { *setskillRider* | *skill? = skill*
• (*ridercoreclasstorider* θ *RiderCoreClass*
↦ *ridercoreclasstorider* θ *RiderCoreClass'*) } }

setskillRiderSystem

$\Delta SRider$
rider?: RIDER
skill?: Skill

$idRider' = idRider \oplus (\{rider?\} \lhd idRider \fatsemi \mathbb{R} setskillRider\ skill?)$

La méthode getter renvoie la valeur de l'attribut privé.

getskillRider

RiderGet
skill!: Skill

---

skill! = skill

II.2.5. Représentation de l'héritage

En modélisation orientée objet, l'héritage exprime le fait que la classe parente ou mère rassemble les attributs et les opérations communs à des classes qui deviennent des classes enfants. Les classes enfants possèdent les caractéristiques de leurs classes parentes avec la possibilité de les spécialiser. La relation d'héritage entre deux classes est traduite, dans une spécification formelle en Z, par l'inclusion du schéma de la classe mère dans la partie déclarative du schéma de la classe enfant.

L'exemple ci-dessous illustre la règle de transformation de la relation d'héritage entre la classe *Teacher* qui hérite de la classe *Rider*.

TeacherCoreClass

RiderCoreClass
qualification: Skill

$self \in TEACHER$

Dans toute relation d'héritage, l'ensemble d'identifiants des objets de la classe enfant représente un sous ensemble des identifiants des objets de la classe mère. Afin d'exprimer cette relation nous définissons le schéma suivant :

RiderHierarchy

$SRider$
$STeacher$

$teacherIds = riderIds \cap TEACHER$
$\forall\ t: teacherIds$
• (λ *TeacherCoreClass*
• θ *RiderCoreClass*) (*teachercoreclasstoteacher* ~ (*idTeacher t*))
= *ridercoreclasstorider* ~ (*idRider t*)

Dans la partie prédicative du schéma, le premier prédicat affirme que les identifiants des enseignants (*Teacher*) représentent un sous ensemble des identifiants des cavaliers (*Rider*). Ainsi, le deuxième prédicat montre que les états des enseignants, lorsqu'ils sont considérés comme cavaliers, correspondent aux états des cavaliers ayant les mêmes identifiants. La fonction lambda λ représente la fonction de projection de l'état *Teacher* à l'état *Rider*.

II.2.6. Représentation des classes associations

Dans notre exemple, les classes associations sont représentées par la classe *Lesson*. La formalisation des classes associations n'est pas fournie par Hall et elle a été introduite dans (Amálio and Polack, 2003) comme une extension au modèle de Hall. Les classes associations sont représentées sous forme d'un schéma qui contient un identifiant et des attributs comme une classe normale en plus d'une relation définissant les classes liées par cette classe association.

Lesson

self: LESSON
time: TIME
maxRiders: Integer
relLesson: RIDER↔HORSE

---

hasMult(*relLesson*, 1..*, 1..*)

La partie prédicative définit les multiplicités de cette association. *hasMult* est un opérateur qui limite le nombre de paires d'une relation liant la classe et la multiplicité des associations correspondantes (Amálio and Polack, 2003). L'opérateur *hasMult* a été proposé par Amalio et Polack (Amálio and Polack, 2003) dans le cadre d'une extension du modèle d'Anthony Hall.

$$[X, Y]$$

$$hasMult1\_ : \mathbb{P}\,((X \leftrightarrow Y) \times \mathbb{P}\,\mathbb{N})$$
$$hasMult\_ : \mathbb{P}\,((X \leftrightarrow Y) \times \mathbb{P}\,\mathbb{N} \times \mathbb{P}\,\mathbb{N})$$

$$(hasMult1\_) = \{\, r : X \leftrightarrow Y ;\ m : \mathbb{P}\,\mathbb{N} \mid \forall x : \mathrm{dom}\ r \bullet \#\,(r\ (\!|\ \{x\}\ |\!)) \in m \}$$
$$(hasMult\_) = \{\, r : X \leftrightarrow Y ;\ m1, m2 : \mathbb{P}\,\mathbb{N} \mid hasMult1(r, m1) \wedge (\forall\, y : \mathrm{ran}\ r \bullet \#(r^{\sim}\ (\!|\ \{\, y \,\}\ |\!)) \in m2)\}$$

Ensuite on procède de la même façon qu'une classe normale pour définir le schéma *SLesson*.

Afin d'avoir une vue générale sur l'ensemble des classes du système et les relations qui les réunissent, nous introduisons deux schémas.

Le schéma *ClassSystem* contient une relation *subSuper* qui définit les couples de classes liées par une relation d'héritage ainsi qu'une fonction *typeOf* qui renvoie le type de la classe à partir d'un identifiant.

ClassSystem

$$classes : \mathbb{P}\ CLASS$$
$$subSuper : CLASS \leftrightarrow CLASS$$
$$typeOf : \mathbb{P}\ OBJECT \rightarrowtail CLASS$$

$$subSuper^{+} \cap \mathrm{id}\ Class = \varnothing$$
$$\mathrm{dom}\ subSuper \subseteq classes$$
$$\mathrm{ran}\ subSuper \subseteq classes$$
$$subSuper = \{TeacherClass \mapsto RiderClass\}$$
$$typeOf = \{RIDER \mapsto RiderClass,\ TEACHER \mapsto TeacherClass,\ NAME \mapsto NameClass,\ SKILL \mapsto SkillClass,\ WEIGHT \mapsto WeightClass,\ TIME \mapsto TimeClass,\ HORSE \mapsto HorseClass,\ INTEGER \mapsto IntegerClass\}$$

Le schéma *System* contient tous les systèmes de classes définies auparavant ainsi que les systèmes d'héritage. Ce schéma nous permettra de définir les contraintes sur le système en sa totalité.

System

$\mathbb{S}$*Rider*
$\mathbb{S}$*Teacher*
$\mathbb{S}$*Name*
$\mathbb{S}$*Skill*
$\mathbb{S}$*Weight*
$\mathbb{S}$*Time*
$\mathbb{S}$*Horse*
$\mathbb{S}$*Integer*
$\mathbb{S}$*Lesson*
RiderHierarchy

Le but principal de notre approche est de vérifier la cohérence des diagrammes de classes à l'aide de spécifications formelles en Z. Nous avons proposé dans un premier lieu un modèle formel d'un sous ensemble des diagrammes de classes illustré à travers un exemple issu de la littérature (Hall, 1994). Ensuite, un ensemble de théorèmes sera proposé afin d'identifier et spécifier les différents cas d'incohérences.

II.3. Identification et spécification des cas d'inconsistances

L'intra-validation consiste à contrôler la conformité et la cohérence entre le modèle et son méta modèle correspondant (OMG UML 2.5.1, 2017). En se basant sur le modèle formel du diagramme de classes présenté précédemment, nous proposons dans la présente partie une liste non exhaustive des inconsistances que nous avons soulevées à travers la littérature ainsi que leurs formalisations en langage Z. Chaque cas d'inconsistance sera étudié à travers un exemple illustratif.

II.3.1. Intra-validation : Violation de la contrainte {disjoint}

Le premier cas d'incohérence qui sera étudié est la violation de la contrainte disjoint. Disjoint est utilisé dans une relation d'héritage pour indiquer qu'une instance de la classe mère ne peut être membre que d'un sous-type au plus. Autrement dit, un héritage multiple de classes disjointes est interdit. La contrainte disjoint est notée en UML entre deux accolades près de la flèche de l'héritage. Le diagramme de classes illustré dans la Figure II-2 présente un exemple de modèle inconsistant en raison de la violation de la contrainte disjoint (Kaneiwa and Satoh, 2006; Liu et

al., 2002). Cette contradiction réside dans le fait que la classe *class1* hérite à la fois de *class2* et *class3* tandis que ces deux dernières sont définies comme étant disjointes.

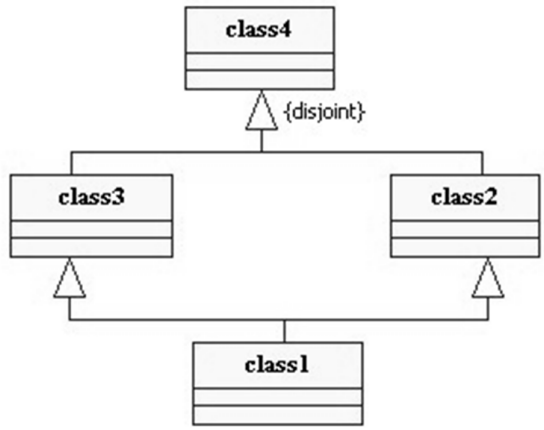


Figure II-2: Diagramme inconsistant - Violation de la contrainte {disjoint}

La contrainte disjoint est formalisée par l'inclusion d'un prédicat qui garantit la disjonction des classes mentionnées avec la contrainte {disjoint}. Ce prédicat précise que les identifiants des classes marquées par une contrainte {disjoint} ne peuvent pas avoir une instance en commun.

System
S*class1*
S*class3*
S*class2*
S*class4*
class3class1Hierarchy
class2class1Hierarchy
class4class2Hierarchy
class4class3Hierarchy

disjoint ⟨ *class2Ids , class3Ids*⟩

Le prédicat *disjoint* ⟨ *class2Ids, class3Ids*⟩ est équivalent à:

$$class2Ids \cap class3Ids = \emptyset. \quad (1)$$

L'inclusion des schémas *class2class1Hierarchy* et *class3class1Hierarchy* représentant les relations d'héritage dans le schéma *System* nous permet d'accéder aux contraintes suivantes :

$$(class1Ids = class2Ids \cap CLASS1) \wedge (class1Ids = class3Ids \cap CLASS1) \quad (2)$$

(1) et (2) impliquent que $class1Ids = \varnothing$ (3)

Si *class1* est instanciée alors $class1Ids \neq \varnothing$, d'où l'incohérence.

Puisque l'héritage multiple de deux classes disjointes est interdit, la classe *class1* ne doit avoir aucune instance. Une fois la classe *class1* instanciée, ce diagramme de classe présente une contradiction. Donc pour vérifier la cohérence de la spécification, un théorème nommé *disjointness* est généré lorsqu'une contrainte {disjoint} est signalée sur le modèle UML.

Theorem *disjointness*
$\exists Sclass1 \mid class1Ids \neq \varnothing \bullet System$

Si le théorème est prouvé, alors le système est incohérent.

II.3.2. Intra-validation : les contraintes de disjonction et de complétude

La contrainte {complete} signifie que chaque instance de la classe mère doit être un membre de l'un de ses sous-types. L'utilisation des contraintes {disjoint} et {complete} dans une relation d'héritage peut causer des problèmes d'incohérence si elles ne sont pas correctement utilisées. Notre approche propose de générer un théorème pour vérifier la cohérence dans ce cas. Voici un exemple illustratif (Kaneiwa and Satoh, 2006).

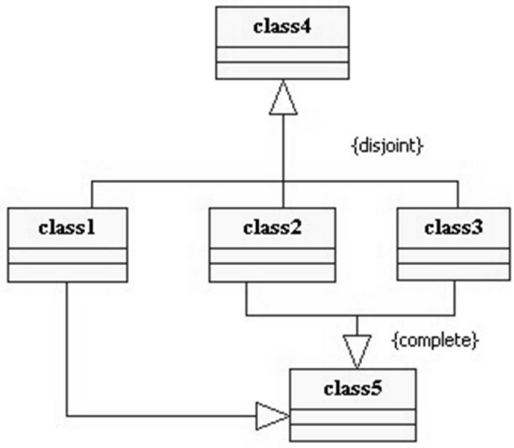


Figure II-3:Exemple de diagramme de classes inconsistant

L'utilisation de la contrainte {complete} dans la Figure II-3 assure que *class5* doit être spécialisée soit en tant que *class2* ou *class3*. La contrainte se traduit par le prédicat suivant dans le schéma *System*:

System

$\mathbb{S}class1$
$\mathbb{S}class2$
$\mathbb{S}class3$
$\mathbb{S}class4$
$\mathbb{S}class5$
class4class1Hierarchy
class4class2Hierarchy
class4class3Hierarchy
class5class1Hierarchy
class5class2Hierarchy
class5class3Hierarchy

---

disjoint ⟨*class1Ids*, *class2Ids*, *class3Ids*⟩
class5Ids = *class2Ids* ∪ *class3*Ids

Lors de l'utilisation des contraintes {disjoint} et {complete} simultanément, le théorème suivant est généré pour vérifier la cohérence du schéma System.

theorem *completeness*
$\exists$ *Sclass1* | *class1Ids* ≠ ∅ • *System*

Si un tel système existe, alors le système est inconsistant.

II.3.3. Intra-validation : L'héritage multiple et les multiplicités

Considérons le diagramme de classes UML suivant qui représente un exemple d'héritage multiple :

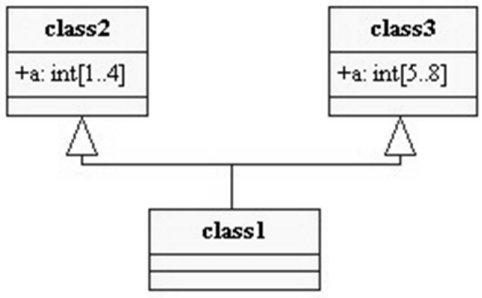


Figure II-4: Exemple de diagramme de classes inconsistant dans le cas de l'héritage multiple

La multiplicité d'un attribut spécifie le nombre de valeurs qui peut contenir. Dans la Figure II-4 , *class2* possède un attribut avec une multiplicité minimale de 1 et maximale de 4, *class3* possède un attribut avec le même nom mais des multiplicités différentes : le minimum est de 5 et

la multiplicité maximale est de 8. L'héritage multiple dans ce cas peut être une source d'incohérence du système. Le schéma suivant illustre la formalisation des multiplicités :

class2

self: CLASS2
a: $\mathbb{P}\mathbb{Z}$

$\#a \in 1..4$

class3

self: CLASS3
a: $\mathbb{P}\mathbb{Z}$

$\#a \in 5..8$

Selon le modèle formel que nous avons présenté, l'héritage multiple est représenté comme suit :

class1

class3
class2

self $\in$ *CLASS1*

Dans cet exemple, la contradiction est immédiatement détectée en Z parce que *class1* hérite deux attributs portant le même nom à partir de deux super-classes *class2* et *class3*. Le système Z/EVES (Meisels, 2004; Saaltink, 1997) découvre immédiatement une telle déclaration redondante. Il est important de mentionner que la norme UML reste ambigüe dans le cas de l'héritage multiple du même attribut.

II.4. Automatisation de la transformation du diagramme de classes vers une spécification Z

Nous avons discuté dans la section précédente (II.2), la formalisation des différentes constructions UML relatives aux diagrammes de classes en utilisant la notation Z. Dans cette section, nous présentons le prototype que nous avons développé pour la transformation automatique des diagrammes de classes vers le langage Z.

Le passage du diagramme de classes vers une spécification formelle en Z s'effectue suivant des règles de traduction que nous avons définies dans un document XSLT.

XSLT ou Extensible Stylesheet Language Transformations (W3C XSLT 3.0, 2017) est un langage destiné à transformer les informations d'un document XML vers un autre type de document comme un document HTML, PDF ou encore une spécification Z. C'est d'ailleurs ce dernier cas que nous avons abordé au cours de cette partie. Dans notre cas, le diagramme de classes est transformé en un document XML. Un fichier XSLT est associé à ce dernier afin de créer automatiquement une spécification Z. la Figure II-5 présente le principe de transformation avec XSLT.

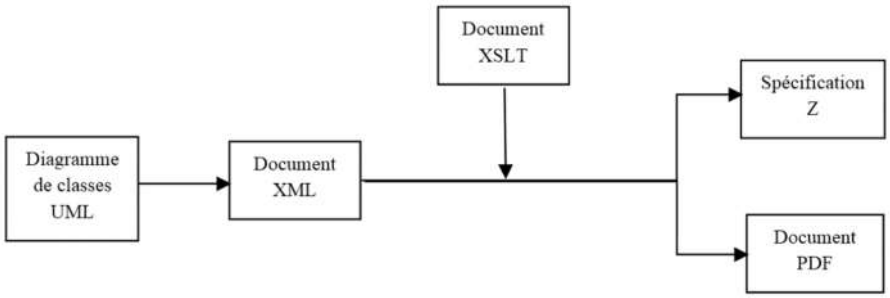


Figure II-5: Processus de transformation automatique d'un diagramme de classes vers Z

En général, une feuille de style XSLT débute par l'élément xsl:stylesheet, qui contient tous les modèles utilisés pour créer le résultat final. Elle déclenche ensuite le traitement de tous les autres modèles (via apply-templates). Il est possible de l'appeler grâce à son nom. Cette fonction accepte donc un attribut "name" qui permet d'identifier le modèle à appeler.

Cette fonction nous permet surtout de bien hiérarchiser nos documents XSLT en créant un modèle associé à un rôle bien précis. Dans notre cas, nous avons plusieurs modèles dont le rôle est de définir les classes, les opérations, l'initialisation, l'héritage et les associations. La relecture et la modification des documents XSLT sont ainsi facilitées.

Afin d'éviter d'alourdir le texte, nous avons joint en annexe les différents modèles utilisés.

Pour notre exemple, la feuille XSLT ci-dessous contient toutes les instructions qui permettent de transformer les données du document XML en une spécification Z. Cette feuille XSLT représente le fichier principal responsable de la transformation. Il fait appel aux autres fichiers XSLT que nous avons séparés par rôle comme précisé précédemment.

Dans un souci de simplification et pour faciliter la lecture de la feuille XSLT responsable de la génération automatique des spécifications Z, chaque étape de la transformation a été annotée à

l'aide d'un commentaire. Un commentaire en XSLT se présente sous forme d'une balise qui commence par **<!--** et finit par **-->.**

```
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://czt.sourceforge.net/zml/zml/Z_2_1.xsd"
    xmlns:UML="href://org.omg/UML/1.3"
    version="2.0">

<xsl:import href="HierarchyTemplates.xsl"/>
<xsl:import href="InitialisationTemplates.xsl"/>
<xsl:import href="ClassTemplates.xsl"/>
<xsl:import href="OperationTemplates.xsl"/>
<xsl:import href="AssociationTemplates.xsl"/>

<xsl:output method='text'  encoding='UTF-8'  />
<xsl:strip-space elements="*"/>
<xsl:template match="*">
 <xsl:copy>
   <xsl:apply-templates/>
 </xsl:copy>
</xsl:template>
<xsl:variable name="NEWLINE" select="'&#xA;'" />
<xsl:variable name="quote">'</xsl:variable>
<xsl:template match="UML:Namespace.ownedElement/UML:Model">
 <xsl:if test="@name='Design Model'">
    <xsl:result-document
            href="AnthonyHallHierarchy.tex"
            method="text"
            indent="yes">
        <xsl:value-of select="concat(
                '\documentclass{article}',$NEWLINE,
                '\usepackage{z-eves}',$NEWLINE,
                '\begin{document}',$NEWLINE,
                 $NEWLINE)"/>

<!-- Nous déclarons les givenSet class et object -->

         <xsl:call-template name="GivenSet">
        </xsl:call-template>

<!-- Nous déclarons les ensembles d'objets(les noms des classes en majuscules) -->

        <xsl:call-template name="ObjectNames">
        </xsl:call-template>
        <xsl:call-template name="ClassNames">
        </xsl:call-template>
```

```
<xsl:for-each select="UML:Namespace.ownedElement/UML:Class">

 <!--Pour chaque classe nous créons le schéma initial par exemple: VisitorCoreClass-->
<!--  Ensuite on ajoute un type énuméré qui introduit la valeur nulle correspondante à chaque classe -->
<!-- Nous commençons par les classes qui n'ont pas d'attributs -->

      <xsl:if test="not(exists(UML:Classifier.feature/UML:Attribute))">
        <xsl:call-template name="UMLClass">
          <xsl:with-param name="ClassName" as="xs:string" select="@name"/>
        </xsl:call-template>

 <!--Pour chaque classe nous créons le schéma system qui représente l'ensemble des instances de la classe-->

        <xsl:call-template name="classSystem">
          <xsl:with-param name="ClassName" as="xs:string" select="@name"/>
        </xsl:call-template>
      </xsl:if>
</xsl:for-each>


<xsl:for-each select="UML:Namespace.ownedElement/UML:Class">

<!-- Ensuite les classes qui contiennent des attributs -->

 <xsl:if test="exists(UML:Classifier.feature/UML:Attribute)">
        <xsl:call-template name="UMLClass">
          <xsl:with-param name="ClassName" as="xs:string" select="@name"/>
        </xsl:call-template>

 <!--Ensuite le schéma system qui représente l'ensemble des instances de la classe-->

        <xsl:call-template name="classSystem">
          <xsl:with-param name="ClassName" as="xs:string" select="@name"/>
        </xsl:call-template>
      </xsl:if>
</xsl:for-each>

<!-- Même traitement pour les classes associations -->

<xsl:for-each select="UML:Namespace.ownedElement/UML:AssociationClass">
 <xsl:if test="exists(UML:Classifier.feature/UML:Attribute)">
        <xsl:call-template name="UMLClass">
          <xsl:with-param name="ClassName" as="xs:string" select="@name"/>
        </xsl:call-template>

 <!--Pour chaque classe association nous créons le schéma system qui représente l'ensemble des instances de la classe-->
```

```
        <xsl:call-template name="classSystem">
          <xsl:with-param name="ClassName" as="xs:string" select="@name"/>
        </xsl:call-template>
      </xsl:if>
</xsl:for-each>

<!--Ensuite pour chaque classe, Nous ajoutons deux schémas d'initialisation  -->

<xsl:for-each select="UML:Namespace.ownedElement/UML:Class">
 <xsl:if test="exists(UML:Classifier.feature/UML:Attribute)">
   <xsl:variable name="ClassName" select="@name"/>

<!-- La première initialisation avec des valeurs nulles -->

     <xsl:call-template name="InitClassByDefault">
       <xsl:with-param name="ClassName" as="xs:string" select="$ClassName"/>
     </xsl:call-template>

<!-- La deuxième initialisation avec des valeurs en entrée -->

     <xsl:call-template name="InitClassWithValues">
       <xsl:with-param name="ClassName" as="xs:string" select="$ClassName"/>
     </xsl:call-template>
 </xsl:if>
</xsl:for-each>

<!-- Pour chaque classe nous créons un schéma qui nous permet d'ajouter un nouvel objet au system -->
<xsl:for-each select="UML:Namespace.ownedElement/UML:Class">
 <xsl:variable name="ClassName" select="@name"/>
     <xsl:call-template name="AddToSystem">
       <xsl:with-param name="ClassName" as="xs:string" select="$ClassName"/>
     </xsl:call-template>
</xsl:for-each>
<xsl:for-each select="UML:Namespace.ownedElement/UML:Class">
        <xsl:call-template name="ClassOp">
          <xsl:with-param name="ClassName" as="xs:string" select="@name"/>
        </xsl:call-template>

        <xsl:call-template name="ClassGet">
          <xsl:with-param name="ClassName" as="xs:string" select="@name"/>
        </xsl:call-template>
</xsl:for-each>
<xsl:for-each select="UML:Namespace.ownedElement/UML:Class">
  <xsl:variable name="ClassName" select="@name"/>
   <xsl:for-each select="UML:Classifier.feature/UML:Attribute">
    <xsl:variable name="attributeName" select="@name"/>
      <xsl:call-template name="setOperation">
        <xsl:with-param name="ClassName" as="xs:string" select="$ClassName"/>
        <xsl:with-param name="attributeName" as="xs:string" select="$attributeName"/>
```

```
    </xsl:call-template>
    <xsl:call-template name="setFunction">
        <xsl:with-param name="ClassName" as="xs:string" select="$ClassName"/>
        <xsl:with-param name="attributeName" as="xs:string" select="$attributeName"/>
    </xsl:call-template>
    <xsl:call-template name="setOperationSystem">
        <xsl:with-param name="ClassName" as="xs:string" select="$ClassName"/>
        <xsl:with-param name="attributeName" as="xs:string" select="$attributeName"/>
    </xsl:call-template>
    <xsl:call-template name="getOperation">
        <xsl:with-param name="ClassName" as="xs:string" select="$ClassName"/>
        <xsl:with-param name="attributeName" as="xs:string" select="$attributeName"/>
    </xsl:call-template>
  </xsl:for-each>
  <xsl:call-template name="Methods">
    <xsl:with-param name="ClassName" as="xs:string" select="$ClassName"/>
  </xsl:call-template>
</xsl:for-each>
 <xsl:for-each select="UML:Namespace.ownedElement/UML:Generalization">
      <xsl:call-template name="Hierarchy">
      </xsl:call-template>
 </xsl:for-each>
<xsl:for-each select="UML:Namespace.ownedElement/UML:AssociationClass">
  <xsl:variable name="classAssociationName" select="@name"/>

    <xsl:for-each select="UML:Association.connection/UML:AssociationEnd">
      <xsl:variable name="associationEndType" select="@type"/>
      <xsl:variable name="associationEndName"
select="//UML:Class[@xmi.id=$associationEndType]/@name"/>
     <xsl:call-template name="Association">
          <xsl:with-param name="classAssociationName" as="xs:string"
select="$classAssociationName"/>
           <xsl:with-param name="associationEndName" as="xs:string"
select="$associationEndName"/>
      </xsl:call-template>
 </xsl:for-each>
</xsl:for-each>

<!-- Nous créons un schéma appelé System contenant toutes les classes system -->

       <xsl:call-template name="System">
       </xsl:call-template>
<xsl:if test="exists(//UML:Generalization)">
         <xsl:call-template name="HierarchySystem"/>
 </xsl:if>
 <xsl:value-of select="concat($NEWLINE,'\end{document}',$NEWLINE)"/>
    </xsl:result-document>
 </xsl:if>
</xsl:template>
```

```
<xsl:template name="ObjectNames">
<xsl:value-of select="concat('\begin{axdef}',$NEWLINE)"/>
 <xsl:for-each select="UML:Namespace.ownedElement/UML:Class">
      <xsl:value-of select="upper-case(@name)"/>
       <xsl:if test="not(position()=last())">
        <xsl:text> , </xsl:text>
       </xsl:if>
 </xsl:for-each>
 <xsl:for-each select="UML:Namespace.ownedElement/UML:AssociationClass">
    <xsl:value-of select="concat(',',upper-case(@name))"/>
 </xsl:for-each>
  <xsl:for-each select="UML:Namespace.ownedElement/UML:AssociationClass" >
      <xsl:variable name="classAssociationName" select="@name"/>
       <xsl:for-each select="UML:Association.connection/UML:AssociationEnd">
      <xsl:variable name="associationEndType" select="@type"/>
      <xsl:variable name="associationEndName"
select="//UML:Class[@xmi.id=$associationEndType]/@name"/>
        <xsl:value-of select="concat(',',upper-case($classAssociationName),upper-
case($associationEndName))"/>
    </xsl:for-each>
 </xsl:for-each>
<xsl:value-of select="concat(': \power OBJECT',$NEWLINE,'\end{axdef}',$NEWLINE)"/>
</xsl:template>
<xsl:template name="GivenSet">
   <xsl:value-of
select="concat('\begin{zed}',$NEWLINE,'[CLASS,OBJECT]','\end{zed}',$NEWLINE)"/>
 <xsl:text>  \begin{axdef}
  Class: \power  CLASS
\end{axdef}
</xsl:text>
</xsl:template>
</xsl:stylesheet>
```

II.5.Conclusion

Ce chapitre a été consacré à la présentation du modèle de transformation des constructions UML vers le langage Z. Le passage du diagramme UML vers une spécification formelle en Z s'effectue suivant des règles de traduction que nous avons présentées à l'aide d'exemples pour faciliter la compréhension.

Au niveau des diagrammes de classes, nous avons formalisé les principales notions liées à la conception orientée objet tel que l'héritage et l'encapsulation en proposant les méthodes setters et getters. Contrairement à l'outil RoZ proposé par (Dupuy et al., 2000), le processus de formalisation dans notre méthode n'a pas besoin d'annotations pour compléter la conception

UML alors que l'outil RoZ nécessite des connaissances en Z à l'étape de conception UML. La séparation entre UML et Z permet de travailler sur des conceptions UML fournies par des ingénieurs de logiciels sans connaissances en Z. En outre, notre approche permet non seulement la génération de spécifications mais aussi la détection des incohérences structurelles, ce qui n'est pas abordée par l'outil RoZ. Ce chapitre illustre également les incohérences UML les plus fréquentes en utilisant le modèle d'Anthony Hall (Hall, 1990, 1994). La plupart de ces inconsistances sont traduites en des prédicats contradictoires ou sous forme de théorèmes. Les spécifications formelles ainsi obtenues sont utilisées pour prouver la cohérence de l'ensemble du système.

Nous avons également développé un prototype basé sur la technologie XSLT afin de traduire automatiquement les conceptions UML en une spécification Z. Ce prototype est structuré en plusieurs feuilles XSLT organisées par rôle.

Chapitre III : SPECIFICATION FORMELLE DE LA VUE DYNAMIQUE DES SYSTEMES ORIENTES OBJET

III.1. Spécification formelle du diagramme de séquences

III.1.1. Introduction

Le diagramme de séquences est l'un des diagrammes UML les plus utilisés pour spécifier les interactions entre les différentes parties du système. En effet, il est important de noter que les diagrammes de séquences modélisent principalement les échanges entre des objets et non des classes.

Le diagramme de séquences montre l'ordre des échanges de messages et le passage du temps. Les principaux concepts sont les objets participants à la séquence, le temps, les messages, et la création et la suppression de participants. Les diagrammes de séquences capturent l'ordre des interactions entre les différentes parties du système. Les diagrammes de séquences sont caractérisés par leur simplicité et leur efficacité pour montrer l'aspect temporel.

Chaque participant possède une ligne de vie représentée par une ligne verticale en pointillée. Une flèche reçue par un participant modélise la réception d'un message et se traduit par l'exécution d'une opération.

Considérons l'étude de cas d'un système de vidéo à la demande (VOD) proposée par (Lopez-Herrejon and Egyed, 2010). Le diagramme de classes illustré dans la Figure III-1 se compose de trois classes : *Service*, *Streamer* et *Program*. Ces classes possèdent plusieurs méthodes. Les classes sont liées à l'aide de deux associations navigables. Une première association allant de *Service* à *Streamer*, et une deuxième de *Streamer* à *Program*.

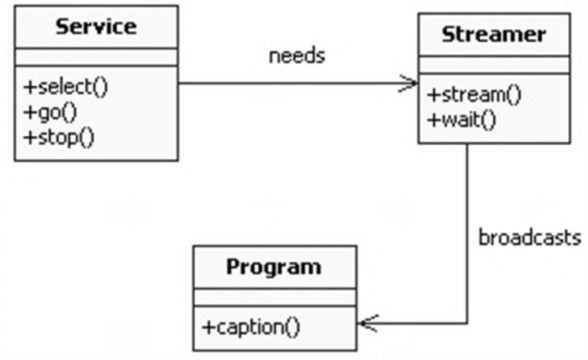


Figure III-1: Diagramme de classes d'un système de vidéo à la demande

La Figure III-2 illustre un diagramme de séquences très simple modélisant un scénario particulier des interactions entre deux objets de type *Service* et *Streamer*. Les messages échangés correspondent aux méthodes *select* et *stream* définies respectivement par les classes *Service* et *Streamer*.

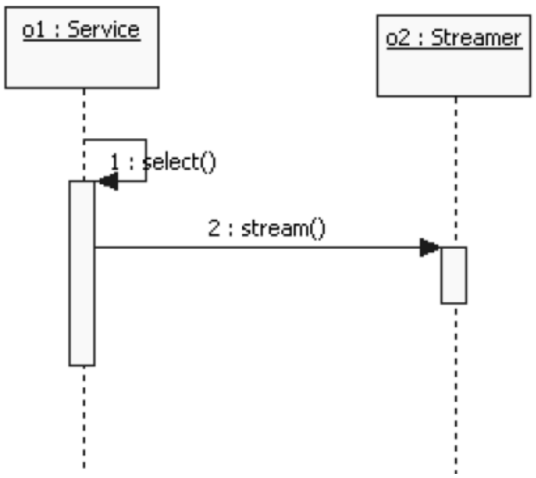


Figure III-2: Exemple d'un diagramme de séquences

Nous présentons dans le reste de cette partie notre contribution dans la formalisation du diagramme de séquences dans le but d'étudier sa consistance. Notre approche traite les différentes règles de cohérence mentionnées dans la littérature en particulier celles correspondantes à la cohérence entre le diagramme de classes et le diagramme de séquences.

III.1.2. Le modèle formel

Les objets *o1* et *o2* sont respectivement les instances des classes *Service* et *Streamer*. Nous déclarons *o1* et *o2* comme *OBJECT* et plus tard, nous préciserons leur appartenance à leurs classes respectives.

$$o1, o2 : OBJECT$$

Pour représenter toutes les opérations, nous introduisons un ensemble énuméré *OP* contenant toutes les méthodes. Les méthodes doivent être auparavant définies dans le modèle formel du diagramme de classes comme expliqué précédemment. Les paramètres des méthodes sont définis comme entrées dans le schéma de chaque opération.

OP ::= *select* ⟨⟨*selectService*⟩⟩

| *go* ⟨⟨*goService*⟩⟩

| *stop* ⟨⟨*stopService*⟩⟩

| *stream* ⟨⟨*streamStreamer*⟩⟩

| *wait* ⟨⟨*waitStreamer*⟩⟩

| *caption* ⟨⟨*captionProgram*⟩⟩

Les messages échangés entre les objets *o1* et *o2* sont des appels aux méthodes définies précédemment. Les appels de méthodes sont représentés par une instance de chaque méthode notée en majuscule.

SELECT : *selectService*
GO : *goService*
STOP : *stopService*
STREAM : *streamStreamer*
WAIT : *waitStreamer*
CAPTION : *captionProgram*

A présent, la sémantique du diagramme de séquences peut être définie de manière complète. Un diagramme de séquences est constitué d'un ensemble d'objets qui interagissent les uns avec les autres au moyen de messages. Il est nécessaire de préciser l'ordre dans lequel les messages sont agencés. Donc pour modéliser le diagramme de séquences en Z, nous avons choisi de le représenter en tant que séquence de messages. Une séquence en Z décrit une collection ordonnée d'objets. Chaque message est défini par un triplet représentant l'émetteur du message, le destinataire du message et le message qui correspond à l'opération.

SequenceDiagram

Objects : ℙ *OBJECT*
Messages : seq (*OBJECT* × *OBJECT* × *OP*)

Objects = {*o*1, *o*2}
∀ *System*
• ∃ *serviceId* : ℙ *serviceIds* ;
streamerId : ℙ *streamerIds*
• *o*1 ∈ *serviceId* ∧ *o*2 ∈ *streamerId*
Messages = ⟨ (*o*1,*o*1, *select SELECT*), (*o*1,*o*2, *stream STREAM*)⟩

Le premier prédicat définit l'ensemble des objets qui participent dans le diagramme de séquences. Ces objets doivent être inclus dans l'ensemble des instances de leurs classes respectives. Cette contrainte s'exprime à travers le deuxième prédicat. Enfin, la séquence de messages est explicitement exprimée par le troisième prédicat.

III.1.3. Inter-validation

Étant donné que le comportement d'un objet est décrit par un diagramme de classes et ses interactions avec les autres objets sont spécifiées avec différents diagrammes de séquence, la consistance multi-vues doit être vérifiée.

La sémantique formelle du diagramme de séquences définie ci-dessus permet de vérifier si un diagramme de séquences est cohérent avec un diagramme de classes.

La description de la syntaxe des diagrammes UML fournie par le standard UML (OMG UML 2.5.1, 2017) se base essentiellement sur des règles de bonne formation exprimées en langage naturel. Pour pallier aux ambiguïtés dues au langage naturel, nous proposons de formaliser en Z ces règles afin de les rendre plus précises (El Miloudi et al., 2013).

Selon la classification proposée par (Lopez-Herrejon and Egyed, 2010), ces règles de cohérence sont définies parmi les règles inter-vues qui décrivent les relations sémantiques entre les composantes des différentes vues.

- **Règle 1** : **Le nom du message doit correspondre à une méthode de la classe.**

Chaque message sur le diagramme de séquences doit correspondre à une méthode définie dans le diagramme de classes. L'ensemble énuméré *OP* contient toutes les méthodes qui sont préalablement définies dans le diagramme de classes. Par conséquent, chaque message correspond nécessairement à une de ces méthodes ; sinon, cette règle est automatiquement détectée en utilisant le système Z/ EVES (Meisels, 2004; Saaltink, 1997).

- **Règle 2** : **Chaque objet doit avoir une classe correspondante dans le diagramme de classes.**

Cette règle est énoncée comme prédicat dans le schéma du diagramme de séquences. Le prédicat suivant spécifie que chaque objet dans le diagramme de séquences doit appartenir à l'ensemble des instances de sa classe définies dans le diagramme de classes. Cette règle est vérifiée explicitement par le modèle formel que nous proposons.

$\forall$ *System*
• $\exists$ *serviceId* : $\mathbb{P}$ *serviceIds* ;
streamerId : $\mathbb{P}$ *streamerIds*
• *o*1 $\in$ *serviceId* $\wedge$ *o*2 $\in$ *streamerId*

Règle 3 : **Un message du diagramme de séquences doit correspondre à une opération de l'objet récepteur.**

Pour illustrer cette règle de cohérence en Z, nous proposons un théorème qui vérifie que chaque message apparaissant dans un diagramme de séquences appartient à l'ensemble des opérations de la classe de l'objet récepteur :

theorem *methods*
$\forall$ *SequenceDiagram*
• $\exists$ *Sender, Receiver: Objects; Operation: OP*
• $\langle$(*Sender*, *Receiver*, *Operation*)$\rangle$ in *Messages*
$\Rightarrow$ *Operation* $\in$ *methodsOfObject Receiver*

Dans la formalisation de la règle de cohérence, une fonction supplémentaire nommée *methodsOfObject* a été utilisée dans la définition du théorème. Cette fonction renvoie l'ensemble des opérations de la classe correspondante à chaque objet.

methodsOfObject: OBJECT $\rightarrow$ $\mathbb{P}$ *OP*

methodsOfObject o1 = { *select SELECT*, *go GO*, *stop STOP* }
*methodsOfObject o*2 = { *stream STREAM*, *wait WAIT* }

Nous définissons d'abord l'ensemble des opérations de la classe de chaque objet à travers la fonction *methodsOfObject.* Ensuite la vérification est effectuée en prouvant le théorème vrai.

III.1.4. Conclusion

Cette partie présente une sémantique formelle du diagramme de séquences permettant la vérification de la cohérence entre la vue statique d'un système exprimée par le diagramme de classes et la vue dynamique exprimée par le diagramme de séquences. Notre approche traite les différentes règles soulevées par (Liu, 2013) en particulier celles correspondantes à la cohérence entre le diagramme de classes et le diagramme de séquences. Notre approche est plus

approfondie par rapport à celle proposée par Dubauskaite et Vasilecas (Dubauskaite and Vasilecas, 2013) grâce à l'utilisation du langage Z. Nous avons choisi Z pour exprimer les règles de cohérences car il présente une sémantique précise basée sur des notations mathématiques permettant ainsi l'élimination des ambiguïtés contrairement à Dubauskaite et Vasilecas qui ont choisi d'utiliser UML pour exprimer des règles de cohérence entre les différentes vues d'un système alors que UML en soi dispose d'une sémantique ambigüe.

Notre approche est entièrement vérifiée à l'aide du système Z/ EVES fournissant l'une des premières spécifications formelles en Z pour le diagramme de séquences.

III.2. Spécification formelle du diagramme d'états

III.2.1. Introduction

Le diagramme d'états est l'un des plus importants diagrammes UML pour spécifier le comportement dynamique des systèmes. Le diagramme d'états permet de modéliser le comportement individuel d'un objet d'une classe. Il montre les différents états d'un objet de la classe ainsi que les évènements provoquant les transitions entre ces états suivant un ensemble de règles de bonne formation. Une transition est définie par un évènement, une condition et une action. Chaque transition est déclenchée par un évènement. Ce dernier est souvent une invocation d'une méthode de la classe. La transition n'est effectuée que si la condition est valide. Cela permettra par la suite l'exécution de l'action. Les deux extrémités d'un diagramme d'états sont déterminées par les pseudo-états initial et final.

Sur la base de nos travaux antérieurs sur la formalisation des diagrammes de classes et de séquences, nous proposons une méthode pour transformer un sous-ensemble des diagrammes d'états en une spécification Z dans le but de vérifier formellement la cohérence dans le cadre d'une modélisation multi-vues. Cette spécification sémantique capture, de façon précise, à la fois la structure et les caractéristiques dynamiques du système modélisé. La consistance de la spécification résultante est garantie en fournissant un ensemble de règles de bonne formation et de cohérence. Il est à noter que notre approche multi-vues est le premier travail sur la formalisation des diagrammes d'états basé sur la notation Z (El Miloudi and Ettouhami, 2015).

Reprenons l'exemple du système de vidéos à la demande illustré par le diagramme de classes dans la Figure III-1 et le diagramme de séquences dans la Figure III-2. La Figure III-3 montre un exemple de diagramme d'états d'un objet de la classe Service. Comme le montre cette figure, les

états possibles d'un objet de la classe *Service* sont *Init* et *Streaming*. Le changement d'états est déclenché par les évènements : *select*, *go* et *stop*.

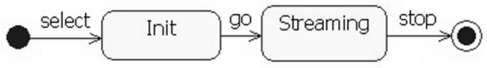


Figure III-3: Diagramme d'états de l'objet Service

Nous proposons dans ce qui suit une sémantique formelle du diagramme d'états en Z à travers l'exemple introduit par la Figure III-3.

III.2.2. Modèle formel

Un diagramme d'états est essentiellement composé d'un ensemble d'états reliés avec des transitions selon un ensemble de règles de bonne formation définies par le standard UML.

Nous commençons par la représentation de la notion d'états. L'ensemble des états est introduit sous forme d'un type basique nommé *STATE*. La spécification formelle du diagramme d'états inclut la déclaration :

[STATE]

Les deux pseudo-états initial et final sont représentés par deux variables de type *STATE*. L'état initial représente le point de départ du diagramme d'états. L'état final montre la fin de l'exécution du diagramme d'états.

$$INITIALSTATE, FINALSTATE : STATE$$

La fonction *statesOfObject* retourne, pour chaque objet, l'ensemble de ses états.

$$statesOfObjects : OBJECT \longrightarrow \mathbb{P}\ STATE$$

Les transitions sont considérées parmi les notions basiques des diagrammes d'états. Les transitions sont des relations entre les états. Une transition désigne le fait qu'un objet changera son état actuel en un autre. Le premier état est appelé état source tandis que le deuxième est appelé état cible. Une transition peut avoir plusieurs sources représentant une jointure à partir de plusieurs états, ainsi que plusieurs cibles dans le cas d'une transformation à plusieurs états. Une

action spécifique est exécutée lorsqu'un événement se produit et que la condition est évaluée. Si la condition est vraie, la transition peut être activée ; sinon, elle est désactivée. Formellement, une transition est spécifiée comme suit :

TRANSITION

event: EVENT
effect: EFFECT
guard: BOOL
source: $\mathbb{P}$ *STATE*
target: $\mathbb{P}$ *STATE*
kind: KIND

$kind = fork \Rightarrow \#source = 1 \land \#target > 1$
$kind = join \Rightarrow \#source > 1 \land \#target = 1$

Selon le standard UML actuel, un évènement peut être sous forme d'un appel ou d'un signal. Afin de formaliser la notion d'évènement, nous déclarons un type énuméré dans lequel chaque élément est soit de type *SignalEvent* ou le retour de la fonction *OperationAsCallEvent* appliquée à un élément de type *OP*. *OP* a été introduit précédemment comme étant un ensemble énuméré représentant toutes les opérations. Un événement d'appel représente la réception de l'appel d'une opération par un objet. Les évènements de type signal ne sont pas détaillés, nous les représentons sous forme de constante.

EVENT ::= *OperationAsCallEvent* 《*OP*》 | *SignalEvent*

Une transition peut avoir une condition de garde. Cette dernière est définie sous forme d'un type booléen, d'où la nécessité d'introduire le type énuméré *BOOL*.

BOOL ::= *True* | *False*

Lorsqu'une transition se déclenche, son effet s'exécute. L'effet d'une transition est un comportement optionnel donc nous le formalisons sous forme d'un type énuméré introduisant une constante nommée *Action* et une constante nommée *NullEffect* dans le cas où la transition n'a pas d'effet.

EFFECT ::= *Action* | *NullEffect*

La fonction *isTargetOf*, est une fonction totale qui prend en entrée un état et une transition et retourne un état où chaque état est lié à un seul état utilisant une transition spécifique. Cette fonction sera utilisée ultérieurement dans la définition du diagramme d'états.

$$isTargetOf: STATE \times TRANSITION \rightarrow STATE$$

Après avoir défini les notions basiques, la sémantique du diagramme d'états peut être entièrement formalisée en utilisant les définitions précédentes. Un schéma nommé *StatechartDiagram* a été défini. Dans la partie déclarative du schéma, deux variables ont été introduites : une variable *Obj* de type *OBJECT* est l'objet dont le comportement est spécifié par le diagramme d'états. La seconde variable appelée *statechart* spécifie les composants du diagramme d'états.

Un diagramme d'états est défini par l'ensemble des états et l'ensemble des transitions les reliant. Nous distinguons entre les états sources et cibles des transitions. Nous introduisons la variable *statechart* en tant qu'un ensemble de produit cartésien composé de tous les triplets de la forme ($STATE \times TRANSITION \times STATE$). Chaque triplet correspond respectivement à l'état source de la transition, la transition et l'état cible.

StatechartDiagram

$$Obj: OBJECT$$
$$statechart: \mathbb{P}\,(STATE \times TRANSITION \times STATE)$$

$$\forall\ Source, Target: STATE;\ Transition: TRANSITION$$
$$\bullet\ \{(Source, Transition, Target)\} \subseteq statechart$$
$$\wedge\ Target = isTargetOf\,(Source, Transition)$$
$$\wedge\ Source \in Transition.source$$
$$\wedge\ Target \in Transition.target$$
$$\wedge\ Source \in statesOfObject\ Obj \setminus \{FINALSTATE\}$$
$$\wedge\ Target \in statesOfObject\ Obj \setminus \{INITIALSTATE\}$$
$$\wedge\ (Transition.event \neq SignalEvent$$
$$\Rightarrow OperationAsCallEvent\ ^\sim\ Transition.event \in methodsOfObject\ Obj$$
$$\wedge\ (\exists\ seqDiagr: SequenceDiagram;\ o: OBJECT$$
$$\bullet\ \langle (o, Obj, OperationAsCallEvent\ ^\sim\ Transition.event)\rangle$$
$$\text{in } seqDiagr.Messages))$$
$$\wedge\ (Source = INITIALSTATE \Rightarrow \#\,(outgoings\ Source) = 1 \wedge \#\,(incomings\ Source) = 0)$$
$$\wedge\ (Target = FINALSTATE \Rightarrow \#\,(outgoings\ Target) = 0)$$

La partie prédicative indique que les états sources et cibles des transitions doivent appartenir à l'ensemble des états de l'objet dont le comportement est décrit par le diagramme d'états. Ce prédicat indique également qu'un état initial ne peut jamais être la cible d'une transition. De même, un état final ne peut être une source de transition. En appliquant cette formalisation à l'exemple du système VOD de la Figure III-3, le diagramme d'états de l'objet *service* sera décrit par l'ensemble suivant :

Statechart = {(*INITIALSTATE*, *select*, *Init*),
(*Init*, *go*, s*treaming*),
(*Streaming*, *stop*, *FINALSTATE*)}

Dans ce cas, le prédicat indiquant que la source n'est jamais un état final et que la cible n'est jamais un état initial est vérifié.

Un schéma d'initialisation est fourni pour définir la valeur initiale du diagramme d'états. Un diagramme d'états est initialement un ensemble vide.

$$StateDiagramInit \mathrel{\widehat=} [StatechartDiagram' \mid statechart' = \varnothing]$$

Pour vérifier que les composants du diagramme d'états sont cohérents, il suffit d'établir qu'il existe un diagramme d'états initial et par conséquent, qu'au moins un diagramme d'états remplit les conditions définies dans la partie prédicative du schéma *statechartDiagram*.

theorem *InitIsOk*
$\exists StatechartDiagram' \bullet StateDiagramInit$

L'opération de changement d'état nécessite deux entrées : l'état actuel de l'objet et la transition choisie. Nous les modélisons comme deux entrées *currentState?* et *transition?,* de types respectifs *STATE* et *TRANSITION*. L'opération de changement d'état est décrite comme suit :

ChangeState

Δ*StatechartDiagram*
currentState?: STATE
transition?: TRANSITION

---

if *transition?.guard = True*
then *statechart'*
= *statechart*
$\cup$ {(*currentState?*, *transition?*,
isTargetOf (*currentState?*, *transition?*))}
else *statechart' = statechart*

L'effet de cette opération est défini seulement si la condition de garde de la transition est satisfaite.

Une fois le diagramme d'états est traduit en une spécification Z, la consistance multi vues peut être étudiée en utilisant les outils offerts par Z. La partie prédicative du schéma *StatechartDiagram* sera discutée en détails dans la section suivante à travers une série de règles relatives à la bonne forme.

III.2.3. Approche de validation

Afin de garantir l'exactitude d'un diagramme d'états et sa cohérence avec les diagrammes de classes et de séquences dans la modélisation multi-vues, un ensemble de règles de bonne formation et de cohérence doit être satisfait. Nous fournissons à travers le modèle proposé la formalisation de ces règles de bonne formation en utilisant la notation Z. Nous avons utilisé les règles de bonne formation publiées pour montrer l'efficacité de notre modèle.

a) Intra-validation

Deux fonctions supplémentaires renvoyant respectivement l'ensemble des transitions partant d'un état spécifique et entrant dans celui-ci sont utilisées dans la définition formelle des règles de cohérence.

$$
\begin{array}{|l}
incomings: STATE \rightarrow \mathbb{P}\ TRANSITION \\
outgoings: STATE \rightarrow \mathbb{P}\ TRANSITION \\
\hline
\forall\ s: STATE \bullet\ incomings\ s = \{\ t: TRANSITION \mid\ t.target = s\ \} \\
\forall\ s: STATE \bullet\ outgoings\ s = \{\ t: TRANSITION \mid\ t.source = s\ \}
\end{array}
$$

- **Règle 1** : **Un état final ne peut avoir aucune transition sortante.**

Cette règle est représentée sous forme de prédicat dans le schéma *statechartDiagram* par l'expression Z suivante à l'aide de la fonction *outgoings* définie ci-dessus.

$$Target = FINALSTATE \Rightarrow \#(outgoings\ Target) = 0$$

- **Règle 2** : **Un état initial peut avoir au plus une transition sortante et aucune transition entrante.**

La formalisation de cette règle est similaire à la règle 1.

$$Source = INITIALSTATE \Rightarrow \#(outgoings\ Source) = 1 \wedge \#(incomings\ Source) = 0$$

b) Inter-validation : Règles de consistance entre le diagramme d'états et le diagramme de classes

Un diagramme d'états peut montrer les différents états d'une instance d'une classe et indiquer également comment une instance d'une classe passe d'un état à un autre à l'aide de transitions. Les objets et les états dans les diagrammes de machine à états sont liés aux diagrammes de classes. Par conséquent, certaines règles de cohérence entre les diagrammes de classes et les diagrammes d'états doivent être satisfaites pour assurer une cohérence multi-vues.

- **Règle 3: Un objet décrit par le diagramme d'états doit correspondre à une instance d'une classe dans les diagrammes de classes** (Liu, 2013).

Pour exprimer cette règle de cohérence, un théorème Z est fourni. Le théorème suivant indique que l'objet dont le comportement est spécifié par le diagramme d'états doit appartenir à l'ensemble des objets d'une classe existante définie dans le diagramme de classes.

theorem *consistencyStateAndClassDiagram*
$\forall$ *s: StatechartDiagram*
- $\exists$ *class: CLASS*
  - *s.Obj* $\in$ *ObjectsOfClass class*

La fonction *ObjectsOfClass* retourne, pour chaque classe, l'ensemble de ses instances. *CLASS* et *OBJECT* sont définis dans la formalisation des diagrammes de classes.

$$ObjectsOfClass: CLASS \rightarrow \mathbb{P}\ OBJECT$$

- **Règle 4: Si l'événement lié à une transition dans un diagramme d'états doit appeler une opération d'une classe, cette opération doit être définie comme une opération de la classe du propriétaire** (Lopez-Herrejon and Egyed, 2010).

La relation inverse de la fonction *OperationAsCallEvent* est utilisée pour atteindre l'opération utilisée dans la définition d'événement. Cette opération doit appartenir à l'ensemble des opérations correspondantes à l'objet *Obj*.

$$Transition.event \neq SignalEvent$$
$$\Rightarrow OperationAsCallEvent^{\sim} Transition.event \in methodsOfObject\ Obj$$

La fonction methodsOfObject, définie précédemment, retourne l'ensemble des opérations correspondantes à chaque objet.

c) Inter-validation : Règles de consistance entre le diagramme d'états et le diagramme de séquences

Le diagramme de séquences est l'un des diagrammes comportementaux qui illustrent l'interaction entre des objets. Certains composants du diagramme de séquences peuvent être décrits par plusieurs diagrammes notamment le diagramme d'états. Par conséquent, la cohérence

du système doit être vérifiée. Dans ce cadre, nous définissons la sémantique du diagramme d'états dans un contexte multi vues pour garantir la cohérence.

Règle 5 : Dans un diagramme d'états, si un événement doit appeler une opération, l'opération doit figurer sous forme de message dans un diagramme de séquences.

Le diagramme de séquences est défini précédemment sous forme de séquence de messages. Chaque message est défini par un triplet représentant l'expéditeur du message, le destinataire du message et l'opération invoquée.

Par conséquent, afin de garantir la cohérence entre le diagramme d'états et le diagramme de séquences, l'opération appelée par un événement d'appel doit apparaître dans un message d'un diagramme de séquence existant.

$$Transition.event \neq SignalEvent$$
$$\Rightarrow (\exists\ seqDiagr: SequenceDiagram;\ o: OBJECT$$
$$\bullet\ \langle (o, Obj, OperationAsCallEvent \sim Transition.event) \rangle$$
$$\text{in } seqDiagr.Messages))$$

Comme indiqué ci-dessus, ce prédicat est inclus dans la partie prédicative du schéma *statechartDiagram* afin de garantir la cohérence multi-vues.

En vérifiant ces propriétés pour les diagrammes de classes et de séquences du système de vidéo à la demande introduits respectivement dans la Figure III-1 et la Figure III-2, les règles de bonne formation sont satisfaites. Les opérations appelées par le diagramme d'états de la Figure III-3 sont définies par la classe *Service* et utilisées en tant que messages dans le diagramme de séquences.

L'ensemble des opérations de l'objet *o1* est le suivant :

$$methodsOfObject\ o1 = \{select, go, stop\}$$

Le diagramme de séquences de la Figure III-2 illustre un appel de la méthode *select* d'un objet *o1* de type *service*. L'opération *select* est également appelée par la transition qui fait passer l'objet de type *Service* de l'état initial à l'état *Init*. Par conséquent, la règle de bonne formation 5 est satisfaite. Les opérations *go* et *stop* appelées dans le diagramme d'états de la Figure III-3 ne

sont pas utilisées par le diagramme de séquences illustré dans la Figure III-2. Dans ce cas, pour garantir la cohérence multi-vue, l'existence d'un autre diagramme de séquences appelant ces opérations doit être assurée. Souvent, dans ce cas, l'examen du modèle aide à comprendre le problème et donc à proposer une correction.

III.2.4. Conclusion

L'objectif de cette partie est de surmonter les principales limitations de la sémantique des diagrammes d'états UML dans la modélisation multi-vues. Une sémantique formelle des diagrammes d'états a été fournie en fonction du modèle formel des diagrammes de classes et de séquences précédemment proposés. Les principaux avantages de notre approche sont, d'une part, la concision et la clarté du modèle formel qui permet de définir une sémantique précise et non ambiguë du diagramme d'états, et d'autre part, l'analyse de la spécification résultante et l'étude de la consistance. Contrairement à (Fernandes and Song, 2014; Ledang and Souquières, 2001b; Mostafa et al., 2007) où seule la modélisation des concepts UML est considérée, nous proposons une formalisation des différentes règles de bonne formation et de consistance pour assurer la cohérence dans un contexte multi vues. Ainsi, notre approche représente l'une des premières spécifications formelles Z du diagramme d'états dans un contexte à vues multiples. Toutes les spécifications présentées ont été minutieusement vérifiées à l'aide du système Z/EVES (Meisels, 2004).

III.3. Spécification formelle du diagramme de cas d'utilisation

III.3.1. Introduction

Les diagrammes de cas d'utilisation UML sont couramment utilisés durant les premières phases de la définition des exigences logicielles et ont été incorporés dans la plupart des méthodes de développement orientées objet grâce à ses conceptions simplistes promouvant de bonnes pratiques en matière d'ingénierie logicielle. Cette partie décrit la première approche de modélisation formelle des diagrammes de cas d'utilisation dans un contexte multi-vues. L'approche est divisée en deux étapes. Dans la première étape, un modèle formel de diagramme de cas d'utilisation UML est proposé à l'aide de la notation Z. Une vérification de la cohérence multi-vues est ensuite présentée. Cette approche garantit la cohérence logicielle, améliorant ainsi la qualité des exigences (El Miloudi and Ettouhami, 2018).

III.3.2. Modèle formel

Les diagrammes de cas d'utilisation sont utilisés pour l'analyse des exigences de haut niveau d'un système afin de capturer sa vue dynamique. Ils sont essentiellement utilisés pour rassembler les exigences et les fonctionnalités d'un système sous forme de cas d'utilisation, ainsi que pour identifier les agents internes et externes interagissant avec le système. Ces agents sont connus sous le nom d'acteurs. La Figure III-4 montre un exemple de diagramme de cas d'utilisation du système ATM (OMG UML 2.5.1, 2017).

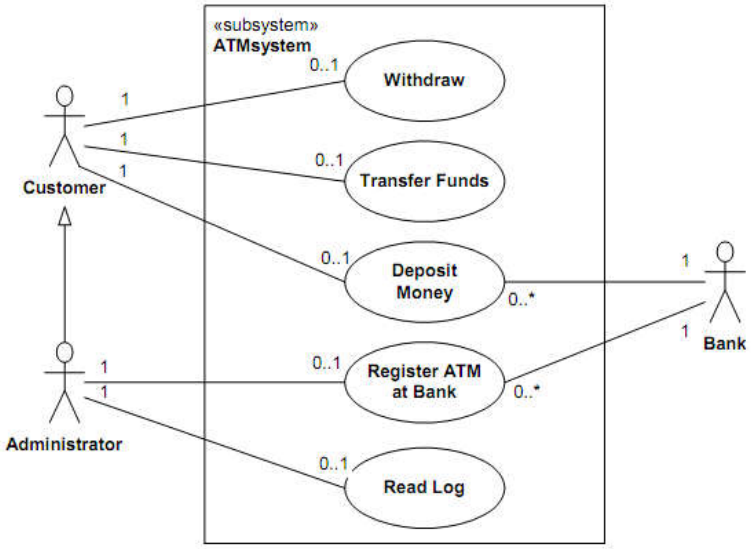


Figure III-4: Exemple de diagramme de cas d'utilisation du système ATM

Pour résumer, les diagrammes de cas d'utilisation sont constitués d'acteurs, de cas d'utilisation et de leurs relations. La Figure III-5 illustre la syntaxe abstraite des diagrammes de cas d'utilisation (OMG UML 2.5.1, 2017).

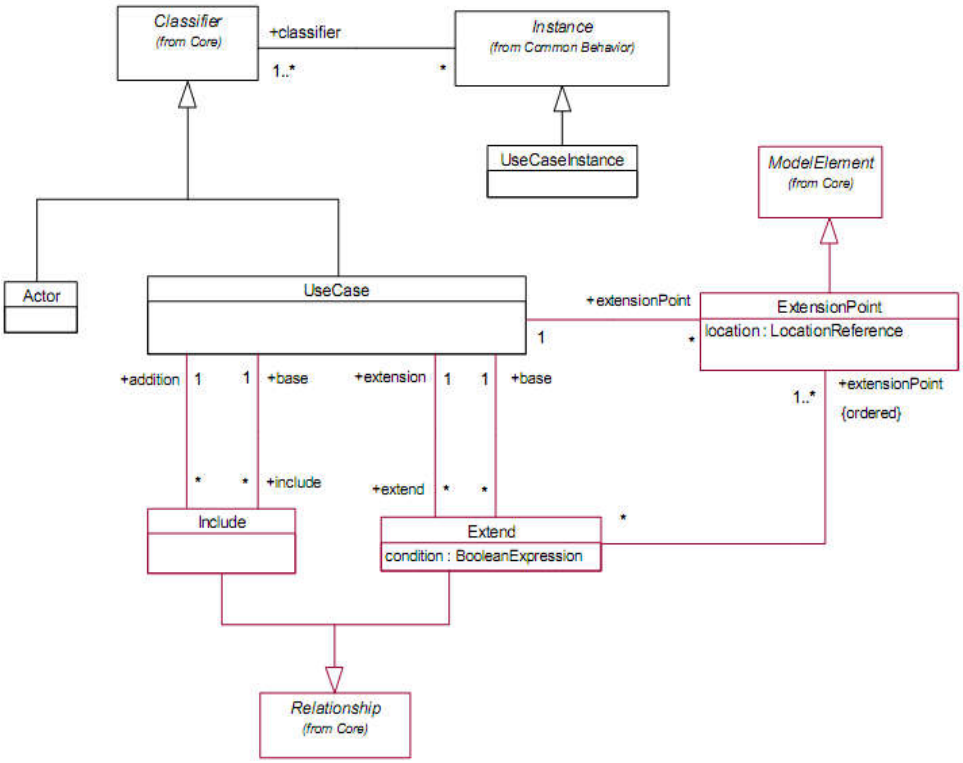


Figure III-5: Syntaxe abstraite du package use cases

La première étape de notre approche consiste à formaliser la notion de cas d'utilisation et les différentes relations utilisées. Pour mieux comprendre le modèle formel proposé, nous définissons d'abord les différents termes impliqués dans la spécification, tels que *included use cases*, *extending use cases* et *extension point*. Nous allons utiliser la notation Z pour spécifier et décrire les différentes notions du diagramme de cas d'utilisation

III.3.2.1. Formalisation des cas d'utilisation

Selon le standard UML, un cas d'utilisation est défini en tant que classificateur. En effet, l'ensemble de tous les identificateurs uniques figurant dans la spécification est désigné par l'ensemble donné *CLASSIFIER*.

Par conséquent, la variable *UCASE* est introduite en tant que classificateur pour spécifier l'ensemble des identifiants des cas d'utilisation.

[CLASSIFIER]

$UCASE : \mathbb{P}\ CLASSIFIER$

Il existe trois types de relations entre les cas d'utilisation : Il s'agit de la relation d'inclusion, de la relation d'extension et de la généralisation, qui sont toutes décrites dans cette section.

Include est une relation entre deux cas d'utilisation utilisée pour montrer «que le comportement du cas d'utilisation inclus est inséré dans le comportement du cas d'utilisation incluant»(OMG UML 2.5.1, 2017). *Extend* est une relation «qui spécifie comment et quand le comportement défini dans le cas d'utilisation d'extension généralement facultatif peut être inséré dans le comportement défini dans le cas d'utilisation étendu» (OMG UML 2.5.1, 2017) .

Les relations *include* et *extend* sont essentiellement représentées par une relation Z qui relie deux cas d'utilisation. La relation *include* est utilisée pour enregistrer le cas d'utilisation directement inclus par le cas d'utilisation d'origine, tandis que la relation *extend* sert à enregistrer le cas d'utilisation qui étend directement le cas d'utilisation d'origine.

Les fonctions *includedUsecases* et *extendingUsecases* sont respectivement obtenues avec la fermeture transitive des relations *include* et *extend*. Elles sont utilisées pour spécifier formellement la fonction qui renvoie l'ensemble des cas d'utilisation liés directement ou indirectement avec les relations *include* ou *extend*.

$$
\begin{array}{|l}
include: UCASE \rightarrow UCASE \\
extend: UCASE \rightarrow UCASE \\
includedUsecases: UCASE \rightarrow \mathbb{P}\ UCASE \\
extendingUsecases: UCASE \rightarrow \mathbb{P}\ UCASE \\
\hline
\forall\ u1: UCASE \bullet\ includedUsecases\ u1 = \{\ u2: UCASE \mid (u1, u2) \in include^{+}\ \} \\
\forall\ u1: UCASE \bullet\ extendingUsecases\ u1 = \{\ u2: UCASE \mid (u2, u1) \in extend^{+}\ \} \\
\end{array}
$$

L'extension peut se produire à un point spécifique du cas d'utilisation étendu appelé point d'extension. Un point d'extension est associé à une contrainte indiquant à quel moment l'extension a lieu. Selon le standard UML, un point d'extension doit avoir un nom. Formellement, un point d'extension est spécifié comme suit :

ExtensionPoint

$$
\begin{array}{|l}
name: NAME \\
constraint: \mathbb{P}\ BOOL \\
\hline
\#\ constraint \leqslant 1 \\
\end{array}
$$

Si aucune contrainte n'est associée, l'extension est inconditionnelle. Dans ce cas, la variable d'état *constraint* est un ensemble vide et sa cardinalité est donc égale à 0.

La relation de généralisation de cas d'utilisation est une relation entre un cas d'utilisation enfant et un cas d'utilisation parent, spécifiant comment un enfant peut spécialiser tous les comportements et caractéristiques décrits pour le parent. Une généralisation de cas d'utilisation est traduite en Z en ajoutant, dans le schéma *Usecase*, la variable d'état *parents* définie comme un ensemble de cas d'utilisation.

Une fois que la terminologie relative aux cas d'utilisation a été clarifiée au moyen d'une spécification écrite en Z, la notion de cas d'utilisation peut ensuite être complètement formalisée.

Par conséquent, la notion de cas d'utilisation est clairement représentée dans le schéma suivant :

Usecase

$$
\begin{array}{l}
self: UCASE \\
name: NAME \\
parents: \mathbb{P}\ UCASE \\
Fragments: \text{seq}\ FRAGMENT \\
includedUC: \mathbb{P}\ UCASE \\
extendingUC: \mathbb{P}\ UCASE \\
extendedUC: \mathbb{P}\ UCASE \\
isExtended: BOOL \\
isExtending: BOOL \\
extensionPoints: \text{seq}\ ExtensionPoint \\
\hline
includedUC = includedUsecases\ self \\
self \notin includedUC \cup extendingUC \\
\textbf{if}\ isExtended = True \\
\quad \textbf{then}\ extendingUC = extendingUsecases\ self \\
\textbf{else}\ extendingUC = \varnothing \ \wedge \ extensionPoints = \langle\rangle
\end{array}
$$

La moitié supérieure du schéma *Usecase* définit les différentes variables d'un cas d'utilisation avec leurs types respectifs. Chaque cas d'utilisation est identifié par son identifiant appelé *self* et son nom. Comme expliqué précédemment, la variable *parents* est utilisée pour définir l'ensemble des cas d'utilisation spécialisés par le cas d'utilisation actuel.

Si un cas d'utilisation est étendu, il est alors composé d'une séquence de fragments et d'une liste ordonnée de points d'extension. Un cas d'utilisation étendu consiste en une ou plusieurs descriptions de fragments de comportement à insérer aux emplacements appropriés du cas

d'utilisation étendu. Les points d'extension spécifient où les fragments de comportement respectifs du cas d'utilisation étendu doivent être insérés (OMG UML 2.5.1, 2017). Si la condition est vraie lorsque le premier point d'extension est atteint, le cas d'utilisation d'extension sera exécuté ; sinon, l'extension ne se produit pas. Si aucune contrainte n'est associée à la relation d'extension, l'extension est inconditionnelle.

Comme indiqué dans le premier prédicat de la seconde moitié du schéma *Usecase*, un cas d'utilisation ne peut pas inclure des cas d'utilisation qui l'incluent directement ou indirectement.

Après avoir défini la notion de cas d'utilisation et la terminologie associée, nous pouvons passer à l'étape suivante, celle de la spécification des acteurs et de leurs relations.

III.3.2.2. Formalisation des acteurs

Un acteur spécifie un rôle joué par un utilisateur ou tout autre système qui interagit avec le sujet (OMG UML 2.5.1 2017). Selon le standard UML, un acteur est défini en tant que classificateur. Comme cité précédemment, l'ensemble donné *CLASSIFIER* est utilisé pour définir l'ensemble de tous les identifiants uniques tout au long de la spécification. La variable *ACTOR* est introduite en tant que classificateur pour spécifier l'ensemble des identifiants des acteurs.

$$ACTOR : \mathbb{P}\ CLASSIFIER$$

L'acteur peut être un utilisateur humain, une application interne ou une application externe. Le type d'acteur est présenté comme un ensemble énuméré représentant deux types d'acteurs.

$$ActorTYPE ::= HUMAN \mid nonHUMAN$$

Pour les acteurs, il existe principalement deux types de relations : 1) une relation entre acteur et cas d'utilisation, 2) une relation entre acteurs qui n'est autre que la généralisation.

La relation entre acteur et cas d'utilisation est essentiellement représentée par une relation Z reliant un acteur à un cas d'utilisation. La relation *associationAU* est utilisée pour enregistrer le cas d'utilisation pouvant être utilisé par l'acteur, tandis que la fonction relatedUsecases renvoie l'ensemble des cas d'utilisation associés à l'acteur.

$$associationAU: ACTOR \rightarrow Usecase$$
$$relatedUsecases: ACTOR \rightarrow \mathbb{P}\ Usecase$$
$$\forall\ a: ACTOR \bullet relatedUsecases\ a = \{\ uc: Usecase \mid (a, uc) \in associationAU\ \}$$

La généralisation des acteurs est utilisée pour factoriser et réutiliser les similitudes entre les acteurs. Cette relation montre qu'un acteur hérite du rôle et des propriétés d'un autre acteur. Pour représenter la généralisation, nous avons choisi d'ajouter une variable d'état à la partie déclarative du schéma *Actor*, comme indiqué ci-dessous. Cette variable d'état est appelée *parents* et est spécifiée en tant qu'ensemble d'acteurs. La relation de généralisation implique également que l'acteur descendant puisse utiliser tous les cas d'utilisation définis pour son ancêtre. Ceci est clairement indiqué dans la partie prédicative.

Actor

$$
\begin{array}{|l}
\hline
self: ACTOR \\
actorName: NAME \\
actorType: ActorTYPE \\
relatedUC: \mathbb{P}\ Usecase \\
parents: \mathbb{P}\ ACTOR \\
inheritedUC: \mathbb{P}\ Usecase \\
\hline
relatedUC = relatedUsecases\ self \cup inheritedUC \\
\hline
\end{array}
$$

La fonction *getActorFromACTOR* prend un identifiant d'acteur et renvoie l'acteur correspondant. Cette fonction sera utilisée ultérieurement dans la formalisation du diagramme de cas d'utilisation.

$$
\begin{array}{|l}
\hline
getActorFromACTOR: ACTOR \rightarrow Actor \\
\hline
\forall\ a: ACTOR; ac: Actor \mid ac.self = a \bullet getActorFromACTOR\ a = ac \\
\hline
\end{array}
$$

De même, nous définissons la fonction *getUsecaseFromUCASE*.

$$
\begin{array}{|l}
\hline
getUsecaseFromUCASE: UCASE \rightarrow Usecase \\
\hline
\forall\ UC: UCASE; uc: Usecase \mid uc.self = UC \bullet getUsecaseFromUCASE\ UC = uc \\
\hline
\end{array}
$$

III.3.2.3. Formalisation du diagramme de cas d'utilisation

En se basant sur la formalisation des cas d'utilisation, des acteurs et de leurs relations, le concept de diagramme de cas d'utilisation peut être entièrement défini. Comme le montre le

schéma suivant, un diagramme de cas d'utilisation a un nom et est composé d'un ensemble d'acteurs et d'un ensemble de cas d'utilisation.

Nous spécifions également le scénario d'exécution des cas d'utilisation en cas d'extension. Nous proposons la fonction *UCexecution* qui prend une séquence de fragments de cas d'utilisation et de points d'extension et renvoie un type booléen. Cette fonction indique si l'extension sera exécutée ou non.

useCaseDiagram

$$
\begin{array}{l}
name: NAME \\
actors: \mathbb{P}\ Actor \\
usecases: \mathbb{P}\ Usecase \\
UCexecution: \text{seq}\ (FRAGMENT \times ExtensionPoint \times FRAGMENT) \rightarrow BOOL \\
\hline
\forall\ a: Actor;\ ap: \mathbb{P}\ ACTOR \mid\ a \in actors \land ap = a.parents \\
\quad \bullet\ a.inheritedUC = \cup \{x: ap \bullet (\ getActorFromACTOR\ x\ ).relatedUC\ \ \} \\
\forall\ a1, a2: Actor \bullet\ a1 \in actors \land a2 \in actors \Leftrightarrow a1.self \neq a2.self \\
\forall\ u1, u2: Usecase \bullet\ u1 \in usecases \land u2 \in usecases \Leftrightarrow u1.self \neq u2.self \\
\forall\ u1, u2: Usecase \mid\ u1 \in usecases \land u2 \in usecases \\
\quad \bullet\ u1.self \in u2.includedUC \Rightarrow u2.self \notin u1.includedUC \\
\forall\ u1, u2: Usecase;\ f1, f2: FRAGMENT;\ e: ExtensionPoint \\
\quad \mid\ u1 \in usecases \\
\quad\ \land u2 \in usecases \\
\quad\ \land getUsecaseFromUCASE^{\sim}\ u2 \in u1.extendingUC \\
\quad\ \land \langle f1 \rangle \subseteq u1.Fragments \\
\quad\ \land \langle f2 \rangle \subseteq u2.Fragments \\
\quad\ \land \langle e \rangle \subseteq u1.extensionPoints \\
\quad \bullet\ \textbf{if}\ e.constraint = \{True\} \lor e.constraint = \varnothing \\
\qquad \textbf{then}\ UCexecution\ \langle (f1, e, f2) \rangle = True \\
\quad\ \textbf{else}\ UCexecution\ \langle (f1, e, f2) \rangle = False
\end{array}
$$

Le premier prédicat indique que, pour chaque acteur, les cas d'utilisation hérités ne sont rien d'autre que les cas d'utilisation liés à ses parents. Il est également important de noter que les identifiants des cas d'utilisation et des acteurs sont uniques. Le quatrième prédicat indique qu'un cas d'utilisation ne peut pas inclure des cas d'utilisation qui l'incluent directement ou indirectement. Le dernier prédicat signifie qu'une extension ne se produit que si la contrainte est vraie ou si l'extension est inconditionnelle.

Une fois le diagramme de cas d'utilisation est formalisé, la cohérence multi vues peut être vérifiée.

Dans la section suivante, nous discutons de la cohérence entre le diagramme de cas d'utilisation, le diagramme de classe et le diagramme de séquence.

III.3.3. Approche de validation

Après avoir présenté comment construire un modèle formel du diagramme de cas d'utilisation à l'aide de la notation Z, nous présentons les moyens de vérifier la cohérence dans un contexte multi-vues.

Une fois que le modèle formel des diagrammes de cas d'utilisation a été défini, il est utile de proposer et de prouver des théorèmes entre les différentes vues. Cela permet de vérifier le système et de rechercher les erreurs. Dans cette partie, nous nous concentrons sur la cohérence de trois différents modèles d'un système, comprenant un diagramme de classes, un diagramme de séquence et un diagramme de cas d'utilisation.

III.3.3.1. Cohérence entre les diagrammes de cas d'utilisation et de séquence

Pour formaliser la notion de cohérence entre le diagramme de cas d'utilisation et le diagramme de séquence, considérons un théorème Z appelé *consistencyUsecaseAndSequenceDiagram*. Ce théorème vérifie si:

- Chaque nom de cas d'utilisation correspond à une trame de diagramme de séquence.
- Les acteurs du diagramme de cas d'utilisation sont définis comme des objets dans le diagramme de séquence.

theorem *consistencyUsecaseAndSequenceDiagram*
$\forall$ *ud: useCaseDiagram; uc: Usecase; a: Actor*
| $uc \in ud.usecases \land a \in ud.actors \land uc \in a.relatedUC$
• $\exists$ *sd: SequenceDiagram* • $sd.Frame = uc.name \land a.self \in sd.Objects$

Pour tout diagramme de cas d'utilisation *ud*, un cas d'utilisation *uc* et un acteur *a* tel que *uc* et *a* appartiennent respectivement aux cas d'utilisation et acteurs de *ud ; uc* appartient à l'ensemble des cas d'utilisation liés à a, il existe alors un diagramme de séquence *sd* dont le nom de la trame correspond au nom de *uc* et que l'identifiant de *a* se trouve dans l'ensemble des objets du diagramme *sd*.

Si le théorème est prouvé, alors chaque contrainte exprimée précédemment pourra être satisfaite et, par conséquent, cela garantira que le système reste toujours dans un état cohérent.

III.3.3.2. Cohérence entre les diagrammes de cas d'utilisation et les diagrammes de classes

Définissons maintenant la notion de cohérence entre le diagramme de cas d'utilisation et le diagramme de classes.

Premièrement, étant donné une instance de diagramme de classes et un diagramme de cas d'utilisation, nous définissons un théorème Z appelé *consistencyUsecaseAndClassDiagram*. Le théorème suivant affirme que les deux vues sont cohérentes, si nous considérons que :

- Les acteurs correspondent à des objets de classes définies dans un diagramme de classes.
- Les cas d'utilisation dans le diagramme de cas d'utilisation correspondent aux opérations définies dans les classes dans le diagramme de classes.

Nous définissons la fonction *op* qui renvoie l'opération correspondant à un cas d'utilisation donné.

$$op: Usecase \rightarrow OP$$

theorem *consistencyUsecaseAndClassDiagram*

$$\forall\ ud: useCaseDiagram;\ a: Actor;\ uc: Usecase \mid a \in ud.actors \land uc \in ud.usecases \bullet \exists\ class: CLASS;\ operation: OP \bullet a.self \in ObjectsOfClass\ class \land (uc, operation) \in op$$

Pour tout diagramme de cas d'utilisation *ud*, un cas d'utilisation *uc* et un acteur *a* tel que *uc* et *a* appartiennent respectivement aux cas d'utilisation et acteurs du diagramme *ud*, il existe alors une classe nommée *class* tel que l'identifiant de *a* appartient a l'ensemble des objets de la classe class et *uc* correspond à l'une de ses opérations.

Le théorème proposé assure que la spécification satisfait les contraintes présentées précédemment. Si les contraintes sont satisfaites, la spécification est cohérente. Le théorème devrait être prouvé comme étant vrai.

III.3.4. Conclusion

« La spécification des exigences est une activité clé pour la réalisation des objectifs de tout projet logiciel. Elle est établie et reconnue depuis longtemps par les chercheurs et les praticiens»(De Oliveira et al., 2014). Dans cette partie, nous avons proposé une spécification formelle des principaux concepts du diagramme de cas d'utilisation. Les objectifs de notre travail sont de fournir une spécification concise et non ambiguë du diagramme de cas d'utilisation en utilisant la notation Z et la vérification de la cohérence au moyen de théorèmes Z. La spécification formelle des diagrammes de cas d'utilisation a été étudiée dans le passé en utilisant différents langages, mais nous estimons que l'expressivité et la base mathématique de la notation Z fournissent un cadre plus riche pour un tel formalisme.

Comparée aux formalisations du diagrammes des cas d'utilisation en Z proposées dans la littérature (Butler et al., 1997; Mostafa et al., 2007; Muhamad et al., 2019; Singh et al., 2016), notre approche met l'accent sur la formalisation et la clarification des notions de base du cas d'utilisation et de ses concepts associés ainsi que sur la spécification des différents composants et contraintes imposées aux cas d'utilisation. La formalisation des acteurs et la relation d'héritage entre eux et les points d'extension sont également ajoutées. Dans notre modèle, chaque notion est formalisée tout en considérant l'ensemble du diagramme ainsi que les différentes vues du système. Le modèle a été amélioré en incluant la notion de scénario en cas d'extension.

Toutes les spécifications présentées ont été vérifiées syntaxiquement et leur véracité a été vérifiée à l'aide du système Z/EVES (Meisels, 2004), un outil d'analyse des spécifications Z prenant en charge la vérification de types et la démonstration de théorèmes.

CONCLUSION GENERALE

Aujourd'hui, la vie quotidienne des gens dépend fortement de logiciels qui deviennent de plus en plus grands et complexes. À mesure que la partie logicielle dans les systèmes actuels grandit, les documents relatifs aux exigences deviennent énormes et l'ambiguïté est inévitablement introduite. De nombreux projets de développement logiciel échouent à cause de problèmes de configuration. En conséquence, il est devenu très important de pouvoir améliorer la qualité des logiciels en accordant une plus grande attention à la modélisation des exigences. Cependant, d'une part, l'absence de formalité dans la définition des diagrammes UML laisse toute la place à l'introduction des erreurs et des incohérences. D'autre part, les modèles UML ne sont pas directement analysables et les types ne peuvent pas être vérifiés automatiquement.

L'enjeu de nos recherches, à travers cette thèse de doctorat, était l'étude des diagrammes UML les plus fréquemment utilisés notamment le diagramme de classes, le diagramme de séquences, le diagramme d'états et le diagramme des cas d'utilisation dans la mesure où une telle étude permettait de vérifier et valider les modèles UML. Au cours de ce travail, nous avons proposé un modèle formel des diagrammes UML en utilisant la notation Z pour bénéficier de la rigueur issue des méthodes formelles. Grâce à ce formalisme, nous surmontons les limitations des diagrammes UML en permettant d'utiliser des vérificateurs de type et des démonstrateurs de théorèmes tels que le système Z / EVES. Il convient de noter que nous avons proposé la première formalisation des diagrammes UML en notation Z qui prend en compte la cohérence entre les différentes vues.

Au niveau des diagrammes de classes, nous avons formalisé les différents composants tels que les classes, les opérations, les associations, les classes associations ainsi que les principales notions liées à la conception orientée objet tel que l'héritage et l'encapsulation. Nous avons également étudié les incohérences UML causées principalement par le manque de documentation précise fournie par le standard UML. La plupart de ces inconsistances ont été traduites en des prédicats contradictoires ou sous forme de théorèmes. Les spécifications formelles ainsi obtenues sont utilisées pour prouver la cohérence de l'ensemble du système.

Nous avons également développé un prototype basé sur la technologie XSLT afin de traduire automatiquement le diagramme de classes en une spécification Z. Le passage du diagramme UML vers une spécification formelle en Z s'effectue suivant des règles de traduction que nous avons présentées à l'aide d'exemples pour faciliter la compréhension.

Au niveau du diagramme de séquences, nous avons proposé une sémantique formelle permettant la vérification de la cohérence entre la vue statique d'un système exprimée par le diagramme de classes et la vue dynamique exprimée par le diagramme de séquences. Notre

approche traite les différentes règles de consistance soulevées par la littérature à l'aide d'un ensemble de prédicats et de théorèmes que nous avons proposés.

Dans le même contexte de la modélisation à vues multiples, une sémantique formelle des diagrammes d'états a été fournie en fonction du modèle formel des diagrammes de classes et de séquences précédemment proposés. Nous avons également présenté l'analyse de la spécification résultante et l'étude de la consistance. Nous avons proposé une formalisation des différentes règles de bonne formation et de consistance pour assurer la cohérence dans un contexte multi vues.

En ce qui concerne le diagramme des cas d'utilisation, nous avons proposé une spécification formelle concise et précise des principaux concepts de ce diagramme. Nous avons mis l'accent sur la formalisation et la clarification des notions de base du cas d'utilisation et de ses concepts associés ainsi que sur la spécification des différents composants et contraintes imposées aux cas d'utilisation. La formalisation des acteurs, des relations d'héritage entre eux et des points d'extension ont également été traités. Nous avons également vérifié la consistance du diagramme des cas d'utilisation ainsi que sa cohérence avec différents diagrammes UML au moyen de théorèmes Z.

Le but de ce travail est de tirer parti de l'expressivité et de la base mathématique des méthodes formelles pour améliorer la qualité des exigences.

Les perspectives des travaux présentés dans ce mémoire suivent différents axes de réflexion et se situent dans les différents contextes de recherche. Nous donnons dans ce paragraphe quelques perspectives qui nous semblent intéressantes :

- Un domaine important pour des recherches ultérieures consiste à trouver des moyens de gérer des vues plus larges afin de détecter davantage d'incohérences. Nous nous sommes intéressés dans notre étude aux diagrammes UML les plus fréquemment utilisés. L'étude des autres diagrammes UML ne pourrait qu'enrichir l'approche.
- Nous prévoyons également d'étendre notre prototype pour générer automatiquement une spécification Z à partir des différents diagrammes UML présentés.
- Exploiter notre travail pour la mise en œuvre d'une approche sur des exemples d'applications industrielles.
- Un autre axe d'amélioration est de générer automatiquement du code à partir d'un modèle UML en passant par le modèle de spécification formelle en Z que nous avons proposé à travers cette thèse.

- Enfin, sûrs de la valeur pédagogique et positive de l'utilisation des méthodes formelles dans la phase de conception des logiciels, nos travaux pourraient donner lieu à des supports d'enseignement du langage UML en présentant les constructions et les cas d'inconsistances possibles.

BIBLIOGRAPHIE

Abrial, J. R.: The B-Book: Assigning Programs to Meanings, Cambridge University Press, Cambridge., 2005.

Amálio, N. and Polack, F.: Comparison of Formalisation Approaches of UML Class Constructs in Z and Object-Z, in ZB 2003: Formal Specification and Development in Z and B, edited by D. Bert, J. P. Bowen, S. King, and M. Waldén, pp. 339–358, Springer Berlin Heidelberg, Turku, Finland., 2003.

Amálio, N., Stepney, S. and Polack, F.: Formal Proof from UML Models, in Formal Methods and Software Engineering, edited by J. Davies, W. Schulte, and M. Barnett, pp. 418–433, Springer Berlin Heidelberg, Seattle, WA, USA., 2004.

Amálio, N., Polack, F. and Stepney, S.: UML+Z: Augmenting UML with Z, in Software Specification Methods, pp. 81–102, John Wiley & Sons, Ltd., 2010.

Barajas, F. V.: A formal model for a requirements engineering tool, in 1st. Alloy Workshop, pp. 63–70, Portland, USA., 2006.

Butler, G., Grogono, P. and Khendek, F.: A Z specification of use cases: a preliminary report, in Proceedings of Joint 4th International Computer Science Conference and 4th Asia Pacific Software Engineering Conference, pp. 505–506, Hong Kong., 1997.

David, A., Deneux, J. and d'Orso, J.: A Formal Semantics for UML Statecharts, Uppsala: Department of Information Technology, Uppsala University. [online] Available from: http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-49014 (Accessed 27 December 2018), 2003.

De Oliveira, R. P., Blanes, D., Gonzalez-Huerta, J., Insfran, E., Abrahão, S. M., Cohen, S. and De Almeida, E. S.: Defining and validating a feature-driven requirements engineering approach, Journal of universal computer science : JUCS, 20(Nr.5), 666–691, doi:10.3217/jucs-020-05-0666, 2014.

Dubauskaite, R. and Vasilecas, O.: Method on Specifying Consistency Rules among Different Aspect Models, expressed in UML, Elektronika ir Elektrotechnika, 19(3), 77–81, doi:10.5755/j01.eee.19.3.2058, 2013.

Dupuy, S., Ledru, Y. and Chabre-Peccoud, M.: An Overview of RoZ : A Tool for Integrating UML and Z Specifications, in Advanced Information Systems Engineering, edited by B. Wangler and L. Bergman, pp. 417–430, Springer Berlin Heidelberg, Stockholm, Sweden., 2000.

El Miloudi, K. and Ettouhami, A.: A Multi-View Approach for Formalizing UML State Machine Diagrams Using Z Notation, WSEAS Transactions on Computers, 14, 72–78, 2015.

El Miloudi, K. and Ettouhami, A.: A Multiview Formal Model of Use Case Diagrams Using Z Notation: Towards Improving Functional Requirements Quality, Journal of Engineering, 2018, 1–9, doi:10.1155/2018/6854920, 2018.

El Miloudi, K., El Amrani, Y. and Ettouhami, A.: An Automated Translation of UML Class Diagrams into a Formal Specification to Detect Inconsistencies, in ICSEA 2011, The Sixth International Conference on Software Engineering Advances, pp. 432–438, Barcelone. [online] Available from: https://www.thinkmind.org/index.php?view=article&articleid=icsea_2011_18_30_10503 (Accessed 6 January 2019), 2011.

El Miloudi, K., El Amrani, Y. and Ettouhami, A.: Using Z Formal Specification for Ensuring Concistency in Multi view Modeling, Journal of Theoretical and Applied Information Technology, 57(3), 407–411, 2013.

Falah, B., Akour, M., Arab, I. and Mhanna, Y.: An Attempt Towards a Formalizing UML Class Diagram Semantics, in New Trends in Information Technology 2017, pp. 21–27, The University of Jordan, Amman, Jordan., 2018.

Fernandes, F. and Song, M.: UML-Checker: An Approach for Verifying UML Behavioral Diagrams, Journal of Software, 9(5), doi:10.4304/jsw.9.5.1229-1236, 2014.

Giese, M. and Heldal, R.: From Informal to Formal Specifications in UML, in «UML» 2004 — The Unified Modeling Language. Modeling Languages and Applications, edited by T. Baar, A. Strohmeier, A. Moreira, and S. J. Mellor, pp. 197–211, Springer Berlin Heidelberg, Lisbon, Portugal., 2004.

Hall, A.: Using Z as a specification calculus for object-oriented systems, in VDM ’90 VDM and Z — Formal Methods in Software Development, edited by D. Bjørner, C. A. R. Hoare, and H. Langmaack, pp. 290–318, Springer Berlin Heidelberg, Kiel, FRG., 1990.

Hall, Anthony. 1994. “Specifying and Interpreting Class Hierarchies in Z.” In *Z User Workshop, Cambridge 1994*, edited by J. P. Bowen and J. A. Hall, 120–38. Workshops in Computing. Springer London.

Ibrahim, N., Ibrahim, R., Saringat, M. Z., Mansor, R. D. and Herawan, T.: Use case driven based rules in ensuring consistency of UML model, AWERProcedia Information Technology and Computer Science, 1, 1485–1491, 2012.

Jones, C.: Systematic Software Development Using VDM, 2nd ed., Prentice Hall, London., 1991.

Kaneiwa, K. and Satoh, K.: Consistency Checking Algorithms for Restricted UML Class Diagrams, in Foundations of Information and Knowledge Systems, edited by J. Dix and S. J. Hegner, pp. 219–239, Springer Berlin Heidelberg, Budapest, Hungary., 2006.

Latella, D., Majzik, I. and Massink, M.: Towards a Formal Operational Semantics of UML Statechart Diagrams, in Formal Methods for Open Object-Based Distributed Systems, edited by P. Ciancarini, A. Fantechi, and R. Gorrieri, pp. 331–347, Springer US, Florence, Italy., 1999.

Ledang, Hung, and Jeanine Souquières. 2001b. “Formalizing UML Behavioral Diagrams with B.” In *Proceedings of the Tenth OOPSLA Workshop on Behavioral Semantics: Back to Basics*, 12 p. https://hal.inria.fr/inria-00107872/document.

Ledang, H. and Souquières, J.: New Approach for Modeling State-Chart Diagrams in B, report. [online] Available from: https://hal.inria.fr/inria-00107544/document (Accessed 27 December 2018b), 2001.

Li, X., Liu, Z. and Jifeng, H.: A formal semantics of UML sequence diagram, in Proceeding of Australian Software Engineering Conference, pp. 168–177, Melbourne, Australia., 2004.

Litvak, B., Tyszberowicz, S. and Yehudai, A.: Behavioral consistency validation of UML diagrams, in Proceeding of the First International Conference onSoftware Engineering and Formal Methods., pp. 118–125, Brisbane, Queensland, Australia., 2003.

Liu, S., Liu, Y., André, É., Choppy, C., Sun, J., Wadhwa, B. and Dong, J. S.: A Formal Semantics for Complete UML State Machines with Communications, in Integrated Formal Methods, edited by E. B. Johnsen and L. Petre, pp. 331–346, Springer Berlin Heidelberg, Turku, Finland., 2013.

Liu, W., Easterbrook, S. and Mylopoulos, J.: Rule-Based Detection of Inconsistency in UML Models, in Workshop on Consistency Problems in UML-Based Software Development, pp. 106–123, San Francisco, USA., 2002.

Liu, X.: Identification and Check of Inconsistencies between UML Diagrams, Journal of Software Engineering and Applications, 06, 73, doi:10.4236/jsea.2013.63B016, 2013.

Lopez-Herrejon, R. E. and Egyed, A.: Detecting Inconsistencies in Multi-View Models with Variability, in Modelling Foundations and Applications, edited by T. Kühne, B. Selic, M.-P. Gervais, and F. Terrier, pp. 217–232, Springer Berlin Heidelberg, Paris, France., 2010.

McUmber, W. E. and Cheng, B. H. C.: A general framework for formalizing UML with formal languages, in Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001, pp. 433–442, Toronto, Ontario, Canada., 2001.

Meisels, I.: Software Manual for Windows Z/EVES v2.3, ORA CANADA., 2004.

Meng, S., Naixiao, Z. and Aichernig, B. K.: The Formal Foundations in RSL for UML Statechart Diagrams, Acta Scientiarum Naturalium Universitatis Pekinensis, 2004. UNU-IIST, P.O. Box 3058, Macau 34., 2005.

Mostafa, A. M., Ismail, M. A., EL-Bolok, H. and Saad, E. M.: Toward a Formalization of UML2.0 Metamodel using Z Specifications, in Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007), vol. 1, pp. 694–701, Qingdao, China., 2007.

Muhamad, Z., Abdulmonim, D. and Alathari, B.: An integration of uml use case diagram and activity diagram with Z language for formalization of library management system, International Journal of Electrical and Computer Engineering, 9, 3069~3076, doi:10.11591/ijece.v9i4. pp. 3069-3076, 2019.

Murali, R., Ireland, A. and Grov, G.: A Rigorous Approach to Combining Use Case Modelling and Accident Scenarios, in NASA Formal Methods, edited by K. Havelund, G. Holzmann, and R. Joshi, pp. 263–278, Springer International Publishing, Pasadena, CA, USA., 2015.

Oliveira, M., Ribeiro, L., Cota, É., Duarte, L. M., Nunes, I. and Reis, F.: Use Case Analysis Based on Formal Methods: An Empirical Study, in Recent Trends in Algebraic Development Techniques, edited by M. Codescu, R. Diaconescu, and I. Țuțu, pp. 110–130, Springer International Publishing, Sinaia, Romania., 2015.

OMG OCL 2.2: The Object Management Group (OMG). The Object Constraint Language v2.2, [online] Available from: https://www.omg.org/spec/OCL/2.2/About-OCL/ (Accessed 13 January 2019), 2010.

OMG UML 2.5.1: The Object Management Group (OMG).The Unified Modeling Language Specification Version 2.5.1, [online] Available from: https://www.omg.org/spec/UML/About-UML/ (Accessed 13 January 2019), 2017.

Saaltink, M.: The Z/EVES system, in ZUM '97: The Z Formal Specification Notation, edited by J. P. Bowen, M. G. Hinchey, and D. Till, pp. 72–85, Springer Berlin Heidelberg, UK., 1997.

Santos, L. B. R. dos, Júnior, V. A. de S. and Vijaykumar, N. L.: Transformation of UML Behavioral Diagrams to Support Software Model Checking, Electronic Proceedings in Theoretical Computer Science, 147, 133–142, doi:10.4204/EPTCS.147.10, 2014.

Satoh, K., Kaneiwa, K. and Uno, T.: Contradiction Finding and Minimal Recovery for UML Class Diagrams, in 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), pp. 277–280, Tokyo., 2006.

Scandurra, P., Arnoldi, A., Yue, T. and Dolci, M.: Functional Requirements Validation by Transforming Use Case Models into Abstract State Machines, in Proceedings of the 27th Annual ACM Symposium on Applied Computing, pp. 1063–1068, ACM, New York, NY, USA., 2012.

Shen, W. and Liu, S.: Formalization, Testing and Execution of a Use Case Diagram, in Formal Methods and Software Engineering, edited by J. S. Dong and J. Woodcock, pp. 68–85, Springer Berlin Heidelberg, Singapore., 2003.

Singh, M., Sharma, A. K. and Saxena, R.: 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, edited by S. C. Satapathy, A. Joshi, N. Modi, and N. Pathak, pp. 25–38, Springer Singapore, Singapore., 2016.

Sinnig, D., Chalin, P. and Khendek, F.: Use Case and Task Models: An Integrated Development Methodology and Its Formal Foundation, ACM Transactions on Software Engineering and Methodology (TOSEM), 22, 31, doi:10.1145/2491509.2491521, 2013.

Snook, C., Savicks, V. and Butler, M.: Verification of UML Models by Translation to UML-B, in Formal Methods for Components and Objects, edited by B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, pp. 251–266, Springer Berlin Heidelberg, Bertinoro, Italy., 2012.

Spivey, J. M.: The Z Notation: A Reference Manual, Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK., 1992.

Tian, B. and Gu, Y.: Formal validation for software modeling, Int. J. Comput. Sci. Issues, 10(2), 308–312, 2013.

W3C XSLT 3.0: XSL Transformations (XSLT) Version 3.0, [online] Available from: https://www.w3.org/TR/xslt-30/#concepts (Accessed 27 July 2019), 2017.

Woodcock, J. and Davies, J.: Using Z: Specification, Refinement, and Proof, Prentice-Hall, Inc., Upper Saddle River, NJ, USA., 1996.

Xia, F. and S Kane, G.: Defining the semantics of UML class and sequence diagrams for ensuring the consistency and executability of OO software specification, in 1st International Workshop on Automated Technology for Verification and Analysis, pp. 77–86, National Taiwan University., 2003.

Yue, T., Briand, L. and Labiche, Y.: Automatically Deriving UML Sequence Diagrams from Use Cases, Carleton University, Canada., 2010.

Yue, T., Briand, L. C. and Labiche, Y.: aToucan: An Automated Framework to Derive UML Analysis Models from Use Case Models, ACM Trans. Softw. Eng. Methodol., 24(3), 13:1–13:52, doi:10.1145/2699697, 2015.

Zafar, N. A.: Formal Specification and Verification of Few Combined Fragments of UML Sequence Diagram, Arabian Journal for Science and Engineering, 41(8), 2975–2986, doi:10.1007/s13369-015-1999-9, 2016.

ANNEXE

- ClassTemplates.xsl : ce fichier contient tous les templates concernant la transformation des classes.

```
<xsl:stylesheet
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns:xs="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://czt.sourceforge.net/zml/zml/Z_2_1.xsd"
     xmlns:UML="href://org.omg/UML/1.3"
     version="2.0">
<xsl:template name="UMLClass">
   <xsl:param name="ClassName" as="xs:string"/>
   <xsl:variable name="XMIID" select="@xmi.id"/>
    <xsl:value-of select="concat('\begin{schema}{',$ClassName,'CoreClass}',$NEWLINE)"/>
 <xsl:choose>
      <xsl:when  test="@xmi.id=//UML:Generalization/@child">
       <xsl:for-each select="//UML:Generalization[@child=$XMIID]">
         <xsl:variable name="SuperClassId" select="@parent"/>
         <xsl:variable name="SuperClassName" select="//UML:Class[@xmi.id=$SuperClassId]/@name"/>
<!-- ici je dois vérifier si la sous classe contient un attribut de même nom qu'un attribut de la classe mère
-->
       <xsl:choose>
          <xsl:when
test="//UML:Attribute[@owner=$XMIID]/@name=//UML:Attribute[@owner=$SuperClassId]/@name">
           <xsl:value-of select="concat('self: ',upper-case($SuperClassName),'\\')"/>
          <!-- ici je dois mettre juste les attributs de la classe mère dont le nom ne figure pas dans la classe
fille -->
           <xsl:for-each select="//UML:Attribute[@owner=$SuperClassId]">
               <xsl:variable name="attributeName" select="@name"/>
              <xsl:variable name="AttributeType" select="@type"/>
              <xsl:variable name="Type" select="//UML:Class[@xmi.id=$AttributeType]/@name"/>
                <xsl:if test="$attributeName!=//UML:Attribute[@owner=$XMIID]/@name">
                    <xsl:value-of select="concat('\\',' ',$attributeName,' : ',$Type)"/>
                </xsl:if>
           </xsl:for-each>
         </xsl:when>
          <xsl:otherwise>
              <xsl:value-of select="concat($SuperClassName,'CoreClass','\\')"/>
          </xsl:otherwise>
    </xsl:choose>
     </xsl:for-each>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="concat(' self: ',upper-case($ClassName),$NEWLINE)"/>
    </xsl:otherwise>
 </xsl:choose>
    <xsl:for-each select="//UML:Attribute[@owner=$XMIID]">
            <xsl:variable name="AttributeType" select="@type"/>
            <xsl:variable name="Type" select="//UML:Class[@xmi.id=$AttributeType]/@name"/>
            <xsl:value-of select="concat ('\\',' ',@name,' : ',$Type)"/>
                <xsl:if test="not(position()=last())">
                    <xsl:text>\\</xsl:text>
                </xsl:if>
            <xsl:value-of select="$NEWLINE"/>
    </xsl:for-each>
 <xsl:if  test="@xmi.id=//UML:Generalization/@child">
    <xsl:value-of select="concat('\where',$NEWLINE,'self \in ',upper-case($ClassName),'\\')"/>
 </xsl:if>
       <xsl:value-of select="concat($NEWLINE,$NEWLINE,'\end{schema}',$NEWLINE)"/>
       <xsl:value-of select="concat('\begin{zed}',$NEWLINE,
```

```
            $ClassName,'::= nil',$ClassName,' |',lower-case($ClassName),'coreclassto',lower-
case($ClassName),' \ldata ',
            $ClassName,'CoreClass',' \rdata',$NEWLINE,
           '\end{zed}')"/>
</xsl:template>
<xsl:template name="classSystem">
  <xsl:param name="ClassName" as="xs:string"/>
      <xsl:value-of select="concat($NEWLINE,'\begin{schema}{S',$ClassName,' }',$NEWLINE)"/>
      <xsl:value-of select="concat(' ',lower-case($ClassName),'s : \power ',$ClassName,'\\')"/>
      <xsl:value-of select="concat($NEWLINE,' id',$ClassName,': ')"/>
      <xsl:value-of select="concat(upper-case($ClassName),' \pfun ',$ClassName,$NEWLINE,'\\')"/>
      <xsl:value-of select="concat(lower-case($ClassName),'Ids : \power ',upper-
case($ClassName),$NEWLINE,'\where',$NEWLINE)"/>
      <xsl:value-of select="concat('id',$ClassName,'= \{ ',lower-case($ClassName),'coreclass:
',$ClassName,'CoreClass @ ','\\',
            lower-case($ClassName),'coreclass.self \mapsto ',lower-case($ClassName),'coreclassto',lower-
case($ClassName),
            '\ ',lower-case($ClassName),'coreclass \}\\')"/>
      <xsl:value-of select="concat(lower-case($ClassName),'Ids = \dom\ id',$ClassName)"/>
      <xsl:value-of select="concat($NEWLINE,'\end{schema}',$NEWLINE) "/>
</xsl:template>
<xsl:template name="System">
   <xsl:value-of select="concat($NEWLINE,'\begin{schema}{ System }',$NEWLINE)"/>
      <xsl:for-each select="UML:Namespace.ownedElement/UML:Class">
          <xsl:value-of select="concat($NEWLINE,'S',@name)"/>
            <xsl:if test="not(position()=last())">
                <xsl:text>\\</xsl:text>
            </xsl:if>
          <xsl:value-of select="$NEWLINE"/>
      </xsl:for-each>
      <xsl:for-each select="UML:Namespace.ownedElement/UML:AssociationClass">
          <xsl:value-of select="concat('\\',$NEWLINE,'S',@name)"/>
      </xsl:for-each>
   <xsl:value-of select="concat($NEWLINE,'\end{schema}',$NEWLINE) "/>
</xsl:template>
</xsl:stylesheet>
```

- OperationTemplates : ce fichier contient tous les modèles nécessaires pour traduire les opérations

```
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://czt.sourceforge.net/zml/zml/Z_2_1.xsd"
    xmlns:UML="href://org.omg/UML/1.3"
    version="2.0">
<xsl:template name="ClassOp">
 <xsl:param name="ClassName" as="xs:string"/>
 <xsl:value-of select="concat($NEWLINE,'\begin{schema}{ ',$ClassName,'Op }',$NEWLINE)"/>
 <xsl:value-of select="concat('\Delta ',$ClassName,'CoreClass',$NEWLINE)"/>
 <xsl:value-of select="concat('\where',$NEWLINE,'self ',$quote,'= self',$NEWLINE)"/>
 <xsl:value-of select="concat('\end{schema}',$NEWLINE)"/>
</xsl:template>

<xsl:template name="ClassGet">
 <xsl:param name="ClassName" as="xs:string"/>
 <xsl:value-of select="concat($NEWLINE,'\begin{schema}{ ',$ClassName,'Get }',$NEWLINE)"/>
```

```
  <xsl:value-of select="concat('\Xi ',$ClassName,'CoreClass',$NEWLINE)"/>
  <xsl:value-of select="concat('\where',$NEWLINE,'self ',$quote,'= self',$NEWLINE)"/>
  <xsl:value-of select="concat('\end{schema}',$NEWLINE)"/>
</xsl:template>

<xsl:template name="AddToSystem">
 <xsl:param name="ClassName" as="xs:string"/>
  <xsl:value-of select="concat($NEWLINE,'\begin{schema}{ Add',$ClassName,'ToSystem }',$NEWLINE)"/>
  <xsl:choose>
    <xsl:when test="not(exists(UML:Classifier.feature/UML:Attribute))">
      <xsl:value-of select="concat($ClassName,'CoreClass \\',$NEWLINE)"/>
      <xsl:value-of select="concat('\Delta\ S',$ClassName,$NEWLINE)"/>
      <xsl:value-of select="concat('\where',$NEWLINE,'self \notin \dom id',$ClassName,'\\',$NEWLINE)"/>
      <xsl:value-of select="concat('id',$ClassName,$quote,' = id',$ClassName,' \cup \{( self \mapsto ')"/>
      <xsl:value-of select="concat(lower-case($ClassName),'coreclassto',lower-case($ClassName),' \theta
',$ClassName,'CoreClass)\}',$NEWLINE)"/>
      <xsl:value-of select="concat('\end{schema}',$NEWLINE)"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="concat('Init',$ClassName,'CoreClassByDefault \\',$NEWLINE)"/>
      <xsl:value-of select="concat('\Delta\ S',$ClassName,$NEWLINE)"/>
      <xsl:value-of select="concat('\where',$NEWLINE,'\theta ',$ClassName,'CoreClass',$quote,'.self \notin \dom
id',$ClassName,'\\',$NEWLINE)"/>
      <xsl:value-of select="concat('id',$ClassName,$quote,' = id',$ClassName,' \cup \{(\theta
',$ClassName,'CoreClass',$quote,'.self \mapsto ')"/>
      <xsl:value-of select="concat(lower-case($ClassName),'coreclassto',lower-case($ClassName),'  \theta
',$ClassName,'CoreClass',$quote,')\}',$NEWLINE)"/>
      <xsl:value-of select="concat('\end{schema}',$NEWLINE)"/>
 </xsl:otherwise>
 </xsl:choose>
</xsl:template>

<xsl:template name="setOperation">
  <xsl:param name="attributeName" as="xs:string"/>
  <xsl:param name="ClassName" as="xs:string"/>

  <xsl:value-of select="concat($NEWLINE,'\begin{schema}{ set',$attributeName,$ClassName,' }',$NEWLINE)"/>
    <xsl:value-of select="concat($ClassName,'Op \\',$NEWLINE)"/>

    <xsl:variable name="type" select="@type"/>
    <xsl:variable name="ParameterType" select="//UML:Class[@xmi.id=$type]/@name"/>

       <xsl:value-of select="concat($attributeName,'?:',$ParameterType)"/>
       <xsl:value-of select="concat($NEWLINE,'\end{schema}',$NEWLINE) "/>

</xsl:template>

<xsl:template name="setFunction">
  <xsl:param name="attributeName" as="xs:string"/>
  <xsl:param name="ClassName" as="xs:string"/>

    <xsl:value-of select="concat($NEWLINE,'\begin{axdef}',$NEWLINE)"/>
    <xsl:value-of select="concat('Rset',$attributeName,$ClassName,': ')"/>
      <xsl:variable name="type" select="@type"/>
      <xsl:variable name="ParameterType" select="//UML:Class[@xmi.id=$type]/@name"/>
        <xsl:value-of select="concat($ParameterType,' \fun ',$ClassName,' \fun ',$ClassName, $NEWLINE)"/>
        <xsl:value-of select="concat('\where',$NEWLINE,'Rset',$attributeName,$ClassName,$NEWLINE,'= \{ ') "/>
        <xsl:value-of select="concat($attributeName,' : ',$ParameterType,'\ @\ ',$attributeName)"/>
        <xsl:value-of select="concat($NEWLINE,' \mapsto \{set',$attributeName,$ClassName,' | ')"/>
        <xsl:value-of select="concat(@name,'? = ',@name)"/>
```

```
      <xsl:value-of select="concat('@ (',lower-case($ClassName),'coreclassto',lower-case($ClassName),' \theta
',$ClassName,'CoreClass \mapsto ',lower-case($ClassName),'coreclassto',lower-case($ClassName),' \theta
',$ClassName,'CoreClass',$quote,' )\} \} ')"/>
      <xsl:value-of select="concat($NEWLINE,'\end{axdef}')"/>
</xsl:template>

<xsl:template name="setOperationSystem">
  <xsl:param name="attributeName" as="xs:string"/>
  <xsl:param name="ClassName" as="xs:string"/>
   <xsl:value-of select="concat('\begin{schema}{ set',$attributeName,$ClassName,'System }',$NEWLINE)"/>
   <xsl:value-of select="concat('\Delta S',$ClassName,' \\',$NEWLINE)"/>
    <xsl:value-of select="concat(lower-case($ClassName),'? : ',upper-case($ClassName),'\\')"/>
    <xsl:variable name="type" select="@type"/>
    <xsl:variable name="ParameterType" select="//UML:Class[@xmi.id=$type]/@name"/>
    <xsl:value-of select="concat($attributeName,'? : ',$ParameterType, '\\',$NEWLINE)"/>
    <xsl:value-of select="concat('\where',$NEWLINE)"/>
    <xsl:value-of select="concat('id',$ClassName,$quote,$NEWLINE,' = ','id',$ClassName,' \oplus (\{',lower-
case($ClassName),'?\} \dres id',$ClassName,' \semi Rset',$attributeName,$ClassName,'\ ',$attributeName,'?')"/>
    <xsl:value-of select="concat(')',$NEWLINE,'\end{schema}')"/>
</xsl:template>

<xsl:template name="getOperation">
  <xsl:param name="attributeName" as="xs:string"/>
  <xsl:param name="ClassName" as="xs:string"/>
  <xsl:value-of select="concat($NEWLINE,'\begin{schema}{ get',$attributeName,$ClassName,' }',$NEWLINE)"/>
  <xsl:value-of select="concat($ClassName,'Get \\',$NEWLINE)"/>
  <xsl:variable name="type" select="@type"/>
   <xsl:variable name="ParameterType" select="//UML:Class[@xmi.id=$type]/@name"/>

     <xsl:value-of select="concat($attributeName,'!:',$ParameterType)"/>
     <xsl:value-of select="concat('\where',$NEWLINE)"/>
     <xsl:value-of select="concat($attributeName,'!=',$attributeName)"/>
      <xsl:value-of select="concat($NEWLINE,'\end{schema}',$NEWLINE) "/>
</xsl:template>

<xsl:template name="testOp">
   <xsl:variable name="XMIID" select="@xmi.id"/>
     <xsl:if test="exists(//UML:Generalization[@child=$XMIID])">
     <xsl:variable name="child" select="//UML:Generalization[@child=$XMIID]/@child"/>
     <xsl:variable name="childName" select="//UML:Class[@xmi.id=$child]/@name"/>
     <xsl:variable name="parent" select="//UML:Generalization[@child=$XMIID]/@parent"/>
     <xsl:variable name="parentName" select="//UML:Class[@xmi.id=$parent]/@name"/>
     <xsl:value-of select="concat('[CHILD',$childName, 'PARENT',$parentName,']')"/>
  </xsl:if>
</xsl:template>

<xsl:template name="Methods">
 <xsl:param name="ClassName" as="xs:string"/>
  <xsl:variable name="XMIID" select="@xmi.id"/>
    <xsl:for-each select="UML:Classifier.feature/UML:Operation">
      <xsl:variable name="OperationName" select="@name"/>
      <xsl:variable name="OperationId" select="@xmi.id"/>
<xsl:choose>
   <xsl:when test="exists(//UML:Generalization[@child=$XMIID])">
     <xsl:variable name="child" select="//UML:Generalization[@child=$XMIID]/@child"/>
     <xsl:variable name="childName" select="//UML:Class[@xmi.id=$child]/@name"/>
     <xsl:variable name="parent" select="//UML:Generalization[@child=$XMIID]/@parent"/>
     <xsl:variable name="parentName" select="//UML:Class[@xmi.id=$parent]/@name"/>
    <xsl:variable name="OpParent" select="substring($OperationName,4)"/>
    <xsl:variable name="OpParentId" select="//UML:Operation[@name=$OpParent]/@xmi.id"/>
```

```
    <xsl:variable name="OpParentId2" select="//UML:Operation[@owner=$parent and
@name=$OperationName]/@xmi.id"/>
      <xsl:choose>

         <!-- si le nom de la méthode correspond à un set de la classe mère -->
         <!-- je vérifie la signature de la méthode (nombre de paramètres et leur type) -->
         <xsl:when test="(substring($OperationName,1,3)='set') and
($OpParent=//UML:Attribute[@owner=$parent]/@name) and
(//UML:Parameter[@behavioralFeature=$OperationId]/@kind='in') and
(count(//UML:Parameter[@behavioralFeature=$OperationId])=1) and
(every $p1 in //UML:Parameter[@behavioralFeature=$OperationId]/@type satisfies
$p1=//UML:Attribute[@owner=$parent and @name=$OpParent]/@type) ">

                <xsl:call-template name="redefinedSet">
                  <xsl:with-param name="OpName" select="$OperationName"/>
                  <xsl:with-param name="ClassName" select="$ClassName"/>
                  <xsl:with-param name="parentName" select="$parentName"/>
                </xsl:call-template>


         </xsl:when>

         <!-- si le nom de la méthode correspond à un get de la classe mère -->
        <!-- je vérifie la signature de la méthode (nombre de paramètres et leur type) -->

         <xsl:when test="(substring($OperationName,1,3)='get' )and
($OpParent=//UML:Attribute[@owner=$parent]/@name) and
(count(//UML:Parameter[@behavioralFeature=$OperationId])=1) and
 (//UML:Parameter[@behavioralFeature=$OperationId]/@kind='return') and
(every $p1 in //UML:Parameter[@behavioralFeature=$OperationId]/@type satisfies
$p1=//UML:Attribute[@owner=$parent and @name=$OpParent]/@type)">

                <xsl:call-template name="redefinedGet">
                  <xsl:with-param name="OpName" select="$OperationName"/>
                  <xsl:with-param name="ClassName" select="$ClassName"/>
                  <xsl:with-param name="parentName" select="$parentName"/>
                </xsl:call-template>
         </xsl:when>

         <!-- si la méthode de la classe fille est une méthode redéfinie -->
       <!-- je vérifie la signature de la méthode (nombre de paramètres et leur type) -->

         <xsl:when test="($OperationName=//UML:Operation[@owner=$parent]/@name) and
(count(//UML:Parameter[@behavioralFeature=$OperationId])=count(//UML:Parameter[@behavioralFeature=$Op
ParentId2])) and
(every $p1 in //UML:Parameter[@behavioralFeature=$OperationId]/@type satisfies
$p1=//UML:Parameter[@behavioralFeature=$OpParentId2]/@type) and
(every $p2 in //UML:Parameter[@behavioralFeature=$OpParentId2]/@type satisfies
$p2=//UML:Parameter[@behavioralFeature=$OperationId]/@type)">

                  <xsl:call-template name="redefinedMethod">
                       <xsl:with-param name="OpName" select="$OperationName"/>
                       <xsl:with-param name="ClassName" select="$ClassName"/>
                       <xsl:with-param name="parentName" select="$parentName"/>
                  </xsl:call-template>
           </xsl:when>
           <xsl:otherwise>
              <xsl:call-template name="classMethod">
                  <xsl:with-param name="OpName" select="$OperationName"/>
                  <xsl:with-param name="ClassName" select="$ClassName"/>
```

```
                </xsl:call-template>
            </xsl:otherwise>
  </xsl:choose>
 </xsl:when>
 <xsl:otherwise>
                <xsl:call-template name="classMethod">
                    <xsl:with-param name="OpName" select="$OperationName"/>
                    <xsl:with-param name="ClassName" select="$ClassName"/>
                </xsl:call-template>
 </xsl:otherwise>
 </xsl:choose>
</xsl:for-each>
</xsl:template>

<xsl:template name="redefinedSet">
  <xsl:param name="OpName" as="xs:string"/>
  <xsl:param name="ClassName" as="xs:string"/>
  <xsl:param name="parentName" as="xs:string"/>
<xsl:variable name="attributeName" select="substring($OpName,4)"/>
<xsl:variable name="type"
select="//UML:Operation[@name=$OpName]/UML:BehavioralFeature.parameter/UML:Parameter[@name=$attri
buteName]/@type"/>
<xsl:variable name="ParameterType" select="//UML:Class[@xmi.id=$type]/@name"/>
  <xsl:value-of select="concat($NEWLINE,'\begin{schema}{ set',$attributeName,$ClassName,' }',$NEWLINE)"/>
    <xsl:value-of select="concat($ClassName,'Op \\',$NEWLINE)"/>
    <xsl:value-of select="concat('set',$attributeName,$parentName)"/>
    <xsl:value-of select="concat($NEWLINE,'\end{schema}',$NEWLINE) "/>

 <xsl:value-of select="concat($NEWLINE,'\begin{axdef}',$NEWLINE)"/>
 <xsl:value-of select="concat('Rset',$attributeName,$ClassName,': ')"/>
 <xsl:value-of select="concat($ParameterType,' \fun ',$ClassName,' \fun ',$ClassName, $NEWLINE)"/>
 <xsl:value-of select="concat('\where',$NEWLINE,'Rset',$attributeName,$ClassName,$NEWLINE,'= \{ ') "/>
 <xsl:value-of select="concat($attributeName,' : ',$ParameterType,'\ @\ ',$attributeName)"/>
 <xsl:value-of select="concat($NEWLINE,' \mapsto \{set',$attributeName,$ClassName,' | ')"/>
 <xsl:value-of select="concat($attributeName,'? = ',$attributeName)"/>
 <xsl:value-of select="concat('@ (',lower-case($ClassName),'coreclassto',lower-case($ClassName),' \theta
',$ClassName,'CoreClass \mapsto ',lower-case($ClassName),'coreclassto',lower-case($ClassName),' \theta
',$ClassName,'CoreClass',$quote,' )\} \} ')"/>

    <xsl:value-of select="concat($NEWLINE,'\end{axdef}')"/>
    <xsl:value-of select="concat('\begin{schema}{ set',$attributeName,$ClassName,'System }',$NEWLINE)"/>
    <xsl:value-of select="concat('\Delta S',$ClassName,' \\',$NEWLINE)"/>
    <xsl:value-of select="concat(lower-case($parentName),'? : ',upper-case($parentName),'\\')"/>
    <xsl:value-of select="concat($attributeName,'? : ',$ParameterType, '\\',$NEWLINE)"/>
    <xsl:value-of select="concat('\where',$NEWLINE)"/>
    <xsl:value-of select="concat('id',$ClassName,$quote,$NEWLINE,' = ','id',$ClassName,' \oplus (\{',lower-
case($parentName),'?\} \dres id',$ClassName,' \semi Rset',$attributeName,$ClassName,'\ ',$attributeName,'?')"/>
    <xsl:value-of select="concat(')',$NEWLINE,'\end{schema}')"/>
</xsl:template>

<xsl:template name="redefinedGet">
  <xsl:param name="OpName" as="xs:string"/>
  <xsl:param name="ClassName" as="xs:string"/>
<xsl:param name="parentName" as="xs:string"/>
  <xsl:variable name="attributeName" select="substring($OpName,4)"/>
  <xsl:value-of select="concat($NEWLINE,'\begin{schema}{ get',$attributeName,$ClassName,' }',$NEWLINE)"/>
    <xsl:value-of select="concat($ClassName,'Get \\',$NEWLINE)"/>
    <xsl:value-of select="concat('get',$attributeName,$parentName)"/>
    <xsl:value-of select="concat($NEWLINE,'\end{schema}',$NEWLINE) "/>
</xsl:template>
```

```
<xsl:template name="redefinedMethod">
  <xsl:param name="OpName" as="xs:string"/>
  <xsl:param name="ClassName" as="xs:string"/>
 <xsl:param name="parentName" as="xs:string"/>
    <xsl:value-of select="concat($NEWLINE,'\begin{schema}{ ',$OpName,$ClassName,' }',$NEWLINE)"/>
    <xsl:value-of select="concat($OpName,$parentName,'\\',$NEWLINE)"/>
    <xsl:value-of select="concat($NEWLINE,'\end{schema}',$NEWLINE) "/>
</xsl:template>

<xsl:template name="classMethod">
  <xsl:param name="OpName" as="xs:string"/>
   <xsl:param name="ClassName" as="xs:string"/>
   <xsl:value-of select="concat($NEWLINE,'\begin{schema}{ ',$OpName,$ClassName,' }',$NEWLINE)"/>
     <xsl:value-of select="concat($ClassName,'Op \\',$NEWLINE)"/>
     <xsl:variable name="XMIID" select="@xmi.id"/>
      <xsl:for-each select="//UML:Parameter[@behavioralFeature=$XMIID]">

         <xsl:variable name="type" select="@type"/>
         <xsl:variable name="ParameterType" select="//UML:Class[@xmi.id=$type]/@name"/>
         <xsl:if test="@kind='in'">
               <xsl:value-of select="concat(@name,'?: ',$ParameterType,$NEWLINE)"/>
         </xsl:if>
         <xsl:if test="@kind='return'">
               <xsl:value-of select="concat(lower-case($ParameterType),'!: ',$ParameterType,$NEWLINE)"/>
         </xsl:if>
         <xsl:if test="not(position()=last())">
                     <xsl:text>\\</xsl:text>
         </xsl:if>
      </xsl:for-each>
      <xsl:value-of select="concat($NEWLINE,'\end{schema}',$NEWLINE) "/>

</xsl:template>
</xsl:stylesheet>
```

- InitialisationTemplates : ce fichier contient tous les templates responsables de l'initialisation des classes

```
<xsl:stylesheet
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns:xs="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://czt.sourceforge.net/zml/zml/Z_2_1.xsd"
     xmlns:UML="href://org.omg/UML/1.3"
     version="2.0">

<!-- schéma d'initialisation de chaque classe -->
<xsl:template name="InitClassByDefault">

 <xsl:param name="ClassName" as="xs:string"/>
  <xsl:variable name="IdClass"  select="@xmi.id"/>
   <xsl:value-of select="concat($NEWLINE,'\begin{schema}{ Init',$ClassName,'CoreClassByDefault
}',$NEWLINE)"/>
   <xsl:value-of select="concat($ClassName,'CoreClass',$quote,'\\',$NEWLINE)"/>
   <!--    <xsl:for-each select="//UML:Attribute[@owner=$IdClass]">
      <xsl:variable name="AttributeType" select="@type"/>
            <xsl:variable name="Type" select="//UML:Class[@xmi.id=$AttributeType]/@name"/>
            <xsl:value-of select="concat(' ',@name,' ?: ',$Type)"/>
                  <xsl:if test="not(position()=last())">
```

```
                    <xsl:text>\\</xsl:text>
                </xsl:if>
            <xsl:value-of select="$NEWLINE"/>
        </xsl:for-each>   -->
        <xsl:value-of select="concat($NEWLINE,'\where',$NEWLINE)"/>
        <xsl:for-each select="//UML:Attribute[@owner=$IdClass]">
        <xsl:variable name="AttributeType" select="@type"/>
            <xsl:variable name="Type" select="//UML:Class[@xmi.id=$AttributeType]/@name"/>
            <xsl:value-of select="concat(' ',@name,$quote,' = nil',$Type)"/>
                <xsl:if test="not(position()=last())">
                    <xsl:text>\\</xsl:text>
                </xsl:if>
            <xsl:value-of select="$NEWLINE"/>
        </xsl:for-each>
    <xsl:value-of select="concat($NEWLINE,'\end{schema}',$NEWLINE) "/>

</xsl:template>

<!-- schéma d'initialisation de chaque classe (avec des valeurs en entrée)-->
<xsl:template name="InitClassWithValues">

 <xsl:param name="ClassName" as="xs:string"/>
  <xsl:variable name="IdClass"  select="@xmi.id"/>
    <xsl:value-of select="concat($NEWLINE,'\begin{schema}{ Init',$ClassName,'CoreClassWithValues
}',$NEWLINE)"/>
    <xsl:value-of select="concat($ClassName,'CoreClass',$quote,'\\',$NEWLINE)"/>
    <xsl:for-each select="//UML:Attribute[@owner=$IdClass]">
        <xsl:variable name="AttributeType" select="@type"/>
            <xsl:variable name="Type" select="//UML:Class[@xmi.id=$AttributeType]/@name"/>
            <xsl:value-of select="concat(' ',@name,' ?: ',$Type)"/>
                <xsl:if test="not(position()=last())">
                    <xsl:text>\\</xsl:text>
                </xsl:if>
            <xsl:value-of select="$NEWLINE"/>
        </xsl:for-each>
        <xsl:value-of select="concat($NEWLINE,'\where',$NEWLINE)"/>
        <xsl:for-each select="//UML:Attribute[@owner=$IdClass]">
        <xsl:variable name="AttributeType" select="@type"/>
            <xsl:variable name="Type" select="//UML:Class[@xmi.id=$AttributeType]/@name"/>
            <xsl:value-of select="concat(' ',@name,$quote,' = ',@name,'?')"/>
                <xsl:if test="not(position()=last())">
                    <xsl:text>\\</xsl:text>
                </xsl:if>
            <xsl:value-of select="$NEWLINE"/>
        </xsl:for-each>
    <xsl:value-of select="concat($NEWLINE,'\end{schema}',$NEWLINE) "/>
</xsl:template>

</xsl:stylesheet>
```

- HierarchyTemplates : ce fichier contient tous les templates concernant la traduction des relations d'héritage

```
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://czt.sourceforge.net/zml/zml/Z_2_1.xsd"
    xmlns:UML="href://org.omg/UML/1.3"
```

```
    version="2.0">

<xsl:variable name="NEWLINE" select="'&#xA;'"/>
<xsl:variable name="quote">'</xsl:variable>

<xsl:template name="Hierarchy">
   <xsl:variable name="childId" select="@child"/>
   <xsl:variable name="parentId" select="@parent"/>
   <xsl:variable name="childName" select="//UML:Class[@xmi.id=$childId]/@name"/>
   <xsl:variable name="parentName" select="//UML:Class[@xmi.id=$parentId]/@name"/>

   <xsl:value-of select="concat('\begin{schema} {',$parentName,$childName,'Hierarchy}')"/>
   <xsl:value-of select="concat('S',$parentName,'\\','S',$childName,$NEWLINE,'\where',$NEWLINE,lower-
case($childName),'Ids = ',lower-case($parentName),'Ids \cap ',upper-case($childName),'\\')"/>
   <xsl:value-of select="concat('\forall\ ',substring(lower-case($childName),1,1),':',lower-case($childName),'Ids
@ (\lambda ',$childName,'CoreClass', '@\ \theta\ ',$parentName,'CoreClass)\ (',lower-
case($childName),'coreclassto',lower-case($childName),'\ \inv\ (id',$childName,'\ ',substring(lower-
case($childName),1,1),'))\ =',lower-case($parentName),'coreclassto',lower-case($parentName),'\ \inv\
(id',$parentName,'\ ',substring(lower-case($childName),1,1),')'  )"/>

    <xsl:value-of select="concat('\end{schema}',$NEWLINE)"/>
</xsl:template>

<xsl:template name="ClassNames">
<xsl:value-of select="concat('\begin{axdef}',$NEWLINE)"/>
  <xsl:for-each select="UML:Namespace.ownedElement/UML:Class">
      <xsl:value-of select="concat(@name,'Class')"/>
       <xsl:if test="not(position()=last())">
        <xsl:text> , </xsl:text>
       </xsl:if>
  </xsl:for-each>
<xsl:value-of select="concat(':CLASS',$NEWLINE,'\end{axdef}',$NEWLINE)"/>
</xsl:template>

<xsl:template name="HierarchySystem">
<xsl:text>
\begin{schema}{ClassSystem}
  classes: \power  CLASS\\
  subSuper: CLASS \rel  CLASS\\
  typeOf: \power  OBJECT \pfun  CLASS
\where
  subSuper \plus  \cap  \id  Class = \emptyset\\
  \dom  subSuper \subseteq  classes\\
  \ran  subSuper \subseteq  classes\\
  subSuper=\{
</xsl:text>

<xsl:for-each select="UML:Namespace.ownedElement/UML:Generalization">
   <xsl:variable name="childId" select="@child"/>
   <xsl:variable name="parentId" select="@parent"/>
   <xsl:variable name="childName" select="//UML:Class[@xmi.id=$childId]/@name"/>
   <xsl:variable name="parentName" select="//UML:Class[@xmi.id=$parentId]/@name"/>

  <xsl:value-of select="concat($childName,'Class \mapsto ',$parentName,'Class')"/>
       <xsl:if test="not(position()=last())">
         <xsl:text>,\\ </xsl:text>
       </xsl:if>
</xsl:for-each>

<xsl:value-of select="concat('\}\\','typeOf =\{')"/>
```

```
<xsl:for-each select="UML:Namespace.ownedElement/UML:Class">
   <xsl:value-of select="concat(upper-case(@name),'\mapsto ',@name,'Class')"/>
        <xsl:if test="not(position()=last())">
          <xsl:text>,\\ </xsl:text>
        </xsl:if>
</xsl:for-each>

<xsl:value-of select="concat('\}',$NEWLINE,'\end{schema}')"/>
</xsl:template>

</xsl:stylesheet>
```

- AssociationTemplates : ce fichier contient les templates dont on a besoin pour représenter les associations

```
<xsl:stylesheet
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns:xs="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://czt.sourceforge.net/zml/zml/Z_2_1.xsd"
     xmlns:UML="href://org.omg/UML/1.3"
     version="2.0">

<xsl:template name="Association">

  <xsl:param name="classAssociationName" as="xs:string"/>
  <xsl:param name="associationEndName" as="xs:string"/>

    <xsl:value-of
select="concat('\begin{schema}{',$classAssociationName,$associationEndName,'}',$NEWLINE)"/>
    <xsl:value-of select="concat('self:',upper-case($classAssociationName),upper-
case($associationEndName),'\\',lower-case($classAssociationName),lower-case($associationEndName),':',upper-
case($classAssociationName),'\ \rel\ ',upper-case($associationEndName),'\where',$NEWLINE)"/>

       <xsl:variable name="BorneInf"
select="UML:AssociationEnd.multiplicity/UML:Multiplicity/UML:Multiplicity.range/UML:MultiplicityRange/@l
ower"/>
       <xsl:variable name="BorneSup"
select="UML:AssociationEnd.multiplicity/UML:Multiplicity/UML:Multiplicity.range/UML:MultiplicityRange/@u
pper"/>

        <xsl:if test="$BorneInf='0' and $BorneSup='1'">
             <xsl:value-of select="concat(' \# (\ran  ',lower-case($classAssociationName),lower-
case($associationEndName),') \leq\ 1')"/>
        </xsl:if>

        <xsl:if test="($BorneInf='-1' and $BorneSup='-1')or ($BorneInf='0' and $BorneSup='-1')">
             <xsl:value-of select="concat(' \# (\ran  ',lower-case($classAssociationName),lower-
case($associationEndName),') \geq\ 0')"/>
        </xsl:if>

        <xsl:if test="$BorneInf='1' and $BorneSup='1'">
             <xsl:value-of select="concat(' \# (\ran  ',lower-case($classAssociationName),lower-
case($associationEndName),')= \ 1')"/>
        </xsl:if>

        <xsl:if test="$BorneInf='1' and $BorneSup='-1'">
             <xsl:value-of select="concat(' \# (\ran  ',lower-case($classAssociationName),lower-
case($associationEndName),') \geq\ 1')"/>
```

```
        </xsl:if>

<xsl:value-of select="concat($NEWLINE,'\end{schema}',$NEWLINE)"/>

</xsl:template>

</xsl:stylesheet>
```

Résumé

L'objectif de nos travaux de recherche consiste à proposer une approche formelle pour la validation et la vérification du standard UML en utilisant la notation Z. Nous avons étudié UML dans le cadre d'une modélisation multi vues à travers ses diagrammes les plus fréquemment utilisés. La modélisation multi vues présente plusieurs défis, notamment ceux liés à la cohérence, qui pourraient potentiellement mettre en péril l'intégrité du système. Nous avons exploité le langage Z qui, d'un côté, dispose d'une sémantique rigoureuse basée sur la logique mathématique et, d'un autre côté, offre suffisamment d'expressivité pour spécifier les différents diagrammes UML autant au niveau statique qu'au niveau dynamique. Nous avons proposé une modélisation formelle d'un ensemble de diagrammes UML en utilisant la notation Z. Une étude de la consistance est proposée à l'aide d'un ensemble de règles et de théorèmes en se basant sur le modèle formel proposé pour chaque diagramme.

Ainsi, les logiciels obtenus sont de qualité suffisante pour assurer une réelle crédibilité et fiabilité à leur contenu. Les solutions proposées dans cette thèse contribuent de manière significative à l'amélioration de la qualité des logiciels et de leurs consistances et à la réduction des coûts lors du processus de développement.

Mots-clefs : Méthodes formelles, UML, Z, Vérification, Inconsistances, Modélisation multi vues.

Abstract

The objective of our research is to propose a formal approach for the validation and verification of the UML standard using Z notation. We studied UML as part of a multi-view modeling through its more frequently used diagrams. Multi-view modeling presents several challenges, including those related to consistency, which could potentially jeopardize the integrity of the system. We showed the interest of using the Z language which, on the one hand, has a rigorous semantics based on mathematical logic and, on the other hand, offers enough expressiveness to specify the different UML diagrams both static and dynamic. We proposed formal modeling of a set of UML diagrams using Z notation. A consistency study is proposed using a set of rules and theorems based on the formal model proposed for each diagram.

Thus, the software obtained is of sufficient quality to ensure real credibility and reliability to their content. The solutions proposed in this thesis contribute significantly to improving the quality of software and their consistencies and reducing costs during the development process.

Keywords : Formal Methods, UML, Z, Consistency Checking, Multi-view Modeling.