Contribution à l'ingénierie des systèmes : Raffinement et Refactoring de spécifications UML

Boulbaba Ben Ammar

▶ To cite this version:

Boulbaba Ben Ammar. Contribution à l'ingénierie des systèmes : Raffinement et Refactoring de spécifications UML. Génie logiciel [cs.SE]. Université de Sfax, 2012. Français. NNT : . tel-00693693

HAL Id: tel-00693693
https://theses.hal.science/tel-00693693

Submitted on 2 May 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Ministre de l'Enseignement Supérieur et de la Recherche Scientifique

Université de Sfax

Faculté des Sciences Economiques et de Gestion de Sfax

THESE DE DOCTORAT EN INFORMATIQUE

Présentée et soutenue publiquement par

Boulbaba BEN AMMAR

Contribution à l'ingénierie des systèmes Raffinement et Refactoring de spécifications UML

Composition du jury

Mr. Faiez Gargouri	Professeur d'Ens. Sup. à l'ISIMS Tunisie	*Président*
Mr. Abdelmajid Ben Hamadou	Professeur d'Ens. Sup. à l'ISIMS Tunisie	*Directeur de thèse*
Mr. Jean Pierre Giraudin	Professeur d'Ens. Sup. à l'Université Pierre Mendès France	*Rapporteur*
Mme. Hanêne Ben Abdallah	Professeur d'Ens. Sup. à la FSEGS Tunisie	*Rapporteur*
Mme. Jeanine Souquières	Professeur d'Ens. Sup. à l'Université de Lorraine France	*Membre*
Mr. Mohamed Tahar Bhiri	Maitre Assistant à la FSS Tunisie	*Co-encadreur*

Année Universitaire | 2011-2012

Remerciements

Mon premier remerciement s'adresse à mes directeurs de recherche Madame le professeur Jeanine Souquières et Monsieur le Professeur Adelmajid Ben Hamadou pour leurs précieux conseils, leur bienveillance, et toute l'attention qu'ils ont accordé à mon égard, notamment en m'aidant à trouver un financement, et en se chargeant de toutes les formalités administratives pour amener ce travail à bon port. Qu'ils trouvent ici l'expression de mon respect pour leurs qualités humaines et scientifiques.

Pour ses réflexions et ses lectures attentives, j'exprime ma plus sincère reconnaissance à Monsieur Mohamed Tahar Bhiri pour son dévouement, sa patience et sa disponibilité, qu'il trouve dans ces quelques mots l'expression de ma profonde gratitude pour m'avoir encadré durant cette thèse.

Reconnaissant envers les membres du jury pour leurs participations à mon jury, je les en remercie. Un très grand merci à M. Jean Pierre Giraudin, Professeur à l'Université Pierre Mendès France et Mme. Hanêne Ben Abdallah, Professeur à la Faculté des Sciences Economiques et de gestion de Sfax d'avoir accepté de rapporter ce travail. Une pensée particulière est adressée à Monsieur le Professur Jean Pierre Giraudin qui a contribué d'une façon ou d'une autre à l'amélioration de ce travail. Enfin, je remercie M. Faiez Gargouri Professeur à l'Institut Supérieur d'Informatique et de Multimédia de Sfax de m'avoir accordé l'honneur d'être le président de mon jury.

Ces remerciements seraient incomplets s'ils ne mentionnaient pas Monsieurs Jean Pierre Jacquot et Francis Alexandre pour l'attention qu'ils ont accordé à mon égard.

Il serait inconcevable de ne pas y associer l'ensemble de l'équipe DEDALE et MIRACL pour leur accueil, leur disponibilité et leurs compétences.

Pour terminer, je souhaite également un prompt et complet rétablissement à Madame le professeur Jeanine Souquières.

Que ce mémoire soit le témoignage de l'amour que je leur voue, de ma reconnaissance infinie pour les sacrifices qu'ils ont consentit jusqu'à ce jour :

A ma cher femme Hajer et à mon fils Ibrahim pour leur amour et leur patience face à un "papa étudiant"

A ma cher femme, pour tous

A mon père, pour son soutien

A ma mère, pour son amour

A mes soeurs, pour leurs encouragements

A tous ceux qui ont contribué à ma formation intellectuelle

A ceux que j'aime et à tous mes amis, comme marque de mon affection et de mon amitié, pour le soutien qu'ils m'ont toujours apporté.

Table des matières

Table des figures 1

Liste des tableaux 5

Introduction générale

1 Qualité du logiciel 9
2 Raffinement 9
3 Refactoring 10
4 Contributions et plan 10
4.1 Contributions 10
4.2 Plan 11
5 Publications 12

Partie I Etat de l'art

Chapitre 1
Raffinement et refactoring de modèles

1.1 Raffinement de modèles 15

1.1.1 Notion de raffinement 15
1.1.2 Méta-modélisation 16
1.1.3 D'UML vers des langages formels 17
1.1.4 Patterns de raffinement 18
1.1.5 Bilan 19
1.1.6 Approche proposée 19
1.2 Refactoring de modèles 21
1.2.1 Notion de refactoring 21
1.2.2 Méta-modélisation 22
1.2.3 D'UML vers des langages formels 22
1.2.4 Règles de refactoring 23
1.2.5 Bilan 24
1.2.6 Approche proposée 24
1.3 Conclusion 25

Chapitre 2
D'UML vers les langages formels

2.1 Aperçu de la méthode B 27
2.1.1 Le langage 27
2.1.2 Les preuves 28
2.1.3 Les outils 28
2.2 Aperçu du langage CSP 29
2.2.1 Les processus 29
2.2.2 Sémantique de CSP 29
2.2.3 Le Raffinement CSP 30
2.2.4 Les outils 30
2.3 Dérivation des aspects structurels d'UML en B 30
2.3.1 Dérivation de classes 31
2.3.2 Dérivation d'attributs 31
2.3.3 Dérivation d'opérations 31
2.3.4 Dérivation des associations 32
2.3.5 Dérivation d'une classe associative 32
2.3.6 Dérivation d'une généralisation 33
2.4 Dérivation de contraintes OCL 34
2.5 Dérivation des aspects comportementaux d'UML en CSP 34
2.5.1 Conventions d'affectation de noms 35
2.5.2 Dérivation d'un état 35

2.5.3 Dérivation d'une sous-machine 35
2.5.4 Dérivation de diagrammes d'états-transitions 35
2.6 Etude de cas : Contrôle d'accès à un bâtiment 36
2.6.1 Objectif de l'étude de cas 36
2.6.2 Présentation générale de l'étude de cas 36
2.6.3 Propriétés de l'étude de cas 36
2.6.4 Diagramme de classes 36
2.6.5 Contraintes OCL et Diagrammes d'états-transitions 37
2.7 Conclusion 43

Partie II Approche de raffinement des spécifications UML

Chapitre 3
Patterns de raffinement

3.1 Introduction 47
3.1.1 Définition d'un pattern de raffinement 47
3.1.2 Canevas de présentation d'un pattern de raffinement 47
3.2 Pattern d'introduction d'une classe intermédiaire : `Class_Helper` 48
3.2.1 Intention 48
3.2.2 Motivation 48
3.2.3 Solution 48
3.2.4 Vérification 50
3.2.5 Exemple 53
3.2.6 Voir aussi 55
3.3 Pattern de réification d'un attribut : `Class_Attribute` 57
3.3.1 Intention 57
3.3.2 Motivation 57
3.3.3 Solution 58
3.3.4 Vérification 58
3.3.5 Exemple 60
3.3.6 Voir aussi 65

3.4 Pattern d'enrichissement d'une association : `Class_Association` 65
3.4.1 Intention 65
3.4.2 Motivation 65
3.4.3 Solution 66
3.4.4 Vérification 66
3.4.5 Exemple 68
3.4.6 Voir aussi 72
3.5 Pattern de décomposition d'une classe : `Class_Decomposition` 72
3.5.1 Intention 72
3.5.2 Motivation 72
3.5.3 Solution 74
3.5.4 Vérification 74
3.5.5 Exemple 75
3.5.6 Voir aussi 77
3.6 Pattern d'introduction d'une nouvelle entité : `Class_NewEntity` 78
3.6.1 Intention 78
3.6.2 Motivation 78
3.6.3 Solution 78
3.6.4 Vérification 79
3.6.5 Exemple 81
3.6.6 Voir aussi 83
3.7 Pattern de raffinement de contrôle d'une classe :
`Refinement_Operation` 85
3.7.1 Intention 85
3.7.2 Motivation 85
3.7.3 Solution 85
3.7.4 Vérification 85
3.7.5 Exemple 86
3.7.6 Voir aussi 87
3.8 Pattern d'abstraction d'une classe : `Class_Abstraction` 88
3.8.1 Intention 88
3.8.2 Motivation 88
3.8.3 Solution 89
3.8.4 Vérification 89
3.8.5 Exemple 92
3.8.6 Voir aussi 96
3.9 Conclusion 96

Chapitre 4
Approche par raffinement des diagrammes de classes UML

4.1 Introduction . . . 97
4.2 Développement conjoint UML/B . . . 97
4.2.1 Réécriture du cahier des charges . . . 97
4.2.2 Stratégie de raffinement . . . 97
4.2.3 Spécification abstraite . . . 98
4.2.4 Raffinement . . . 98
4.3 Etude de cas : Contrôle d'accès aux bâtiments . . . 98
4.3.1 Cahier des charges . . . 98
4.3.2 Réécriture du cahier des Charges . . . 99
4.3.3 Stratégie de raffinement . . . 100
4.3.4 Etapes de raffinement . . . 100
4.4 Conclusion . . . 110

Partie III Approche de refactoring des spécifications UML

Introduction **117**
1 Canevas de présentation d'un schéma de refactoring . . . 118
1.1 Identification des paramètres . . . 118
1.2 Vérification des conditions d'application . . . 119
1.3 Evolution de la spécification . . . 120
1.4 Correction du schéma . . . 120
2 Plan . . . 120

Chapitre 5
Schéma de refactoring : introduction de la notion d'héritage

5.1 Etape 1. Identification des paramètres . . . 121
5.2 Etape 2. Vérification des conditions d'application . . . 121
5.2.1 Définition de la consistance de la relation d'héritage . . . 121
5.2.2 Vérification . . . 122

5.3 Etape 3. Evolution de la spécification . . . 123
5.4 Etape 4. Correction du schéma . . . 123
5.5 Etude d'un exemple . . . 124
5.5.1 Etape 1. Identification des paramètres . . . 124
5.5.2 Etape 2. Vérification des conditions d'application . . . 126
5.5.3 Etape 3. Evolution de la spécification . . . 128
5.5.4 Etape 4. Correction du schéma . . . 128
5.6 Etude de cas . . . 128
5.6.1 Etape 1. Identification des paramètres . . . 128
5.6.2 Etape 2. Vérification des conditions d'application . . . 130
5.6.3 Etape 3. Evolution de la spécification . . . 131
5.6.4 Etape 4. Correction du schéma . . . 131
5.7 Autres schémas de refactoring . . . 133
5.7.1 Schéma de refactoring : introduction de la notion de redéfinition . . . 133
5.7.2 Schéma de refactoring : introduction de la notion de classe abstraite . 134
5.7.3 Schéma de refactoring : introduction de la notion de polymorphisme . 135
5.8 Conclusion . . . 136

Chapitre 6
Schémas de refactoring : introduction des notions d'association, de délégation et de généricité

6.1 Schéma de refactoring : introduction de la notion d'association . . . 137
6.1.1 Etape 1. Identification des paramètres . . . 137
6.1.2 Etape 2. Vérification des conditions d'application . . . 138
6.1.3 Etape 3. Evolution de la spécification . . . 138
6.1.4 Etape 4. Correction du schéma . . . 139
6.1.5 Etude d'un exemple . . . 139
6.2 Schéma de refactoring : introduction de la notion de délégation . . . 144
6.2.1 Etape 1. Identification des paramètres . . . 144
6.2.2 Etape 2. Vérification des conditions d'application . . . 145
6.2.3 Etape 3. Evolution de la spécification . . . 145
6.2.4 Etape 4. Correction du schéma . . . 146
6.2.5 Etude de cas . . . 148
6.3 Schéma de refactoring : introduction de la notion de généricité . . . 148
6.3.1 Etape 1. Identification des paramètres . . . 150
6.3.2 Etape 2. Vérification des conditions d'application . . . 150
6.3.3 Etape 3. Evolution de la spécification . . . 150

6.3.4 Etape 4. Correction du schéma 152
6.3.5 Etude de cas 152
6.4 Conclusion 152

Conclusions et perspectives

157
1 Raffinement 157
1.1 Bilan 157
1.2 Perspectives 157
2 Refactoring 159
2.1 Bilan 159
2.2 Perspectives 159

Bibliographie **161**

Table des figures

2.1 Cycle de développement en B 28
2.2 Dérivation d'une classe 31
2.3 Dérivation d'un attribut 32
2.4 Dérivation d'une opération 32
2.5 Dérivation d'une association 33
2.6 Dérivation d'une classe associative 33
2.7 Dérivation d'une sous-classe 34
2.8 Un premier diagramme de classes 37
2.9 Un état de la modélisation de contrôle d'accès à un bâtiment 38
2.10 `STD_Building` 39
2.11 `STD_Light` 40
2.12 `STD_GreenLight` 40
2.13 `STD_RedLight` 41
2.14 `STD_Card` 41
2.15 `STD_CardReader` 42

3.1 Relation abstraite `reside` 48
3.2 Introduction de la notion de `Chambre` 49
3.3 Payement dans un SGH 49
3.4 Introduction de la notion de `Personne` 50
3.5 Pattern d'introduction d'une classe intermédiaire : `Class_Helper` 50
3.6 Modélisation en **B** de l'état de spécification abstraite 51
3.7 Modélisation en **B** de l'état de spécification concrète 52
3.8 Relation abstraite `embarquement` 53
3.9 Introduction de la notion de `Vol` 54
3.10 Dépendances des machines `Context`, `B_Avion`, `B_Avion_r` et `B_Vol` 54
3.11 Modélisation en **B** de l'état de spécification abstraite 55
3.12 Modélisation en **B** de l'état de spécification concrète 56
3.13 `Helper` reliée à `P1` et `P2` avec la même nature de relation 57
3.14 `Helper` reliée à `P1` et `P2` avec deux relations de nature différente 57
3.15 Pattern de réification d'un attribut : `Class_Attribute` 58
3.16 Modélisation en **B** de l'état de spécification abstraite 58
3.17 Modélisation en **B** de l'état de spécification concrète 59
3.18 Classe `Livre` 60
3.19 Réification de l'attribut `etat` 62
3.20 Dépendances des machines `Context`, `B_Livre`, `B_Livre_r`, `B_Disponible` et `B_Emprunte` 63
3.21 Modélisation en **B** de l'état de spécification abstraite 63

3.22 Modélisation en **B** de l'état de spécification concrète 64
3.23 Classes `Société` et `Personne` 65
3.24 Adjonction de la classe associative `Emploi` 66
3.25 Pattern d'enrichissement d'une association : `Class_Association` 66
3.26 Modélisation en **B** de l'état de spécification abstraite 67
3.27 Modélisation en **B** de l'état de spécification concrète 67
3.28 Un premier diagramme de classes 68
3.29 Adjonction de la classe associative `Enseigne` 69
3.30 Dépendances des machines `Context`, `B_Professeur`, `B_Professeur_r` et `B_Enseigne` 69
3.31 Modélisation en **B** de l'état de spécification abstraite 70
3.32 Modélisation en **B** de l'état de spécification concrète 71
3.33 Classe `Personne` 72
3.34 Décomposition de la classe `Personne` 73
3.35 Classe `Rectangle` 73
3.36 Décomposition de la classe `Rectangle` 73
3.37 Pattern de décomposition d'une classe 74
3.38 Modélisation en **B** de l'état de spécification abstraite 74
3.39 Modélisation en **B** de l'état de spécification concrète 75
3.40 Dépendances des machines `Context`, `B_Rectangle` et `B_Carre` 76
3.41 Modélisation en **B** de l'état de spécification abstraite 76
3.42 Modélisation en **B** de l'état de spécification concrète 77
3.43 Classe `Document` 77
3.44 Décomposition de la classe `Document` 78
3.45 Diagramme de classes simplifié 78
3.46 Introduction de la notion d'`Autorité` 79
3.47 Pattern d'introduction d'une nouvelle entité : `Class_NewEntity` 79
3.48 Modélisation en **B** de l'état de spécification abstraite 80
3.49 Modélisation en **B** de l'état de spécification concrète 80
3.50 Diagramme de classes simplifié 81
3.51 Introduction de la notion de `Autorité` 82
3.52 Dépendances des machines `Context`, `B_Jeton`, `B_Jeton_r` et `B_Autorite` 82
3.53 Modélisation en **B** de l'état de spécification abstraite 83
3.54 Modélisation en **B** de l'état de spécification concrète 84
3.55 Pattern de raffinement de contrôle d'une opération 85
3.56 Classe `Horloge` 86
3.57 Raffinement de contrôle de `tictac` de la classe `Horloge` 86
3.58 Modélisation en **B** de la classe `Horloge` 87
3.59 Modélisation en **B** de la classe `Horloge` après raffinement 88
3.60 Affinage d'un ensemble par une suite 88
3.61 Diagramme de classes non factorisé 89
3.62 Raffinement par abstraction des classes `Enseignant` et `Etudiant` 90
3.63 Pattern de raffinement par abstraction 90
3.64 Modélisation en **B** de l'état de spécification abstraite 91
3.65 Modélisation en **B** de l'état de spécification concrète 92
3.66 Diagramme de classes non factorisé 93
3.67 Raffinement par abstraction des classes `Film` et `Livre` 94
3.68 Modélisation en **B** de l'état de spécification abstraite 94
3.69 Modélisation en **B** de l'état de spécification concrète 95

4.1 Etape de raffinement 98
4.2 Modèle initial 101
4.3 Modélisation en **B** de l'état de spécification abstraite 101
4.4 Premier raffinement 102
4.5 Deuxième raffinement 103
4.6 Modélisation en **B** de l'état de deuxième raffinement 103
4.7 Troisième raffinement 104
4.8 Modélisation en **B** de l'état de troisième raffinement 105
4.9 Quatrième raffinement 106
4.10 Modélisation en **B** de l'état de quatrième raffinement 107
4.11 Cinquième raffinement 108
4.12 Modélisation en **B** de l'état du cinquième raffinement 109
4.13 Sixième raffinement 111
4.14 Septième raffinement 112
4.15 Modélisation en **B** de l'état de septième raffinement 113

I Paramètres avant refactoring 119

5.1 Paramètres après refactoring 124
5.2 Diagramme de classes avant refactoring 124
5.3 Diagramme d'états-transitions de la classe Process 125
5.4 Diagrammes d'états-transitions de la classe PrioProcess 126
5.5 Modèles B 127
5.6 Vérification avec FDR2 129
5.7 Diagramme de classes après refactoring 129
5.8 Modèles B 130
5.9 Diagramme de classes après refactoring 132
5.10 Paramètres avant refactoring 133
5.11 Paramètres après refactoring 133
5.12 Paramètres après refactoring 134
5.13 Paramètres avant refactoring 135
5.14 Paramètres après refactoring 136

6.1 Paramètres avant refactoring 138
6.2 Paramètres après refactoring 139
6.3 Diagramme de classes avant refactoring 140
6.4 Diagrammes d'états-transitions des classes : DBCreate, Parser, DBFill, DBCheck, Analyse, StatCalc et StatFilter 141
6.5 Diagramme d'états-transitions de Statistics 141
6.6 Diagramme d'états-transitions de DB avant refactoring 142
6.7 Diagramme d'états-transitions de Saat avant refactoring 142
6.8 Diagramme de classes après refactoring 143
6.9 Diagramme d'états-transitions de Saat après refactoring 143
6.10 Diagramme d'états-transitions de DB après refactoring 144
6.11 `STD_Class` avant refactoring 146
6.12 `STD_Class` après refactoring 147
6.13 `STD_ComponentOfClass` 147
6.14 Paramètres après refactoring 147

6.15 Diagramme de classes après refactoring . . . 149
6.16 `STD_Building` après refactoring . . . 149
6.17 `STD_Door` . . . 149
6.18 Paramètres avant refactoring . . . 150
6.19 Paramètres après refactoring . . . 151
6.20 Diagramme de classes après refactoring . . . 153

Liste des tableaux

1.1 Tableau de synthèse des travaux liés au raffinement de modèles 20
1.2 Tableau de synthèse des travaux liés au refactoring de modèles 24

2.1 Tableau des outils B 29
2.2 Tableau des outils CSP 30
2.3 Tableau de conventions d'affectation de noms 35

3.1 Tableau de l'état de la machine B_Class_Helper_a 51
3.2 Tableau de l'état de la machine B_Class_Helper_r 52
3.3 Tableau de l'état de la machine `B_Avion` 53
3.4 Tableau de l'état de la machine `B_Vol` 55
3.5 Tableau de l'état de la machine `B_Avion_r` 55
3.6 Tableau de l'état de la machine B_Class_Attribute_a 59
3.7 Tableau de l'état de la machine B_Class_Attribute_r 60
3.8 Tableau de l'état de la machine `B_Livre` 61
3.9 Tableau de l'état de la machine `B_Disponible` 61
3.10 Tableau de l'état de la machine `B_Emprunte` 65
3.11 Tableau de l'état de la machine `B_Livre_r` 65
3.12 Tableau de l'état de la machine B_Class_Association_a 67
3.13 Tableau de l'état de la machine B_Class_Association_r 68
3.14 Tableau de l'état de la machine `B_Professeur` 70
3.15 Tableau de l'état de la machine `B_Enseigne` 71
3.16 Tableau de l'état de la machine `B_Professeur_r` 71
3.17 Tableau de l'état de la machine `B_Class_Decomposition` 74
3.18 Tableau de l'état de la machine `B_Class_Decomposition_r` 75
3.19 Tableau de l'état de la machine `B_Rectangle` 76
3.20 Tableau de l'état de la machine `B_Carre` 77
3.21 Tableau de l'état de la machine B_Class_NewEntity_a 80
3.22 Tableau de l'état de la machine B_Class_NewEntity_r 81
3.23 Tableau de l'état de la machine `B_Jeton` 83
3.24 Tableau de l'état de la machine `B_Autorite` 84
3.25 Tableau de l'état de la machine `B_Jeton_r` 84
3.26 Tableau de l'état de la machine `B_Horloge` 87
3.27 Tableau de l'état de la machine `B_Horloge_r` 87
3.28 Tableau de l'état de la machine `B_Class_Abstraction_a` 90
3.29 Tableau de l'état de la machine `B_Class_Abstraction_r` 93
3.30 Tableau de l'état de la machine `B_Oeuvre` 95

3.31 Tableau de l'état de la machine B_Oeuvre_r 96

4.1 Tableau de synthèse de stratégie de raffinement 100
4.2 Tableau de l'état de la machine ModelInitial 102
4.3 Tableau de l'état de la machine DeuxiemeRaffinement 103
4.4 Tableau de l'état de la machine TroisiemeRaffinement 105
4.5 Tableau de l'état de la machine QuatriemeRaffinement 108
4.6 Tableau de l'état de la machine CinquiemeRaffinement 110
4.7 Tableau de l'état de la machine SeptiemeRaffinement 114

5.1 Tableau de l'état de la machine Process 126
5.2 Tableau de l'état de la machine PrioProcess 126
5.3 Tableau de l'état de la machine Light 130
5.4 Tableau de l'état de la machine GreenLight 131

6.1 Règles de réécriture des contraintes dans le contexte de Class 146
6.2 Règle de réécriture des contraintes dans le contexte de ComponentOfClass 146
6.3 Règles de réécriture de OCL_Class_G[G] et de OCL_Class2 151

I Contribution de raffinement proposée vis-à-vis des approches existantes 158
II Contribution de refactoring proposée vis-à-vis des approches existantes 160

Introduction générale

1 Qualité du logiciel

Le but du génie logiciel est de produire un logiciel de qualité. Mais la qualité du logiciel est une notion complexe et difficile à maîtriser. Pour la cerner d'une façon satisfaisante, il est nécessaire d'analyser plusieurs facteurs de qualité, dont principalement : correction, réutilisabilité, extensibilité et efficacité [71, 72]. La correction est la capacité que possède un produit logiciel de mener à bien sa tâche, telle qu'elle a été définie par sa spécification. La réutilisabilité est la capacité des éléments logiciels à servir à la construction de nouveaux logiciels. L'extensibilité est la facilité d'adaptation des produits logiciels aux changements de spécifications. Enfin, l'efficacité est la capacité d'un système logiciel à utiliser le minimum de ressources matérielles.

Dans l'approche Orientée Objet, ces facteurs de qualité doivent être soignés dès les premières étapes du processus de développement : analyse et conception. Egalement, ils doivent être observés voire vérifiés sur tous les artefacts logiciels (diagrammes de classes dotées des contraintes OCL, diagrammes d'états-transitions,...) et non pas uniquement sur le code.

Dans le cadre UML, plusieurs techniques favorisant l'obtention de logiciels de qualité sont utilisées par les concepteurs telles que : patterns d'analyse [33], patterns de conception notamment ceux de GoF [36], spécifications pré/post exprimées en OCL [77], activités de test liées aux modèles UML [18] et types abstraits de données [59].

Dans cette thèse, nous explorons l'utilisation des techniques de raffinement [37] et de refactoring [8] afin d'aider le concepteur à aboutir à des modèles UML de qualité.

2 Raffinement

D'une façon informelle, un raffinement est un processus permettant de transformer une spécification abstraite en une spécification concrète. Il vise le développement incrémental de systèmes corrects par construction. Le raffinement est défini d'une façon rigoureuse dans divers langages formels tels que B [6], Event-B [24], CSP [49], Z [93] et Object-Z [40]. Le langage UML ne supporte pas le concept du raffinement. Il offre une relation de dépendance stéréotypée «refine» permettant de relier un client (élément raffiné ou concret) à un fournisseur (élément abstrait). Cette relation est sujette à plusieurs interprétations [42] et n'offre pas une assistance méthodologique liée à la manière de raffiner un modèle UML existant. En outre, UML ne permet pas de vérifier si un modèle raffine un autre.

Après avoir classé et étudié les différentes approches visant l'intégration du concept de raffinement dans un cadre semi-formel (UML), nous apportons une approche fondée sur l'utilisation conjointe d'UML et B. En fait, nous proposons des patterns de raffinement décrits selon un canevas précis et formalisé en B en se servant des règles de transformation systématique d'UML/OCL vers B [57, 73]. Ces patterns concernent la construction des diagrammes de classes UML/OCL et permettent de guider le concepteur lors d'un développement incrémental des diagrammes de classes dotées des contraintes OCL. La vérification de la relation de raffinement induite par l'application du pattern de raffinement est assurée par l'Atelier B.

3 Refactoring

Le refactoring est une activité de restructuration permettant d'améliorer la structure interne d'un système en préservant son comportement externe. Une telle activité favorise l'obtention de logiciels de qualité : extensibles (lors de la maintenance évolutive), réutilisables et efficaces.

Plusieurs travaux liés à l'application de la technique de refactoring sur le code existent. Par exemple, [34] propose un catalogue des règles de refactoring applicables sur la partie statique d'un programme Java. Parmi ces règles, nous citons : RenameClass, ExtractClass, MoveOperation, MoveAttribute, RenameOperation. Egalement des outils de refactoring [85] sont disponibles pour la plupart des langages orientés objets comme Java, Smalltalk, C++, C#, Delphi et Eiffel et les environnements de développement intégrés comme Eclipse, NetBeans, Oracle JDeveloper. Mais ces règles de refactoring applicables sur le code sont définies d'une façon informelle.

Récemment, plusieurs chercheurs travaillent sur l'application de la technique de refactoring sur les modèles et notamment sur ceux d'UML [67].

Dans le carde de cette thèse, nous apportons une nouvelle approche de refactring basée sur l'utilisation combinée d'UML, de B et de CSP. Les modèles UML sont décrits par des diagrammes de classes, des contraintes OCL et des diagrammes d'états-transitions. Plus précisément, nous proposons des schémas de refactoring décrits selon un canevas précis et formalisé en B et CSP. Ces schémas de refactoring couvrent les notions fondamentales de l'approche par objets : relations conceptuelles entre classes (association et généralisation/spécialisation), polymorphisme, redéfinition, classe abstraite, délégation et généricité. La préservation du comportement après avoir appliqué le refactoring est confiée aux outils de vérification formelle associés à B (le prouveur de l'***Atelier B*** [26]) et à CSP (le model-checker ***FDR2*** [38]).

4 Contributions et plan

4.1 Contributions

Comme nous venons de l'indiquer, cette thèse préconise l'utilisation de deux techniques de raffinement et de refactoring afin d'établir des modèles UML de qualité c'est-à-dire corrects par construction, extensibles, réutilisables et efficaces. En outre, elle plaide en faveur de l'utilisation conjointe de la méthode semi-formelle UML et des méthodes formelles comme B et CSP.

Les principales contributions de cette thèse sont :

- Proposition de sept patterns de raffinement de diagrammes de classes UML/OCL afin de guider le concepteur lors de la modélisation statique de son application :
  1. Introduction d'une classe intermédiaire : `Class_Helper`.
  2. Réification d'un attribut : `Class_Attribute`.
  3. Enrichissement d'une association : `Class_Association`.
  4. Décomposition d'une classe : `Class_Decomposition`.
  5. Introduction d'une nouvelle entité : `Class_NewEntity`.
  6. Raffinement de contrôle d'une classe : `Refinement_Operation`.
  7. Abstraction d'une classe : `Class_Abstraction`.
- Proposition d'une approche par raffinement des diagrammes de classes UML comportant quatre phases : Réécriture du cahier des charges, Stratégie de raffinement, Spécification abstraite et Raffinement.

- Proposition de sept schémas de refactoring des modèles UML décrits par des diagrammes de classes, des contraintes OCL et des diagrammes d'états-transitions afin d'aider le concepteur lors de la restructuration des modèles UML :
  1. Introduction de la notion d'héritage.
  2. Introduction de la notion de redéfinition.
  3. Introduction de la notion de classe abstraite.
  4. Introduction de la notion de polymorphisme.
  5. Introduction de la notion d'association.
  6. Introduction de la notion de délégation.
  7. Introduction de la notion de généricité.

4.2 Plan

Cette thèse est organisée en trois parties.

1. Première partie est consacrée à l'état de l'art. Elle comporte deux chapitres.
   - Le chapitre 1 présente la notion de raffinement, les différentes approches favorisant l'intégration de cette notion dans UML et notre façon d'aborder le problème d'intégration du concept de raffinement en UML. Il présente aussi le concept de refactoring, les différentes approches liées au refactoring des modèles UML et notre façon d'aborder le problème d'intégration du concept de refactoring en UML.
   - Le chapitre 2 présente les dérivations d'UML vers des langages formels (B et CSP) utilisées dans la suite de cette thèse. Également, il présente l'étude de cas contrôle d'accès à un ensemble de bâtiments.
2. Deuxième partie propose, dans deux chapitres, une approche de raffinement des spécifications UML. Elle débute par une introduction comportant notre description générale d'un pattern de raffinement.
   - Le chapitre 3 propose les définitions des patterns de raffinement proposés, à savoir :
     - Introduction d'une classe intermédiaire : `Class_Helper`.
     - Réification d'un attribut : `Class_Attribute`.
     - Enrichissement d'une association : `Class_Association`.
     - Décomposition d'une classe : `Class_Decomposition`.
     - Introduction d'une nouvelle entité : `Class_NewEntity`.
     - Raffinement de contrôle d'une classe : `Refinement_Operation`.
     - Abstraction d'une classe : `Class_Abstraction`.
   - Le chapitre 4 apporte une approche par raffinement des diagrammes de classes UML. Il comporte deux sections :
     - La première section préconise une démarche de développement des diagrammes de classes UML guidée par les patterns de raffinement proposés.
     - La deuxième applique la démarche préconisée sur une étude de cas : Contrôle d'accès aux bâtiments.
3. La troisième partie détaille, dans deux chapitres, notre approche de refactoring des spécifications UML. Une introduction comportant notre description générale d'un schéma de refactoring est fournie au début de cette partie.
   - Le chapitre 5 décrit le schéma de refactoring : "introduction de la notion d'héritage", ainsi que trois schémas de refactoring sous-jacents à la notion d'héritage. Ils sont : "introduction de la notion de redéfinition", "introduction de la notion de classe abstraite" et "introduction de la notion de polymorphisme".

– Le chapitre 6 décrit les schémas de refactoring : "introduction de la notion d'association", "introduction de la notion de délégation" et "introduction de la notion de généricité".

La thèse comporte aussi une conclusion synthétisant nos contributions de recherche et esquissant les perspectives envisagées pour notre travail.

5 Publications

– B. Ben Ammar, M. T. Bhiri, and J. Souquières. Control access case study : an incremental development of UML specifications. Technical report, LORIA, 2007.
– B. Ben Ammar, M. T. Bhiri, and J. Souquières. Quelques patrons de raffinement pour le développement de diagrammes de classes UML. In 6ème atelier sur les Objets, Composants et Modèles dans l'ingénierie des Systèmes d'Information, OCM-SI, couplé avec le 15ème congrès INFORSID, Perros-Guirec France, 2007.
– B. Ben Ammar, M. T. Bhiri, and J. Souquières. Schéma de refactoring de diagrammes de classes basé sur la notion de délégation. In 7ème atelier sur l'Evolution, Réutilisation et Traçabilité des Systèmes d'Information, ERTSI, couplé avec le XXVIème congrès INFORSID, Fontainebleau France, 2008.
– B. Ben Ammar, M. T. Bhiri, and J. Souquières. Modélisation événementielle pour la construction de diagrammes de classes. RSTI - ISI, 13 :131-155, 2008.
– B. Ben Ammar, M. T. Bhiri, and J. Souquières. Incremental development of uml specifications using operation refinements. Innovations in Systems and Software Engineering, 4 :259-266, 2008.

Première partie

Etat de l'art

Chapitre 1

Raffinement et refactoring de modèles

Dans ce chapitre, nous présentons les deux concepts de raffinement et de refactoring de modèles, les travaux réalisés autour de ces deux concepts dans le cadre d'UML et enfin une idée générale sur notre façon d'aborder les problèmes d'intégration de ces deux concepts en UML.

1.1 Raffinement de modèles

Après présentation de la notion de raffinement, nous étudions les différentes approches favorisant l'intégration de cette notion dans un cadre semi-formel celui d'UML.

Une classification de ces différentes approches est proposée. Enfin, nous esquissons la manière avec laquelle nous abordons le problème d'intégration du concept de raffinement en UML.

1.1.1 Notion de raffinement

D'une façon informelle, le raffinement [37, 103] est un processus de transformation d'une spécification abstraite en une spécification concrète tout en vérifiant à chaque étape que la spécification obtenue est correcte vis-à-vis de la spécification précédente.

J. R. Abrial définit l'essence de raffinement comme suit *"acquiring a refined model instead of an abstraction must not perceptible by the buyer"* [3].

En informatique, on distingue deux types de raffinement [3] : raffinement horizontal et raffinement vertical. Dans un processus de développement formel de logiciels, le raffinement horizontal précède le raffinement vertical. Le raffinement horizontal consiste à établir pas-à-pas une spécification cohérente du futur logiciel ou système. Il démarre à partir d'un modèle très abstrait issu du cahier des charges du logiciel. Raffinement par raffinement, il introduit des détails jusqu'à la prise en compte de tous les comportements et contraintes issus du cahier des charges. Le modèle ultime engendré par la phase de raffinement horizontal constitue une spécification du futur logiciel ou système à développer.

Ensuite, on passe la main au raffinement vertical qui consiste à réaliser la spécification issue de la phase de raffinement horizontal en prenant d'une façon graduelle des décisions de conception et/ou d'implémentation. En général, le modèle ultime issu de la phase de raffinement vertical est transformé en code grâce au générateur de code faisant partie intégrante des plate-formes formelles comme B [6]. Dans les méthodes formelles comme B et Event-B [3], la correction de chaque étape de raffinement est vérifiée par la démonstration des théorèmes appelés encore Obligations de Preuves (OP). Ces OP sont produites par un composant appelé générateur

des obligations de preuves. Et elles sont déchargées (ou démontrées) grâce au prouveur interactif.

Le raffinement constitue la pierre angulaire des méthodes formelles comme B [6], Event-B [24], CSP [49], Z [93] et Object-Z [40]. Mais, il existe selon les méthodes utilisées de nombreuses notions de raffinement qui ne sont pas toujours équivalentes. En effet, bien que le raffinement ait pour objectif la préservation de la correction, il s'exprime et se vérifie de différentes manières. Par exemple, en B et Event-B, la correction du raffinement se vérifie grâce à la technique de preuve formelle : ***Atelier B*** [26] et plate-forme ***RODIN*** [2]. Par contre en CSP, elle se vérifie grâce à la technique de model-checking : l'outil ***FDR2*** [87, 88, 89]. En outre, la notion de raffinement en B diffère de celle d'Event-B. Event-B supporte la possibilité d'agir sur la signature du raffinement en rajoutant des événements. Dans la suite de ce chapitre, nous étudions les différentes approches favorisant l'intégration de la notion de raffinement dans un cadre semi-formel celui d'UML.

1.1.2 Méta-modélisation

D'après [66], le raffinement peut être considéré comme une opération de transformation de modèles selon l'approche IDM. En effet, la réalisation d'un raffinement consiste à prendre un modèle et à le modifier. Ceci correspond en fait à une transformation de modèle. Le raffinement implique une opération de transformation de modèles *endogènes* : celui d'avant le raffinement (le modèle source) et celui d'après (le modèle cible) sont du même type, c'est à dire conformes au même méta-modèle. En outre, le raffinement est une opération de transformation plutôt *verticale*, c'est à dire entraînant souvent un changement de niveau d'abstraction (par exemple d'un modèle d'analyse vers un modèle de conception).

Le travail décrit dans [31] propose une méthode basée sur une approche IDM à caractère général permettant entre autres de spécifier des opérations de raffinement en utilisant des contrats de transformation, exprimés en OCL, attachés au niveau méta (M2). Un contrat de transformation est défini par trois ensembles de contraintes [31] :

Contraintes sur le modèle source : contraintes à respecter par un modèle pour pouvoir être transformé (ou raffiné).

Contraintes sur le modèle cible : contraintes générales –indépendamment du modèle source– à respecter par un modèle pour qu'il soit le résultat valide de la transformation.

Contraintes d'évolution d'éléments : contraintes à respecter sur l'évolution de certains éléments entre le modèle source et le modèle cible, pour que le modèle cible soit le résultat valide de la transformation par rapport au modèle source.

Nous apprécions le caractère global de cette méthode de spécification et de validation de raffinement de modèles. En effet, cette méthode considère un couple de modèles l'un étant la source et l'autre la cible d'une transformation ou d'un raffinement et vérifie que ce couple respecte bien le contrat de la transformation. Le modèle cible généré peut être modifié à la main par le concepteur.

Mais la vérification des contrats de transformation de modèles (la correction du raffinement : cas des raffinements de modèles) pose des problèmes liés notamment à la vérification des contraintes d'évolution entre les modèles source et cible. Ceci nécessite l'écriture de fonctions utilitaires souvent complexes en OCL en utilisant la construction `def` permettant l'implantation des correspondances entre les éléments des modèles source et cible. Ces correspondances peuvent être de trois natures : correspondance totale, correspondance partielle et pas de correspondance.

Le travail décrit dans [19, 21] est intéressant. Il propose une taxonomie comportant 47 changements atomiques applicables sur les diagrammes de classes UML tels que : adjonction d'une

classe, suppression d'une association, déplacement d'un attribut entre deux classes, adjonction d'un attribut, adjonction d'une interface, adjonction d'une opération. En se basant sur ces 47 changements atomiques, les auteurs proposent 31 raffinements atomiques applicables sur les diagrammes de classes tels que : introduction d'une sous-classe (TopDownGen), introduction d'une nouvelle classe en utilisant la composition (TopDownCom), transformation d'un attribut en une classe (TurnAttriIntoClass). Par exemple, le raffinement TopDownGen [19] est dérivé (ou détecté) suite aux deux changements atomiques AddedClass et AddedGeneralization. Le concepteur peut intervenir afin de guider l'outil VIATool (Vertical Impact Analysis Tool) [21] lors de la détection des raffinements. La composition des raffinements est également possible. La traçabilité entre les éléments des modèles source et cible est supportée par l'outil VIATool. Ce dernier est centré autour d'une approche de type IDM. L'outil VIATool intègre un méta-modèle comportant deux fragments liés. L'un formalise les notions de changement atomique, raffinement atomique, raffinement composé et traçabilité en utilisant OCL. L'autre concerne le méta-modèle UML2.0. Mais, la non prise en compte des aspects sémantiques, tels que l'invariant de classe, la spécification pré/post associée aux opérations, risque de poser des problèmes de cohérence. En effet, des changements atomiques comme « AddedClass » et « AddedOperation » peuvent entraîner des incohérences. Par exemple, l'invariant de classe ne peut pas être déduit de la post-condition de l'opération ajoutée à cette classe.

Le langage UML offre une dépendance stéréotypée « refine » permettant de spécifier graphiquement une relation de raffinement entre deux modèles UML décrits par deux packages. Mais cette possibilité n'est pas suffisante pour décrire les liens de traçabilité entre des éléments UML appartenant aux packages (par exemple entre deux classes) et des raffinements composés. Pour faire face à cette situation, le travail décrit dans [28] propose une extension d'UML sous forme d'un profil permettant de mieux spécifier les raffinements UML. Pour y parvenir, il propose des stéréotypes tels que : « SimpleRefinement », « CompoundRefinement », « RefinementComposition » et « RefinedElement ». Ces stéréotypes sont équipés des contraintes OCL. Mais ce profil vise uniquement l'enrichissement syntaxique d'UML vis-à-vis de la description des raffinements des modèles UML.

Le travail [101] propose des règles de raffinement permettant de raffiner des diagrammes de classes UML. Ces règles concernent le concept relation d'UML : association, composition et généralisation/spécialisation. L'idée générale des règles de raffinement proposées consiste à transformer une relation UML vers plusieurs relations (une-vers-plusieurs) en introduisant des classes dites intermédiaires. Les auteurs de ce travail se limitent à la transformation une-vers-deux. Afin de vérifier la correction du raffinement entre deux modèles UML successifs (niveau abstrait et niveau concret) exprimés par deux diagrammes de classes, les auteurs de ce travail intègrent leurs règles de raffinement dans le méta-modèle UML sous forme des stéréotypes : « Refined_Gen », « Refining_Assoc », etc. Les contraintes relatives à ces stéréotypes sont formalisées en OCL. Ainsi, la vérification de la relation de raffinement revient à voir si M1 (diagramme de classes UML) est conforme à son M2 (méta-modèle UML étendu avec les règles de raffinement).

1.1.3 D'UML vers des langages formels

L'idée générale est d'ouvrir UML sur des langages formels dotés des outils de vérification.

Le travail décrit dans [29] propose une approche permettant de traduire des modèles orientés objets décrits en UML (les diagrammes de classes, les diagrammes d'objets, les diagrammes d'états-transitions et les diagrammes de séquence) vers des processus CSP. Dans un premier temps, les auteurs de ce travail définissent la sémantique de chaque type de diagramme UML utilisé en CSP. Ensuite, ils combinent ces processus obtenus en utilisant l'opérateur de composi-

tion parallèle de CSP. Plusieurs vérifications peuvent être effectuées sur les modèles CSP obtenus telles que : raffinement de deux modèles UML convertis en deux modèles CSP, cohérence inter-diagrammes UML en utilisant des outils de vérification formelle associés à CSP comme FDR.

Le travail décrit dans [94] propose une approche permettant de formaliser une opération de raffinement de classes très utile : introduction d'une sous-classe. Le comportement d'une classe est défini comme la combinaison de son *protocol state machine* et de tous les diagrammes de séquence où des instances appartenant à cette classe évoluent. La relation de raffinement entre la classe ascendante et la classe descendante est spécifiée en termes de diagrammes de séquence et des protocol state machines. La vérification de cette relation de raffinement nécessite la traduction d'UML vers un formalisme logique basé sur la logique du premier ordre appelé DLs (Description Logics) [11]. Un prototype sous forme de plug-in a été développé par les auteurs de ce travail afin de valider l'approche proposée.

Le travail décrit dans [104] propose une approche basée sur la théorie des graphes permettant de formaliser les deux diagrammes de classes et d'objets. Un diagramme de classes est formalisé par un graphe orienté valué. Les sommets modélisent soit des classes, soit des types primitifs, et les arcs étiquetés par des symboles représentent une relation d'héritage simple, une association ou le nom d'un attribut. Le diagramme d'objets représente l'espace d'états de l'application. Il est formalisé par un graphe orienté valué ayant un sommet jouant le rôle d'une racine (objet racine). Les auteurs de ce travail proposent une définition formelle de la notion de raffinement structurel en termes de transformations de graphes [90]. Pour y parvenir, ils définissent un ensemble de règles permettant d'étendre un diagramme de classes par décomposition et adjonction des classes.

Les design patterns sont traditionnellement décrits de manière informelle. Ian Bayley [12] a formalisé les patterns de GoF en utilisant la logique des prédicats. Son approche consiste à déployer la logique des prédicats afin de préciser les conditions sur les structures des 23 patterns. Ceci lui permet de tirer des conditions sur l'utilisation de ces patterns ainsi de reconnaître les modèles de conception dans le code existant et le code refactorisé.

UML offre une relation de raffinement stéréotypée «refine» (une sorte de dépendance) permettant de relier un client (élément raffiné) à un fournisseur (élément abstrait). Mais la relation «refine» d'UML est sujette à plusieurs interprétations et ne peut pas guider le concepteur dans un développement incrémental. En outre, UML ne permet pas de vérifier si un modèle raffine un autre. Le travail décrit dans [44] apporte une contribution à la formalisation de la notion de relation de raffinement en UML. Les auteurs de ce travail retiennent le concept de collaboration afin de décrire le système à modéliser. Ceci permet de modéliser aussi bien les aspects statiques que dynamiques de l'application. Une collaboration est un ensemble de diagrammes de communication. Sachant qu'un diagramme de communication est une instance du diagramme de collaboration où les liens (instances d'association) sont étiquetés par un ensemble de messages partiellement ordonnés. Egalement, le travail présenté dans [44] définit des règles liées au raffinement structurel (diagramme de collaboration) et des règles liées au raffinement comportemental (diagramme de communication). Ces règles sont formalisées en utilisant la théorie des ensembles. En outre, les auteurs de ce travail introduisent le concept de sous-collaboration permettant le raffinement systématique d'un diagramme de collaboration. Enfin, un prototype sous forme d'un plug-in permettant de mettre en oeuvre cette approche est également fourni.

1.1.4 Patterns de raffinement

L'idée générale consiste à proposer des solutions génériques c'est-à-dire paramétrées permettant de résoudre des problèmes de raffinement plus ou moins récurrents liés à la modélisation OO en UML.

Le travail décrit dans [23] propose une approche permettant de découvrir des raffinements cachés en partant du niveau concret. De tels raffinements cachés sont induits par l'utilisation des concepts UML comme spécialisation (héritage), composition, adjonction d'un attribut ou adjonction d'une opération. Egalement, des raffinements cachés liés à l'utilisation des concepts relatifs aux diagrammes de cas d'utilisation sont traités. La découverte des raffinements cachés favorise l'obtention d'une documentation entre deux modèles UML successifs en utilisant la dépendance stéréotypée «refine» d'UML explicitant les correspondances entre les différents éléments UML grâce au méta-attribut appelé *mapping*.

Les règles de raffinement décrites dans [101] et présentées dans 1.1.2 peuvent être assimilées à des patterns de raffinement liés au concept relation d'UML.

Egalement, les raffinements décrits dans [19, 21] et présentés dans 1.1.2 peuvent être considérés comme des patterns de raffinement de diagrammes de classes.

Le travail décrit dans [80] plaide en faveur de l'utilisation des structures de raffinement venant des méthodes formelles dans le cadre d'UML. Deux patterns de raffinement sont proposés : décomposition d'une entité composite et décomposition d'une opération non atomique (ou encore élaborée). Pour chaque pattern, l'auteur de ce travail donne une instanciation décrite en Object-Z. Ensuite, la même instanciation est traduite en UML/OCL. Des insuffisances liées à OCL concernant le mapping entre les deux modèles (abstrait et concret) sont soulignées et des solutions basées sur la construction `def` (fonctions utilitaires) sont proposées.

Le travail décrit dans [53] propose des patterns de raffinement favorisant une implémentation efficace des diagrammes de classes tels que : transformation d'une association par un attribut, remplacement d'une association par des clés étrangères, raffinement d'une opération par affaiblissement de sa pré-condition et/ou renforcement de sa post-condition, élimination d'une classe associative.

1.1.5 Bilan

Le Tableau 1.1 récapitule les points forts et faibles des différentes approches étudiées précédemment favorisant l'intégration de la notion de raffinement dans un cadre semi-firmel celui d'UML.

Les approches étudiées apportent peu voire pas d'assistance méthodologique au spécifieur afin de suivre un processus de développement basé sur le concept raffinement. En outre, aucune approche étudiée ne couvre toutes les constructions relatives à la modélisation de la partie statique d'une application UML à savoir classe, relation et contraintes OCL. Enfin, les techniques de vérification de la correction d'une étape de raffinement apportées par les approches étudiées posent des problèmes : écriture des fonctions utilitaires OCL, attachement des contrats OCL au niveau méta et évaluation des contraintes OCL.

1.1.6 Approche proposée

Dans la deuxième partie de cette thèse et précisément dans le chapitre 3, nous proposons des patterns de raffinement permettant le développement incrémental des diagrammes de classes UML. Ces patterns sont construits pour résoudre des problèmes récurrents lors de l'élaboration de la partie statique d'une application OO tels que : introduction d'une classe intermédiaire, réification d'un attribut, enrichissement d'une association, décomposition d'un agrégat et introduction d'une nouvelle entité. Ces patterns sont présentés selon un canevas précis comportant six rubriques exhibant les aspects fondamentaux d'un pattern de raffinement. Ces rubriques sont Intention, Motivation, Solution, Vérification, Exemple et Voir aussi. En outre, les patterns de

Approche de	Points forts	Points faibles
N. Belloir et *al.* [31]	– une définition de contrats de transformation de modèles – la prise en compte des contraintes OCL – la vérification est effectuée au niveau méta (M2)	– la vérification nécessite l'écriture de fonctions utilitaires souvent complexes en OCL en utilisant la construction `def`
L. C. Briand et *al.* [21, 19, 20]	– 47 changements atomiques dont 31 raffinements atomiques applicables sur les diagrammes de classes – la composition des raffinements – la détection des raffinements via l'outil VIATool	– la non prise en compte des aspects sémantiques – des problèmes de cohérence
N. Correa et *al.* [28]	– une extension d'UML sous forme d'un profil – la prise en compte des contraintes OCL	– la description des raffinements des modèles UML concerne uniquement l'enrichissement syntaxique d'UML
W. L. Low et *al.* [101]	– des règles de raffinement des diagrammes de classes – la vérification est effectuée au niveau méta (M2) – la prise en compte des contraintes OCL	– les règles de raffinement concernent uniquement le concept relation d'UML : association, composition et généralisation/spécialisation
R. Van Der Streaten et *al.* [94]	– la formalisation d'une opération de raffinement de classes – la prise en compte du comportement d'une classe – la vérification via la traduction d'UML vers un formalisme logique basé sur la logique du premier ordre – un prototype sous forme d'un plug-in	– une unique opération de raffinement : introduction d'une sous-classe
L. Zhao et *al.* [104]	– un ensemble de règles permettant d'étendre un diagramme de classes par décomposition et adjonction des classes – une définition formelle de la notion de raffinement structurel en termes de transformations de graphes	– la non prise en compte des contraintes OCL – des règles de raffinement qui se limitent à la décomposition et l'adjonction des classes
B. Hnatkowska et *al.* [44]	– des règles liées au raffinement structurel – des règles liées au raffinement comportemental – l'utilisation de la théorie des ensembles – un prototype sous forme d'un plug-in	– des règles de raffinement qui se limitent aux diagrammes de collaboration et de communication
C. Pons et *al.* [23, 83, 80, 81, 82]	– 5 patterns de raffinement – l'utilisation des structures de raffinement venant des méthodes formelles (Object-Z) – la prise en compte des contraintes OCL – la découverte des raffinements cachés – la documentation entre deux modèles UML successifs en utilisant la dépendance stéréotypée "refine"	– des insuffisances liées à OCL concernant le mapping entre les deux modèles (abstrait et concret)

TABLE 1.1 – Tableau de synthèse des travaux liés au raffinement de modèles

raffinement proposés sont formalisés en **B** en se servant des règles systématiques de transformation d'UML vers B [73, 57]. Ceci permet d'identifier avec précision les conditions d'application, l'évolution d'un diagramme de classes UML et la correction de la relation de raffinement. Une telle formalisation en **B** peut être utilisée avec profit lors de l'instanciation de ces patterns par le concepteur. Ainsi, dans un développement conjoint UML-B, le concepteur choisit et applique un pattern de raffinement sur sa spécification abstraite. Il obtient alors une spécification concrète comportant entre autres des propriétés liées au pattern de raffinement appliqué. La vérification de la correction de la relation de raffinement entre deux spécifications est confiée à l'***Atelier B***.

Enfin, dans le chapitre 4, nous proposons une démarche de développement incrémental avec preuves des diagrammes de classes UML – en utilisant conjointement UML/OCL et B – guidée par les patterns de raffinement. Notre démarche comporte quatre phases : Réécriture du cahier des charges, Stratégie de raffinement, Spécification abstraite et Raffinement. Afin de montrer la faisabilité de la démarche proposée, nous allons l'appliquer sur l'étude de cas : Contrôle d'accès aux bâtiments [1, 13, 7].

1.2 Refactoring de modèles

Nous mettons l'accent sur la notion de refactoring dans un premier temps puis nous étudions les différentes approches favorisant l'intégration de cette notion dans le cadre semi-formel : UML. Enfin, nous présentons notre façon d'aborder le problème d'intégration du concept refactoring en UML.

1.2.1 Notion de refactoring

L'activité de refactoring ou de restructuration est bien connue en génie logiciel. Elle vise à améliorer certains facteurs de qualité tels que l'extensibilité, la réutilisabilité et l'efficacité de l'application. Cette activité a tout d'abord porté sur la restructuration de code par modification de sa structure interne sans changement de son comportement externe [79]. D'après Martin Fowler *"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure"* [35]. Ainsi, l'activité de refactoring est une technique qui consiste à retravailler le code. Elle n'ajoute pas de fonctionnalités supplémentaires mais elle vise à améliorer les facteurs de qualité. Elle ne doit pas changer le comportement externe du code source.

Dans [79], Opdyke a proposé la première définition de la préservation du comportement. Il stipule que, pour les mêmes valeurs d'entrée, l'ensemble des valeurs de sortie résultant doit être le même avant et après refactoring. Toutefois, cette définition ne prend pas en compte tous les aspects comportementaux, tels que les contraintes de temps, de mémoire. En outre, la mise en oeuvre de cette définition exige souvent une activité de test. Ceci apporte plutôt une validation partielle.

Le refactoring est largement appliqué sur le code. Plusieurs catalogues de règles de refactoring liés aux langages tels que C++, Java, Smalltalk, Eiffel sont proposés [85]. Par exemple, Fowler [35] propose des règles de refactoring élémentaires liées à la partie statique des programmes écrits en Java. Parmi ces règles, nous citons : «RenameClass, RenameAttribute, RenameOperation, MoveAttribute, MoveOperation, ExtractClass». Mais ces règles de refactoring sont définies d'une façon informelle.

Dans l'ingénierie logicielle dirigée par les modèles, les techniques de refactoring sont peu nombreuses [8]. Selon Mens [39, 65, 68, 69], l'un des défis consiste à prendre en compte tout le processus de refactoring de modèles qui se divise en six activités :

1. identifier quelles parties du modèle devraient être refactorisées,
2. déterminer quelles règles de refactoring devraient être appliquées à ces endroits,
3. garantir qu'une fois appliqué, le refactoring des modèles préserve le comportement et la cohérence,
4. automatiser l'application du refactoring,
5. evaluer l'impact du refactoring sur des critères de qualité logicielle (complexité, lisibilité, adaptabilité) ou du processus (productivité, coût, effort),
6. synchroniser le modèle refactorisé et les autres artefacts tels que le code source, la documentation, les spécifications, les tests.

Dans la suite, nous nous limitons au refactoring des modèles UML. Nous passons en revue les travaux associées selon la même classification adoptée dans le chapitre précédent concernant le raffinement des modèles UML. Notons au passage que [75] propose une classification des approches de restructuration des modèles dans une optique d'IDM.

1.2.2 Méta-modélisation

Le refactoring peut être considéré comme une opération de transformation de modèles selon l'approche IDM [66]. En effet, le refactoring consiste à réorganiser (sans ajouter des détails) la structure d'un modèle. Contrairement au raffinement, qui est considéré comme une opération de transformation verticale, le refactoring est considéré comme une opération de transformation horizontale.

Ainsi, une opération de refactoring n'entraîne pas un changement de niveau d'abstraction : le modèle source (avant le refactoring) et le modèle cible (après le refactoring) demeurent sur le même niveau d'abstraction.

Le travail décrit dans [19, 21] et présenté dans 1.1.2 ne distingue pas entre les opérations de raffinement et refactoring. En effet, des opérations comme «ExtractClass», «ExtractSubClass», «CollapseHierarchy» et «InlineClass» sont plutôt des opérations de refactoring. Par exemple, l'opération «CollapseHierarchy» permettant de fusionner une super-classe et une sous-classe afin d'éviter des redondances inutiles est une opération de refactoring car elle n'introduit pas des nouveaux détails.

Le travail décrit dans [62] propose un catalogue d'opérations de refactoring inspiré du catalogue de Fowler [35]. Les opérations proposés sont applicables sur des diagrammes de classes. Elles sont exprimées par une transformation de modèle formalisée en QVT [78]. L'impact d'une opération de refactoring sur les contraintes OCL et les diagrammes d'objets est également traité.

Le travail décrit dans [39] propose une extension du méta-modèle d'UML. Cette extension permet une meilleure spécification des pré/postcontions de deux opérateurs de refactoring : «Pull Up Method» et «Extract Method». De plus, cette extension donne la possibilité à des outils de : vérifier les pré/post-conditions, composer des séquences d'opérations de refactoring et utiliser le moteur de requête OCL pour détecter les "design smells" (défauts de conception).

1.2.3 D'UML vers des langages formels

L'idée générale est d'ouvrir UML sur des langages formels dotés d'outils de vérification.

Le travail décrit dans [99] propose une approche permettant de traduire des diagrammes d'états-transitions UML vers des processus CSP. Ainsi, après transformation de modèle, les vérifications de la préservation du comportement sont effectuées sur les processus CSP obtenus par traduction des diagrammes d'états-transitions avant et après refactoring. En effet, le processus

CSP après refactoring doit être un raffinement par échecs-divergences($\sqsubseteq_{FD}$ voir 2.2.3) de son homologue avant refactoring.

Le travail décrit dans [29] et présenté dans 1.1.3 définit formellement le refactoring sur deux modèles CSP issus de deux modèles UML par double raffinement par échecs-divergences :

$$\text{ModelAvantRefactoring} \sqsubseteq_{FD} \text{ModelApresRefactoring}$$
$$\text{et}$$
$$\text{ModelApresRefactoring} \sqsubseteq_{FD} \text{ModelAvantRefactoring}$$

Le travail décrit dans [94] et présenté dans 1.1.3 utilise la même approche pour formaliser la préservation du comportement entre une classe et sa nouvelle version. Sachant que le comportement d'une classe est défini comme la combinaison de son *protocol state machine* et de tous les diagrammes de séquence où des instances appartenant à cette classe évoluent.

Les travaux effectués par Tom Mens et al. [39, 65, 68, 69] correspondent à l'étude la plus complète. Mens propose d'utiliser la théorie de transformation de graphes pour la spécification des opérations de refactoring de modèles afin de prouver la consistance et la préservation du comportement. De plus, les auteurs de ce travail proposent une technique d'analyse permettant de détecter les dépendances implicites entre les différentes opérations de refactoring. Ceci permet aux concepteurs d'avoir une information sur le refactoring le mieux adapté dans un contexte donné.

En utilisant l'outil de transformation de graphes Fujaba, Mens [65] explique comment un plug-in de refactoring peut être développé pour transformer des diagrammes de classes, si chaque refactoring est exprimé par des règles de transformation de graphes dans la bibliothèque de l'outil.

1.2.4 Règles de refactoring

L'idée générale consiste à proposer des règles de refactoring élémentaires ou atomiques. Elles peuvent être assimilées à des règles de réécriture. De telles règles peuvent assurer des restructurations de base cohérentes. Ainsi, l'erreur potentielle induite par l'activité de refactoring est largement réduite.

Marković et Baar [63, 64] proposent quelques règles de refactoring de base pour des diagrammes de classes en tenant compte des contraintes OCL [77], inspirées des règles de refactoring proposées dans les langages orientés objet [69, 85]. Les auteurs de ce travail définissent le refactoring de modèle comme un ensemble de règles de transformation. Ainsi, ils ont proposé un catalogue de sept règles de refactoring avec ou sans influence sur la syntaxe des contraintes OCL attachées aux diagrammes de classes refactorisés. Afin de vérifier la préservation du comportement, les auteurs de ce travail utilisent un formalisme basé sur les grammaires de graphes.

Le travail décrit dans [96] propose deux catalogues de règles de refactoring. Le premier applicable sur le diagramme de classes comporte cinq opérations de base : « addition », « removal », « move », « generalization » et « specialization » d'un élément. Sachant qu'un élément peut être une classe, un attribut, une opération ou une extrémité d'association. Le second catalogue, applicable sur le diagramme d'états-transitions, comporte sept opérations de base : « Unfold Exit Action », « Group States », « Fold Outgoing Transition », « Unfold Outgoing Transition », « Move State into Composite », « Move State out of Composite » et « Same Label ». En outre, la sémantique de ces opérations est définie en OCL.

Le travail décrit dans [27] préconise l'utilisation de la technique de refactoring afin d'améliorer la compréhension et la maintenance des spécifications OCL. Les auteurs de ce travail identifient les mauvaises utilisations d'OCL (OCL smells) et proposent une collection d'opérations de refactoring permettant d'écarter ces OCL smells. Parmi les OCL smells identifiés (une douzaine) par

ces auteurs, nous citons : «Implies chain», «Redundancy», «Non-atomic rule», «And chain», «ForAll chain» et «Long Journey».

1.2.5 Bilan

Le TABLEAU 1.2 récapitule les aptitudes des approches de refactoring des modèles UML étudiées vis-à-vis des critères d'évaluation retenus.

Approche de	S. Markovic et *al.*	P. Gorp et *al.*	M. V. Kempen et *al.*	T. Mens et *al.*	S. Markovic et *al.*	G. Sunyé et *al.*	A. Correa et *al.*
Prise en compte de diagramme de classes	oui	partielle	non	oui	oui	oui	non
Prise en compte de diagramme d'états-transition	non	non	oui	oui	non	oui	non
Prise en compte de contraintes OCL	oui	oui	non	non	oui	non	oui
Préservation du comportement	transformation de modèle formalisée en QVT	méta-modélisation	UML vers des processus CSP	UML vers des graphes	grammaires de graphes	réécriture	réécriture
Outil	supportant QVT	moteur de requête OCL	supportant CSP	Fujaba pour la transformation de graphes	formalisme basé sur les grammaires de graphes	non	non
Détection du refactoring	non	design smells	non	le refactoring le mieux adapté	non	non	OCL smells

TABLE 1.2 – Tableau de synthèse des travaux liés au refactoring de modèles

Les approches examinées ne permettent pas de traiter des modèles UML décrits à la fois par des diagrammes de classes, des diagrammes d'états-transition et des contraintes OCL. Ceci ne favorise pas la vérification de la préservation après refactoring de deux propriétés essentielles à savoir propriété de sûreté (diagramme de classes et contraintes OCL) et propriété de vivacité (diagramme d'états-transition).

1.2.6 Approche proposée

Dans la troisième partie de cette thèse, nous proposons des schémas de refactoring permettant la réorganisation de la structure interne des diagrammes de classes UML en tenant compte des contraintes OCL attachées à ces diagrammes. Ces schémas permettent aussi de transformer les diagrammes d'états-transitions en tenant compte des modifications apportées aux diagrammes de classes. De tels schémas peuvent être assimilés à des règles de refactoring. Les schémas de refactoring proposés permettent l'amélioration des facteurs de qualité tels que : réutilisabilité, extensibilité et efficacité. Ils permettent aux concepteurs d'introduire des notions telles que : l'héritage, le polymorphisme, la classe abstraite, la redéfinition, l'association, la délégation et la généricité. Ils sont présentés selon un canevas précis comportant quatre rubriques exhibant les aspects fondamentaux d'un schéma de refactoring. En outre, les schémas de refactoring proposés sont formalisés en **B** en se servant des règles systématiques de transformation d'UML vers B [73, 57] et d'OCL vers **B** [60, 61] et en CSP en se servant de la fonction $\varphi_{UML \to CSP}$ de traduction d'un diagramme d'états-transitions vers des processus CSP [84]. Ceci permet d'identifier avec précision les conditions d'application, l'évolution d'un diagramme de classes UML, des contraintes

OCL et des diagrammes d'états-transitions et la correction du schéma. Les différentes vérifications sont confiées à l'***Atelier B*** pour les spécifications B obtenues et à ***FDR2*** pour les processus CSP obtenus. Ces propositions peuvent servir comme jalons pour l'automatisation, à terme, des schémas de refactoring.

1.3 Conclusion

Dans ce chapitre, nous avons étudié l'état de l'art lié au raffinement et refactoring des modèles UML. En ce qui concerne le raffinement, nous avons identifié les limites de différentes approches étudiées favorisant l'intégration du concept raffinement dans un cadre semi-formel celui d'UML. Ces limites sont : peu ou pas d'assistance méthodologique au spécifieur, pas de couverture de toutes les constructions de modélisation (classe, relation et contrainte OCL) et vérification basée sur OCL. Pour surmonter ces limites, nous avons proposé une approche basée sur les patterns de raffinement et un processus de développement conjoint UML/B guidé par ces patterns.

En ce qui concerne le refactoring, les approches étudiées ne permettent pas de traiter des modèles UML riches décrits à la fois par des diagrammes de classes, des diagrammes d'états-transition et contraintes OCL. Pour y parvenir, nous avons proposé une approche permettant le refactoring des modèles UML riches. Notre approche combine UML (diagramme de classes, diagramme d'états-transition et contraintes OCL), B et CSP.

Dans le chapitre suivant, nous allons présenter les aspects fondamentaux des deux langages formels B et CSP et les règles de transformation d'UML vers ce deux langages.

Chapitre 2

D'UML vers les langages formels

Pour des raisons de vérification formelle, les patterns de raffinement et les schémas de refactoring proposés dans la deuxième et la troisième partie de ce document utilisent les deux langages formels **B** d'Abrial et **CSP** de Hoare. Ce chapitre essaye d'apporter un éclairage sur ces langages et leurs utilisations pour spécifier des modèles UML. La première section présente un aperçu de la méthode **B**. La deuxième section présente un aperçu du langage **CSP**. Les règles de dérivation des aspects structurels d'UML vers **B**, dérivation des contraintes OCL en **B** et des aspects comportementaux d'UML vers **CSP** réutilisées dans ce travail sont présentées respectivement dans les sections 3, 4 et 5. Enfin, la section 6 présente l'étude de cas : contrôle d'accès à un bâtiment.

2.1 Aperçu de la méthode B

En s'inspirant des travaux de E. W. Dijkstra [30] et de C. A. R. Hoare [46], la méthode **B** a été inventée par Jean-Raymond ABRIAL et a été définie dans le **B** Book [6] en 1996. C'est une méthode formelle qui couvre toutes les étapes de développement d'un logiciel, de la spécification jusqu'à l'implantation, grâce au concept du raffinement et qui permet d'exprimer d'une façon rigoureuse, dans un langage spécifique, les propriétés exigées par un cahier des charges. L'objectif de cette méthode est de produire un logiciel sûr et correct par construction.

2.1.1 Le langage

Le langage **B** est considéré comme une évolution du langage Z, adapté à une utilisation industrielle, et à l'ensemble du cycle de développement (voir FIGURE 2.1).

Le langage **B** est fondé sur le langage des composants, le langage des substitutions généralisées et les concepts mathématiques de la théorie des ensembles. Fondamentalement, la théorie des ensembles est présentée comme un cadre pour la description des modèles mathématiques. La notion de substitution généralisée est utilisée pour construire les prédicats à prouver. Le langage des composants est utilisé pour représenter une machine abstraite, un raffinement ou une implantation.

Machine abstraite. La machine abstraite est l'élément de base d'un modèle B. Elle permet de modéliser de façon abstraite des données encapsulées, des propriétés et des opérations, permettant d'accéder à ces données et leurs propriétés exigées par un cahier des charges. C'est un concept très proche des classes ou encore des types de données abstraits.

Raffinement et implantation. Le raffinement est utilisé dans la méthode **B** pour concrétiser les modèles (machine abstraite ou raffinement). Il consiste à reformuler les données, les

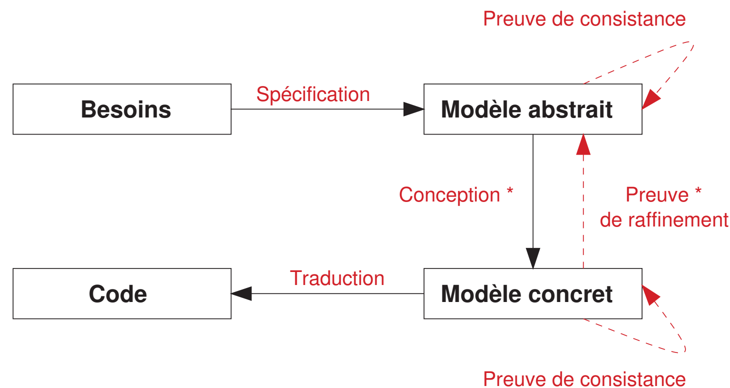


FIGURE 2.1 – Cycle de développement en B

propriétés et les opérations de la machine abstraite en introduisant les détails de conception du cahier des charges n'ayant pas été pris en compte. L'implantation est le dernier niveau de raffinement.

2.1.2 Les preuves

La méthode **B** distingue entre deux types de preuves : la conservation d'invariants et la correction du raffinement. Les preuves de conservation d'invariants vérifient la cohérence du modèle et les propriétés invariantes qui devront être conservées à l'initialisation et avant/après l'exécution des opérations. Les preuves de correction du raffinement garantissent la conformité du modèle concret par rapport à son homologue abstrait. Ces obligations de preuve sont générées automatiquement grâce aux outils associés au langage B.

2.1.3 Les outils

Un ensemble d'outils performants est développé pour une utilisation opérationnelle de la méthode B. Ces outils permettent :

- la vérification syntaxique des modèles,
- la génération automatique des obligations de preuve,
- une aide à la preuve : preuve automatique ou interactive des obligations de preuve,
- la traduction automatique des implantations **B** vers les langages C ou Ada,
- une aide au développement : gestion automatique des dépendances entre modèles et génération de documentation.

La TABLE 2.1 représente les principaux outils associés au langage B.

Dans la suite du document, nous utilisons l'outil ***Atelier B 4.0*** "*atelier de génie logiciel permettant de développer des logiciels prouvés sans défaut*" pour la preuve de la cohérence et du raffinement des modèles **B** issus des modèles UML.

Atelier B	Outil industriel qui permet une utilisation opérationnelle de la méthode **B** [26]
BEditor	Plugin Eclipse permettant d'éditer un modèle Atelier **B** avec colorisation des mots-clefs [25]
RODIN	Plateforme ouverte pour la modélisation et la preuve de systèmes complexes en **B** événementiel [2]
B-Toolkit	Outil industriel qui permet une utilisation opérationnelle de la méthode **B** [10]
Click'n'Prove	Interface du prouveur interactif de l'Atelier **B** et de l'outil B4Free [4]

TABLE 2.1 – Tableau des outils B

2.2 Aperçu du langage CSP

CSP, *Communicating Sequential Processes*, est défini par C. A. R. Hoare en 1978 dans [47, 48, 49]. Il permet de décrire le comportement de systèmes finis via des processus intercommuniquants.

CSP est une algèbre de processus qui permet, par le biais de processus de contrôle et de simulation à événements discrets, de spécifier, de concevoir, de mettre en oeuvre, de vérifier et de valider des systèmes informatiques complexes.

2.2.1 Les processus

Les processus CSP sont définis en terme d'événements considérés comme pertinents pour une description d'un objet. L'ensemble des noms de ces événements est appelé un alphabet. Le comportement le plus simple d'un processus est de ne rien faire : un tel processus est dénoté par `STOP`. Pour décrire des comportements plus élaborés, **CSP** offre des opérateurs tels que : Préfixe, Récursivité, Opérateurs de choix, Événements cachés, Composition parallèle, Entrées/Sorties, Entrelacement et Quantification.

2.2.2 Sémantique de CSP

Les trois principaux modèles sémantiques [87] sont les traces, les échecs stables et les échecs-divergences.

Le modèle des traces associe à chaque processus les séquences finies d'événements admises par ce processus. Ce modèle permet donc de représenter les comportements possibles de processus sous forme de traces. Les traces du processus `P` sont dénotées par `traces(P)`.

Le modèle des échecs stables associe à chaque processus `P` les couples de la forme `(t,E)`, où `t` est une trace finie admise par `P` et `E` est l'ensemble des événements que le processus ne peut pas exécuter après avoir exécuté les événements de `t`. L'ensemble de ces couples est noté `failures(P)`. Ce modèle permet de caractériser les blocages de `P`. En effet, si E est égal à l'ensemble des événements exécutables par `P`, alors `P` se trouve bloqué.

Enfin, le modèle des échecs-divergences associe à chaque processus `P` l'ensemble de ses échecs stables et l'ensemble de ses divergences. Un processus `P` n'est divergent que s'il se trouve dans un état dans lequel les seuls événements possibles sont les événements internes. Cet état est dit divergent. L'ensemble des divergences de `P` noté `divergences(P)`, est l'ensemble des traces `t`

telles que le processus se retrouve dans un état divergent après avoir exécuté `t`. Si le processus est déterministe, alors `divergences(P)` est vide.

2.2.3 Le Raffinement CSP

Le raffinement consiste à calculer et à comparer les modèles sémantiques de deux processus. Le raffinement dépend donc du modèle considéré. Par exemple, dans le cas du modèle des échecs-divergences, si `P` et `Q` sont deux processus, alors `Q` raffine `P`, noté

$$\texttt{P} \sqsubseteq_{FD} \texttt{Q} \text{ si : } \texttt{failures(Q)} \subseteq \texttt{failures(P)} \wedge \texttt{divergence(Q)} \subseteq \texttt{divergence(P)}$$

Concrètement, cela signifie qu'un observateur ne peut pas distinguer si un processus a été substitué à un autre.

2.2.4 Les outils

Un ensemble d'outils performants est développé pour une utilisation opérationnelle du langage CSP. Ces outils permettent :
- les vérifications syntaxiques des processus,
- la vérification des règles : de déterminisme, de non-déterminisme et de récursivité des processus,
- la vérification des propriétés des processus,
- la preuve de l'absence : de blocage et de divergence,
- la vérification du raffinement par : trace, échec et échec-divergence.

La Table 2.2 représente les principaux outils associés au langage CSP.

FDR2	Vérificateur automatique (model-checker) des modèles CSP et leurs raffinements [87, 88, 89]
ProBE	Animateur qui permet d'afficher les actions possibles et les états d'un processus [91]
CSP typechecker	Outil de vérification des processus CSP mais qui reste encore dans sa version beta [97]
CCSP	Compilation CSP vers C, exécution des processus CSP en tant que processus Unix (canaux ≡ sockets) [76]
JCSP	Bibliothèque pour threads Java ayant la sémantique de CSP [102]

Table 2.2 – Tableau des outils CSP

Dans la suite du document, nous utilisons l'outil ***FDR2*** pour la vérification du raffinement des processus CSP issus des modèles UML.

2.3 Dérivation des aspects structurels d'UML en B

Plusieurs travaux ont été proposés pour la dérivation des diagrammes UML en B [50]. Les travaux de Meyer [73] et Ledang [57] ont ciblé une démarche exhaustive portant sur la prise en compte simultanée de plusieurs diagrammes UML. En outre, les travaux de Laleau [52] se sont spécialisés dans les domaines de bases de données afin de produire du code SQL sûr. Enfin, les

travaux de Lano [54] et Snook [92] proposent d'ouvrir UML sur **B** en définissant un profil **B** pour UML.

Cette section présente des propositions de Meyer et Ledang, concernant les diagrammes de classes, avec une mise en évidence des dérivations des concepts suivants : classe, attribut, opération, association et généralisation.

2.3.1 Dérivation de classes

Une classe `Class` est dérivée en **B** par la construction d'une machine abstraite appelée `B_Class`, avec :

B_CLASS est définie comme une constante de la machine abstraite B. Elle représente un sous ensemble de `B_OBJECTS`, avec `B_OBJECTS` est l'ensemble de tous les objets possibles.

B_class est définie comme une variable de la machine abstraite B. Elle représente un sous ensemble de `B_CLASS`. Elle est initialisée à l'ensemble vide.

Types est une machine abstraite spéciale dans laquelle sont modélisés les différents types des attributs qui ne sont pas prédéfinis en **B**.

La FIGURE 2.2 illustre la dérivation en **B** d'une classe.

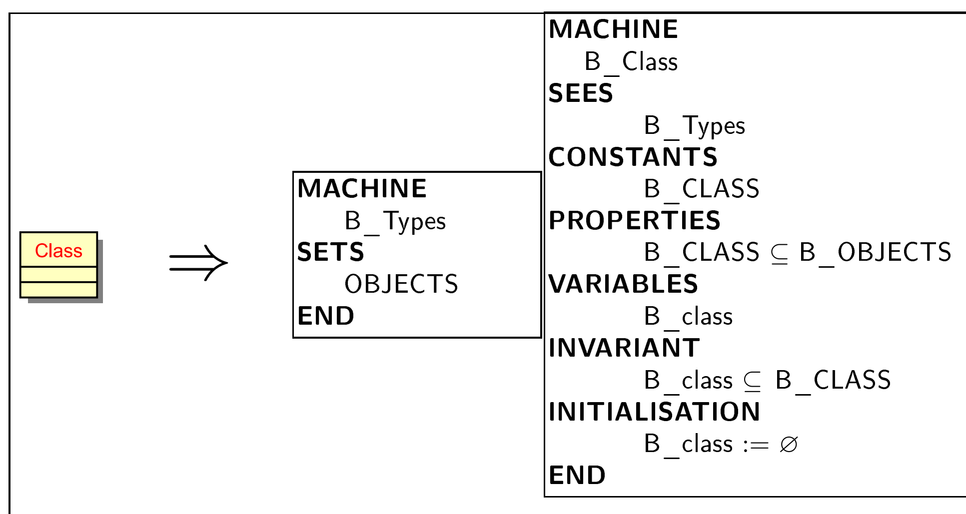


FIGURE 2.2 – Dérivation d'une classe

2.3.2 Dérivation d'attributs

Un attribut `attribute` est dérivé en **B** par une nouvelle variable `B_attribute` dans la machine abstraite associée à la classe de l'attribut. Il est typé dans l'invariant de la machine correspondante par une relation ou une fonction entre l'ensemble des objets instanciés `B_class` et le type de l'attribut `Type`. Il est initialisé à l'ensemble vide.

La FIGURE 2.3 illustre la dérivation en **B** d'un attribut.

2.3.3 Dérivation d'opérations

Une opération `operation` d'une classe `Class` est dérivée en **B** par une opération `B_operation` dans la machine abstraite `B_Class`, avec :

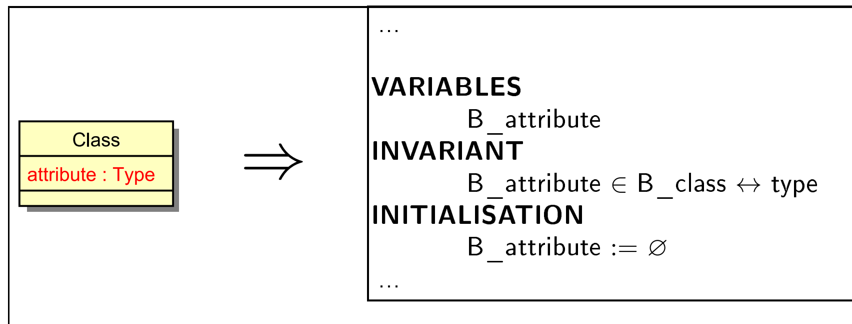


FIGURE 2.3 – Dérivation d'un attribut

- une variable `out` comme paramètre de sortie d'`operation`. Elle est définie dans le corps et de type `typeRetour`,
- une variable `cc`, comme paramètre formel d'`operation`, précisant l'instance qui va recevoir le service. Elle est définie dans la pré-condition et de type `B_class`,
- une variable `param`, comme paramètre formel d'`operation`, précisant un paramètre de la signature. Elle est définie dans la pré-condition et de type `B_class`.

La FIGURE 2.4 illustre la dérivation en **B** d'une opération.

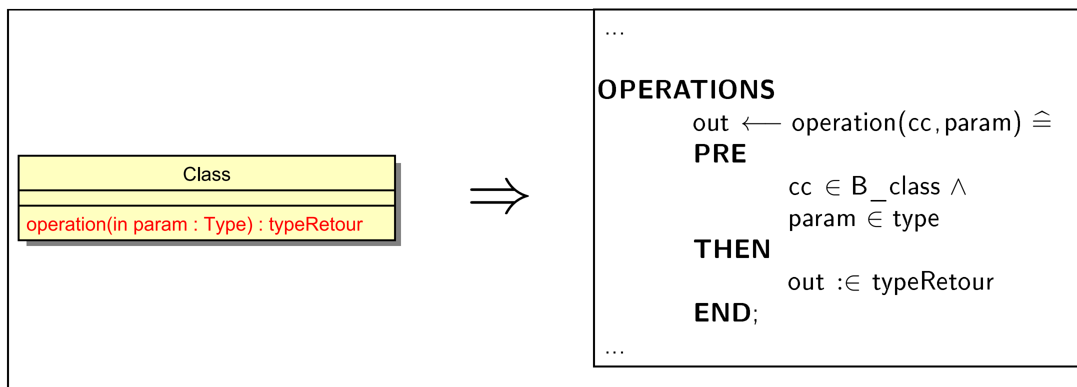


FIGURE 2.4 – Dérivation d'une opération

2.3.4 Dérivation des associations

Une association `association`, qui relie `Class1` et `Class2`, est dérivée en **B** par une nouvelle variable `B_association`. Cette variable est typée dans la clause INVARIANT par une relation ou une fonction entre les deux ensembles des objets instanciés `B_class1` et `B_class2`. Elle est initialisée à l'ensemble vide.

La FIGURE 2.5 illustre la dérivation en **B** d'une association.

2.3.5 Dérivation d'une classe associative

La classe associative `AssociationClass` est considérée en tant que classe liée par deux associations simples (rr1 et rr2) aux deux classes extrémités de la classe associative. Ainsi, un invariant de typage s'impose :

$$\begin{array}{c} \mathsf{rr1} \in \mathsf{associationClass} \rightarrow \mathsf{class1} \ \wedge \\ \mathsf{rr2} \in \mathsf{associationClass} \rightarrow \mathsf{class2} \end{array}$$

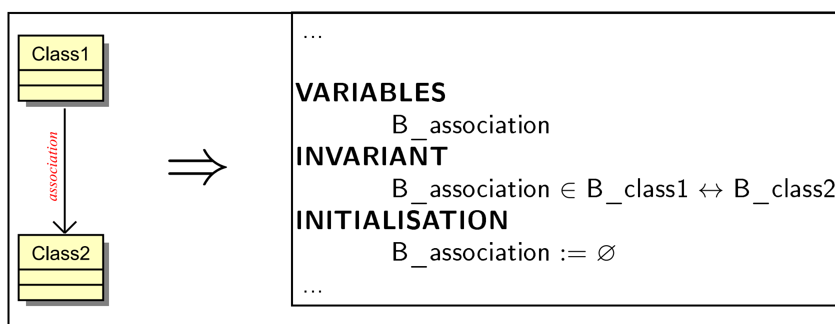


FIGURE 2.5 – Dérivation d'une association

Un invariant supplémentaire doit être rajouté en vue de désigner que chaque couple d'instances des classes associées à la classe associative détermine au maximum une instance de la classe associative :

$$\text{rr1} \times \text{rr2} \in \text{associationClass} \rightarrowtail \text{class1} \times \text{class2}$$

La FIGURE 2.6 illustre la dérivation en **B** d'une association.

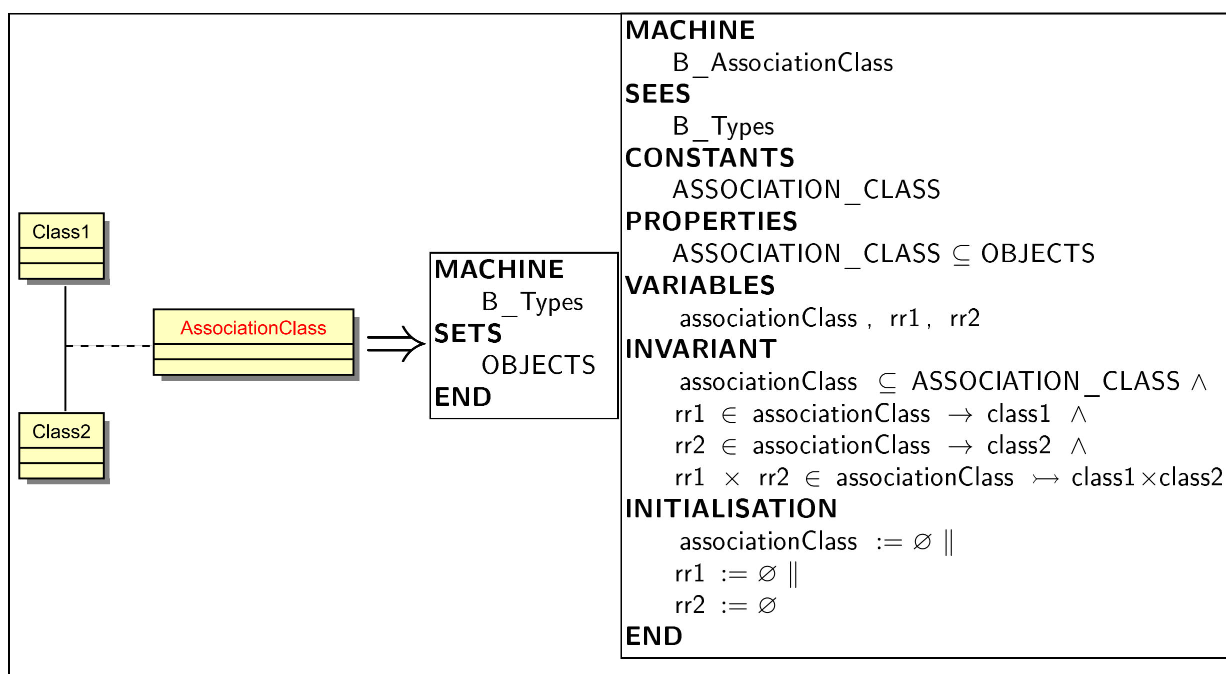


FIGURE 2.6 – Dérivation d'une classe associative

2.3.6 Dérivation d'une généralisation

La généralisation d'une sous-classe `SubClass` par une super-classe `SuperClass` est formalisée en **B** par :

- la dérivation de `SuperClass`, en utilisant les règles de dérivation d'une classe ordinaire présentées précédemment,
- la dérivation de `SubClass` de la manière suivante :

- la machine B_SubClass utilise la machine B_SuperClass. Pour cela, nous utilisons la clause USE. L'utilisation de cette clause permet de modéliser l'ensemble des objets possibles de B_SubClass par la constante **B** B_SUPERCLASS.
- l'ensemble des objets instanciés B_subclass est un sous ensemble de celui de sa superclasse. Il est initialisé à l'ensemble vide.
- les attributs de B_SubClass sont dérivés de la même façon que pour une classe ordinaire.

La Figure 2.7 illustre la dérivation en **B** d'une généralisation.

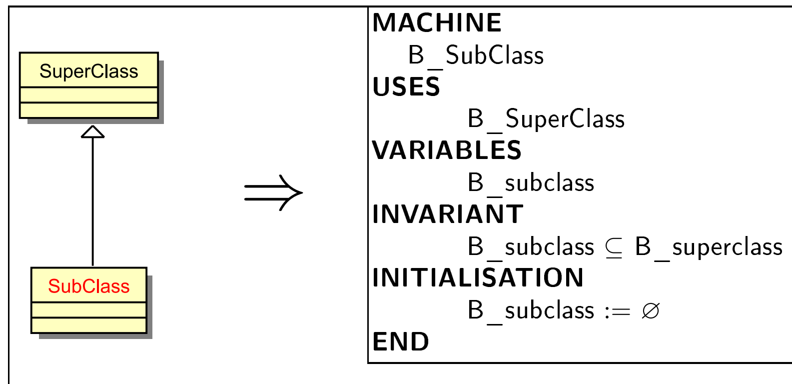


Figure 2.7 – Dérivation d'une sous-classe

2.4 Dérivation de contraintes OCL

Dans cette section, nous rappelons les points essentiels des travaux de transformation des contraintes OCL vers B, proposés dans [58, 60] .

En OCL, des types de base sont prédéfinis. Ils sont dérivés en B par un type correspondant à l'exception de `real`, qui sera traduit par une fraction définie par un couple de valeurs entières. De même, les opérations OCL applicables sur les collections (`Set`, `Bag` et `Sequence`) sont traduites en B par des opérations ensemblistes de B. Enfin, les expressions OCL permettent principalement d'exprimer deux types de contraintes sur l'état d'un objet ou d'un ensemble d'objets :

Invariant d'une classe. Il est exprimé dans la clause **INVARIANT** de la machine B correspondante notée B_Class.

Des pré et post-conditions d'une opération ne décrivent pas comment l'opération est réalisée mais des contraintes sur l'état avant et après son exécution.

Pré-condition. Elle est traduite par l'adjonction d'une pré-condition à l'opération B correspondante.

Post-condition. Elle est traduite par l'adjonction d'une substitution à l'opération B correspondante.

2.5 Dérivation des aspects comportementaux d'UML en CSP

Dans cette section, nous rappelons les points essentiels des travaux, proposés par Rasch et Wehrheim [84] de dérivation des diagrammes états-transitions vers des processus CSP.

2.5.1 Conventions d'affectation de noms

Le TABLE 2.3 représente les conventions d'affectation de noms utilisées dans la suite du document.

$\mathcal{SM}$	le diagramme états-transitions UML
s	un état simple ou composé
$\mathcal{C}_s$	l'ensemble des tous les successeurs directs de s
$\mathcal{T}_s$	l'ensemble de tous les couples (e, t) des successeurs t de s atteint par le biais d'une transition déclenchée par e
$\mathcal{M}$	une sous-machine de $\mathcal{SM}$
$\mathcal{M}_{top}$	la machine d'état de haut niveau
$\mathcal{I}_{\mathcal{M}}$	l'ensemble des états initiaux de $\mathcal{M}$
$\varphi_{UML \to CSP}$	la fonction de traduction de diagrammes états-transitions vers des processus CSP
$SKIP$	fin du processus CSP
$STOP$	`deadlock`
$\vert\vert\vert$	une composition parallèle
;	une composition séquentielle

TABLE 2.3 – Tableau de conventions d'affectation de noms

2.5.2 Dérivation d'un état

La fonction de traduction $\varphi_{UML \to CSP}$ d'un état s en CSP est définie :

$$\varphi_{UML \to CSP}(s) \equiv \begin{cases} \mathcal{P}_s = SKIP & s \text{ est un état final,} \\ \mathcal{P}_s = \Box_{(e,t) \in \mathcal{T}_s} e \to \mathcal{P}_t & s \text{ est un état simple et } \mathcal{C}_s = \emptyset, \\ \mathcal{P}_s = \sqcap_{t \in \mathcal{C}_s} \mathcal{P}_t & s \text{ est un état simple et } \mathcal{C}_s \neq \emptyset \text{ ou } s \text{ est un état initial,} \\ \mathcal{P}_s = (|||_{i=1}^{n} \mathcal{P}_{\mathcal{M}_i}); ((\sqcap_{t \in \mathcal{C}_s} \mathcal{P}_t) \not< \mathcal{C}_s \neq \emptyset \not> STOP) & s \text{ est un état composé de sous-machines } \mathcal{M}_i \text{, avec } 1 \leq i \leq n. \end{cases}$$

2.5.3 Dérivation d'une sous-machine

La fonction de traduction $\varphi_{UML \to CSP}$ d'une sous-machine $\mathcal{M}$ en CSP est définie :

$$\varphi_{UML \to CSP}(\mathcal{M}) \equiv \mathcal{P}_{\mathcal{M}} = \sqcap_{t \in \mathcal{I}_{\mathcal{M}}} \mathcal{P}_t$$

2.5.4 Dérivation de diagrammes d'états-transitions

Après le calcul de $\varphi_{UML \to CSP}(s)$ pour tout état s et $\varphi_{UML \to CSP}(\mathcal{M})$ pour toute sous-machine $\mathcal{M}$ de $\mathcal{SM}$, le processus CSP correspondant au diagramme états-transitions $\mathcal{SM}$ peut être calculé par la combinaison des fonctions $\varphi_{UML \to CSP}$ obtenues et l'évaluation de la machine d'états $\mathcal{M}_{top}$:

$$\mathcal{PROC}_{\mathcal{SM}} = \mathcal{M}_{top}$$

2.6 Etude de cas : Contrôle d'accès à un bâtiment

Afin d'illustrer nos propos, nous présentons le cahier de charge d'un système de contrôle d'accès à un bâtiment. Ce système est utilisé comme une étude de cas tout au long de ce document. Ceci nous permettra d'illustrer la mise en oeuvre de nos propositions.

2.6.1 Objectif de l'étude de cas

Notre objectif est de développer un système chargé de contrôler l'accès de personnes aux divers bâtiments d'un lieu de travail [1, 13, 7]. Le contrôle s'effectue à partir des autorisations affectées aux personnes concernées. Une autorisation permet à une personne, sous le contrôle du système, d'entrer dans certains bâtiments et pas dans d'autres. Les autorisations sont permanentes, c'est-à-dire qu'elles ne peuvent pas être modifiées pendant le fonctionnement du système. Lorsqu'une personne est à l'intérieur d'un bâtiment, sa sortie doit aussi être contrôlée de façon à ce qu'il soit possible de connaître, à tout instant, qui se trouve dans un bâtiment donné.

2.6.2 Présentation générale de l'étude de cas

Nous allons maintenant procéder à la spécification informelle de notre système. L'objectif de ce document est de présenter des schémas et des patterns qui permettent une construction incrémentale de spécifications UML. Ainsi, nous avons opté pour une énumération des principales propriétés du système. Une telle présentation permet au concepteur de se concentrer sur les propriétés une par une.

2.6.3 Propriétés de l'étude de cas

Les principales propriétés du système sont les suivantes :

- le système est chargé de contrôler l'accès d'un ensemble de personnes à un ensemble de bâtiments,
- impossibilité pour une même personne de se trouver simultanément dans deux bâtiments distincts : une personne se trouve dans au plus un bâtiment à la fois,
- toute personne se trouvant dans un bâtiment est autorisée à y être,
- une personne ne peut se déplacer d'un bâtiment à un autre que si ces deux bâtiments communiquent entre eux,
- chaque porte permet de passer d'un bâtiment origine à un bâtiment destination,
- une personne peut entrer dans un bâtiment en franchissant une porte si elle est débloquée,
- les portes étant physiquement bloquées, une porte donnée se débloque pour une seule personne autorisée demandant à entrer dans le bâtiment,
- un voyant vert associé à chaque porte est allumé lorsque l'accès demandé est autorisé, condition nécessaire au déblocage de la porte,
- un voyant rouge associé à chaque porte s'allume lorsque l'accès demandé pour cette porte est refusé,
- chaque personne dispose d'une carte magnétique,
- des lecteurs de cartes sont installés à chaque porte permettant de lire les informations contenues sur une carte.

2.6.4 Diagramme de classes

Notre objectif dans cette thèse est de construire un modèle UML en utilisant le technique de raffinement. Pour cela, nous commençons le développement par un diagramme de classes rudimentaire et très abstrait (voir FIGURE 2.8).

La partie 3 de ce document est consacrée aux schémas de refactoring qui sont utilisés pour restructurer un modèle UML afin d'améliorer ces facteurs de qualité ou encore détecter et corriger des erreurs. Pour cela, nous avons besoin d'un état plus avancé que celui présenté par la FIGURE 2.8. Ce dernier est donné dans la FIGURE 2.9.

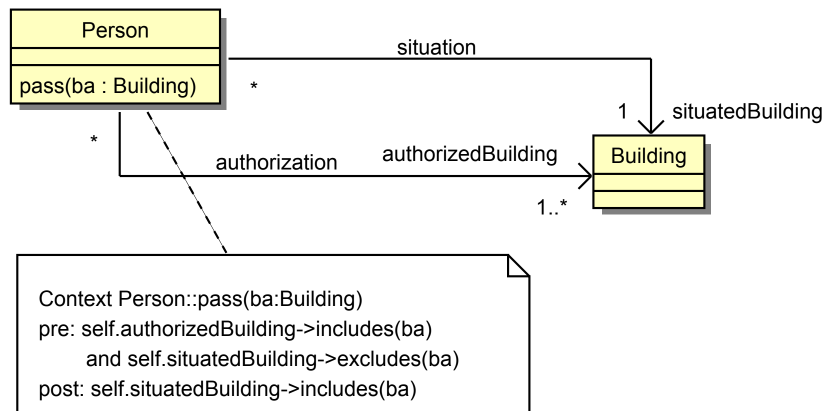


Figure 2.8 – Un premier diagramme de classes

2.6.5 Contraintes OCL et Diagrammes d'états-transitions

Dans les sections suivantes, nous présentons les contraintes OCL ainsi que les diagrammes d'états-transitions correspondantes aux classes utilisées dans le reste de ce document.

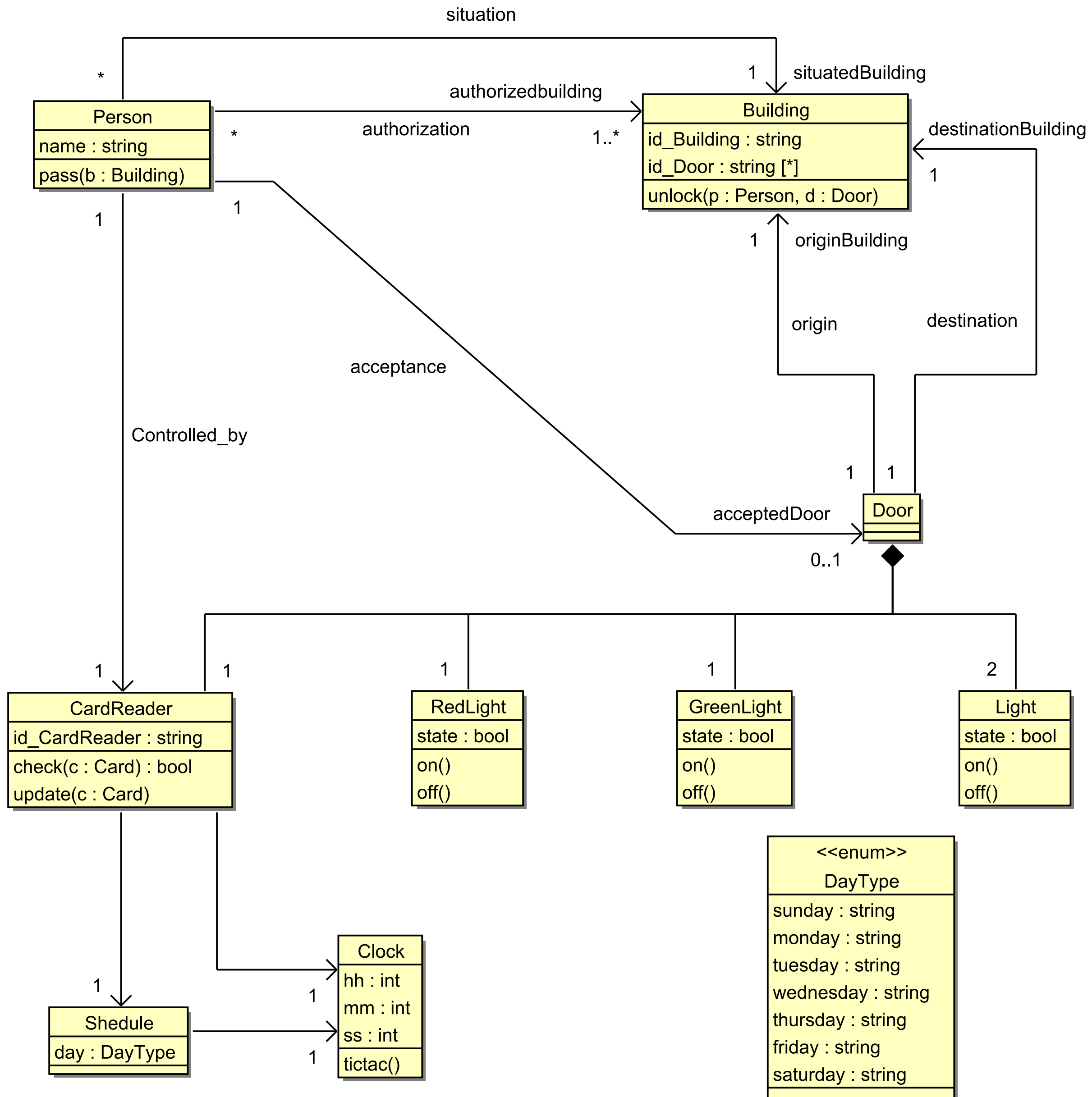


FIGURE 2.9 – Un état de la modélisation de contrôle d'accès à un bâtiment

Person

OCL_Person

/ Une personne peut passer d'un bâtiment à un autre si elle est autorisée */*

```
Context Person::pass(b:Building)
pre : self.building <> b and
      self.authorization->includes(b)
post : self.building = b
```

Building

OCL_Building

/ Un bâtiment possède un identifiant et toutes ses portes possèdent des identifiants */*

```
Context Building
inv I_Building : self.id_Building.size() > 0 and
                 self.id_Door.forAll(d|d.size() > 0)
```

/ La porte est débloquée lorsqu'une personne désirant entrer dans un bâtiment donné est autorisée à entrer dans le bâtiment */*

```
Context Building::unlock(p : Person, d : Door)
pre P_unlock_Building : d.originBuilding = p.situatedBuilding and
                        p.authorizedBuilding->includes(d.destinationBuilding)
post Q_unlock_Building : p.acceptedDoor = d
```

STD_Building

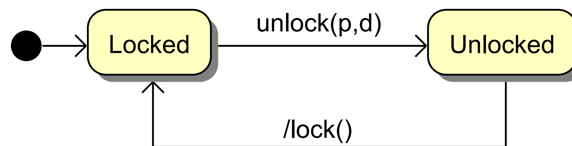


FIGURE 2.10 – STD_Building

Light

OCL_Light

```
Context Light
inv I_Light : True

Context Light::on()
pre P_on_Light : state = False
post Q_on_Light : state = True

Context Light::off()
pre P_off_Light : state = True
post Q_off_Light : state = False
```

STD_ Light

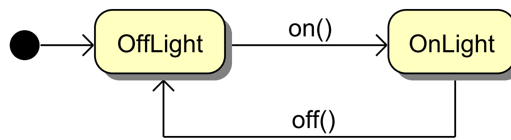


FIGURE 2.11 – STD_Light

GreenLight

OCL_ GreenLight

```
Context GreenLight
inv I_GreenLight : True

Context GreenLight::on()
pre P_on_GreenLight : state = False
post Q_on_GreenLight : state = True

Context GreenLight::off()
pre P_off_GreenLight : state = True
post Q_off_GreenLight : state = False
```

STD_ GreenLight

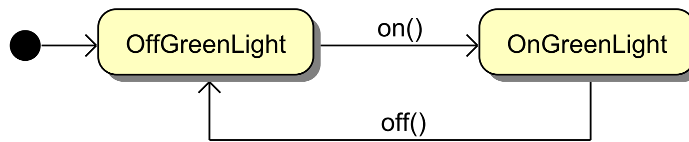


FIGURE 2.12 – STD_GreenLight

RedLight

OCL_ RedLight

```
Context RedLight
inv I_RedLight : True

Context RedLight::on()
pre P_on_RedLight : state = False
post Q_on_RedLight : state = True

Context RedLight::off()
pre P_off_RedLight : state = True
post Q_off_RedLight : state = False
```

STD_RedLight

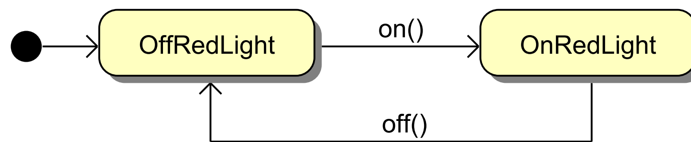


FIGURE 2.13 – `STD_RedLight`

Card

OCL_Card

```
Context Card
inv I_Card : True

Context Card::getValidity():bool
pre P_getValidity_Card : True
post Q_getValidity_Card : result = validity
```

STD_Card

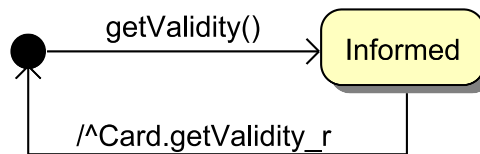


FIGURE 2.14 – `STD_Card`

CardReader

OCL_CardReader

/ Le lecteur de carte permet de vérifier la validité d'une carte donnée et de contrôler l'heure et le jour */*

```
Context CardReader
inv I_CardReader : True

Context CardReader::check(c:Card)
pre P_check_CardReader : shedule.day <> sunday and
    clock.hh > 8 and clock.hh < 18 and
    c = self.door.person.card
post Q_check_CardReader : c.getValidity()
```

STD_ CardReader

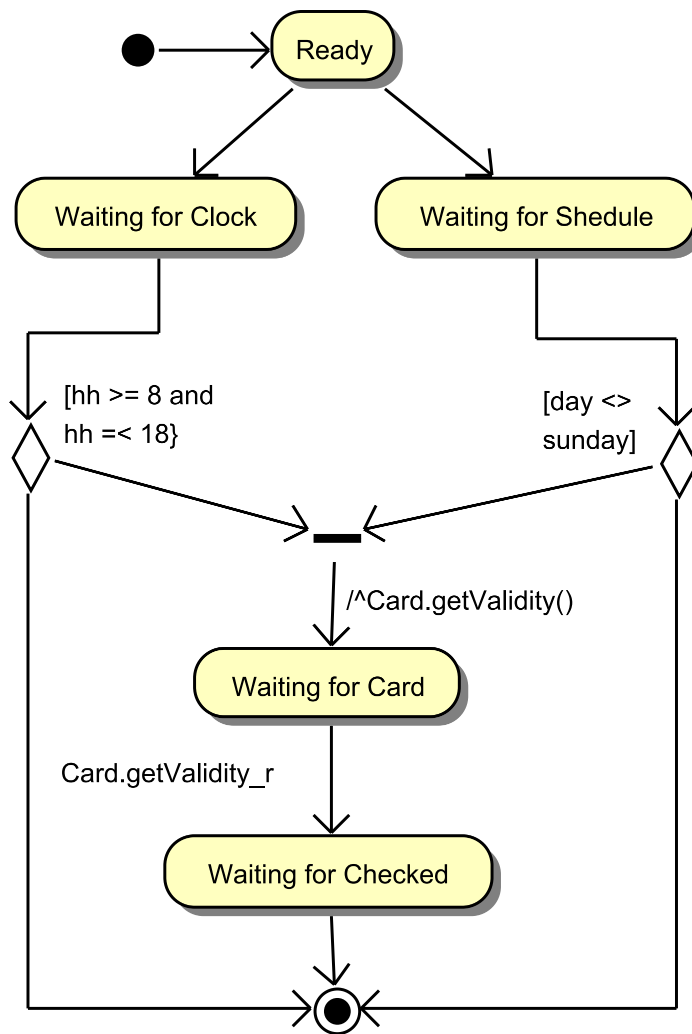


FIGURE 2.15 – `STD_CardReader`

2.7 Conclusion

Dans ce chapitre, nous avons présenté les aspects fondamentaux de deux langages formels B et CSP. En outre, nous avons fourni des règles de dérivation d'UML vers ce deux langages. Enfin, nous avons exposé les aspects généraux de l'étude de cas : Contrôle d'accès aux bâtiments. Le langage formel B est retenu pour ses aptitudes à vérifier les propriétés de sûreté : cohérence et correction des raffinements successifs. Tandis que le langage formel CSP est choisi pour ses aptitudes à vérifier les propriétés de vivacité : enchaînement d'appels d'opérations. Enfin, l'application retenue "Contrôle d'accès aux bâtiments" possède plusieurs mérites : divers concepts métier plus ou moins abstraits (communication, affectation, carte, voyant, porte, etc) et plusieurs propriétés de sûreté.

Deuxième partie

Approche de raffinement des spécifications UML

Chapitre 3

Patterns de raffinement

3.1 Introduction

3.1.1 Définition d'un pattern de raffinement

Contrairement aux patterns d'architecture [22], d'analyse [33] et de conception [36], un pattern de raffinement possède un caractère dynamique. Appliqué à une description de niveau i, un pattern de raffinement produit une description de niveau i+1. Récemment, des patterns de raffinement commencent à apparaître visant des formalismes comme Event-B [5, 45], KAOS [100] et B [86]. Dans ce chapitre, nous proposons des patterns de raffinement permettant de résoudre des problèmes récurrents dans un développement incrémental de la partie statique d'une application OO en utilisant UML/OCL. Un pattern de raffinement possède deux parties ; Spécification et Raffinement. La partie Spécification décrit le modèle UML/OCL de niveau i. Et la partie Raffinement décrit le modèle UML/OCL de niveau i+1 produit en appliquant le pattern de raffinement sur le modèle de niveau i.

3.1.2 Canevas de présentation d'un pattern de raffinement

Les patterns de raffinement proposés dans ce chapitre sont décrits selon le même canevas comportant les six rubriques suivantes : Intention, Motivation, Solution, Vérification, Exemple et Voir Aussi.

Intention

Une description informelle du pattern de raffinement, permettant de répondre aux questions suivantes : Que fait effectivement le pattern de raffinement ? Quel est son but ? quel problème particulier de raffinement concerne-t-il ?, débutera chaque description d'un pattern de raffinement.

Motivation

Cette rubrique a pour objectif de décrire le pourquoi du pattern de raffinement en se basant sur des situations typiques issues des systèmes d'information concrets.

Solution

Cette rubrique exhibe la solution offerte par le pattern de raffinement en utilisant UML/OCL.

Vérification

Dans cette rubrique, nous apportons une formalisation en B du pattern de raffinement. Pour y parvenir, nous réutilisons les règles de traduction systèmatique d'UML vers B (voir chapitre 2). De plus, nous élaborons les conditions de vérification de la correction du pattern décrites en B sous forme d'**invariant de collage**. Celui-ci permet de relier les deux parties d'un pattern à savoir Spécification et Raffinement.

Exemple

Un exemple de taille raisonnable aussi bien en UML/OCL qu'en B permettant de réutiliser le pattern est développé. Pour y parvenir, nous proposons une démarche favorisant l'application rigoureuse de ce pattern.

Voir Aussi

Dans cette section, nous précisions les prolongements possibles de ce pattern. En outre, nous discutons les relations potentielles inter-patterns.

3.2 Pattern d'introduction d'une classe intermédiaire : `Class_Helper`

3.2.1 Intention

Il permet d'introduire une classe **intermédiaire** `Class_Helper` entre deux classes jugées **importantes** vis-à-vis de l'étape de raffinement considérée. La relation directe entre les deux classes importantes est raffinée par un chemin liant ces deux classes en passant par la classe intermédiaire introduite.

3.2.2 Motivation

Un diagramme de classes UML comporte quatre types de relations inter-classes : généralisation (ou héritage), association, agrégation et dépendance. Dans une modélisation OO incrémentale, on a intérêt à démarrer avec des relations abstraites inter-classes. Ceci favorise ultérieurement l'introduction des détails via des classes intermédiaires permettant de raffiner ces relations abstraites. A titre d'exemple, dans un Système de Gestion d'Hôtels (SGH), les deux abstractions métier `Hotel` et `Client` peuvent être reliées par l'association `reside` (voir FIGURE 3.1).

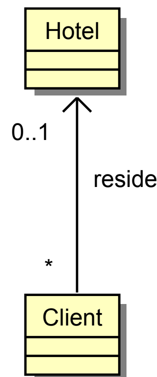


FIGURE 3.1 – Relation abstraite `reside`

Un raffinement à ce modèle peut introduire la précision suivante : en fait, un client réside dans une chambre donnée appartenant à cet hôtel. Pour y parvenir, on propose une classe intermédiaire `Chambre` entre les deux classes principales `Hotel` et `Client` (voir FIGURE 3.2).

L'aspect payement d'un SGH est modélisé par la FIGURE 3.3.

L'introduction du détail consistant à dire qu'un client est une personne est traduite par l'apparition d'une classe `Personne` comme classe ascendante à `Client` (voir FIGURE 3.4).

3.2.3 Solution

Nous nous plaçons dans le cadre suivant :

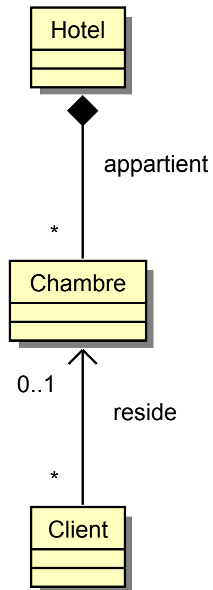


FIGURE 3.2 – Introduction de la notion de `Chambre`

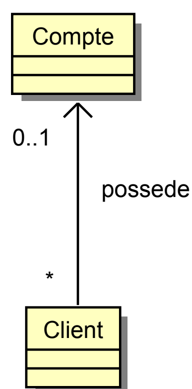


FIGURE 3.3 – Payement dans un SGH

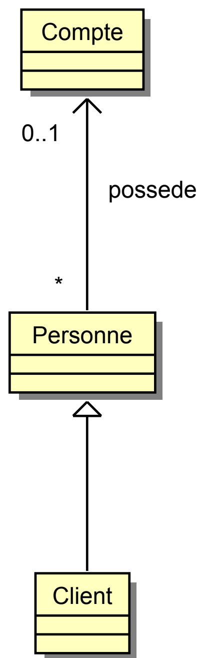
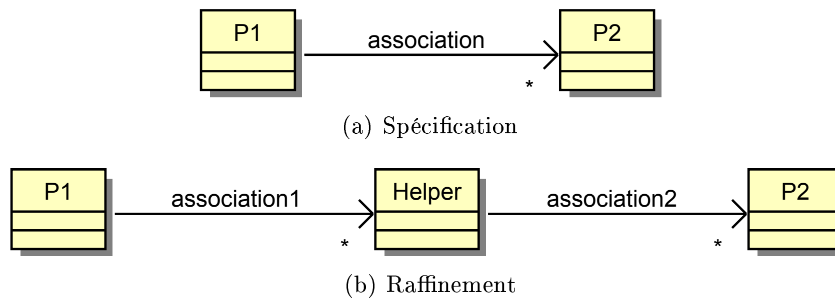


FIGURE 3.4 – Introduction de la notion de `Personne`

- les deux classes principales (ou importantes) `P1` et `P2` sont reliées par une association dirigée de `P1` vers `P2`,
- les deux autres propriétés (attributs et opérations) de `P1` et `P2` ne sont pas prises en compte.

Les deux parties Spécification et Raffinement de ce pattern sont données dans la FIGURE 3.5.



(a) Spécification

(b) Raffinement

FIGURE 3.5 – Pattern d'introduction d'une classe intermédiaire : `Class_Helper`

3.2.4 Vérification

La FIGURE 3.6 présente la machine abstraite `B_Class_Helper_a` correspondante à la transformation en **B** de la partie Spécification présentée par la FIGURE 3.5(a).

L'ensemble OBJECTS est défini par extension. Il comporte des objets respectivement de type `P1 (p1i)`, `P2 (p2i)` et `Helper (hi)`. De même, les deux constantes abstraites `P1` et `P2` sont définies par extension. Elles mémorisent des objets potentiels de type `P1` et `P2`. Ceci permet d'initialiser (voir opération INITIALISATION) les variables `p1`, `p2` et `association`. Une telle initialisation correspond à un diagramme d'objets issu du diagramme de classes de la partie Spécification du pattern de raffinement proposé (voir FIGURE 3.5(a)). Les obligations de preuve générées relatives à la machine `B_Class_Helper_a`

```
MACHINE
    B_Class_Helper_a
SETS
    OBJECTS = {p11, p12, p13,
               p21, p22, p23,
               h1, h2, h3}
ABSTRACT_CONSTANTS
    P1, P2
PROPERTIES
    P1 ⊆ OBJECTS ∧
    P2 ⊆ OBJECTS ∧
    P1 ∩ P2 = ∅ ∧
    P1 = {p11, p12, p13} ∧
    P2 = {p21, p22, p23}
VARIABLES
    p1, p2, association
INVARIANT
    p1 ⊆ P1 ∧
    p2 ⊆ P2 ∧
    association ∈ p1 ↔ p2
INITIALISATION
    p1 := {p11,p12, p13} ||
    p2 := {p21, p22, p23} ||
    association := {p11 ↦ p21, p11 ↦ p22,
        p11 ↦ p23, p12 ↦ p21, p12 ↦ p22,
        p12 ↦ p23, p13 ↦ p21, p13 ↦ p22,
        p13 ↦ p23}
END
```

FIGURE 3.6 – Modélisation en **B** de l'état de spécification abstraite

ont été toutes déchargées automatiquement (voir Tableau 3.1). Ceci valide **en partie** les propriétés invariantes liées à l'état de la machine B_Class_Helper_a.

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	7	0	7	0	100
B_Class_Helper_a	7	0	7	0	100

TABLE 3.1 – Tableau de l'état de la machine B_Class_Helper_a

La FIGURE 3.7 présente la traduction en **B** de l'état de la partie Raffinement du pattern donnée dans la FIGURE 3.5(b).

La machine B_Class_Helper_r introduit une constante abstraite HELPER pour mémoriser les objets potentiels de type Helper. La clause PROPERTIES stipule que les constantes P1, P2 et HELPER sont deux à deux disjointes (voir $P1 \cap P2 = \oslash$ venant de la machine B_Class_Helper_a). L'état variable de la machine B_Class_Helper_r est défini par les cinq variables p1, p2, helper, association1 et association2. Les deux variables p1 et p2 proviennent de la machine abstraite B_Class_Helper_a. Tandis que les trois autres variables sont introduites par la partie Raffinement du pattern Class_Helper. La variable abstraite association a été supprimée. La clause INVARIANT de la machine B_Class_Helper_r définit le typage et les contraintes liées aux variables concrètes helper, association1 et association2. Egalement, elle comporte l'invariant de collage suivant :

```
dom(association) = dom(association1)          (inv1)
ran(association) = ran(association2)          (inv2)
ran(association1)= dom(association2)          (inv3)
association = (association1;association2)     (inv4)
```

$(inv_collage_Class_Helper)$ (3.1)

Cet invariant de collage garantit la correction du pattern de raffinement Class_Helper. Il pourrait être instancié et réutilisé avec profit dans un développement conjoint UML/B guidé par des patterns de raffinement.

Les obligations de preuve générées relatives à la machine B_Class_Helper_r ont été déchargées dont trois de façon interactive en utilisant les deux prouveurs interactifs ss et pr (voir Tableau 3.2). Ceci valide **en partie** les propriétés invariantes, notamment l'invariant de collage, liées à l'état de la machine B_Class_Helper_r.

```
REFINEMENT
   B_Class_Helper_r
REFINES
   B_Class_Helper_a
ABSTRACT_CONSTANTS
    HELPER
PROPERTIES
    HELPER ⊆ OBJECTS ∧
    HELPER ∩ P1 = ∅ ∧
    HELPER ∩ P2 = ∅ ∧
    HELPER = {h1, h2, h3}
ABSTRACT_VARIABLES
   p1 ,
   p2 ,
   helper ,
   association1 ,
   association2
INVARIANT
        helper ⊆ HELPER ∧
        association1 ∈ p1 ↔ helper ∧
        association2 ∈ helper ↔ p2 ∧
        ran( association1 ) = dom(association2) ∧
        /* Invariant de collage */
        dom(association) = dom(association1) ∧
        ran( association ) = ran( association2 ) ∧
        ran( association1 ) = dom(association2) ∧
        association = ( association1 ; association2 )
INITIALISATION
   p1 := {p11 , p12 , p13} ||
   p2 := {p21 , p22 , p23} ||
   helper := {h1, h2, h3} ||
   association1 := {p11↦h1, p11↦h2, p11↦h3,
                    p12↦h1, p12↦h2, p12↦h3,
                    p13↦h1, p13↦h2, p13↦h3} ||
   association2 := {h1↦p21, h1↦p22, h1↦p23,
                    h2↦p21, h2↦p22, h2↦p23,
                    h3↦p21, h3↦p22, h3↦p23}
END
```

FIGURE 3.7 – Modélisation en **B** de l'état de spécification concrète

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	9	3	6	0	100
B_Class_Helper_r	9	3	6	0	100

TABLE 3.2 – Tableau de l'état de la machine B_Class_Helper_r

3.2.5 Exemple

Dans un système de réservation aérienne, les deux classes concepts métier Personne et Avion peuvent être reliées par l'association embarquement. Une spécification pré/post de l'opération reserver est également fournie en OCL (voir FIGURE 3.8).

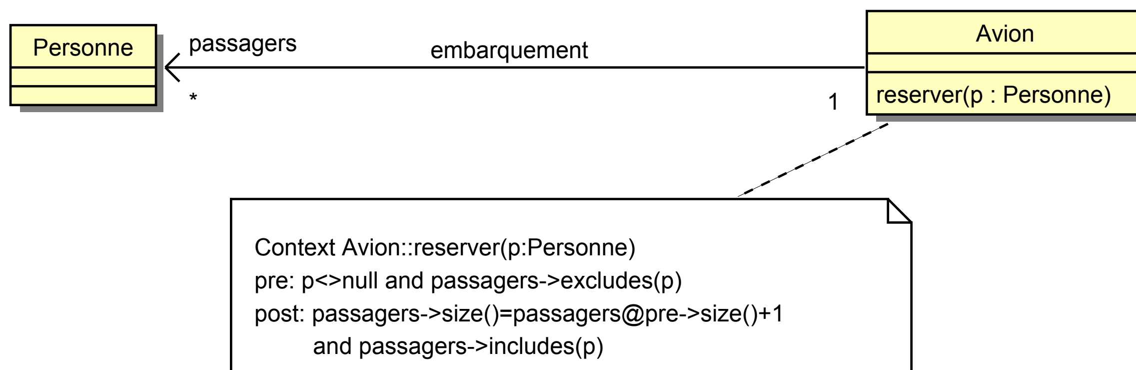


FIGURE 3.8 – Relation abstraite embarquement

Application du pattern Class_Helper

L'introduction de la notion de Vol est matérialisée par une étape de raffinement en appliquant le pattern de raffinement Class_Helper. Ceci est illustré par la FIGURE 3.9.

La spécification pré/post en OCL de l'opération reserver de la classe Avion entraîne celle de l'opération reserver de la classe Vol. En fait, l'opération reserver de la classe Avion devrait appliquer reserver de la classe Vol sur l'occurrence v passée comme paramètre de reserver de la classe Avion.

Vérification de l'application du pattern

Afin de formaliser en **B** le raffinage entre la spécification et le raffinement fournis respectivement par les deux FIGURES 3.8 et 3.9, nous proposons l'architecture des machines **B** présentée par la FIGURE 3.10.

Les deux machines Context et B_Avion formalisent en **B** la relation abstraite embarquement fournie par la FIGURE 3.8. La machine Context (voir FIGURE 3.11) factorise des ensembles abstraits AVION, PERSONNE et VOL utiles pour les deux niveaux : spécification et raffinement.

Quant à la machine B_Avion (voir FIGURE 3.11), elle regroupe les caractéristiques reserver et passagers venant du modèle UML de la FIGURE 3.8. La sémantique pré/post en OCL de la méthode reserver appartenant à la classe Avion est préservée par l'opération de même nom appartenant à la machine B_Avion.

Les obligations de preuve liées à la machine B_Avion sont données dans le tableau 3.3. Elles ont été déchargées automatiquement.

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	1	0	1	0	100
reserver	1	0	1	0	100
B_Avion	2	0	2	0	100

TABLE 3.3 – Tableau de l'état de la machine B_Avion

Les deux machines B_Vol et B_Avion_r (voir FIGURE 3.12) formalisent en **B** le raffinement de la relation abstraite embarquement fourni par la FIGURE 3.9. La machine B_Vol formalise en **B** la classe Vol du

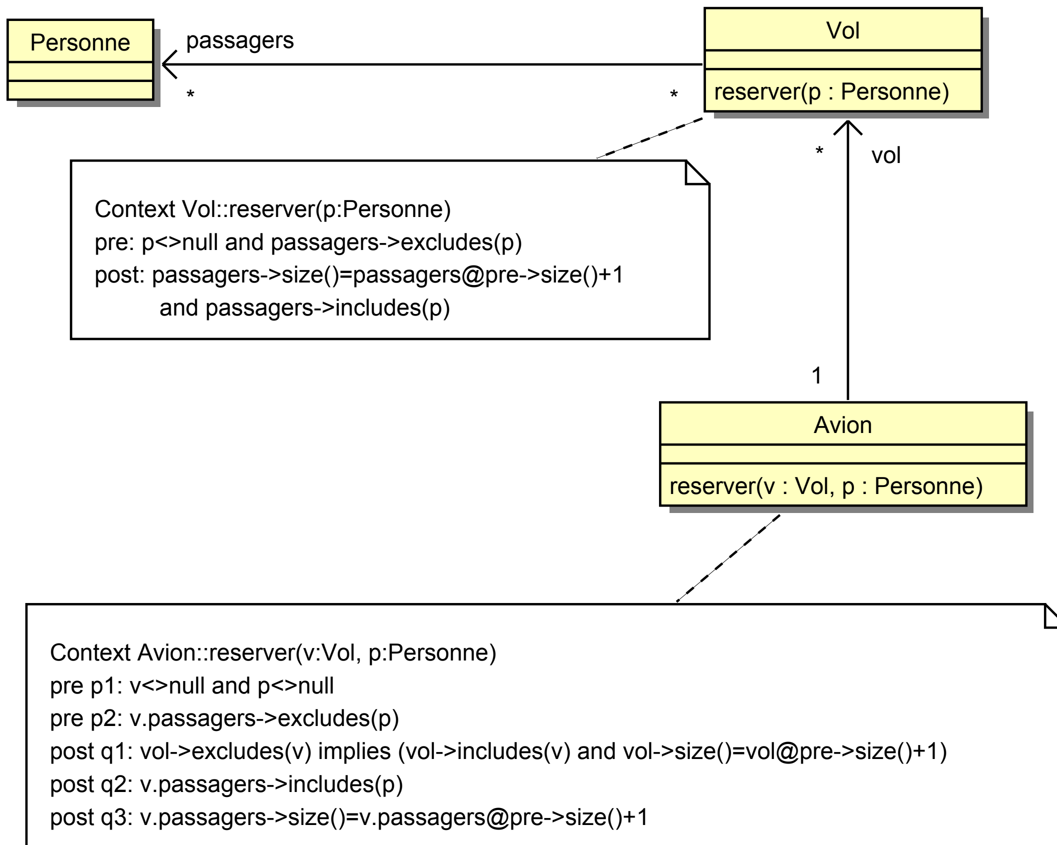


FIGURE 3.9 – Introduction de la notion de `Vol`

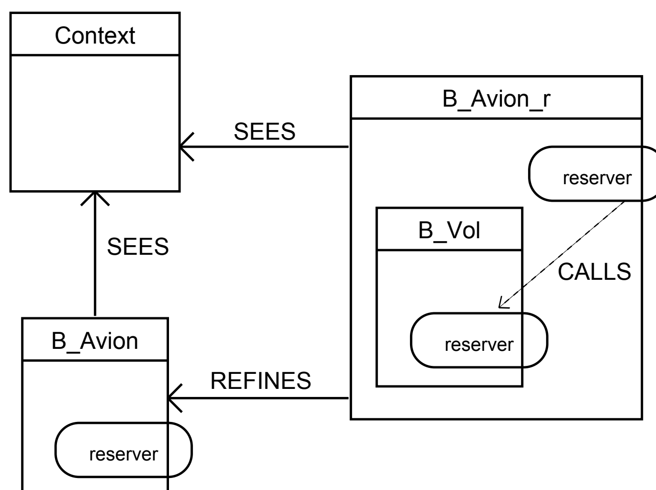


FIGURE 3.10 – Dépendances des machines `Context`, `B_Avion`, `B_Avion_r` et `B_Vol`

```
MACHINE
    Context
SETS
        AVION;PERSONNE;VOL
END
```

```
MACHINE
    B_Avion
SEES
        Context
VARIABLES
        passagers
INVARIANT
        passagers ∈ AVION ⇸ PERSONNE
INITIALISATION
        passagers := ∅
OPERATIONS
reserver (av1,pr1) =
        PRE
            av1 ∈ AVION ∧
            pr1 ∈ PERSONNE ∧
            pr1 ∉ ran(passagers)
        THEN
                passagers(av1) := pr1
        END
END
```

FIGURE 3.11 – Modélisation en **B** de l'état de spécification abstraite

modèle UML présenté par la même figure. Elle introduit la variable `vol_personne` (relation) permettant de modéliser l'association entre `Vol` et `Personne`. L'opération `reserver` de la machine `B_Vol` est spécifiée en tenant compte de la sémantique pré/post en OCL de la méthode `reserver` de la classe `Vol` de la FIGURE 3.9. La machine `B_Avion_r` raffine `B_Avion` en incluant une instance de la machine `B_Vol` (voir la clause INCLUDES). Elle introduit la variable `avion_vol` (fonction partielle) permettant de modéliser l'association entre `Avion` et `Vol`. L'invariant de la machine `B_Avion_r` est issu de l'invariant de collage du pattern `Class_Helper` (Inv_Collage_Class_Helper) (3.1). Les trois variables `association`, `association1` et `association2` sont remplacées respectivement par `passagers`, `avion_vol` et `ff.avion_personne`.

Les obligations de preuve liées à la machine `B_Vol` données dans le tableau 3.4 ont été déchargées automatiquement.

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	1	0	1	0	100
reserver	1	0	1	0	100
B_Vol	2	0	2	0	100

TABLE 3.4 – Tableau de l'état de la machine B_Vol

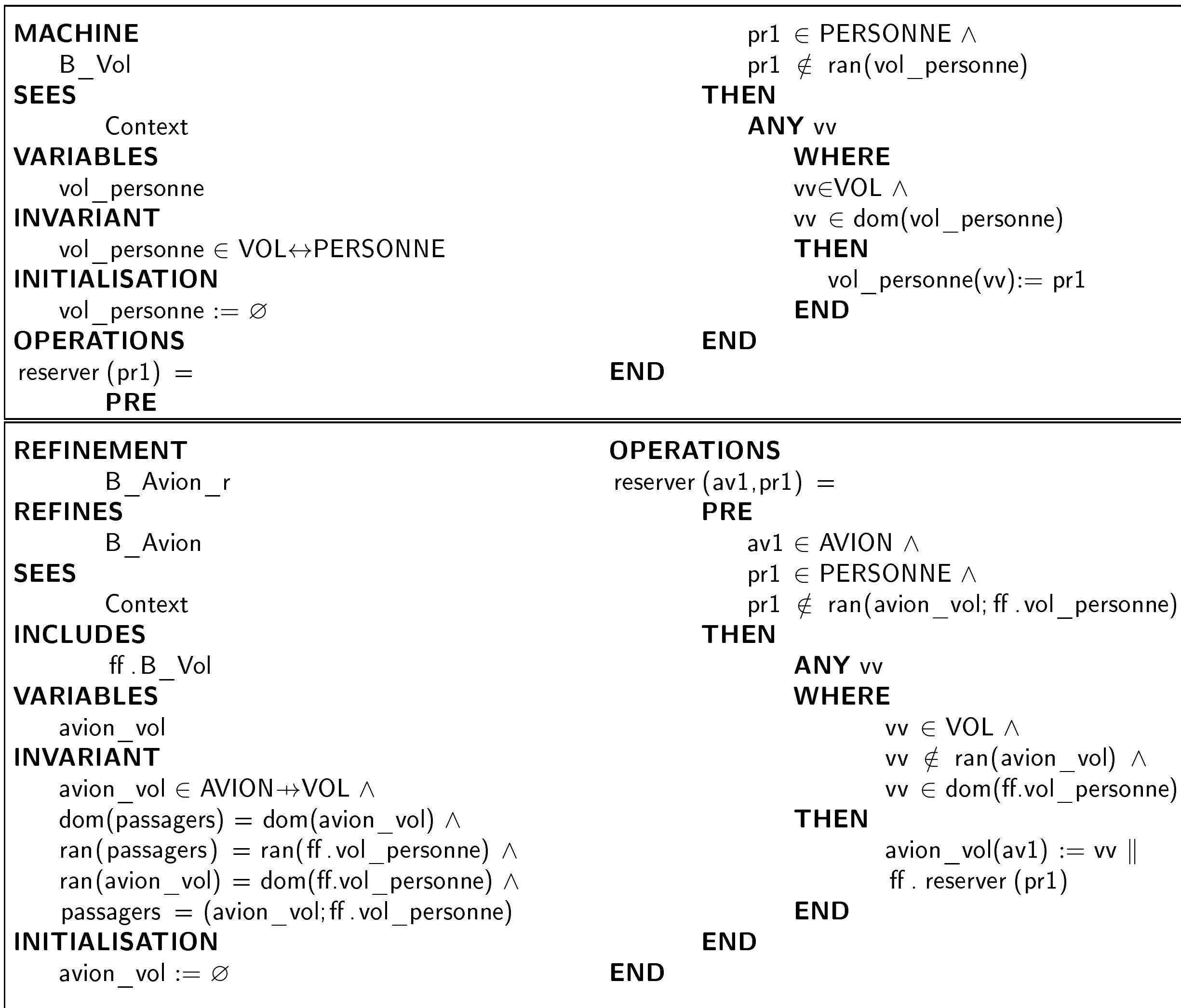
Les obligations de preuve liées à la machine `B_Avion_r` données dans le tableau 3.5 ont été déchargées automatiquement.

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	5	0	5	0	100
reserver	7	0	7	0	100
B_Avion_r	12	0	12	0	100

TABLE 3.5 – Tableau de l'état de la machine B_Avion_r

3.2.6 Voir aussi

Le pattern `Class_Helper` dépend de la nature de la relation liant les deux classes importantes `P1` et `P2` : généralisation, association, agrégation, composition et dépendance. En outre, la classe intermédiaire

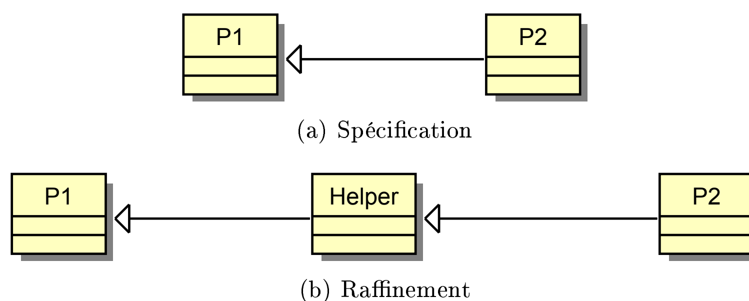


```
MACHINE
    B_Vol
SEES
        Context
VARIABLES
    vol_personne
INVARIANT
    vol_personne ∈ VOL↔PERSONNE
INITIALISATION
    vol_personne := ∅
OPERATIONS
 reserver (pr1) =
        PRE
            pr1 ∈ PERSONNE ∧
            pr1 ∉ ran(vol_personne)
        THEN
            ANY vv
                WHERE
                vv∈VOL ∧
                vv ∈ dom(vol_personne)
                THEN
                   vol_personne(vv):= pr1
                END
        END
END
```

```
REFINEMENT
        B_Avion_r
REFINES
        B_Avion
SEES
        Context
INCLUDES
        ff . B_Vol
VARIABLES
    avion_vol
INVARIANT
    avion_vol ∈ AVION⇸VOL ∧
    dom(passagers) = dom(avion_vol) ∧
    ran(passagers) = ran(ff . vol_personne) ∧
    ran(avion_vol) = dom(ff.vol_personne) ∧
    passagers = (avion_vol; ff . vol_personne)
INITIALISATION
    avion_vol := ∅
OPERATIONS
 reserver (av1,pr1) =
        PRE
            av1 ∈ AVION ∧
            pr1 ∈ PERSONNE ∧
            pr1 ∉ ran(avion_vol; ff . vol_personne)
        THEN
                ANY vv
                WHERE
                        vv ∈ VOL ∧
                        vv ∉ ran(avion_vol) ∧
                        vv ∈ dom(ff.vol_personne)
                THEN
                        avion_vol(av1) := vv ||
                        ff . reserver (pr1)
                END
        END
END
```

Figure 3.12 – Modélisation en **B** de l'état de spécification concrète

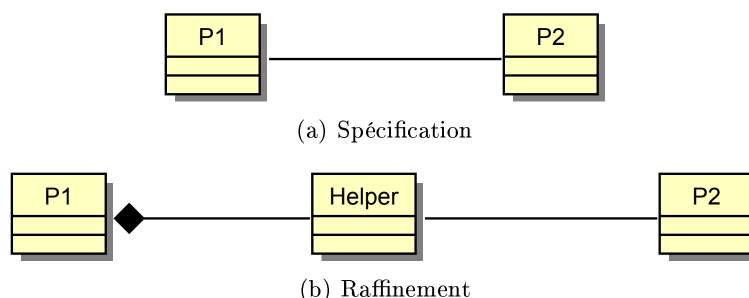
introduite `Helper` peut être reliée à `P1` et `P2` en utilisant la même nature de relation ou deux relations de nature différente (voir FIGURE 3.13 et FIGURE 3.14).



(a) Spécification

(b) Raffinement

FIGURE 3.13 – `Helper` reliée à `P1` et `P2` avec la même nature de relation



(a) Spécification

(b) Raffinement

FIGURE 3.14 – `Helper` reliée à `P1` et `P2` avec deux relations de nature différente

Le pattern `Class_Helper` peut être appliqué dans un ordre inverse de concret vers l'abstrait. Ce processus d'abstraction –par opposition au raffinement– peut être utilisé avec profit dans une activité de rétro-ingénierie ou encore rétro-conception.

3.3 Pattern de réification d'un attribut : `Class_Attribute`

3.3.1 Intention

Une classe englobante comporte un attribut modélisant une notion jugée intéressante et dotée des opérations bien définies. Ce pattern permet de réifier cet attribut en une classe `Class_Attribute`. Une relation de type agrégation est introduite entre la classe englobante –tout– et la classe réifiant l'attribut concerné –partie–.

3.3.2 Motivation

Pour des raison de simplification, à un certain niveau d'abstraction élevé, une notion peut être modélisée par un attribut. Ensuite, en fonction de détails issus du cahier des charges, la même notion peut être retenue comme classe. Ceci est justifié par l'identification des opérations bien définies applicables sur cette notion en analysant les détails introduits. Le type de l'attribut est plutôt discret : entier, énuméré ou alphanumérique.

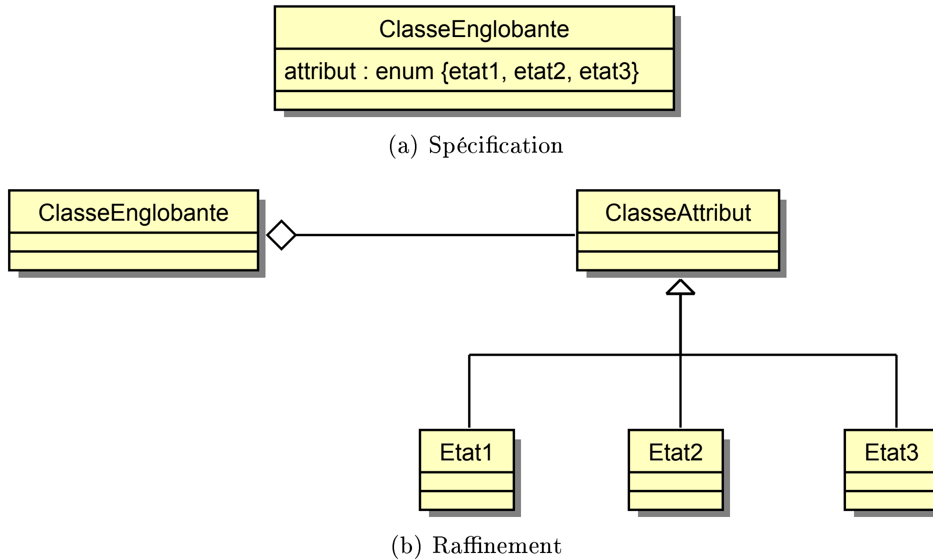
Par exemple, dans un système bancaire, la notion de compte peut être modélisée par un attribut appelé numéro de type alphanumérique appartenant à la classe `Banque`. De même, dans un système de gestion des bibliothèques, l'état d'un livre (classe `Livre`) peut être modélisé par un attribut énuméré : disponible ou emprunté.

3.3.3 Solution

Nous nous plaçons dans le cadre suivant :

- l'attribut à réifier est de type énuméré,
- les autres aspects statiques et dynamiques de la classe englobante ne sont pas pris en compte.

Les deux parties Spécification et Raffinement de ce pattern sont données dans la FIGURE 3.5.



(a) Spécification

(b) Raffinement

FIGURE 3.15 – Pattern de réification d'un attribut : `Class_Attribute`

3.3.4 Vérification

La FIGURE 3.16 présente la machine abstraite `B_Class_Attribute_a` correspondante à la transformation en **B** de la partie Spécification présentée par la FIGURE 3.15(a). Pour y parvenir, nous avons réutilisé avec profit les idées introduites lors de la formalisation en B du pattern précédent `Class_Helper`.

```
MACHINE
    B_Class_Attribute_a
SETS
    OBJECTS = {ce1, ce2, ce3, etat1, etat2, etat3}
ABSTRACT_CONSTANTS
    CLASS_ENGLOBANTE
PROPERTIES
    CLASS_ENGLOBANTE ⊆ OBJECTS ∧
    CLASS_ENGLOBANTE = {ce1, ce2, ce3}
VARIABLES
    class_englobante, attribut
INVARIANT
  class_englobante ⊆ CLASS_ENGLOBANTE ∧
  attribut ∈class_englobante↔{etat1, etat2, etat3}
INITIALISATION
  class_englobante := {ce1, ce2, ce3} ||
  attribut := {ce1↦etat1, ce2↦etat2, ce3↦etat3}
END
```

FIGURE 3.16 – Modélisation en **B** de l'état de spécification abstraite

Les obligations de preuve générées relatives à cette machine ont été toutes déchargées automatiquement (voir Tableau 3.6). Ceci valide en partie les propriétés invariantes liées à l'état de la machine `B_Class_Attribute_a`.

La FIGURE 3.17 présente la traduction en **B** de l'état de la partie Raffinement, donnée dans la FIGURE 3.15(b), du pattern `Class_Attribute`.

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	4	0	4	0	100
B_Class_Attribute_a	4	0	4	0	100

TABLE 3.6 – Tableau de l'état de la machine B_Class_Attribute_a

```
REFINEMENT
   B_Class_Attribute_r
REFINES
   B_Class_Attribute_a
ABSTRACT_CONSTANTS
    CLASS_ATTRIBUT
PROPERTIES
    CLASS_ATTRIBUT ⊆ OBJECTS ∧
    CLASS_ATTRIBUT = {etat1, etat2, etat3}
ABSTRACT_VARIABLES
   class_englobante ,
   class_attribut ,
   class_etat1,
   class_etat2,
   class_etat3,
   composition
INVARIANT
   class_attribut  ⊆ CLASS_ATTRIBUT ∧
   composition∈class_englobante↔class_attribut ∧
   class_etat1 ⊆ class_attribut ∧
   class_etat2 ⊆ class_attribut ∧
   class_etat3 ⊆ class_attribut ∧
   class_etat1∩class_etat2∩class_etat3=∅ ∧
   class_etat1∪class_etat2∪
              class_etat3 = class_attribut ∧
   /∗ invariant de collage∗/
   composition = attribut
INITIALISATION
   class_englobante := {ce1, ce2, ce3} ||
   class_attribut := {etat1, etat2, etat3} ||
   class_etat1 := {etat1} ||
   class_etat2 := {etat2} ||
   class_etat3 := {etat3} ||
   composition := {ce1↦etat1, ce2↦etat2,
              ce3↦etat3}
END
```

FIGURE 3.17 – Modélisation en **B** de l'état de spécification concrète

La machine `B_Class_Attribute_r` introduit une constante abstraite `CLASS_ATTRIBUT` pour mémoriser les objets potentiels de type `ClasseAttribut`. L'état variable de la machine `B_Class_Attribute_r` est défini par les six variables `class_etat1`, `class_etat2`, `class_etat3`, `composition`, `class_attribut` et `class_englobante`. La variable `class_englobante` provient de la machine `B_Class_Attribute_a`. La variable abstraite `attribut` a été supprimée. Tandis que les cinq autres variables sont introduites par la partie Raffinement du pattern. La clause INVARIANT de la machine `B_Class_Attribute_r` définit le typage et les contraintes liées aux variables concrètes `class_attribut`, `class_etat1`, `class_etat2`, `class_etat3` et `composition`. Egalement, elle comporte l'invariant de collage suivant :

`attribut = composition` **(inv)** $\quad (inv_collage_Class_Attribute) \quad (3.2)$

Cet invariant de collage garantit la correction du pattern de raffinement. Les obligations de preuve générées relatives à la machine `B_Class_Attribute_r` ont été toutes déchargées automatiquement (voir Tableau 3.7). Ceci valide en partie les propriétés invariantes, notamment l'invariant de collage, liées à l'état de la machine `B_System_r`.

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	7	0	7	0	100
B_Class_Attribute_r	7	0	7	0	100

TABLE 3.7 – Tableau de l'état de la machine B_Class_Attribute_r

3.3.5 Exemple

A titre d'exemple, la FIGURE 3.18 décrit en partie un système de gestion des bibliothèque[1]. Un tel système est modélisé par une classe `Livre` caractérisée par un `etat`, qui peut prendre la valeur disponible ou emprunté, et contenant trois opérations `emprunter`, `rendre` et `est_il_disponible`. Les contraintes OCL décrivent la sémantique des différentes opérations : `emprunter` pour emprunter un livre, `rendre` pour rendre un livre et `est_il_disponible` pour voir si un livre est disponible ou non.

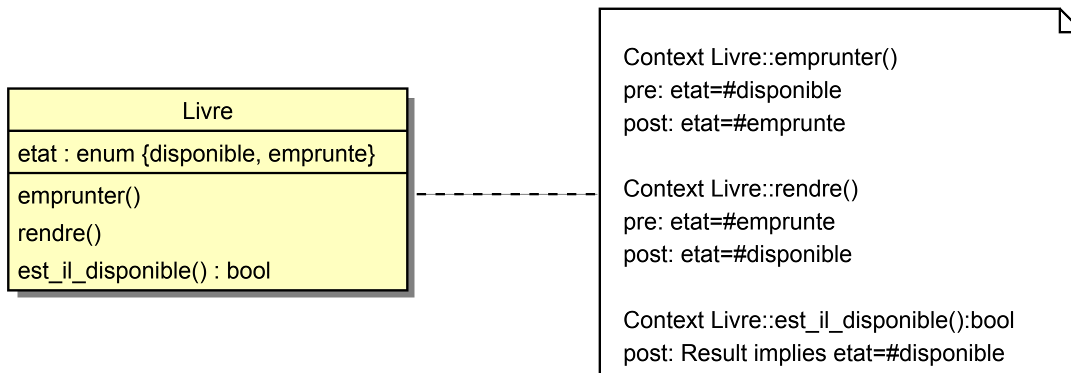


FIGURE 3.18 – Classe `Livre`

Application du pattern `Class_Attribute`

Un raffinement à cette classe peut être envisagé comme suit : l'état d'un livre est retenu comme classe indépendante. En effet, les trois opérations `emprunter`, `rendre` et `est_il_disponible` peuvent

1. Nous nous sommes inspirés de la modélisation aussi bien en UML qu'en B de cet exemple fournie en [42].

être considérées comme des opérations propres à cette classe. Pour cela, nous réutilisons avec profit le pattern de réification d'un attribut : `Class_Attribut`. Ici, l'attribut à réifier est `etat` ayant deux positions. L'application de notre pattern `Class_Attribut` sur le modèle UML donné par la FIGURE 3.18 génère le modèle UML exhibé par la FIGURE 3.19. Les services offerts par la classe `Livre` sont **délégués** à la classe abstraite `Etat` via la relation d'agrégation entre `Livre` et `Etat`. Celle-ci admet des classes descendantes effectives `Disponible` et `Emprunte` permettant d'implémenter ou de définir les méthodes venant de la classe ascendante `Etat`. Enfin, les contraintes OCL attachées aux classes `Livre`, `Etat`, `Disponible` et `Emprunte` du modèle UML donné par la FIGURE 3.19 expriment des propriétés invariantes (clause inv) et la sémantique des opérations `emprunter`, `rendre` et `est_il_disponible`.

Vérification de l'application du pattern

Afin de formaliser en **B** le raffinage entre la spécification et le raffinement fournis respectivement par les deux FIGURES 3.18 et 3.19, nous proposons l'architecture des machines **B** présentée par la FIGURE 3.20.

Les deux machines `Context` et `B_Livre` formalisent en **B** la classe abstraite `Livre` fournie par la FIGURE 3.18. La machine `Context` (voir FIGURE 3.21) factorise l'ensemble abstrait LIVRE utile pour les deux niveaux : spécification et raffinement.

Quant à la machine `B_Livre` (voir FIGURE 3.21), elle regroupe les caractéristiques `emprunter`, `rendre` et `est_il_disponible` venant du modèle UML de la FIGURE 3.18. La sémantique pré/post en OCL des méthodes de la classe `Livre` sont préservées par les opérations de même noms appartenant à la machine `B_Livre`.

Les obligations de preuve liées à la machine `B_Livre` sont données dans le tableau 3.8. Elles ont été déchargées automatiquement.

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	3	0	3	0	100
emprunter	2	0	2	0	100
rendre	2	0	2	0	100
est_il_disponible	0	0	0	0	100
B_Livre	7	0	7	0	100

TABLE 3.8 – Tableau de l'état de la machine `B_Livre`

Les trois machines `B_Livre_r`, `B_Disponible` et `B_Emprunte` (voir FIGURE 3.22) formalisent en **B** le niveau Raffinement présenté par la FIGURE 3.19. Les machines `B_Disponible` et `B_Emprunte` formalisent, en **B**, respectivement les classes `Disponible` et `Emprunte` du modèle UML présenté par la même figure. Elle permettent de déclarer respectivement deux variables `livre_disponible` et `livre_emprunte` modélisant les livres disponibles et empruntés. La machine `B_Livre_r` raffine `B_Livre` en incluant une instance de la machine `B_Disponible` et une instance de la machine `B_Emprunte` (voir la clause INCLUDES). L'invariant de la machine `B_Livre_r` est issue de l'invariant de collage du pattern `Class_Attribute` (Inv_Collage_Class_Attribute) (3.2). La variable `etat` est remplacée par `ll1.livre_disponible` et `ll2.livre_emprunte`.

Les obligations de preuve données dans les tableaux 3.9 et 3.10 ont été déchargées automatiquement.

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	1	0	1	0	100
emprunter	1	0	1	0	100
rendre	0	0	0	0	100
est_il_disponible	0	0	0	0	100
B_Disponible	2	0	2	0	100

TABLE 3.9 – Tableau de l'état de la machine `B_Disponible`

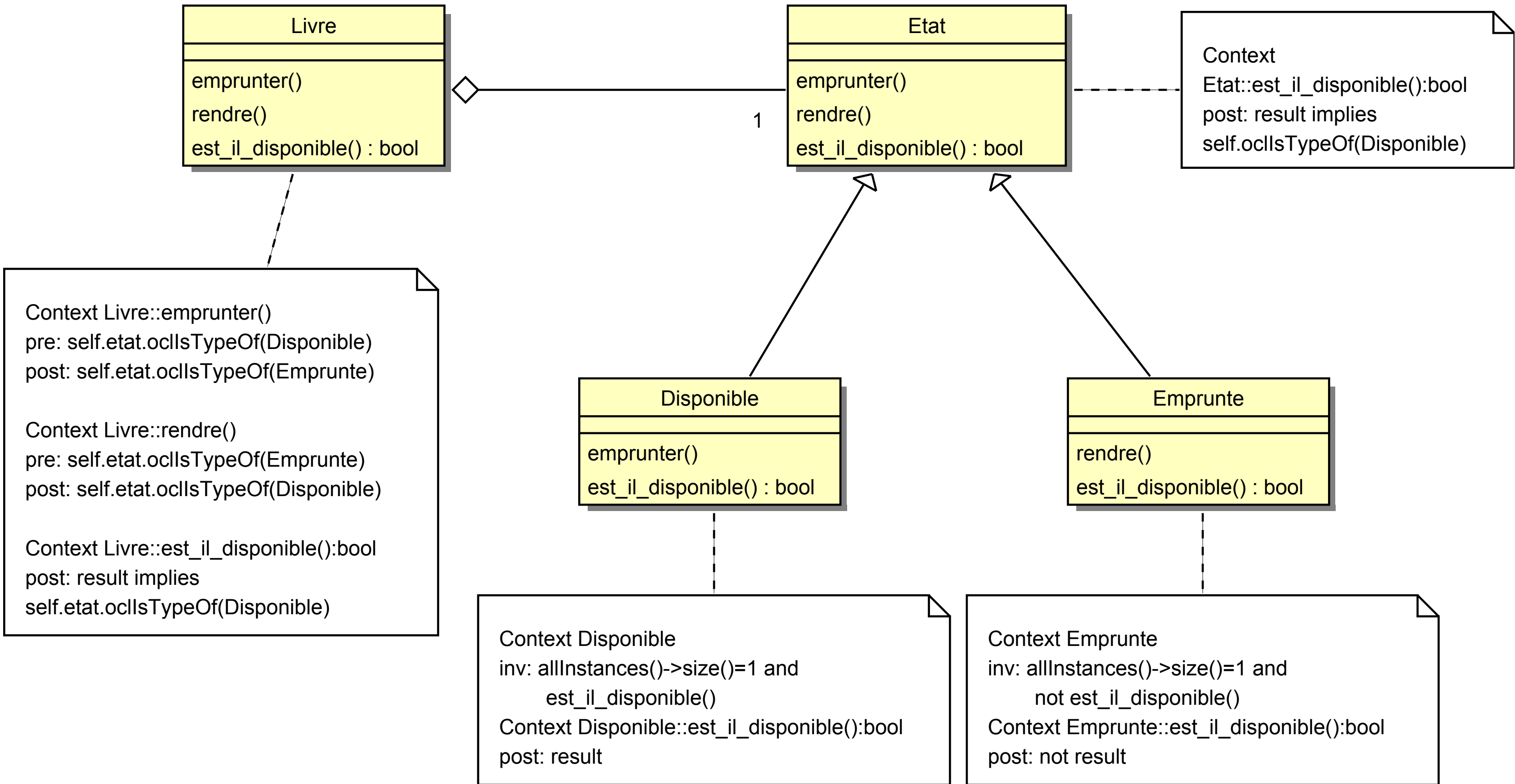


FIGURE 3.19 – Réification de l'attribut `etat`

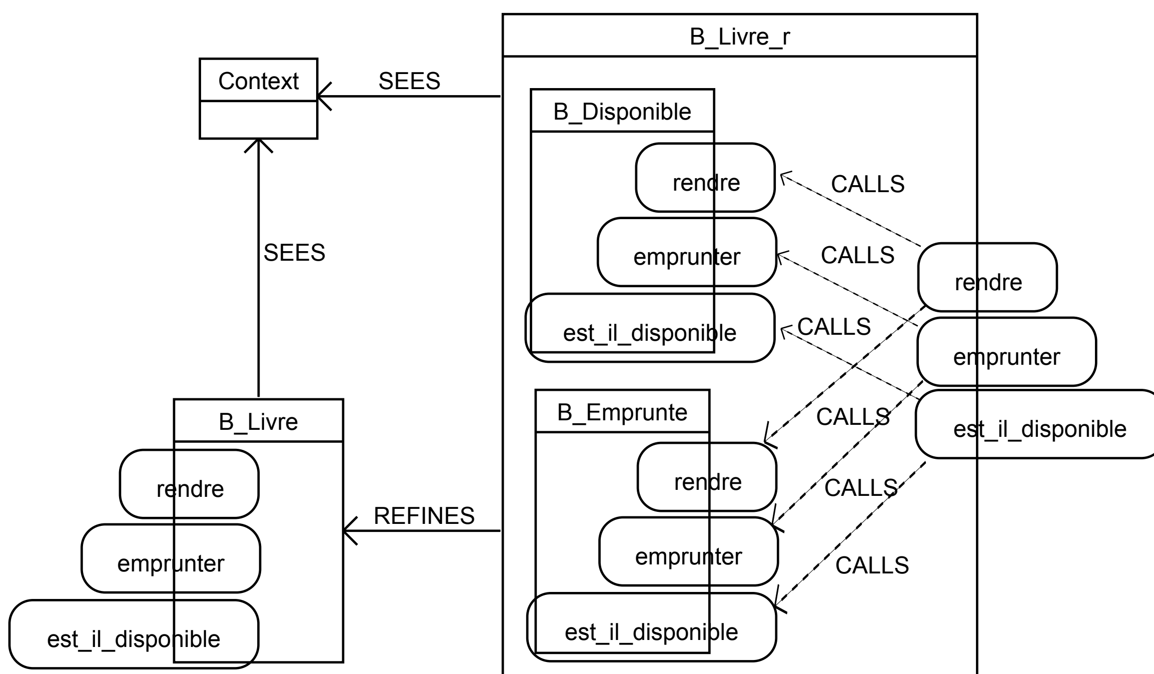


FIGURE 3.20 – Dépendances des machines `Context`, `B_Livre`, `B_Livre_r`, `B_Disponible` et `B_Emprunte`

```
MACHINE
    Context
SETS
    LIVRE
END
```

```
MACHINE
    B_Livre
SEES
    Context
SETS
    ETAT = {disponible, emprunte}
VARIABLES
    etat , livre
INVARIANT
    livre ⊆ LIVRE ∧
    etat ∈ livre → ETAT
INITIALISATION
    livre := ∅ ||
    etat := ∅
OPERATIONS
    emprunter( ll ) =
    PRE
        ll ∈ livre ∧
        etat( ll ) = disponible
    THEN
        etat( ll ) := emprunte
    END;
    rendre( ll ) =
    PRE
        ll ∈ livre ∧
        etat( ll ) = emprunte
    THEN
        etat( ll ) := disponible
    END;
    res ⟵ est_il_disponible( ll ) =
    PRE
        ll ∈ livre ∧ res ∈ 𝔹
    THEN
        res := bool(etat(ll) = disponible)
    END
END
```

FIGURE 3.21 – Modélisation en **B** de l'état de spécification abstraite

```
MACHINE
    B_Disponible
SEES
    Context
VARIABLES
    livre_disponible
INVARIANT
    livre_disponible ⊆ LIVRE
INITIALISATION
    livre_disponible := ∅
OPERATIONS
    emprunter(ll) = PRE
        ll ∈ LIVRE ∧ ll ∈ livre_disponible
    THEN
        livre_disponible := livre_disponible−{ll}
    END;
    rendre(ll) = PRE
        ll ∈ LIVRE ∧ ll ∉ livre_disponible
    THEN
        livre_disponible := livre_disponible∪{ll}
    END;
    res ⟵ est_il_disponible(ll) = PRE
        ll ∈ LIVRE ∧ res ∈ 𝔹
    THEN
        res := bool(ll ∈ livre_disponible)
    END
END
```

```
MACHINE
    B_Emprunte
SEES
    Context
VARIABLES
    livre_emprunte
INVARIANT
    livre_emprunte ⊆ LIVRE
INITIALISATION
    livre_emprunte := ∅
OPERATIONS
    emprunter(ll) = PRE
        ll ∈ LIVRE ∧ ll ∉ livre_emprunte
    THEN
        livre_emprunte := livre_emprunte ∪ {ll}
    END;
    rendre(ll) = PRE
        ll ∈ LIVRE ∧ ll ∈ livre_emprunte
    THEN
        livre_emprunte := livre_emprunte − {ll}
    END;
    res ⟵ est_il_disponible(ll) = PRE
        ll ∈ LIVRE ∧ res ∈ 𝔹
    THEN
        res := bool(ll ∉ livre_emprunte)
    END
END
```

```
REFINEMENT
    B_Livre_r
REFINES
    B_Livre
SEES
    Context
INCLUDES
    ll1.B_Disponible, ll2.B_Emprunte
INVARIANT
    ll1.livre_disponible ∩ ll2.livre_emprunte=∅ ∧
    ll1.livre_disponible ∪ ll2.livre_emprunte=livre ∧
    etat⁻¹[{disponible}] = ll1.livre_disponible ∧
    etat⁻¹[{emprunte}] = ll2.livre_emprunte
OPERATIONS
    emprunter(ll) = PRE
        ll ∈ LIVRE ∧
        ll ∉ ll2.livre_emprunte ∧
        ll ∈ ll1.livre_disponible
    THEN
        ll2.emprunter(ll) || ll1.emprunter(ll)
    END;
    rendre(ll) = PRE
        ll ∈ LIVRE ∧
        ll ∉ ll1.livre_disponible ∧
        ll ∈ ll2.livre_emprunte
    THEN
        ll1.rendre(ll) || ll2.rendre(ll)
    END;
    res ⟵ est_il_disponible(ll) = PRE
        ll ∈ LIVRE
    THEN
        res ⟵ ll2.est_il_disponible(ll)
    END
END
```

Figure 3.22 – Modélisation en **B** de l'état de spécification concrète

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	1	0	1	0	100
emprunter	0	0	0	0	100
rendre	1	0	1	0	100
est_il_disponible	0	0	0	0	100
B_Emprunte	2	0	2	0	100

TABLE 3.10 – Tableau de l'état de la machine B_Emprunte

Les obligations de preuve liées à la machine B_Livre_r sont données dans le tableau 3.11. Elles sont au nombre de 19 dont 17 déchargées d'une façon automatiquement et 2 d'une façon interactive.

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	3	0	3	0	100
emprunter	7	1	6	0	100
rendre	7	0	7	0	100
est_il_disponible	2	1	1	0	100
B_Livre_r	19	2	17	0	100

TABLE 3.11 – Tableau de l'état de la machine B_Livre_r

3.3.6 Voir aussi

L'idée de réification d'un attribut peut être utilisée avec profit dans une activité de restructuration (ou refactoring) des modèles OO existants. D'ailleurs, dans le chapitre 6, nous proposons un schéma de refactoring basé sur la réification d'un attribut : introduction de la notion de délégation.

3.4 Pattern d'enrichissement d'une association : Class_Association

3.4.1 Intention

Ce pattern permet d'**augmenter** le pouvoir d'une association en la considérant à la fois comme classe et association : classe associative. Ceci peut être justifié par l'émergence des *détails* propres à l'association. En effet, de tels détails ne peuvent être attachés aux classes reliées par l'association. La classe associative ajoutée ne peut exister que si l'association existe.

3.4.2 Motivation

Parfois, une association doit posséder des propriétés. Ces propriétés ne peuvent pas être rattachées aux extrémités de cette association. Par exemple, prenons le diagramme de classes composé de l'association emploi et ses deux extrémités Société et Personne (voir FIGURE 3.23).

FIGURE 3.23 – Classes Société et Personne

L'association emploi possède comme propriétés : salaire et dateEmbauche. En effet, ces deux propriétés ne peuvent pas être attachées ni à la classe Société, qui emploie plusieurs personnes, ni à la classe

`Personne`, qui peuvent avoir plusieurs emplois. Il s'agit donc de propriétés de l'association `Emploi`. Comme les associations ne pouvant posséder de propriété, il faut introduire le concept de classe associative, pour modéliser cette situation (voir FIGURE 3.24).

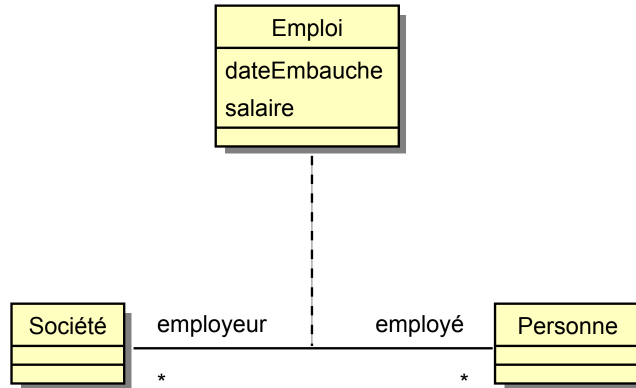
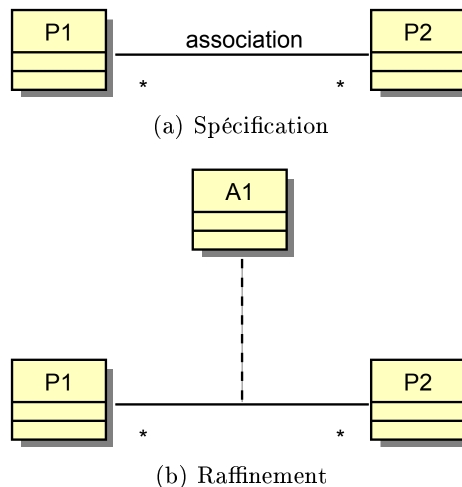


FIGURE 3.24 – Adjonction de la classe associative `Emploi`

3.4.3 Solution

Nous nous plaçons dans le cadre suivant : les propriétés (attributs et opérations) de `P1` et `P2` ne sont pas prises en compte.

Les deux parties Spécification et Raffinement de ce pattern sont données dans la FIGURE 3.25.



(a) Spécification

(b) Raffinement

FIGURE 3.25 – Pattern d'enrichissement d'une association : `Class_Association`

3.4.4 Vérification

La FIGURE 3.26 présente la machine abstraite `B_Class_Association_a` correspondante à la transformation en **B** de la partie Spécification présentée par la FIGURE 3.25(a).

La machine abstraite `B_Class_Association_a` possède une structure similaire à celles représentant les deux patterns précédents `Class_Helper` et `Class_Attribute`. Les obligations de preuve générées relatives à cette machine ont été toutes déchargées automatiquement (voir Tableau 3.12). Ceci valide en partie les propriétés invariantes liées à l'état de la machine `B_Class_Association_a`.

```
MACHINE
    B_Class_Association_a
SETS
    OBJECTS = {p11, p12, p21,
               p22, p23, p24,
               a11, a22, a33, a44}
ABSTRACT_CONSTANTS
    P1, P2
PROPERTIES
    P1 ⊆ OBJECTS ∧
    P2 ⊆ OBJECTS ∧
    P1 ∩ P2 = ∅ ∧
    P1 = {p11, p12} ∧
    P2 = {p21, p22, p23, p24}
VARIABLES
    p1, p2, association
INVARIANT
    p1 ⊆ P1 ∧
    p2 ⊆ P2 ∧
    association ∈ p1 ↔ p2
INITIALISATION
    p1 := {p11,p12} ||
    p2 := {p21, p22, p23, p24} ||
    association := {p11↦p21, p11↦p22,
                    p12↦p23, p12↦p24}
END
```

FIGURE 3.26 – Modélisation en **B** de l'état de spécification abstraite

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	7	0	7	0	100
B_Class_Association_a	7	0	7	0	100

TABLE 3.12 – Tableau de l'état de la machine B_Class_Association_a

La FIGURE 3.27 présente la traduction en **B** de l'état de la partie Raffinement du pattern donnée dans la FIGURE 3.25(b).

```
REFINEMENT
   B_Class_Association_r
REFINES
   B_Class_Association_a
ABSTRACT_CONSTANTS
    A1
PROPERTIES
    A1 ⊆ OBJECTS ∧
    A1 ∩ P1 = ∅ ∧
    A1 ∩ P2 = ∅ ∧
    A1 = {a11, a22, a33, a44}
ABSTRACT_VARIABLES
   p1, p2, a1, rr1 , rr2
INVARIANT
    a1 ⊆ A1 ∧
    rr1 ∈ a1 → p1 ∧
    rr2 ∈ a1 → p2 ∧
    rr1 × rr2 ∈ a1 ↣ p1×p2 ∧
    /∗ Invariant de collage∗/
    ran(rr1 × rr2) = association
INITIALISATION
   p1 := { p11 , p12} ||
   p2 := { p21 , p22 , p23 , p24} ||
   a1 := { a11, a22, a33 , a44} ||
   rr1 := {a11 ↦ p11, a22 ↦ p11,
           a33 ↦ p12, a44 ↦ p12} ||
   rr2 := {a11 ↦ p21, a22 ↦ p22,
           a33 ↦ p23, a44 ↦ p24}
END
```

FIGURE 3.27 – Modélisation en **B** de l'état de spécification concrète

La machine `B_Class_Association_r` introduit une constante abstraite `A1` pour mémoriser les objets potentiels de type `A1`. La clause PROPERTIES stipule que la constante `A1` et les constantes `P1` et `P2` sont deux à deux disjointes. L'état variable de la machine `B_Class_Association_r` est défini par les cinq variables `p1`, `p2`, `a1`, `rr1` et `rr2`. Les deux variables `p1` et `p2` proviennent de la machine abstraite `B_Class_Association_a`. Tandis que les trois autres variables sont introduites par la partie Raffinement du pattern `Class_Association`. La variable abstraite `association` a été supprimée. La clause INVA-

RIANT de la machine `B_Class_Association_r` définit le typage et les contraintes liées aux variables concrètes `a1`, `rr1` et `rr2`. Egalement, elle comporte l'invariant de collage suivant :

`ran(rr1 × rr2) = association` **(inv)**

$$(inv_collage_Class_Association) \quad (3.3)$$

Cet invariant de collage garantit la correction du pattern de raffinement `Class_Association`. Il pourrait être instancié et réutilisé avec profit dans un développement conjoint UML/B guidé par des patterns de raffinement.

Les obligations de preuve générées relatives à la machine `B_Class_Association_r` ont été déchargées dont trois de façon interactive en utilisant les deux prouveurs interactifs `ss` et `pr` (voir Tableau 3.13). Ceci valide en partie les propriétés invariantes, notamment l'invariant de collage, liées à l'état de la machine `B_Class_Association_r`.

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	12	3	9	0	100
B_Class_Association_r	12	3	9	0	100

TABLE 3.13 – Tableau de l'état de la machine B_Class_Association_r

3.4.5 Exemple

Prenons l'exemple suivant : `Professeur` modélise l'ensemble des professeurs ; `Matière` modélise l'ensemble des matières. Il existe un lien entre ces deux classes modélisant le fait qu'un professeur enseigne une ou plusieurs matières et qu'une matière est enseignée par un ou plusieurs professeurs. Ceci est donnée par la FIGURE 3.28.

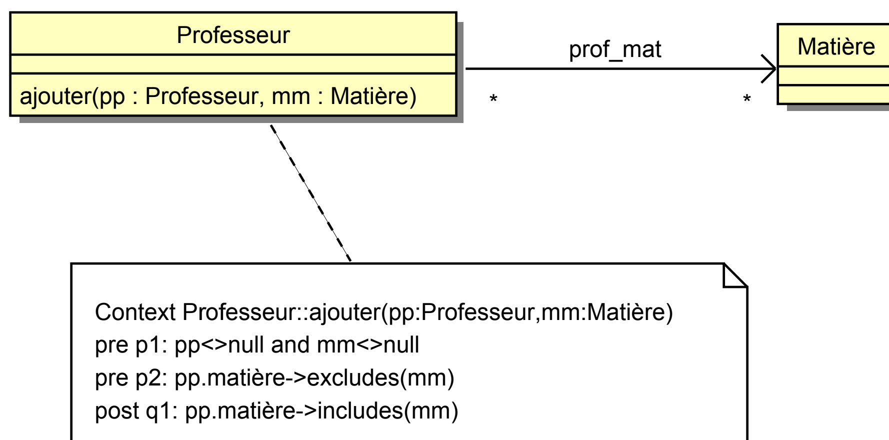


FIGURE 3.28 – Un premier diagramme de classes

Application du pattern `Class_Association`

Dans une étape plus avancée du développement, nous remarquons la nécessité d'ajouter les détails suivants : le jour, l'heure de début et de fin de leçon. En effet, de tels détails n'appartiennent ni à la classe `Professeur` ni à la classe `Matière`. Ainsi, nous pouvons réutiliser le pattern d'enrichissement d'association : `Class_Association` pour introduire une classe associative appelée `Enseigne` (voir FIGURE 3.29).

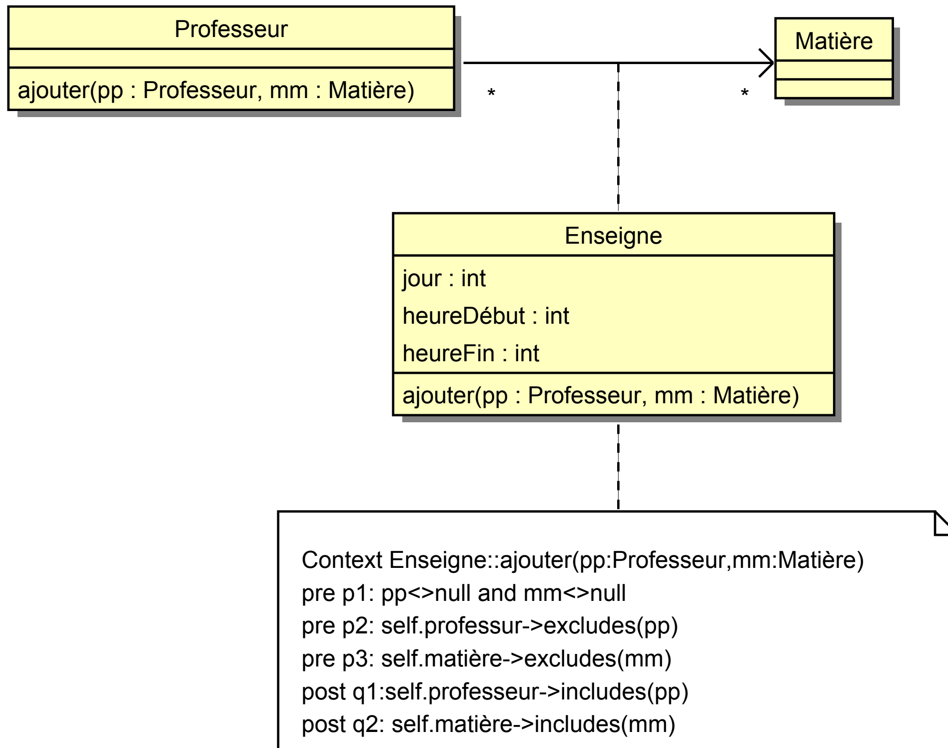


FIGURE 3.29 – Adjonction de la classe associative `Enseigne`

Vérification de l'application du pattern

Afin de formaliser en **B** le raffinage entre la spécification et le raffinement fournis respectivement par les deux FIGURES 3.28 et 3.29, nous proposons l'architecture des machines **B** présentée par la FIGURE 3.30.

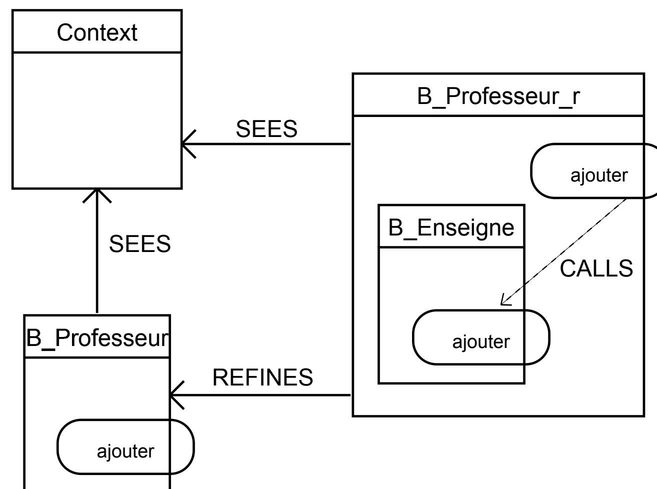


FIGURE 3.30 – Dépendances des machines `Context`, `B_Professeur`, `B_Professeur_r` et `B_Enseigne`

Les deux machines `Context` et `B_Professeur` formalisent en **B** la relation abstraite `prof_mat` fournie

par la FIGURE 3.28. La machine `Context` (voir FIGURE 3.31) factorise des ensembles abstraits PROFESSEUR, MATIERE et ENSEIGNE utiles pour les deux niveaux : spécification et raffinement.

Quant à la machine `B_Professeur` (voir FIGURE 3.31), elle introduit l'opération `ajouter` venant du modèle UML de la FIGURE 3.28. La sémantique pré/post en OCL de la méthode `ajouter` appartenant à la classe `Professeur` est préservée par l'opération de même nom appartenant à la machine `B_Professeur`.

```
MACHINE
    Context
SETS
    PROFESSEUR;MATIERE;ENSEIGNE
END
```

```
MACHINE
    B_Professeur
SEES
        Context
VARIABLES
    prof_mat,
    professeur ,
    matiere
INVARIANT
    professeur ⊆ PROFESSEUR ∧
    matiere ⊆ MATIERE ∧
    prof_mat ∈ professeur ↔ matiere
INITIALISATION
    professeur := ∅ ||
    matiere := ∅ ||
    prof_mat := ∅
OPERATIONS
    ajouter (pp,mm)=
    PRE
        pp ∈ professeur ∧
        mm ∈ matiere
        THEN
        prof_mat(pp) := mm
        END
END
```

FIGURE 3.31 – Modélisation en **B** de l'état de spécification abstraite

Les obligations de preuve liées à la machine `B_Professeur` sont données dans le tableau 3.14. Elles ont été déchargées automatiquement.

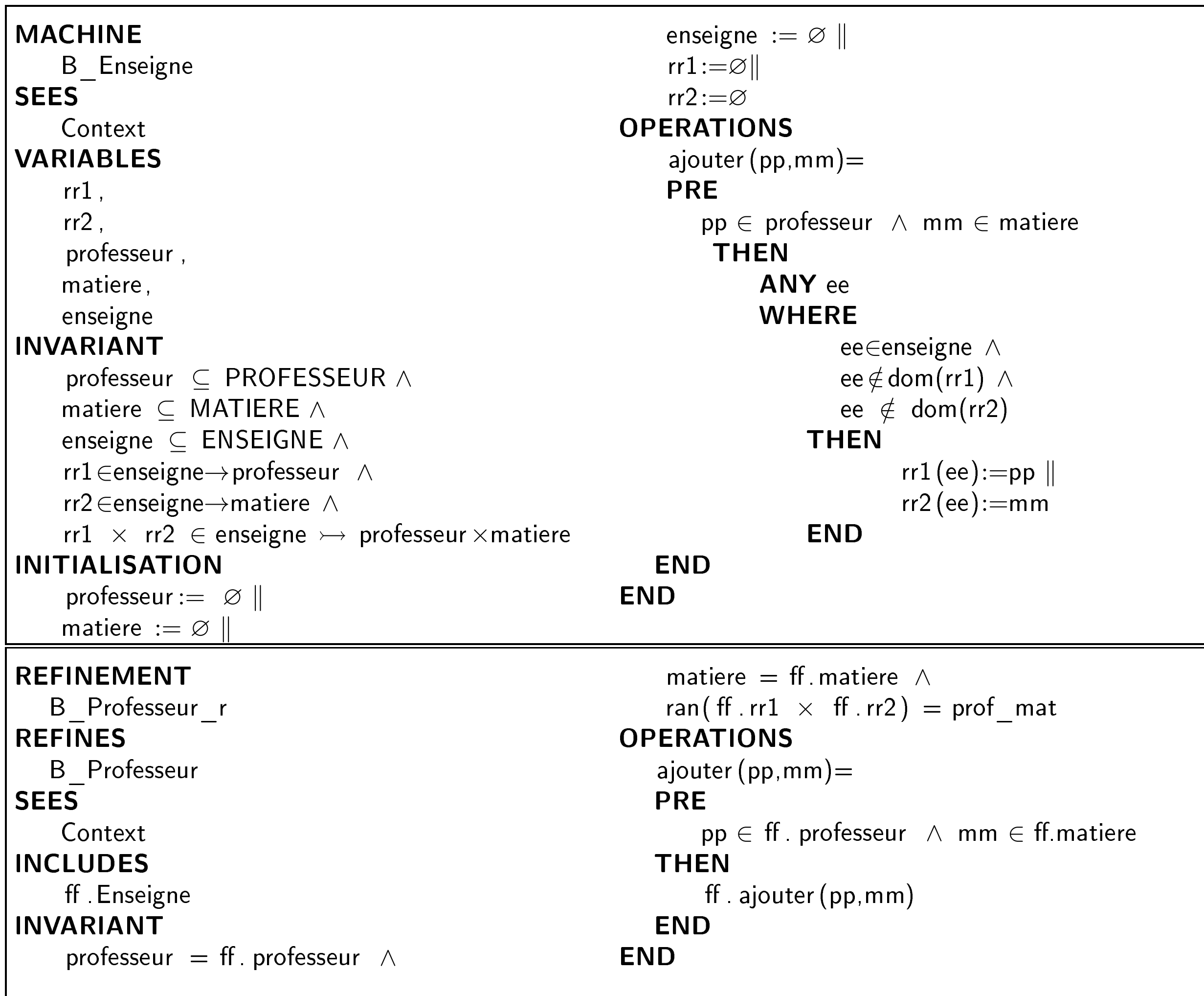
	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	3	0	3	0	100
reserver	1	0	1	0	100
B_Professeur	4	0	4	0	100

TABLE 3.14 – Tableau de l'état de la machine `B_Professeur`

Les deux machines `B_Enseigne` et `B_Professeur_r` (voir FIGURE 3.32) formalisent en **B** le raffinement de la relation abstraite `prof_mat` fourni par la FIGURE 3.29. La machine `B_Enseigne` formalise en **B** la classe d'association `Enseigne` du modèle UML présenté par la même figure. Elle introduit les variables `rr1` et `rr2` (fonctions totales) permettant de modéliser respectivement les relations entre `Enseigne` et `Professeur` et entre `Enseigne` et `Matière`. En effet, le produit direct de ces deux fonctions doit correspondre à une injection totale entre `Enseigne` et le couple `Professeur` et `Matière`. L'opération `ajouter` de la machine `B_Enseigne` est spécifiée en tenant compte de la sémantique pré/post en OCL de la méthode `ajouter` de la classe `Enseigne` de la FIGURE 3.29. La machine `B_Professeur_r` raffine en incluant une instance de la machine `B_Enseigne` (voir la clause INCLUDES). L'invariant de la machine `B_Profeseur_r` est issu de l'invariant de collage du pattern `Class_Association` (Inv_Collage_Class_Association) (3.5).

Les obligations de preuve liées à la machine `B_Enseigne` données dans le tableau 3.15 ont été déchargées automatiquement.

Les obligations de preuve liées à la machine `B_Professeur_r` sont données dans le tableau 3.16. Elles ont été déchargées automatiquement.



```
MACHINE
    B_Enseigne
SEES
    Context
VARIABLES
    rr1 ,
    rr2 ,
    professeur ,
    matiere ,
    enseigne
INVARIANT
    professeur ⊆ PROFESSEUR ∧
    matiere ⊆ MATIERE ∧
    enseigne ⊆ ENSEIGNE ∧
    rr1 ∈enseigne→professeur ∧
    rr2 ∈enseigne→matiere ∧
    rr1 × rr2 ∈ enseigne ↣ professeur ×matiere
INITIALISATION
    professeur := ∅ ||
    matiere := ∅ ||
    enseigne := ∅ ||
    rr1:=∅||
    rr2:=∅
OPERATIONS
    ajouter (pp,mm)=
    PRE
       pp ∈ professeur ∧ mm ∈ matiere
     THEN
         ANY ee
         WHERE
             ee∈enseigne ∧
             ee∉dom(rr1) ∧
             ee ∉ dom(rr2)
         THEN
                 rr1 (ee):=pp ||
                 rr2 (ee):=mm
         END
    END
END
```

```
REFINEMENT
    B_Professeur_r
REFINES
    B_Professeur
SEES
    Context
INCLUDES
    ff . Enseigne
INVARIANT
    professeur = ff . professeur ∧
    matiere = ff . matiere ∧
    ran( ff . rr1 × ff . rr2 ) = prof_mat
OPERATIONS
    ajouter (pp,mm)=
    PRE
       pp ∈ ff . professeur ∧ mm ∈ ff.matiere
    THEN
       ff . ajouter (pp,mm)
    END
END
```

FIGURE 3.32 – Modélisation en **B** de l'état de spécification concrète

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	8	0	8	0	100
ajouter	7	0	7	0	100
B_Enseigne	15	0	15	0	100

TABLE 3.15 – Tableau de l'état de la machine B_Enseigne

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	1	0	1	0	100
ajouter	3	0	3	0	100
B_Professeur_r	4	0	4	0	100

TABLE 3.16 – Tableau de l'état de la machine B_Professeur_r

3.4.6 Voir aussi

La partie Spécification du pattern `Class_Helper` est identique à celle du pattern `Class_Association`. Mais, les deux parties Raffinement de ces deux patterns sont différentes.

3.5 Pattern de décomposition d'une classe : `Class_Decomposition`

3.5.1 Intention

Il permet de **détailler** les responsabilités d'une classe originale par l'introduction des nouvelles classes. La classe originale et les classes résultantes sont reliées par des relations de généralisation (héritage). Le nombre des classes résultantes est au moins égal à un. Ce pattern favorise une approche de modélisation descendante.

La relation de généralisation couvre principalement les deux cas suivants [72] :

Héritage de sous-type : Vous êtes en train de modéliser un système externe dans lequel une catégorie d'objets (externes) peut être décomposée en sous catégories disjointes.
Nous incitons pour que le parent, A, soit retardé de sorte qu'il décrive un ensemble pas entièrement spécifié d'objets. L'héritier B peut être effectif ou retardé.

Héritage de restriction : L'héritage de restriction s'applique si les instances de B sont, parmi les instances de A, celles qui vérifient une contrainte, exprimée, dans l'invariant de B et absente de l'invariant de A. A et B devraient être toutes deux retardées ou toutes deux effectives.

3.5.2 Motivation

Héritage de sous-type :

Dans une approche de modélisation descendante, une personne peut être modélisée par une classe `Personne` caractérisée par un nom, un prénom et une adresse (voir Figure 3.33).

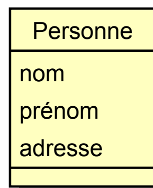


Figure 3.33 – Classe `Personne`

Un raffinement consiste à dire qu'une personne peut avoir un salaire si ce dernier est un salarié. Ceci revient à dire qu'un salarié est considéré comme une spécialisation de la classe `Personne` (voir Figure 3.34).

Héritage de restriction :

De la même façon, nous pouvons représenter l'ensemble des rectangles par la classe `Rectangle`(voir Figure 3.35).

La classe `Rectangle` pourra modéliser les deux ensembles : rectangles et carrés. En effet, l'ensemble des carrés est inclus dans l'ensemble des rectangles. Un raffinement consiste à dire que l'ensemble des carrés vérifie une contrainte absente dans l'invariant de la classe `Rectangle`, disant que la largeur et la longueur d'un carré sont égaux (voir Figure 3.36).

.

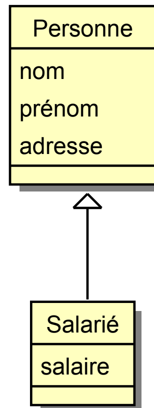


FIGURE 3.34 – Décomposition de la classe `Personne`

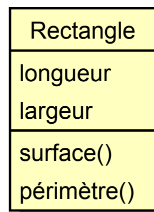


FIGURE 3.35 – Classe `Rectangle`

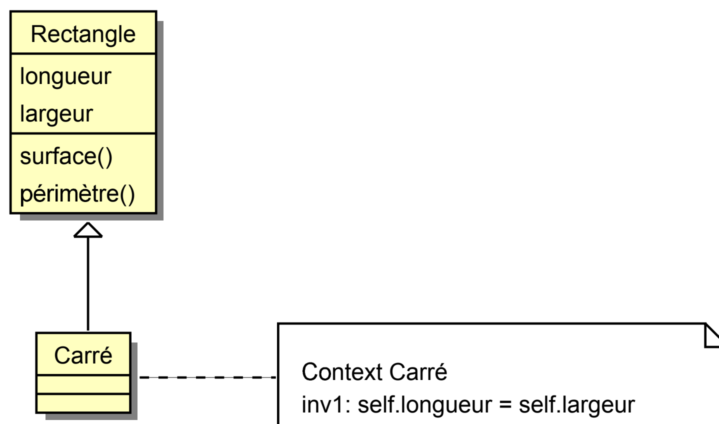


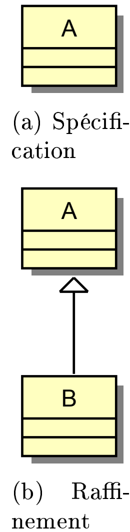
FIGURE 3.36 – Décomposition de la classe `Rectangle`

3.5.3 Solution

Nous traitons seulement le cas de l'héritage de restriction.

La classe B apporte une nouvelle propriété invariante liée à l'état statique hérité de la classe ascendante A.

Les deux parties Spécification et Raffinement de ce pattern sont données dans la FIGURE 3.37.



(a) Spécification

(b) Raffinement

FIGURE 3.37 – Pattern de décomposition d'une classe

3.5.4 Vérification

La FIGURE 3.38 présente la machine abstraite `B_Class_Decomposition` correspondante à la transformation en **B** de la partie Spécification présentée par la FIGURE 3.37(a).

```
MACHINE
    B_Class_Decomposition
SETS
    OBJECTS = {a1, a2, a3, b1, b2, b3}
ABSTRACT_CONSTANTS
    AA
PROPERTIES
    AA ⊆ OBJECTS ∧
    AA = {a1, a2, a3, b1, b2, b3}
VARIABLES
    aa
INVARIANT
    aa ⊆ AA
INITIALISATION
    aa := {a1, a2, a3, b1, b2, b3}
END
```

FIGURE 3.38 – Modélisation en **B** de l'état de spécification abstraite

Les obligations de preuve liées à la machine `B_Class_Decomposition` sont données dans le tableau 3.17. Elles ont été démontrées automatiquement.

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	6	0	6	0	100
B_Class_Decomposition	6	0	6	0	100

TABLE 3.17 – Tableau de l'état de la machine `B_Class_Decomposition`

La FIGURE 3.39 présente la traduction en **B** de l'état de la partie Raffinement du pattern donnée dans la FIGURE 3.37(b).

```
REFINEMENT
  B_Class_Decomposition_r
REFINES
  B_Class_Decomposition
ABSTRACT_CONSTANTS
   BB
PROPERTIES
   BB ⊆ OBJECTS ∧
   BB ⊆ AA ∧
     BB = {b1, b2, b3}
ABSTRACT_VARIABLES
  aa, bb
INVARIANT
  bb ⊆ aa
INITIALISATION
  aa := {a1, a2, a3, b1, b2, b3} ||
  bb := {b1, b2, b3}
END
```

FIGURE 3.39 – Modélisation en **B** de l'état de spécification concrète

Les obligations de preuve liées à la machine `B_Class_Decomposition_r` sont données dans le tableau 3.18. Elles ont été démontrées automatiquement.

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	3	0	3	0	100
B_Class_Decomposition_r	3	0	3	0	100

TABLE 3.18 – Tableau de l'état de la machine `B_Class_Decomposition_r`

3.5.5 Exemple

A titre d'exemple, nous reprendrons l'exemple présenté par la FIGURE 3.35 qui décrit une classe `Rectangle` contenant deux opérations `surface` et `périmètre` permettant respectivement de calculer la surface et le périmètre d'un rectangle.

Application du pattern `Class_Decomposition`

Un raffinement consiste à dire que l'ensemble des carrés vérifie une contrainte absente dans l'invariant de la classe `Rectangle`, stipulant que la largeur et la longueur d'un carré sont égaux. Ainsi, nous pouvons appliquer le pattern de décomposition d'une classe afin de modéliser cette nouvelle contrainte (voir FIGURE 3.36).

Vérification de l'application du pattern

Afin de formaliser en **B** le raffinage entre la spécification et le raffinement fournis respectivement par les deux FIGURES 3.35 et 3.36, nous proposons l'architecture des machines **B** présentée par la FIGURE 3.40.

Les deux machines `Context` et `B_Rectangle` formalisent en **B** la classe abstraite `Rectangle` fournie par la FIGURE 3.35. La machine `Context` (voir FIGURE 3.41) présente l'ensemble abstrait RECTANGLE utile pour les deux niveaux : spécification et raffinement.

Quant à la machine `B_Rectangle` (voir FIGURE 3.41), elle introduit les attributs `longueur` et `largeur` et les opérations `surface` et `perimetre` venant du modèle UML de la FIGURE 3.35.

Les obligations de preuve liées à la machine `B_Rectangle` sont données dans le tableau 3.19. Elles ont été déchargées automatiquement.

La machine `B_Carre` (voir FIGURE 3.42) formalise en **B** le raffinement fourni par la FIGURE 3.36. Elle introduit la variable `carre` permettant de modéliser l'ensemble des carrés, qui sont des rectangles

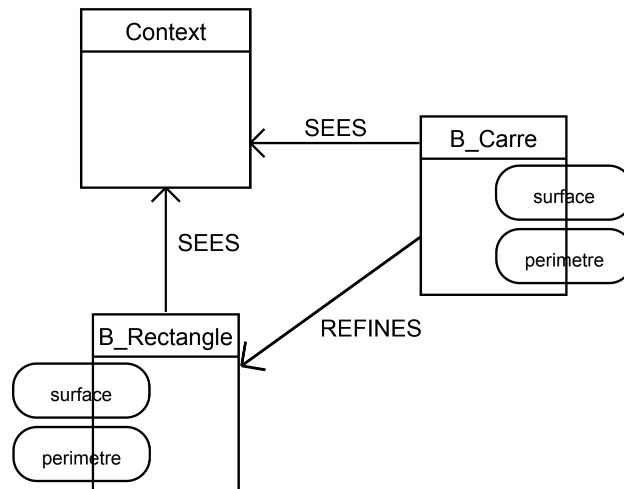


FIGURE 3.40 – Dépendances des machines `Context`, `B_Rectangle` et `B_Carre`

```
MACHINE
    Context
SETS
    RECTANGLE
END
```

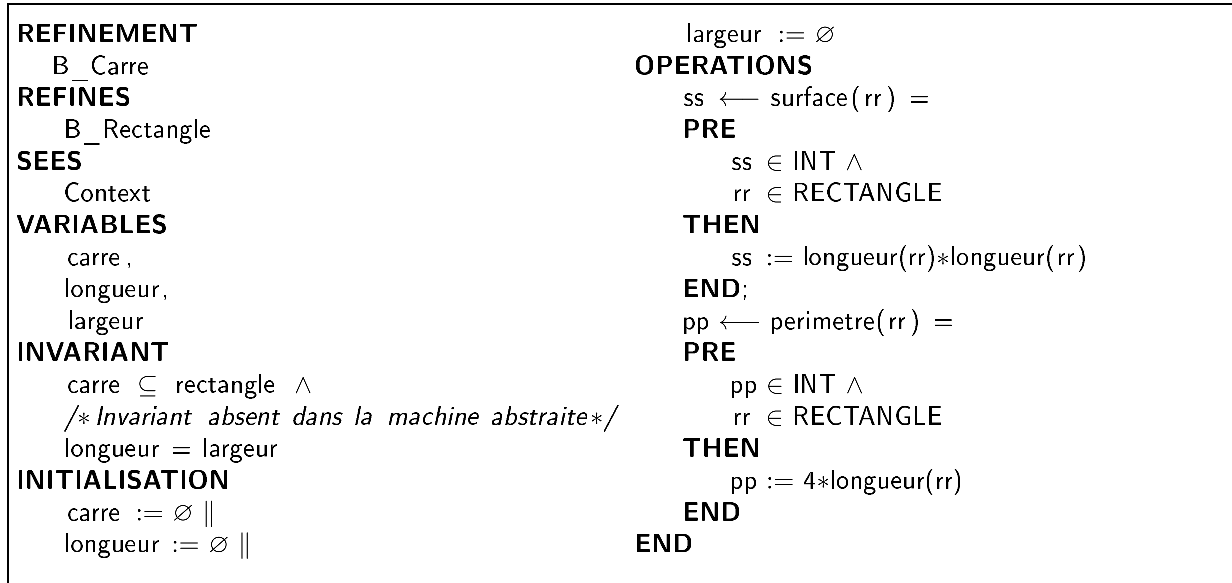
```
MACHINE
    B_Rectangle
SEES
    Context
VARIABLES
    rectangle ,
    longueur,
    largeur
INVARIANT
    rectangle ⊆ RECTANGLE ∧
    longueur ∈ rectangle ↔ NAT ∧
    largeur ∈ rectangle ↔ NAT
INITIALISATION
    rectangle := ∅ ||
    longueur := ∅ ||
    largeur := ∅
OPERATIONS
    ss ⟵ surface(rr) =
    PRE
        ss ∈ INT ∧
        rr ∈ RECTANGLE
    THEN
        ss := longueur(rr)∗largeur(rr)
    END;
    pp ⟵ perimetre(rr) =
    PRE
        pp ∈ INT ∧
        rr ∈ RECTANGLE
    THEN
        pp := 2∗(longueur(rr)+largeur(rr))
    END
END
```

FIGURE 3.41 – Modélisation en **B** de l'état de spécification abstraite

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	2	0	2	0	100
surface	0	0	0	0	100
perimetre	0	0	0	0	100
B_Rectangle	4	0	4	0	100

TABLE 3.19 – Tableau de l'état de la machine `B_Rectangle`

particuliers. Ce qui explique la relation de raffinement entre les machines `B_Rectangle` et `B_Carre`. De plus, elle introduit l'invariant spécifiant la différence entre un carré et un rectangle. En effet, la longueur et la largeur d'un carré sont égaux. Enfin, elle actualise la spécification de deux opérations `surface` et `perimetre` afin de tenir compte de ce nouvel invariant.



```
REFINEMENT
   B_Carre
REFINES
   B_Rectangle
SEES
   Context
VARIABLES
   carre ,
   longueur ,
   largeur
INVARIANT
   carre ⊆ rectangle ∧
   /* Invariant absent dans la machine abstraite*/
   longueur = largeur
INITIALISATION
   carre := ∅ ||
   longueur := ∅ ||
   largeur := ∅
OPERATIONS
   ss ⟵ surface( rr ) =
   PRE
      ss ∈ INT ∧
      rr ∈ RECTANGLE
   THEN
      ss := longueur(rr)*longueur(rr)
   END;
   pp ⟵ perimetre( rr ) =
   PRE
      pp ∈ INT ∧
      rr ∈ RECTANGLE
   THEN
      pp := 4*longueur(rr)
   END
END
```

FIGURE 3.42 – Modélisation en **B** de l'état de spécification concrète

Les obligations de preuve données dans le tableau 3.20 ont été déchargées automatiquement.

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	1	0	1	0	100
surface	4	0	4	0	100
perimetre	4	0	4	0	100
B_Carre	9	0	9	0	100

TABLE 3.20 – Tableau de l'état de la machine `B_Carre`

3.5.6 Voir aussi

Le pattern `Class_Decomposition` a introduit l'idée d'une décomposition d'une classe via la relation d'héritage. Les deux relations UML agrégation et composition peuvent être utilisées avec profit afin de décomposer une entité agrégat modélisée par une classe UML.

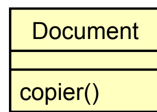


FIGURE 3.43 – Classe `Document`

Par exemple, la classe `Document` présentée dans la FIGURE 3.43 peut être raffinée par la FIGURE 3.44 en respectant l'exigence suivante : un document est un tout dont les parties sont des paragraphes. L'opération `copier` de la classe `Document` délègue en partie son travail à l'opération `copier` de la classe `Paragraphe`.

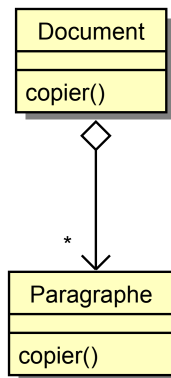


FIGURE 3.44 – Décomposition de la classe `Document`

3.6 Pattern d'introduction d'une nouvelle entité : `Class_NewEntity`

3.6.1 Intention

Il permet d'introduire une classe UML modélisant une entité à part entière. La classe introduite par ce pattern est reliée aux autres classes formant le niveau abstrait par des relations de type association.

3.6.2 Motivation

Dans une modélisation OO incrémentale, on a intérêt à démarrer avec un nombre minimum d'entités appelées entités très abstraites. Ceci favorise ultérieurement l'introduction des détails via des entités dites moins abstraites voire concrètes (équipements par exemple) permettant d'aller du monde abstrait vers le monde concret. A titre d'exemple, dans un Système de Contrôle d'Accès, nous pouvons commencer par un diagramme de classes simplifié (voir FIGURE 3.45).

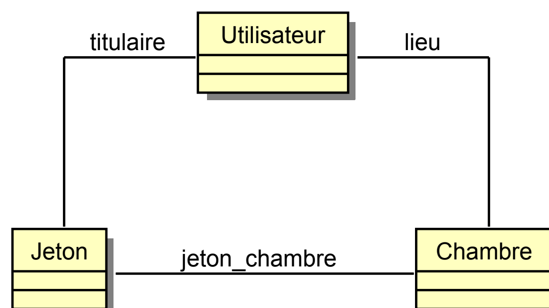


FIGURE 3.45 – Diagramme de classes simplifié

Un raffinement à ce modèle peut introduire la précision suivante : en fait, les jetons sont délivrés par une autorité centrale. Pour y parvenir, on propose d'ajouter une classe `Autorité` liée à toutes les classes `Jeton`, `Chambre` et `Utilisateur` (voir FIGURE 3.46) par une relation de type association.

3.6.3 Solution

Nous nous plaçons dans le cadre suivant :

- à chaque entité de type `A` doit correspondre une et une seule entité de type `B`,

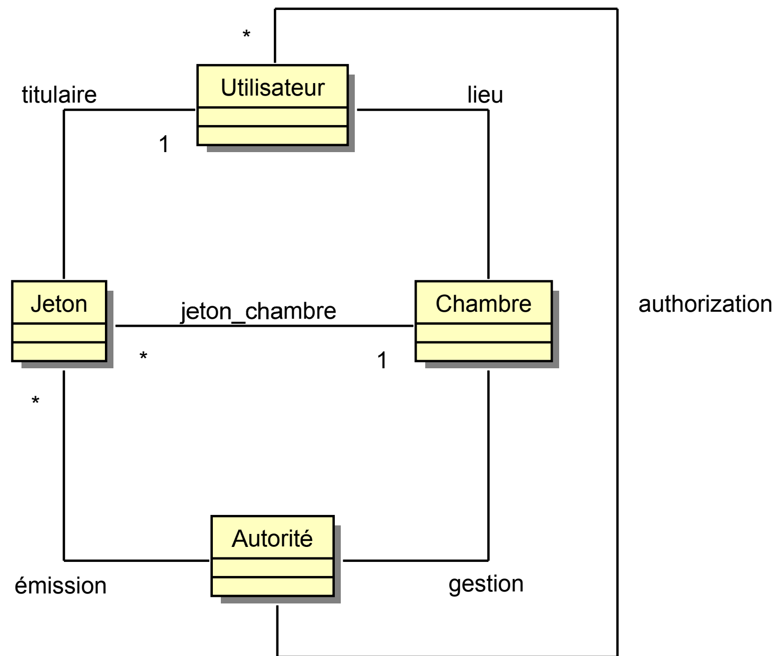
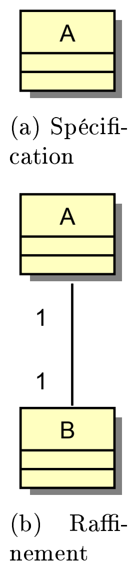


FIGURE 3.46 – Introduction de la notion d'`Autorité`

– les deux propriétés (attributs et opérations) de `A` et `B` ne sont pas prises en compte.

Les deux parties Spécification et Raffinement de ce pattern sont données dans la FIGURE 3.47.



(a) Spécification

(b) Raffinement

FIGURE 3.47 – Pattern d'introduction d'une nouvelle entité : `Class_NewEntity`

3.6.4 Vérification

La FIGURE 3.48 présente la machine abstraite `B_Class_NewEntity_a` correspondante à la transformation en **B** de la partie Spécification présentée par la FIGURE 3.47(a).

```
MACHINE                                     AA = {a1, a2, a3}
    B_Class_NewEntity_a                 VARIABLES
SETS                                        aa
    OBJECTS = {a1, a2, a3,              INVARIANT
               b1, b2, b3}                  aa ⊆ AA
ABSTRACT_CONSTANTS                      INITIALISATION
    AA                                      aa := {a1, a2, a3}
PROPERTIES                              END
    AA ⊆ OBJECTS ∧
```

FIGURE 3.48 – Modélisation en **B** de l'état de spécification abstraite

L'ensemble OBJECTS est défini par extension. Il comporte des objets respectivement de type `AA (ai)` et `BB (bi)`. Ceci permet d'initialiser (voir opération INITIALISATION) les variables `aa` et `bb`. Une telle initialisation correspond à un diagramme d'objets issu du diagramme de classes de la partie Spécification du pattern de raffinement proposé (voir FIGURE 3.47(a)). Les obligations de preuve générées relatives à la machine `B_Class_NewEntity_a` ont été toutes déchargées automatiquement (voir Tableau 3.21). Ceci valide en partie les propriétés invariantes liées à l'état de la machine `B_Class_NewEntity_a`.

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	3	0	3	0	100
B_Class_NewEntity_a	3	0	3	0	100

TABLE 3.21 – Tableau de l'état de la machine B_Class_NewEntity_a

La FIGURE 3.49 présente la traduction en **B** de l'état de la partie Raffinement du pattern donnée dans la FIGURE 3.47(b).

```
REFINEMENT                                  bb,
    B_Class_NewEntity_r                     association
REFINES                                 INVARIANT
    B_Class_NewEntity_a                     bb ⊆ BB ∧
ABSTRACT_CONSTANTS                          /* Invariant de collage*/
    BB                                      association ∈ aa ⤖ bb
PROPERTIES                              INITIALISATION
    BB ⊆ OBJECTS ∧                          aa := {a1, a2, a3} ||
    BB ∩ AA = ∅ ∧                           bb := {b1, b2, b3} ||
    BB = {b1, b2, b3}                       association := {a1↦b1, a2↦b2, a3↦b3}
ABSTRACT_VARIABLES                      END
    aa,
```

FIGURE 3.49 – Modélisation en **B** de l'état de spécification concrète

La machine `B_Class_NewEntity_r` introduit une constante abstraite `BB` pour mémoriser les objets potentiels de type `bb`. L'état variable de la machine `B_Class_NewEntity_r` est défini par les trois variables `aa`, `bb` et `association`. La variable `aa` provient de la machine abstraite `B_Class_NewEntity_a`. Tandis que les deux autres variables sont introduites par la partie Raffinement du pattern Class_NewEntity. La variable concrète `association` stipule qu'à toute entité de `aa` doit correspondre une et une seule entité de `bb`. Ceci définit l'invariant de collage suivant :

association $\in$ aa $\rightarrowtail\!\!\!\!\twoheadrightarrow$ bb (**inv**)

$$(inv_collage_Class_NewEntity) \quad (3.4)$$

Cet invariant de collage garantit la correction du pattern de raffinement `Class_NewEntity`. Il pourrait être instancié et réutilisé avec profit dans un développement conjoint UML/B guidé par des patterns de raffinement.

Les obligations de preuve générées relatives à la machine `B_Class_NewEntity_r` ont été toutes déchargées automatiquement (voir Tableau 3.22). Ceci valide en partie les propriétés invariantes, notamment l'invariant de collage, liées à l'état de la machine `B_Class_NewEntity`.

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	7	0	7	0	100
B_Class_NewEntity_r	7	0	7	0	100

TABLE 3.22 – Tableau de l'état de la machine B_Class_NewEntity_r

3.6.5 Exemple

A titre d'exemple, nous reprendrons l'exemple présenté par la FIGURE 3.45. Dans un système de contrôle d'accès, un utilisateur se trouve dans une chambre s'il possède un jeton permettant l'accès à cette chambre. Dans ce qui suit, nous nous limitons l'étude à la notion jeton. Celle-ci est caractérisée par une validité (voire FIGURE 3.50).

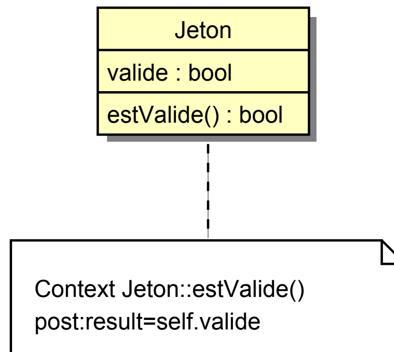


FIGURE 3.50 – Diagramme de classes simplifié

Application du pattern Class_NewEntity

Des détails supplémentaires peuvent être ajoutés :
- les jetons sont délivrés par une autorité,
- les autorisations des jetons sont valides ou non.

L'introduction de la notion d'Autorité est matérialisée par une étape de raffinement en appliquant le pattern de raffinement `Class_NewEntity`. Ceci est illustré par la FIGURE 3.51.

Vérification de l'application du pattern

Afin de formaliser en **B** le raffinage entre la spécification et le raffinement fournis respectivement par les deux FIGURES 3.50 et 3.51, nous proposons l'architecture des machines **B** présentée par la FIGURE 3.52.

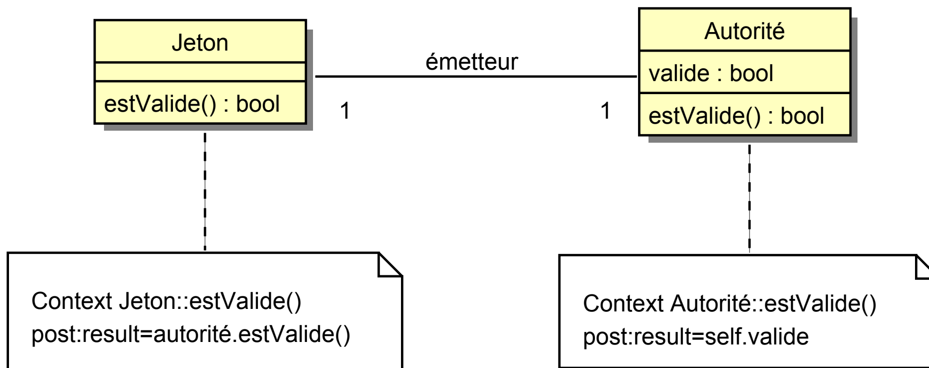


FIGURE 3.51 – Introduction de la notion de `Autorité`

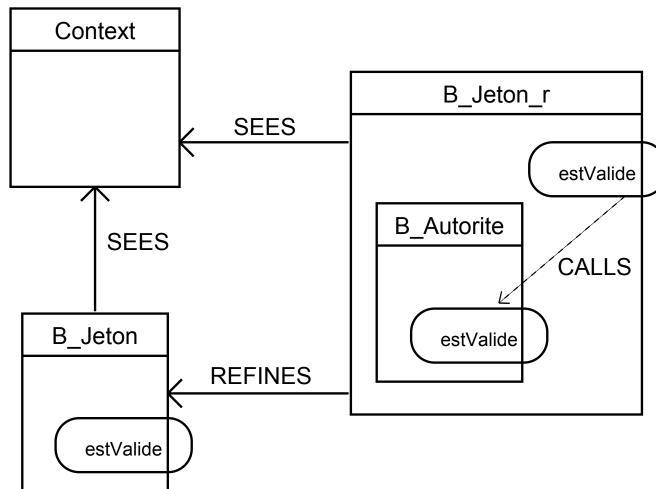


FIGURE 3.52 – Dépendances des machines `Context`, `B_Jeton`, `B_Jeton_r` et `B_Autorite`

Les deux machines Context et B_Jeton formalisent en **B** la classe abstraite Jeton fournie par la FIGURE 3.50. La machine Context (voir FIGURE 3.53) factorise des ensembles abstraits JETON et AUTORITE utiles pour les deux niveaux : spécification et raffinement.

Quant à la machine B_Jeton (voir FIGURE 3.53), elle introduit l'attribut valide et l'opération estValide venant du modèle UML de la FIGURE 3.50. La sémantique pré/post en OCL de la méthode estValide appartenant à la classe Jeton est préservée par l'opération de même nom appartenant à la machine B_Jeton.

```
MACHINE
    Context
SETS
    JETON; AUTORITE
END
```

```
MACHINE
    B_Jeton
SEES
    Context
VARIABLES
    jeton,
    valide
INVARIANT
    jeton ⊆ JETON ∧
    valide ∈ jeton ↔ 𝔹
INITIALISATION
    jeton := ∅ ||
    valide := ∅
OPERATIONS
    ss ⟵ estValide( jj ) =
    PRE
        ss ∈ 𝔹 ∧
        jj ∈ jeton
    THEN
        ss := valide(jj)
    END
END
```

FIGURE 3.53 – Modélisation en **B** de l'état de spécification abstraite

Les obligations de preuve liées à la machine B_Rectangle sont données dans le tableau 3.23. Elles ont été déchargées automatiquement.

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	2	0	2	0	100
estValide	0	0	0	0	100
B_Jeton	2	0	2	0	100

TABLE 3.23 – Tableau de l'état de la machine B_Jeton

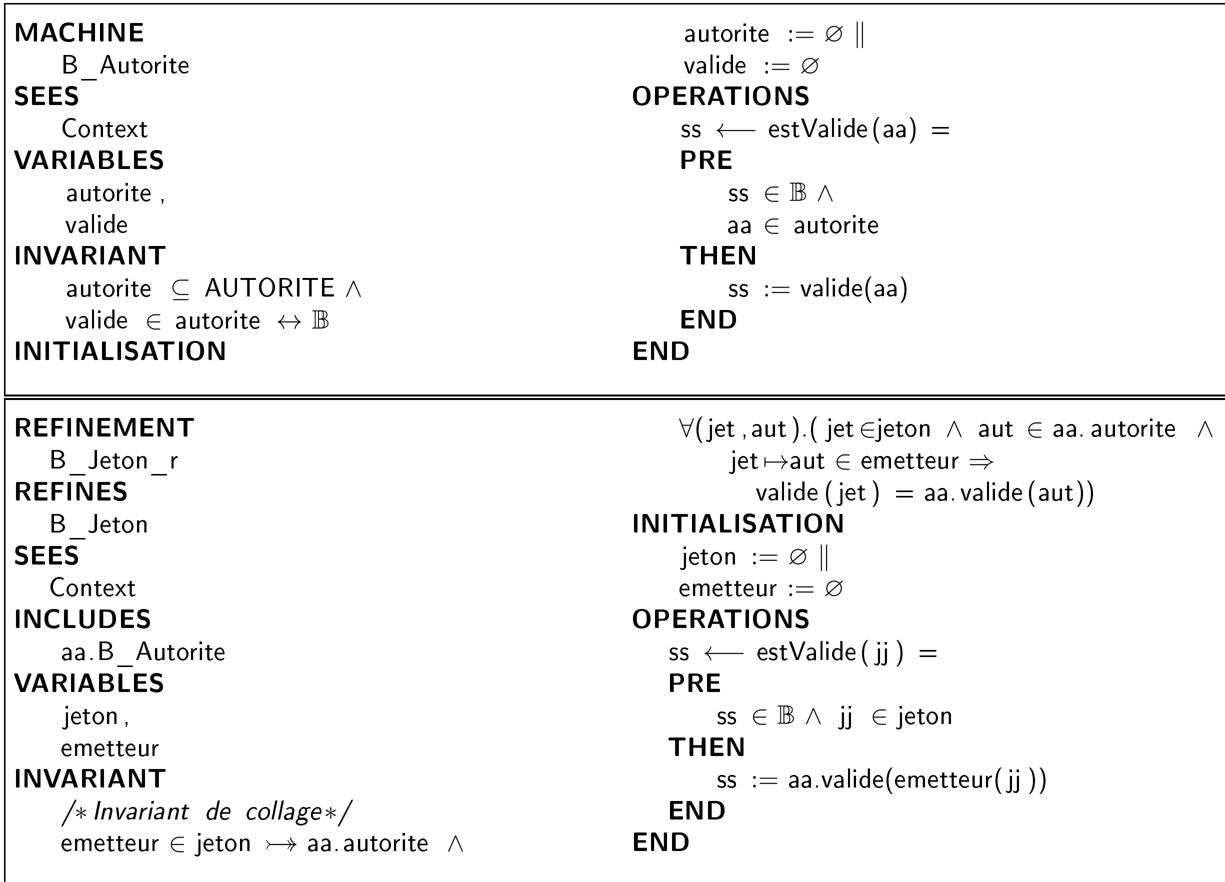
Les deux machines B_Autorite et B_Jeton_r (voir FIGURE 3.54) formalisent en **B** le raffinement fourni par la FIGURE 3.51. La machine B_Autorite formalise en **B** la classe Autorité du modèle UML présenté par la même figure. La machine B_Jeton_r raffine B_Jeton en incluant une instance de la machine B_Autorite (voir la clause INCLUDES). Elle introduit la variable emetteur (bijection totale) permettant de modéliser l'association entre entre Jeton et Autorité. En effet, l'invariant de la machine B_Jeton_r est issue de l'invariant de collage du pattern Class_NewEntity (Inv_Collage_Class_NewEntity)(3.4).

Les obligations de preuve, liées à la machine B_Autorite, données dans le tableau 3.24 ont été déchargées automatiquement.

Les obligations de preuve liées à la machine B_Jeton_r sont données dans le tableau 3.25. Elles ont été déchargées dont une de façon interactive en utilisant le prouveur pp1.

3.6.6 Voir aussi

Sur le plan forme, les deux patterns Class_Helper (voir section 3.2) et Class_NewEntity produisent des effets similaires. Mais, sur le plan contenu, ils divergent. En effet, ils possèdent deux invariants de collage différents. En outre, le pattern Class_NewEntity est plutôt orienté raffinement horizontal (phase



```
MACHINE
    B_Autorite
SEES
    Context
VARIABLES
    autorite ,
    valide
INVARIANT
    autorite ⊆ AUTORITE ∧
    valide ∈ autorite ↔ 𝔹
INITIALISATION
    autorite := ∅ ||
    valide := ∅
OPERATIONS
    ss ⟵ estValide(aa) =
    PRE
        ss ∈ 𝔹 ∧
        aa ∈ autorite
    THEN
        ss := valide(aa)
    END
END
```

```
REFINEMENT
    B_Jeton_r
REFINES
    B_Jeton
SEES
    Context
INCLUDES
    aa.B_Autorite
VARIABLES
    jeton ,
    emetteur
INVARIANT
    /* Invariant de collage */
    emetteur ∈ jeton ⤀ aa.autorite ∧
    ∀(jet ,aut).( jet ∈ jeton ∧ aut ∈ aa.autorite ∧
        jet ↦ aut ∈ emetteur ⇒
            valide ( jet ) = aa.valide (aut))
INITIALISATION
    jeton := ∅ ||
    emetteur := ∅
OPERATIONS
    ss ⟵ estValide( jj ) =
    PRE
        ss ∈ 𝔹 ∧ jj ∈ jeton
    THEN
        ss := aa.valide(emetteur( jj ))
    END
END
```

FIGURE 3.54 – Modélisation en **B** de l'état de spécification concrète

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	2	0	2	0	100
estValide	0	0	0	0	100
B_Autorite	2	0	2	0	100

TABLE 3.24 – Tableau de l'état de la machine `B_Autorite`

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	4	0	4	0	100
estValide	2	1	1	0	100
B_Jeton_r	6	1	5	0	100

TABLE 3.25 – Tableau de l'état de la machine `B_Jeton_r`

spécification) favorisant la construction pas-à-pas d'un modèle métier de l'application. Tandis que le pattern `Class_Helper` est plutôt orienté raffinement vertical (phase conception) favorisant la construction graduelle d'un modèle conceptuel de l'application.

3.7 Pattern de raffinement de contrôle d'une classe : `Refinement_Operation`

3.7.1 Intention

Ce pattern assure le passage d'une spécification abstraite d'une opération UML vers une spécification moins abstraite.

3.7.2 Motivation

La méthode formelle B autorise plusieurs types de raffinement : de données, de contrôle et algorithmique. Dans le raffinement de contrôle, on observe les faits suivants :

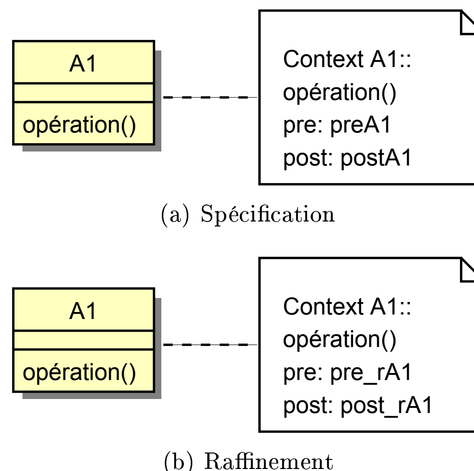
- l'opération B à raffiner conserve la même signature,
- sa précondition peut être élargie,
- son comportement non déterministe décrit par des substitutions B peut être réduit.

Dans le cadre UML, nous pouvons décrire le raffinement de contrôle en se servant d'OCL afin de décrire aussi bien la spécification abstraite que celle moins abstraite (concrète) de l'opération à raffiner.

3.7.3 Solution

Nous prendrons la classe originale `A1` comme classe isolée.

Les deux parties Spécification et Raffinement de ce pattern sont données dans la FIGURE 3.55.



(a) Spécification

(b) Raffinement

FIGURE 3.55 – Pattern de raffinement de contrôle d'une opération

3.7.4 Vérification

La vérification de ce pattern peut être confiée à l'***Atelier B*** moyennant la traduction systématique de deux parties Spécification et Raffinement du pattern `Refinement_Operation`.

3.7.5 Exemple

La spécification pré/post en OCL d'une opération UML peut être obtenue de manière graduelle. A titre d'exemple, nous prenons la classe `Horloge`, qui permet de décompter le temps, de 00 heure 00 minute, à 23 heures 59 minutes. La classe `Horloge` est caractérisée par l'opération `tictac` permettant de calculer l'écoulement du temps. Une première spécification qui ignore la mémorisation des minutes est proposée dans FIGURE 3.56.

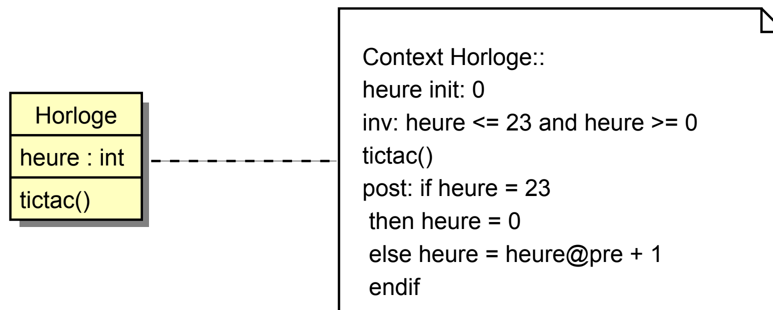


FIGURE 3.56 – Classe `Horloge`

Application du pattern `Refinement_Operation`

Un raffinement à cette classe peut être effectué de la façon suivante : notre horloge peut prendre en compte la mémorisation des minutes (Horloge avec minutes). Dans ce cas, l'opération `tictac` doit être raffinée comme montre la FIGURE 3.57.

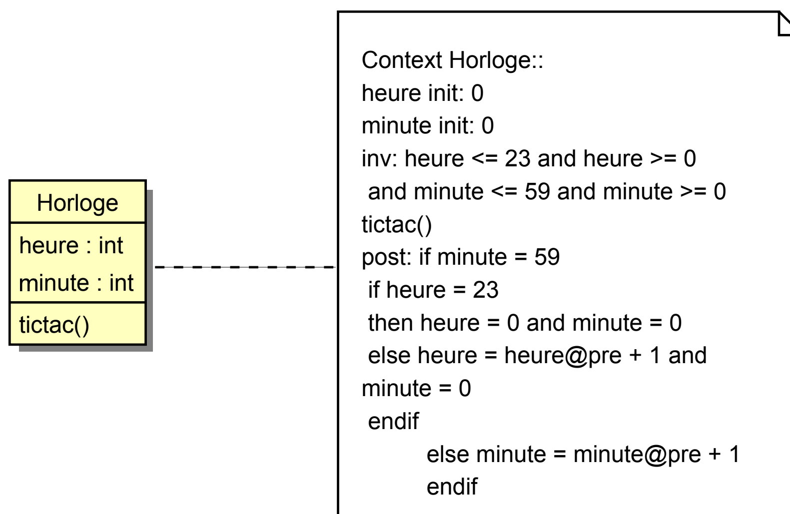


FIGURE 3.57 – Raffinement de contrôle de `tictac` de la classe `Horloge`

Vérification de l'application du pattern

La machine B_Horloge[2], présentée dans la FIGURE 3.58, formalise en **B** la classe abstraite Horloge fournie par la FIGURE 3.56. La sémantique pré/post en OCL de la méthode tictac appartenant à la classe Horloge est préservée par l'opération de même nom appartenant à la machine B_Horloge.

```
MACHINE
    B_Horloge
VARIABLES
    heure
INVARIANT
    heure ∈ 0..23
INITIALISATION
    heure := 0
OPERATIONS
    tictac =
          CHOICE
              IF heure = 23 THEN
                  heure := 0
              ELSE
                  heure := heure + 1
              END
          OR
              skip
          END
      END
```

FIGURE 3.58 – Modélisation en **B** de la classe Horloge

Les obligations de preuve liées à la machine B_Horloge sont données dans le tableau 3.26. Elles ont été déchargées automatiquement.

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	1	0	1	0	100
tictac	2	0	2	0	100
B_Horloge	3	0	3	0	100

TABLE 3.26 – Tableau de l'état de la machine B_Horloge

La machine B_Horloge_r (voir FIGURE 3.59) formalise en **B** le niveau Raffinement présenté par la FIGURE 3.57. L'opération tictac de la machine B_Horloge_r est spécifiée en tenant compte de la sémantique pré/post en OCL de la méthode tictac affinée fournie par la FIGURE 3.57.

Les obligations de preuve liées à la machine B_Horloge sont données dans le tableau 3.27. Elles ont été déchargées automatiquement.

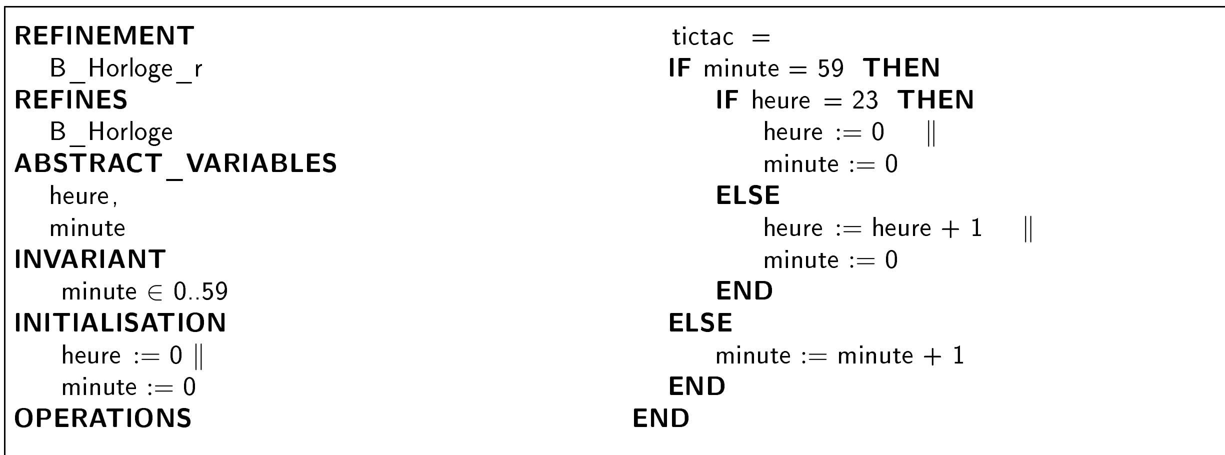
	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	1	0	1	0	100
tictac	5	0	5	0	100
B_Horloge_r	6	0	6	0	100

TABLE 3.27 – Tableau de l'état de la machine B_Horloge_r

3.7.6 Voir aussi

Le pattern Refinement_Operation introduit l'idée de raffinement de contrôle. Mais, nous pouvons définir de la même manière un pattern de raffinement de données. Ce dernier permet d'introduire des

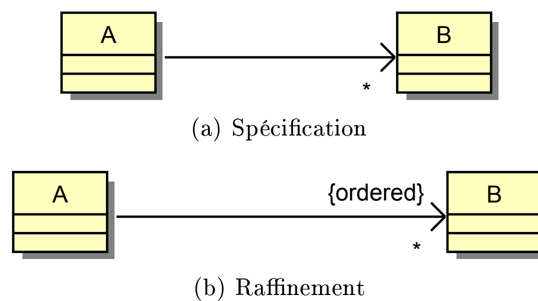
2. L'exemple d'horloge est abordé dans le cours de Dominique Cansell afin d'introduire le raffinement en B *http ://www.loria.fr/c̃ansell/*



```
REFINEMENT
    B_Horloge_r
REFINES
    B_Horloge
ABSTRACT_VARIABLES
    heure,
    minute
INVARIANT
     minute ∈ 0..59
INITIALISATION
     heure := 0 ||
     minute := 0
OPERATIONS
    tictac =
    IF minute = 59 THEN
        IF heure = 23 THEN
            heure := 0 ||
            minute := 0
        ELSE
            heure := heure + 1 ||
            minute := 0
        END
    ELSE
        minute := minute + 1
    END
END
```

FIGURE 3.59 – Modélisation en **B** de la classe `Horloge` après raffinement

variables concrètes par rapport aux variables abstraites de la partie Spécification. Dans ce cas, **un invariant de collage** relie variables abstraites et concrètes doit être explicité. Par exemple, on raffine un ensemble UML par une suite UML. Ceci est illustré par la FIGURE 3.60.



(a) Spécification

(b) Raffinement

FIGURE 3.60 – Affinage d'un ensemble par une suite

Le raffinement de contrôle et de données ne sont pas exclusifs : ils peuvent être opérés dans la même étape de raffinement. Il est évident que le pattern `Refinement_Operation` peut être appliqué en combinant ces deux types de raffinement.

3.8 Pattern d'abstraction d'une classe : `Class_Abstraction`

3.8.1 Intention

Le pattern `Class_Abstraction` introduit des qualités logicielles comme efficacité, réutilisabilité et évolutivité dans un développement logiciel guidé par des raffinements successifs. Ainsi, il permet de **factoriser** les propriétés communes –attributs, opérations et relations– à plusieurs classes au sein d'une classe fondatrice.

3.8.2 Motivation

Dans un processus de développement, chaque entité est modélisée par une classe. Mais souvent, on rencontre des classes qui sont en fait des variantes d'une même notion. Plusieurs classes d'un diagramme de classes ont des caractéristiques communes. On dit que ces classes indépendantes au départ peuvent être dérivées d'un ancêtre commun.

L'idée est d'améliorer la modélisation, d'obtenir une meilleure représentation et de faciliter la mémorisation des données, donc éviter la redondance des caractéristiques. Pour cela nous pouvons factoriser ces caractéristiques communes entre les différentes classes dans une nouvelle classe ancêtre.

A titre d'exemple, nous prenons un diagramme de classes simplifié permettant aux enseignants et aux étudiants d'emprunter ou de commander des livres. Ce diagramme comporte trois classes : `Enseignant`, `Etudiant` et `Livre`. Les deux classes `Enseignant` et `Etudiant` sont caractérisées par un nom, un prénom, une opération `rendre` permettant de rendre un livre et une opération `emprunter` permettant d'emprunter un livre. Mais, seul l'enseignant peut commander des livres (voir FIGURE 3.61).

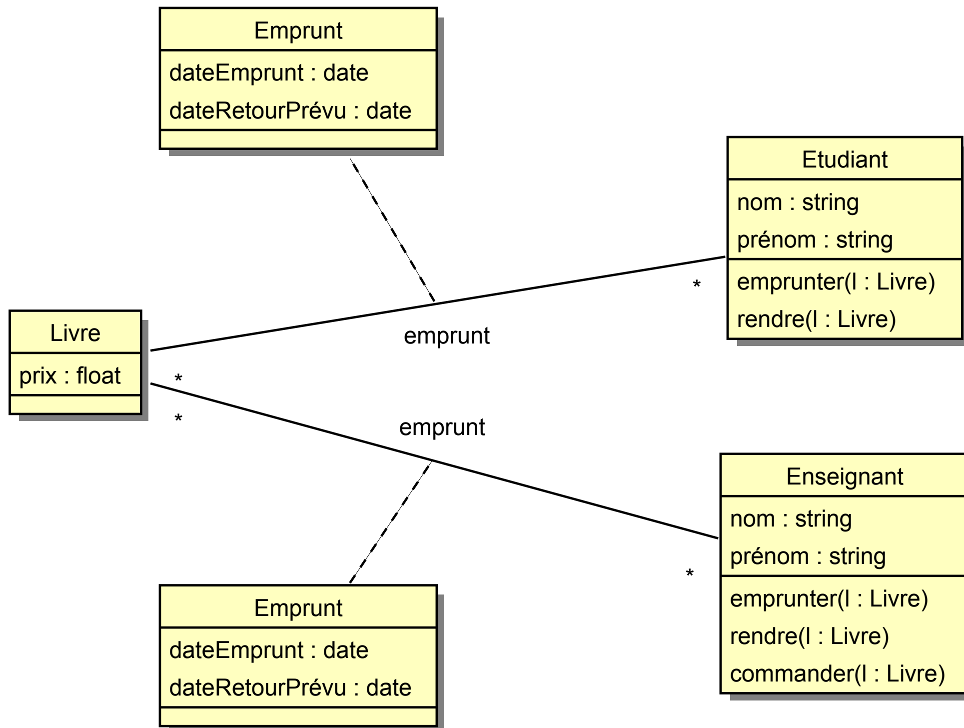


FIGURE 3.61 – Diagramme de classes non factorisé

Un raffinement à ce diagramme de classes peut être basé sur l'exigence suivante : regrouper les caractéristiques communes aux deux classes `Enseignant` et `Etudiant` –nom, prénom, emprunt, emprunter() et rendre()– dans une classe fondatrice `Adhérant` comme montre la FIGURE 3.62.

3.8.3 Solution

Ce pattern concerne uniquement les propriétés `att1` et `op1` communes aux deux classes `A11` et `A12`. Les autres propriétés (`att2`, `op2`, `att3` et `op3`) demeurent à leurs places initiales.

Les deux parties Spécification et Raffinement de ce pattern sont données dans la FIGURE 3.63.

3.8.4 Vérification

La FIGURE 3.65 présente la machine abstraite `B_Class_Abstraction_a` correspondante à la transformation en **B** de la partie Spécification présentée par la FIGURE 3.63(a).

La machine abstraite `B_Class_Abstraction_a` est définie de la même façon que celles des patterns présentés précédemment. Les obligations de preuve générées relatives à cette machine ont été toutes déchargées automatiquement (voir Tableau 3.28). Ceci valide en partie les propriétés invariantes liées à l'état de la machine `B_Class_Association_a`.

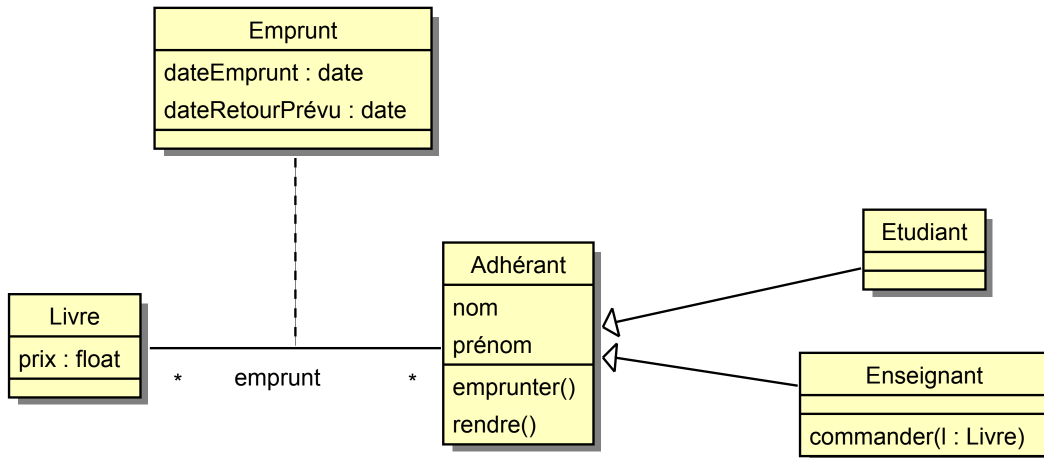
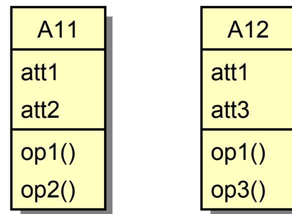
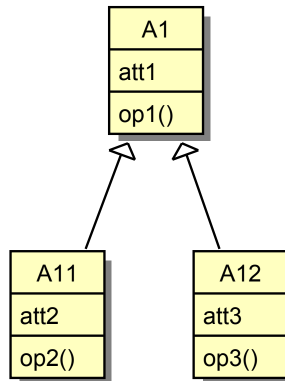


FIGURE 3.62 – Raffinement par abstraction des classes `Enseignant` et `Etudiant`



(a) Spécification



(b) Raffinement

FIGURE 3.63 – Pattern de raffinement par abstraction

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	10	0	10	0	100
a11_op1	0	0	0	0	100
op2	0	0	0	0	100
a12_op1	0	0	0	0	100
op3	0	0	0	0	100
B_Class_Abstraction_a	7	0	7	0	100

TABLE 3.28 – Tableau de l'état de la machine `B_Class_Abstraction_a`

```
MACHINE
    B_Class_Abstraction_a
SETS
    AA = {a111, a112, a113, a121, a122, a123} ;
    TYPES = {type, type1, type2}
ABSTRACT_CONSTANTS
    A11, A12
PROPERTIES
    A11 ⊆ AA ∧
    A12 ⊆ AA ∧
    A11 ∩ A12 = ∅ ∧
    A11 = {a111, a112, a113} ∧
    A12 = {a121, a122, a123}
VARIABLES
    a11, a12, a11_att1, att2, a12_att1, att3
INVARIANT
    a11 ⊆ A11 ∧
    a12 ⊆ A12 ∧
    a11_att1 ∈ a11 ↔ TYPES ∧
    att2 ∈ a11 ↔ TYPES ∧
    a12_att1 ∈ a12 ↔ TYPES ∧
    att3 ∈ a12 ↔ TYPES
INITIALISATION
    a11 := {a111, a112, a113} ||
    a12 := {a121, a122, a123} ||
    a11_att1 := {a111↦type,
                 a112↦type, a113↦type} ||
    att2 := {a111↦type1,
             a112↦type1, a113↦type1} ||
    a12_att1 := {a121↦type,
                 a122↦type, a123↦type} ||
    att3 := {a121↦type2,
             a122↦type2, a123↦type2}
OPERATIONS
    a11_op1(pp) =
    PRE pp ∈ a11
    THEN skip
    END;
    op2(pp) =
    PRE pp ∈ a11
    THEN skip
    END;
    a12_op1(pp) =
    PRE pp ∈ a12
    THEN skip
    END;
    op3(pp) =
    PRE pp ∈ a12
    THEN skip
    END
END
```

Figure 3.64 – Modélisation en **B** de l'état de spécification abstraite

La FIGURE 3.65 présente la traduction en **B** de l'état de la partie Raffinement du pattern donnée dans la FIGURE 3.63(b).

```
REFINEMENT
  B_Class_Abstraction_r
REFINES
  B_Class_Abstraction_a
VARIABLES
    aa, a11, a12, att1, att2, att3
INVARIANT
    aa ⊆ AA ∧
    att1 ∈ aa ↔ TYPES ∧
    dom(a11_att1) ⊆ dom(att1) ∧
    dom(a12_att1) ⊆ dom(att1) ∧
    dom(a11_att1) ∩ dom(a12_att1) =∅ ∧
    ∀ll .( ll ∈a11 ⇒ a11_att1(ll) = att1( ll )) ∧
    ∀ff .( ff ∈a12 ⇒ a12_att1(ff) = att1( ff ))
INITIALISATION
    aa := {a111, a112, a113, a121, a122, a123} ||
    a11 := {a111, a112, a113} ||
    a12 := {a121, a122, a123} ||
    att1 := {a111↦type,
             a112↦type, a113↦type,
             a121↦type, a122↦type,
             a123↦type} ||
    att2 := {a111↦type1,
             a112↦type1, a113↦type1} ||
    att3 := {a121↦type2,
             a122↦type2, a123↦type2}
OPERATIONS
    a11_op1(pp) =
    PRE pp ∈ a11
    THEN skip
    END;
    op2(pp) =
    PRE pp ∈ a11
    THEN skip
    END;
    a12_op1(pp) =
    PRE pp ∈ a12
    THEN skip
    END;
    op3(pp) =
    PRE pp ∈ a12
    THEN skip
    END
END
```

FIGURE 3.65 – Modélisation en **B** de l'état de spécification concrète

La machine `B_Class_Association_r` introduit une variable `att1` commune entre les deux classes abstraites. Une telle variable remplace les deux variables `a11_att1` et `a12_att1`, qui ont été supprimées dans la partie Raffinement. La clause INVARIANT de la machine `B_Class_Abstraction_r` définit l'invariant de collage liant les variables `a11_att1`, `a12_att1` et `att1` :

```
dom(a11_att1) ⊆ dom(att1)                  (inv1)
dom(a12_att1) ⊆ dom(att1)                  (inv2)
dom(a11_att1) ∩ dom(a12_att1) =∅           (inv3)
∀ll.(ll∈a11 ⇒ a11_att1(ll) = att1(ll)) (inv4)
∀ff.(ff∈a12 ⇒ a12_att1(ff) = att1(ff)) (inv5)
```

$(inv_collage_Class_Abstraction)$ (3.5)

Cet invariant de collage garantit la correction du pattern de raffinement `Class_Abstraction`. Il pourrait être instancié et réutilisé avec profit dans un développement conjoint UML/B guidé par des patterns de raffinement.

Les obligations de preuve générées relatives à la machine `B_Class_Abstraction_r` ont été déchargées automatiquement (voir Tableau 3.29). Ceci valide en partie les propriétés invariantes, notamment l'invariant de collage, liées à l'état de la machine `B_Class_Association_r`.

3.8.5 Exemple

A titre d'exemple, le diagramme de classes donné dans la FIGURE 3.66 présente deux classes `Livre` et `Film`. La classe `Livre` est caractérisée par un nombre d'auteurs et un nombre de pages. Elle est caractérisée aussi par deux opérations : `getNbAuteur` permettant de rendre le nombre des auteurs d'un livre et `getNbPages` permettant de rendre le nombre de pages d'un livre. La classe `Film` est caractérisée

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	6	0	6	0	100
a11_op1	0	0	0	0	100
op2	0	0	0	0	100
a12_op1	0	0	0	0	100
op3	0	0	0	0	100
B_Class_Abstraction_r	6	0	6	0	100

TABLE 3.29 – Tableau de l'état de la machine B_Class_Abstraction_r

par un nombre d'auteurs et une durée. Elle est caractérisée aussi par deux opérations : getNbAuteur permettant de rendre le nombre des auteurs d'un film et getDurée permettant de rendre la durée d'un livre.

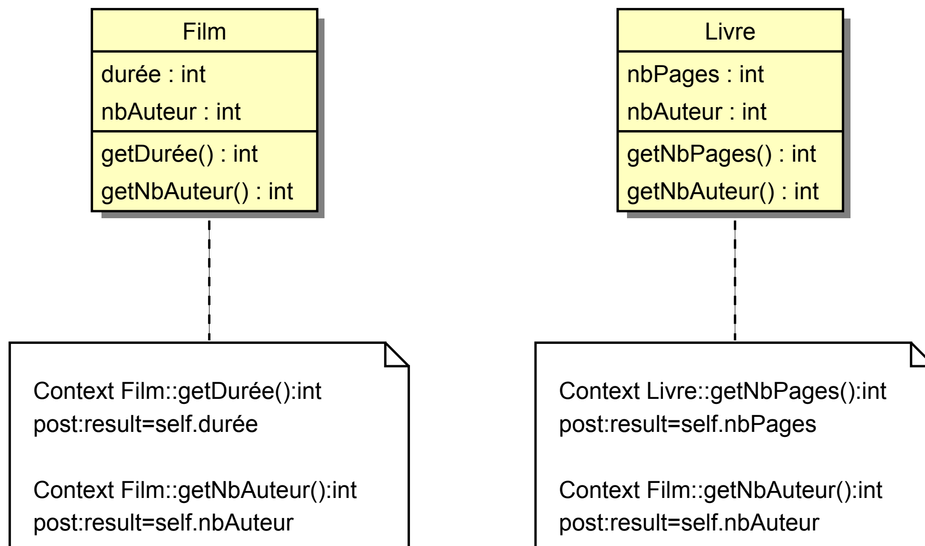


FIGURE 3.66 – Diagramme de classes non factorisé

Application du pattern Class_Abstraction

La factorisation des propriétés communes aux classes Livre et Film en une nouvelle classe Oeuvre est matérialisée par une étape de raffinement en appliquant le pattern de raffinement Class_Abstraction. Ceci est illustré par la FIGURE 3.67.

La spécification pré/post en OCL de l'opération commune getNbAuteur entraîne celle de l'opération getNbAuteur de la classe Oeuvre.

Vérification de l'application du pattern

La FIGURE 3.68 présente la machine abstraite B_Oeuvre correspondante à la transformation en **B** du diagramme de classes présenté par la FIGURE 3.66.

La machine abstraite B_Oeuvre est définie de la même façon que la machine B_Class_Abstraction_a (voir FIGURE 3.65). Les obligations de preuve générées relatives à cette machine ont été toutes déchargées automatiquement (voir Tableau 3.30). Ceci valide en partie les propriétés invariantes liées à l'état de la machine B_Oeuvre.

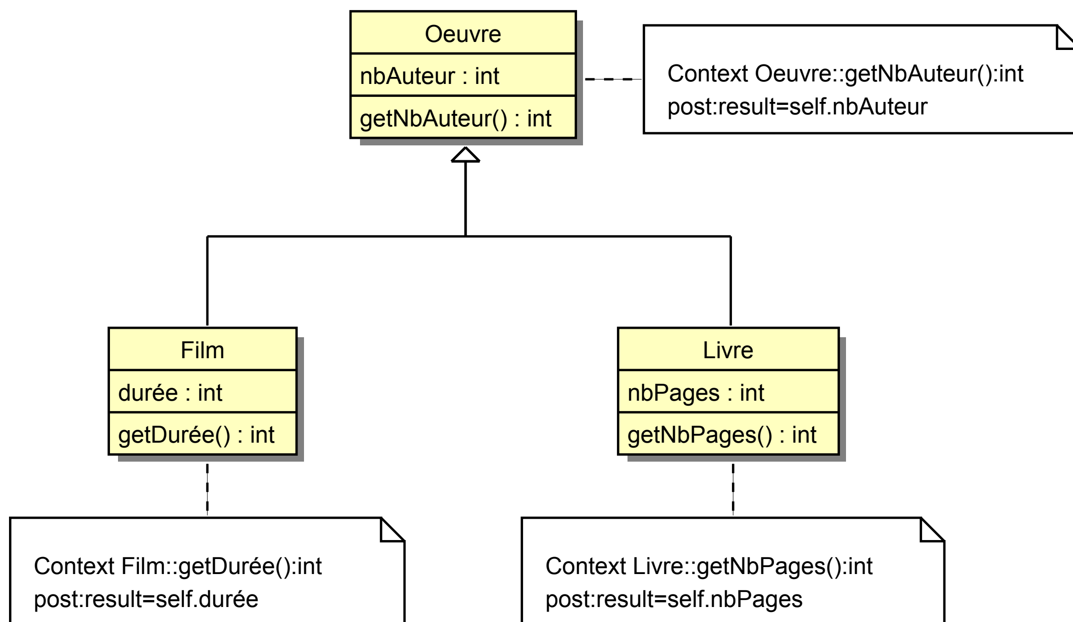
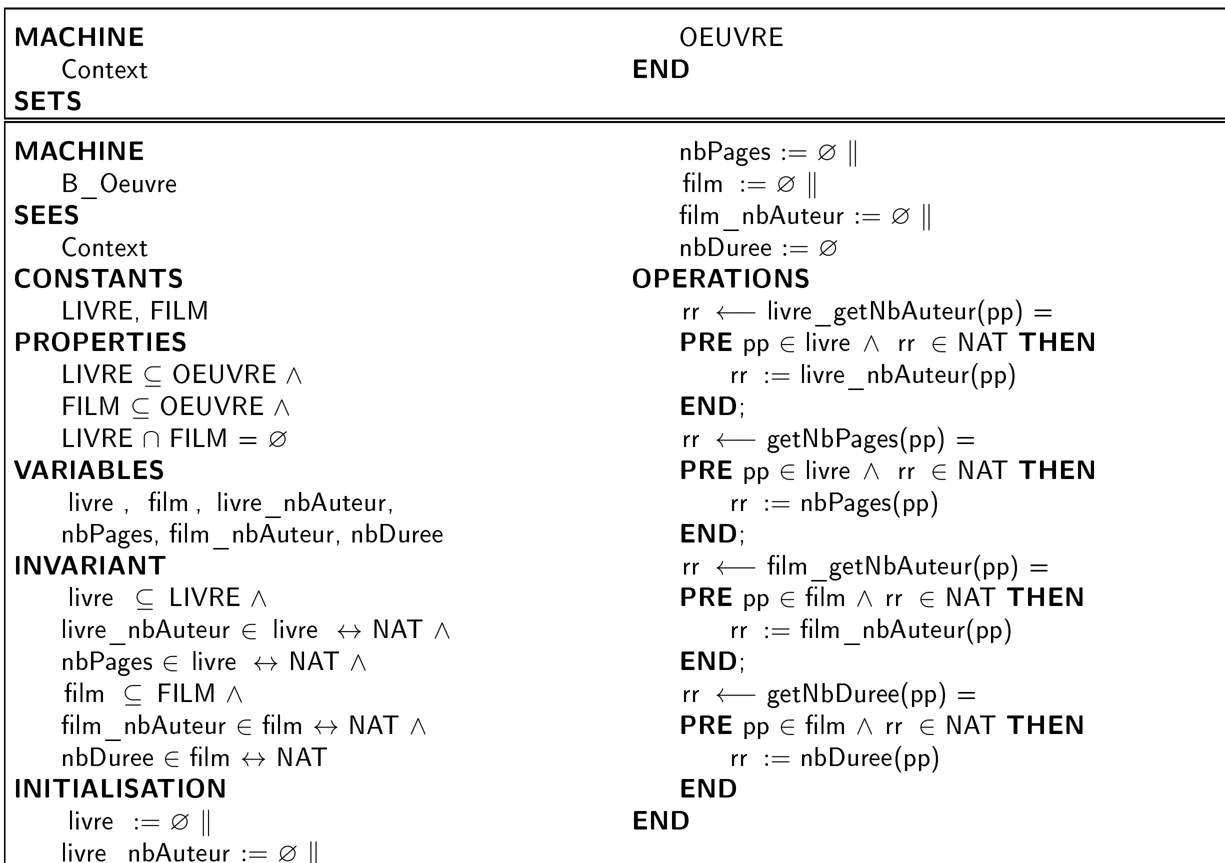


FIGURE 3.67 – Raffinement par abstraction des classes `Film` et `Livre`



```
MACHINE
    Context
SETS
    OEUVRE
END
```

```
MACHINE
    B_Oeuvre
SEES
    Context
CONSTANTS
    LIVRE, FILM
PROPERTIES
    LIVRE ⊆ OEUVRE ∧
    FILM ⊆ OEUVRE ∧
    LIVRE ∩ FILM = ∅
VARIABLES
    livre , film , livre_nbAuteur,
    nbPages, film_nbAuteur, nbDuree
INVARIANT
    livre ⊆ LIVRE ∧
    livre_nbAuteur ∈ livre ↔ NAT ∧
    nbPages ∈ livre ↔ NAT ∧
    film ⊆ FILM ∧
    film_nbAuteur ∈ film ↔ NAT ∧
    nbDuree ∈ film ↔ NAT
INITIALISATION
    livre := ∅ ||
    livre_nbAuteur := ∅ ||
    nbPages := ∅ ||
    film := ∅ ||
    film_nbAuteur := ∅ ||
    nbDuree := ∅
OPERATIONS
    rr ⟵ livre_getNbAuteur(pp) =
    PRE pp ∈ livre ∧ rr ∈ NAT THEN
        rr := livre_nbAuteur(pp)
    END;
    rr ⟵ getNbPages(pp) =
    PRE pp ∈ livre ∧ rr ∈ NAT THEN
        rr := nbPages(pp)
    END;
    rr ⟵ film_getNbAuteur(pp) =
    PRE pp ∈ film ∧ rr ∈ NAT THEN
        rr := film_nbAuteur(pp)
    END;
    rr ⟵ getNbDuree(pp) =
    PRE pp ∈ film ∧ rr ∈ NAT THEN
        rr := nbDuree(pp)
    END
END
```

FIGURE 3.68 – Modélisation en **B** de l'état de spécification abstraite

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	3	0	3	0	100
livre_getNbAuteur	0	0	0	0	100
getNbPages	0	0	0	0	100
film_getNbAuteur	0	0	0	0	100
getDuree	0	0	0	0	100
B_Oeuvre	3	0	3	0	100

TABLE 3.30 – Tableau de l'état de la machine B_Oeuvre

La FIGURE 3.69 présente la traduction en **B** de l'état du diagramme de classes factorisé donnée dans la FIGURE 3.67.

```
REFINEMENT
   B_Oeuvre_r
REFINES
   B_Oeuvre
SEES
   Context
ABSTRACT_VARIABLES
   livre ,
   film ,
   oeuvre ,
   nbPages ,
   nbDuree ,
   nbAuteur
INVARIANT
    oeuvre ⊆ OEUVRE ∧
    nbAuteur ∈ oeuvre ↔ NAT ∧
    dom(livre_nbAuteur) ⊆ dom(nbAuteur) ∧
    dom(film_nbAuteur) ⊆ dom(nbAuteur) ∧
    dom(livre_nbAuteur)∩dom(film_nbAuteur)=∅ ∧
    ∀ll .( ll ∈ livre ⇒
          livre_nbAuteur( ll ) = nbAuteur(ll)) ∧
    ∀ff .( ff ∈film ⇒
          film_nbAuteur(ff) = nbAuteur(ff))
INITIALISATION
   livre := ∅ ||
   oeuvre := ∅ ||
   nbPages := ∅ ||
   film := ∅ ||
   nbAuteur := ∅ ||
   nbDuree := ∅
OPERATIONS
   rr ⟵ livre_getNbAuteur ( pp ) =
   PRE
      pp ∈ livre ∧ rr ∈ NAT
   THEN
      rr := nbAuteur ( pp )
   END;
   rr ⟵ getNbPages ( pp ) =
   PRE
      pp ∈ livre ∧ rr ∈ NAT
   THEN
      rr := nbPages ( pp )
   END;
   rr ⟵ film_getNbAuteur ( pp ) =
   PRE
      pp ∈ film ∧ rr ∈ NAT
   THEN
      rr := nbAuteur ( pp )
   END;
   rr ⟵ getNbDuree ( pp ) =
   PRE
      pp ∈ film ∧ rr ∈ NAT
   THEN
      rr := nbDuree ( pp )
   END
END
```

FIGURE 3.69 – Modélisation en **B** de l'état de spécification concrète

La machine B_Oeuvre_r factorise la variable **nbAuteur**. Une telle variable remplace les deux variables **livre_nbAuteur** et **film_nbAuteur**, qui ont été supprimées dans la partie Raffinement.

Les obligations de preuve générées relatives à la machine B_Oeuvre_r ont été déchargées automatiquement (voir Tableau 3.31).

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	4	0	4	0	100
livre_getNbAuteur	4	0	4	0	100
getNbPages	4	0	4	0	100
film_getNbAuteur	4	0	4	0	100
getDuree	4	0	4	0	100
B_Oeuvre_r	20	0	20	0	100

TABLE 3.31 – Tableau de l'état de la machine `B_Oeuvre_r`

3.8.6 Voir aussi

Le schéma d'évolution introduit par ce pattern peut être utilisé avec profit dans un processus de refactoring (voir partie III) permettant de faire améliorer la structure (ou la qualité) d'un logiciel OO existant. Un processus de raffinement avec preuve favorise plutôt l'obtention d'un logiciel correct par construction. Le pattern `Class_Abstraction` milite en faveur de la prise en compte des autres qualités logicielles telles que efficacité, évolutivité dès les phases initiales d'un processus de développement guidé par des raffinements successifs. D'ailleurs le risque de négliger la qualité efficacité lors d'un processus de raffinement avec preuve est évoqué dans [41].

3.9 Conclusion

Après avoir identifié des problèmes récurrents lors de la construction incrémentale des diagrammes de classes UML/OCL, nous avons apporté des solutions génériques à ces problèmes sous forme des patterns de raffinement. Ceux-ci sont décrits selon le même canevas comportant six rubriques : Intention, Motivation, Solution, Vérification, Exemple et Voir aussi. De plus, nous avons formalisé en B les patterns de raffinement proposés. Ceci nous a permis de définir d'une façon formelle en B **l'invariant de collage** liant les deux niveaux Spécification et Raffinement d'un pattern. Un tel invariant de collage peut être instancié avec profit lors d'un développement conjoint UML/B guidé par l'application des patterns de raffinement.

Dans le chapitre suivant, nous allons réutiliser les patterns de raffinement proposés afin de développer en UML/B pas-à-pas l'application Contrôle d'accès aux bâtiments [1, 13, 7].

Chapitre 4

Approche par raffinement des diagrammes de classes UML

4.1 Introduction

Dans ce chapitre, nous proposons une approche par raffinement des diagrammes de classes UML. Un tel chapitre comporte deux sections. La première section préconise une démarche de développement des diagrammes de classes UML guidée par les patterns de raffinement proposés entre autres dans le chapitre précédent. La deuxième applique la démarche préconisée sur une étude de cas : Contrôle d'accès aux bâtiments [1, 13, 7].

4.2 Développement conjoint UML/B

Dans cette section, nous proposons une démarche de développement des diagrammes de classes UML guidée par des patterns de raffinement. Une telle démarche permet à terme d'établir un diagramme de classes UML qui modélise les concepts métier de l'application et possède des propriétés jugées cohérentes couvrant les contraintes de l'application issues de son cahier des charges. La vérification et la validation des modèles UML/OCL sont assurées par des outils associés à la méthode formelle B tels que : prouveur des obligations de preuves, model-checker et animateur.

La démarche préconisée comporte quatre phases : Réécriture du cahier des charges, Stratégie de raffinement, Spécification abstraite et Raffinement.

4.2.1 Réécriture du cahier des charges

Actuellement, les cahiers des charges sont souvent de mauvaise qualité. J. R. Abrial [1, 95] leur reproche d'être trop orientés vers une solution et de présenter des mécanismes de réalisation au détriment de l'explicitation des propriétés du système à concevoir. Nous préconisons de réécrire le cahier des charges de façon à mettre en exergue les propriétés du futur système et faciliter l'élaboration d'une stratégie de raffinement appropriée. Pour y parvenir, nous utilisons les recommandations de J. R. Abrial [3] distinguant les propriétés fonctionnelles, de sûreté et de vivacité.

4.2.2 Stratégie de raffinement

La réécriture du cahier des charges facilite l'élaboration d'une stratégie de raffinement adéquate. Mais, ceci ne garantit pas l'obtention d'une stratégie de raffinement "optimale". Des travaux permettant de comparer des stratégies de raffinement alternatives pour un domaine d'application donné, en occurrence des systèmes réactifs, commencent à apparaître [95, 3].

4.2.3 Spécification abstraite

Cette phase a pour objectif d'établir un modèle abstrait UML/OCL décrit par un diagramme de classes en se basant sur la stratégie de raffinement arrêtée précédemment. Le diagramme de classes UML/OCL produit est traduit en B afin de vérifier formellement sa cohérence.

4.2.4 Raffinement

Le processus de raffinement comporte plusieurs étapes. Chaque étape de raffinement prend en entrée trois paramètres : le diagramme de classes de niveau i, le catalogue des patterns de raffinement proposé (voir chapitre 3) et les propriétés issues du cahier des charges à prendre en considération et produit en sortie le diagramme de classes de niveau i+1 (FIGURE 4.1).

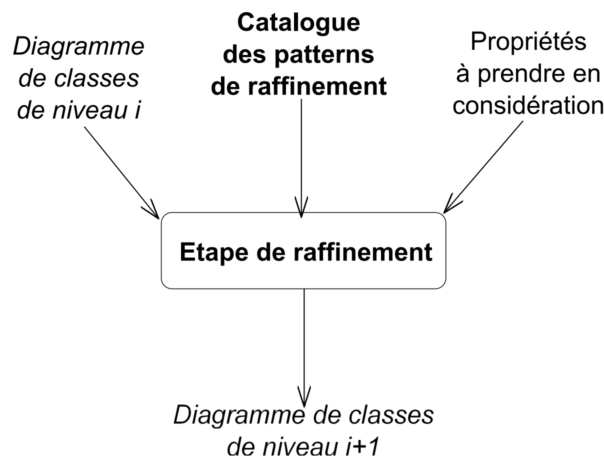


FIGURE 4.1 – Etape de raffinement

La prise en compte des propriétés qui guident le processus de raffinement peut être réalisée en appliquant des patterns de raffinement. La vérification formelle de la correction de l'étape de raffinement est confiée à l'***Atelier B*** moyennant la traduction de deux diagrammes de classes de deux niveaux en B. L'invariant de collage en B liant les deux niveaux (abstrait et affiné) peut être établi en réutilisant la formalisation en B des patterns de raffinement proposés dans le chapitre 3.

Le processus de raffinement se termine quand toutes les propriétés explicitées par le cahier des charges sont prises en compte conformément à la stratégie de raffinement adoptée. Ainsi, l'ultime diagramme de classes UML/OCL obtenu modélise les concepts métier du système à réaliser. En outre, il renferme des propriétés essentielles jugées formellement cohérentes.

4.3 Etude de cas : Contrôle d'accès aux bâtiments

Dans cette section, nous appliquons la démarche préconisée dans la section précédente sur l'étude de cas : Contrôle d'accès aux bâtiments [1, 13, 7].

4.3.1 Cahier des charges

Notre objectif est de développer un système chargé de contrôler l'accès de personnes aux divers bâtiments d'un lieu de travail [1, 7]. Le contrôle s'effectue à partir des autorisations affectées aux personnes concernées. Une autorisation permet à une personne, sous le contrôle du système, d'entrer dans certains bâtiments et pas dans d'autres. Les autorisations sont permanentes, c'est-à-dire qu'elles ne peuvent pas être modifiées pendant le fonctionnement du système. Lorsqu'une personne est à l'intérieur d'un bâtiment,

sa sortie doit aussi être contrôlée de façon à ce qu'il soit possible de connaître, à tout instant, qui se trouve dans un bâtiment donné.

Une personne ne peut se déplacer d'un bâtiment à un autre que si ces deux bâtiments communiquent entre eux. La communication entre les bâtiments s'effectue à travers des portes à sens unique. Chaque porte a un bâtiment origine et un bâtiment destination. Une personne peut entrer dans un bâtiment en franchissant une porte si elle est débloquée. Les portes étant physiquement bloquées, une porte se débloque pour une seule personne autorisée demandant à entrer dans le bâtiment.

Un voyant vert associé à chaque porte est allumé lorsque l'accès demandé est autorisé, condition nécessaire au déblocage de la porte. De même, un voyant rouge associé à chaque porte s'allume lorsque l'accès demandé pour cette porte est refusé.

Chaque personne dispose d'une carte magnétique. Des lecteurs de cartes sont installés à chaque porte permettant de lire les informations contenues sur une carte. A proximité de chaque lecteur, on trouve un tourniquet qui est normalement bloqué : personne ne peut le franchir sans le contrôle du système. Chaque tourniquet est équipé par une horloge qui conditionne en partie son comportement.

4.3.2 Réécriture du cahier des Charges

La réécriture du cahier des charges de l'application contrôle d'accès aux bâtiments présenté précédemment a pour objectif de mettre en avant les propriétés de cette application. Afin de classer ces propriétés, nous avons utilisé les étiquettes suivantes :

- EQU-Equipement permettant de référencer la description d'un équipement utilisé par l'application.
- FUN-Equipement/Acteur permettant de référencer une fonctionnalité attachée à un équipement ou un acteur.
- MODELE-FUN-Numéro permettant de référencer une fonctionnalité assurée par l'application.
- FUN-MODELE permettant de référencer la fonction principale de l'application.

Dans la suite, nous allons énumérer les différents propriétés de l'application contrôle d'accès aux bâtiments. Chaque propriété est décrite par un texte relativement court et une référence.

Propriété	Référence
Le système est chargé de contrôler l'accès d'un ensemble de personnes à un ensemble de bâtiments.	FUN-MODELE
Chaque personne est autorisée à pénétrer dans certains bâtiments (et pas dans d'autres). Les bâtiments non consignés dans cette autorisation sont implicitement interdits. Il s'agit d'une affectation permanente.	MODELE-FUN-1
Toute personne se trouvant dans un bâtiment est bien autorisée à y être.	MODELE-FUN-2
La géométrie des bâtiments sert à définir quels bâtiments peuvent communiquer entre eux et dans quel sens.	MODELE-FUN-3
Un bâtiment ne communique pas avec lui-mémé.	MODELE-FUN-4
Une personne ne peut se déplacer d'un bâtiment où elle se trouve à un autre où elle désire aller que si ces deux bâtiment communiquent bien entre eux.	MODELE-FUN-5
Toute personne autorisée à se trouver dans un bâtiment doit aussi être autorisée à aller dans un autre bâtiment qui communique avec le premier.	MODELE-FUN-6
Les bâtiments communiquent entre eux au moyen de portes, qui sont à sens unique. On peut donc parler des bâtiments origine et destination de chaque porte.	EQU-DOOR
Une porte ne peut être franchie que si elle est débloquée. Une porte ne peut être débloquée que pour une seule personne à la fois. Inversement toute personne impliquée dans le déblocage d'une porte ne peut pas l'être dans celui d'une autre.	FUN-DOOR-1

Lorsqu'une porte est débloquée pour une certaine personne, celle-ci se trouve dans le bâtiment origine de la porte en question. Par ailleurs, cette personne est bien autorisée à aller dans le bâtiment destination de cette même porte.	FUN-DOOR-2
Lorsqu'une porte est débloquée pour une certaine personne, celle-ci se trouve dans le bâtiment origine de la porte en question. Par ailleurs, cette personne est bien autorisée à aller dans le bâtiment destination de cette même porte.	FUN-PERSON
Un voyant vert associé à chaque porte	EQU-GREENLIGHT
Un voyant est allumé lorsque l'accès demandé est autorisé, condition nécessaire au déblocage de la porte,	FUN-GREENLIGHT
Un voyant rouge associé à chaque porte	EQU-REDLIGHT
Le voyant rouge d'une porte dont l'accès vient d'être refusé s'allume.	FUN-REDLIGHT
Les voyants rouge et vert d'une même porte ne peuvent pas être allumés simultanément.	FUN-LIGHT
Chaque personne dispose d'une carte magnétique qui présente ses autorisations aux différents bâtiments.	EQU-CARD
Des lecteurs de cartes sont installés à chaque porte permettant de lire les informations contenues sur une carte.	EQU-CARDREADER

4.3.3 Stratégie de raffinement

Le tableau 4.1 précise l'ordre de prise en compte des propriétés ou exigences de l'application contrôle d'accès aux bâtiments. Ceci définit notre stratégie de raffinement pour le développement incrémental conjoint UML/B de cette application. Le modèle initial se limite aux propriétés de base abstraites de l'application. Chaque étape de raffinement intègre un petit nombre des propriétés en allant de l'abstrait vers le concret. Le processus de raffinement se termine lorsque toutes les propriétés issues du cahier des charges réécris ont été bel et bien prises en compte. Les équipements tels que porte, carte, voyant utilisés par l'application contrôle d'accès aux bâtiments sont introduits lors des étapes de raffinement finales du processus de raffinement adopté.

Modèle	Equipements et Fonctions
Initial	FUN-MODELE, MODELE-FUN-2
Premier	MODELE-FUN-1
Deuxième	MODELE-FUN-3, MODELE-FUN-4, MODELE-FUN-5, MODELE-FUN-6
Troisième	EQU-DOOR, FUN-DOOR-1
Quatrième	FUN-DOOR-2, FUN-PERSON
Cinquième	EQU-GREENLIGHT, FUN-GREENLIGHT, EQU-REDLIGHT, FUN-REDLIGHT
Sixième	FUN-LIGHT
Septième	EQU-CARD, EQU-CARDREADER

TABLE 4.1 – Tableau de synthèse de stratégie de raffinement

4.3.4 Etapes de raffinement

Modèle initial

Nous commençons le développement par un diagramme de classes simple et très abstrait qui prend en considération seulement les propriétés (FUN-MODELE et MODELE-FUN-2) (voir FIGURE 4.2).

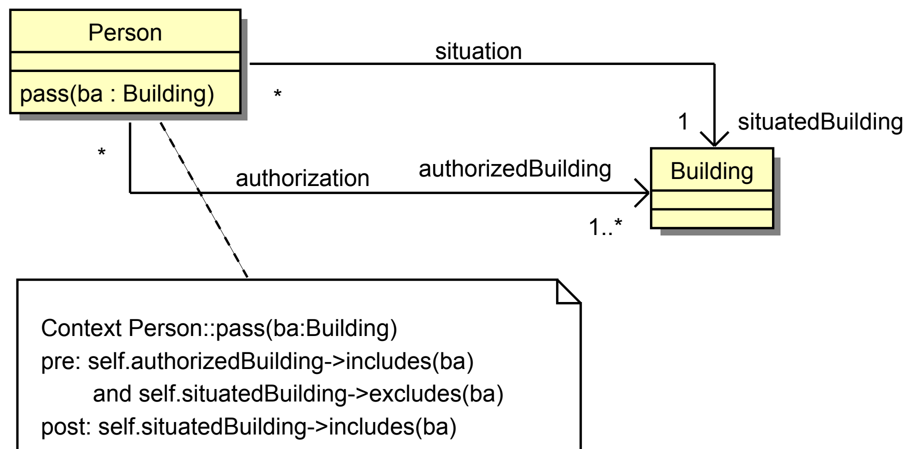
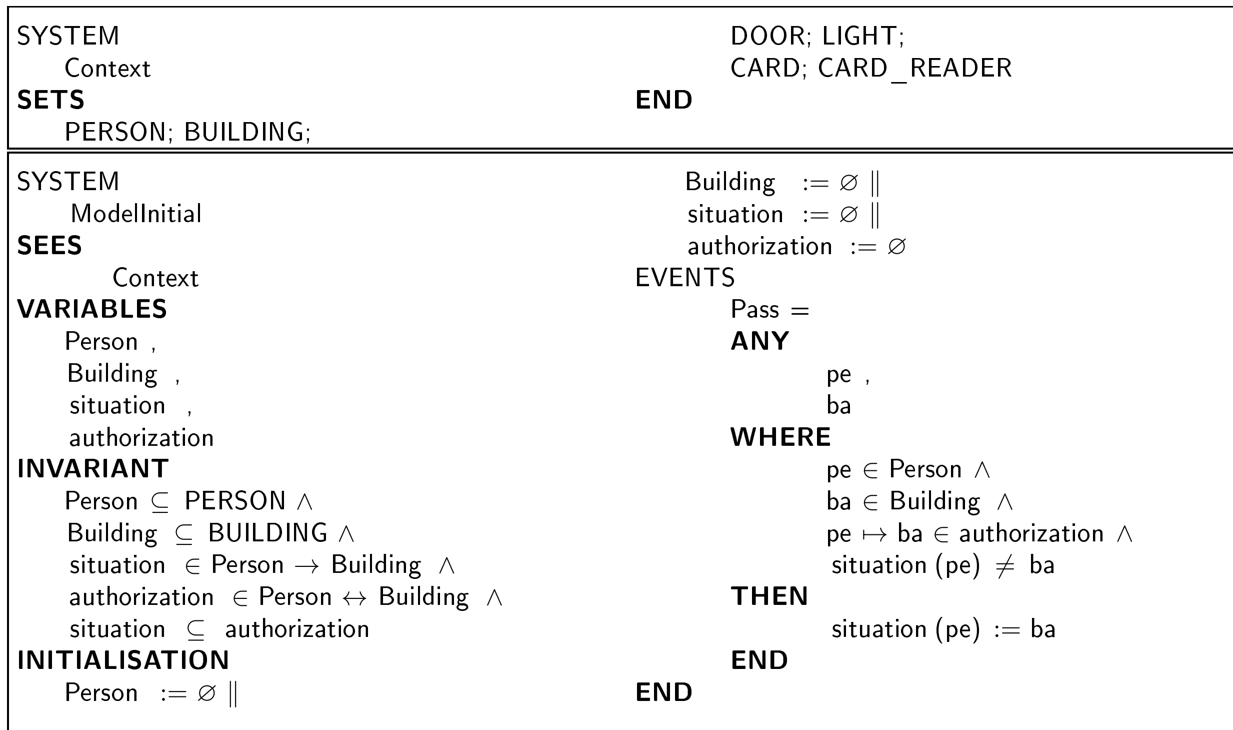


FIGURE 4.2 – Modèle initial

Vérification du modèle

La FIGURE 4.3 représente la transformation en B du modèle initial.



```
SYSTEM
    Context
SETS
    PERSON; BUILDING;
    DOOR; LIGHT;
    CARD; CARD_READER
END
```

```
SYSTEM
    ModelInitial
SEES
    Context
VARIABLES
    Person ,
    Building ,
    situation ,
    authorization
INVARIANT
    Person ⊆ PERSON ∧
    Building ⊆ BUILDING ∧
    situation ∈ Person → Building ∧
    authorization ∈ Person ↔ Building ∧
    situation ⊆ authorization
INITIALISATION
    Person := ∅ ||
    Building := ∅ ||
    situation := ∅ ||
    authorization := ∅
EVENTS
    Pass =
    ANY
        pe ,
        ba
    WHERE
        pe ∈ Person ∧
        ba ∈ Building ∧
        pe ↦ ba ∈ authorization ∧
        situation (pe) ≠ ba
    THEN
        situation (pe) := ba
    END
END
```

FIGURE 4.3 – Modélisation en **B** de l'état de spécification abstraite

Les obligations de preuve générées relatives à la machine `ModelInitial` ont été toutes déchargées dont une de façon interactive et huit de façon automatique (voir Tableau 4.2).

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	6	0	6	0	100
pass	3	1	2	0	100
ModelInitial	9	0	9	0	100

TABLE 4.2 – Tableau de l'état de la machine `ModelInitial`

Premier raffinement

Dans cette étape, nous tenons compte de la propriété (MODELE-FUN-1). Cette propriété consiste à dire que les autorisations, présentées par l'association `authorization` sont permanentes. Pour cela, nous pouvons appliqué le pattern de raffinement de données présenté dans la section 3.7.6. L'application de ce pattern sur le modèle présenté dans la FIGURE 4.2 génère le diagramme de classes donné dans la FIGURE 4.4.

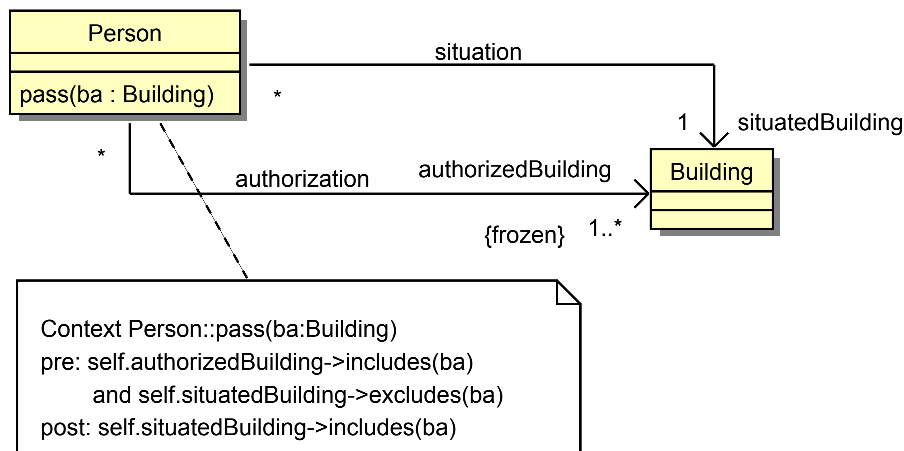


FIGURE 4.4 – Premier raffinement

Vérification du modèle

Le premier raffinement ne nécessite pas une vérification particulière.

Deuxième raffinement

Dans cette étape, nous injectons dans notre système les propriétés (MODELE-FUN-3, MODELE-FUN-4, MODELE-FUN-5 et MODELE-FUN-6). Ces propriétés permettent d'introduire la notion de communication entre bâtiments. Une personne ne peut se déplacer d'un bâtiment à un autre que si ces deux bâtiments communiquent entre eux.

La communication est introduite dans le diagramme de classes par une association récursive sur la classe `Building` (voir FIGURE 4.5). Un tel raffinement nécessite une réécriture de la sémantique OCL de l'opération `pass`. Ainsi, nous allons réutilisé le pattern de raffinement `Refinement_Operation`.

Vérification du modèle

La FIGURE 4.6 représente la transformation en B du deuxième raffinement.

Les obligations de preuve générées relatives à la machine `DeuxiemeRaffinement` ont été toutes déchargées automatiquement (voir Tableau 4.3).

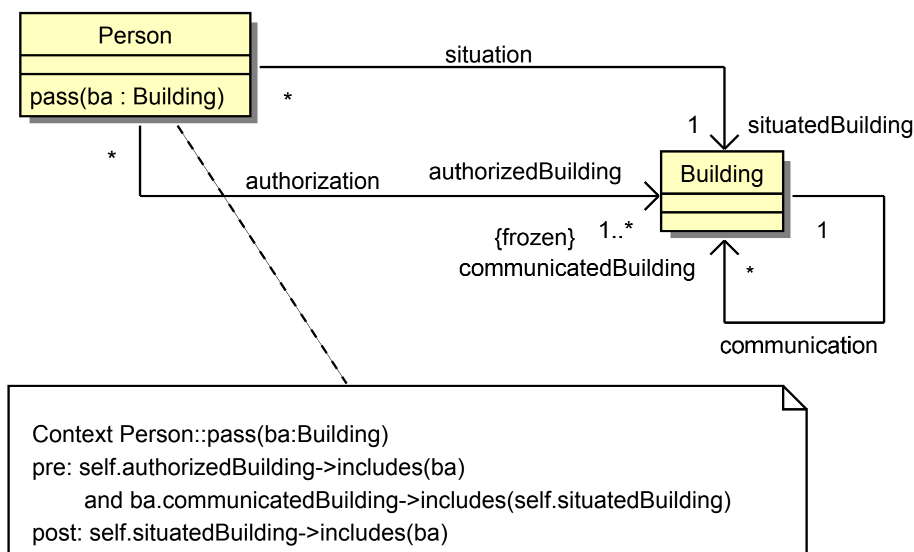
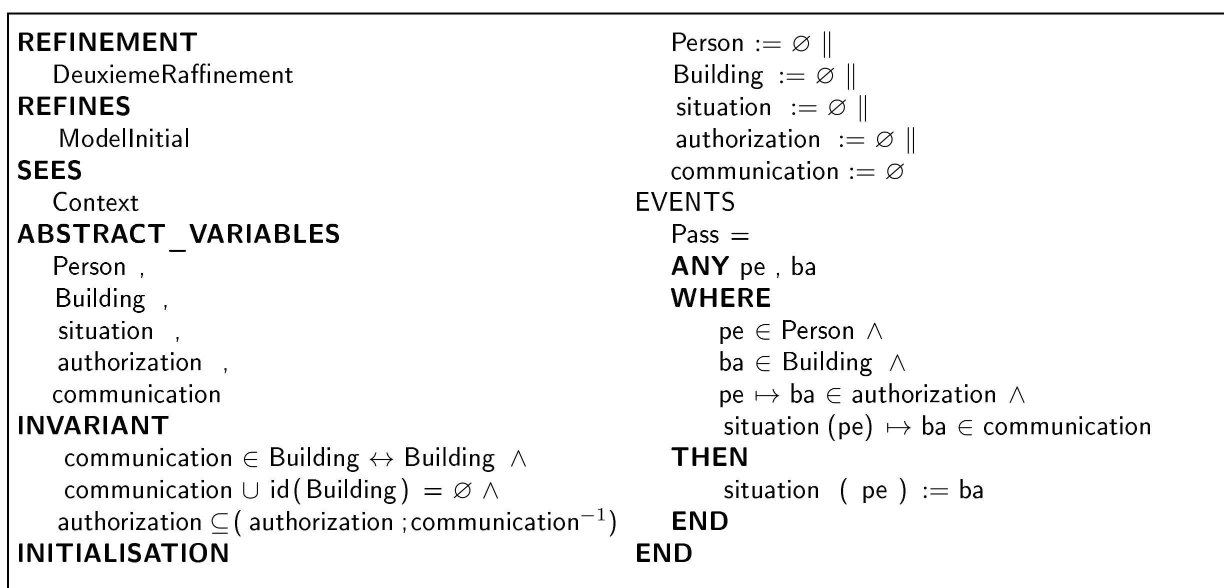


FIGURE 4.5 – Deuxième raffinement



```
REFINEMENT
   DeuxiemeRaffinement
REFINES
   ModelInitial
SEES
   Context
ABSTRACT_VARIABLES
   Person ,
   Building ,
   situation ,
   authorization ,
   communication
INVARIANT
   communication ∈ Building ↔ Building ∧
   communication ∪ id(Building) = ∅ ∧
   authorization ⊆ ( authorization ;communication⁻¹)
INITIALISATION
   Person := ∅ ||
   Building := ∅ ||
   situation := ∅ ||
   authorization := ∅ ||
   communication := ∅
EVENTS
   Pass =
   ANY pe , ba
   WHERE
      pe ∈ Person ∧
      ba ∈ Building ∧
      pe ↦ ba ∈ authorization ∧
      situation (pe) ↦ ba ∈ communication
   THEN
      situation ( pe ) := ba
   END
END
```

FIGURE 4.6 – Modélisation en **B** de l'état de deuxième raffinement

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	3	0	3	0	100
pass	1	0	1	0	100
DeuxiemeRaffinement	4	0	4	0	100

TABLE 4.3 – Tableau de l'état de la machine `DeuxiemeRaffinement`

Troisième raffinement

Le troisième raffinement consiste à ajouter un nouveau équipement (EQU-DOOR et FUN-DOOR-2). Une porte permet de faire la liaison entre deux bâtiments. Ceci nous amène à préciser la notion de bâtiment : chaque porte a un bâtiment origine et un bâtiment destination. En effet, la communication entre les bâtiments est à travers une porte. Ainsi, nous devons supprimer l'association `communication`, introduite dans le raffinement précédent, et de la remplacer par deux associations `destination` entre `Door` et `Building` et `origin` entre `Door` et `Building`. Cette modification peut être obtenue par l'application du pattern de raffinement `Class_Helper`, avec :

- `communication` correspond à `association`,
- `origin` correspond à `association1`,
- `destination` correspond à `association2`,
- `Door` correspond à `Helper`,
- `Building` correspond à la fois aux `P1` et `P2`.

Enfin, la propriété (FUN-DOOR-2) consiste à dire qu'une porte est un composant d'un bâtiment. Ainsi, nous introduisons une relation de composition entre `Door` et `Building` (voir FIGURE 4.7).

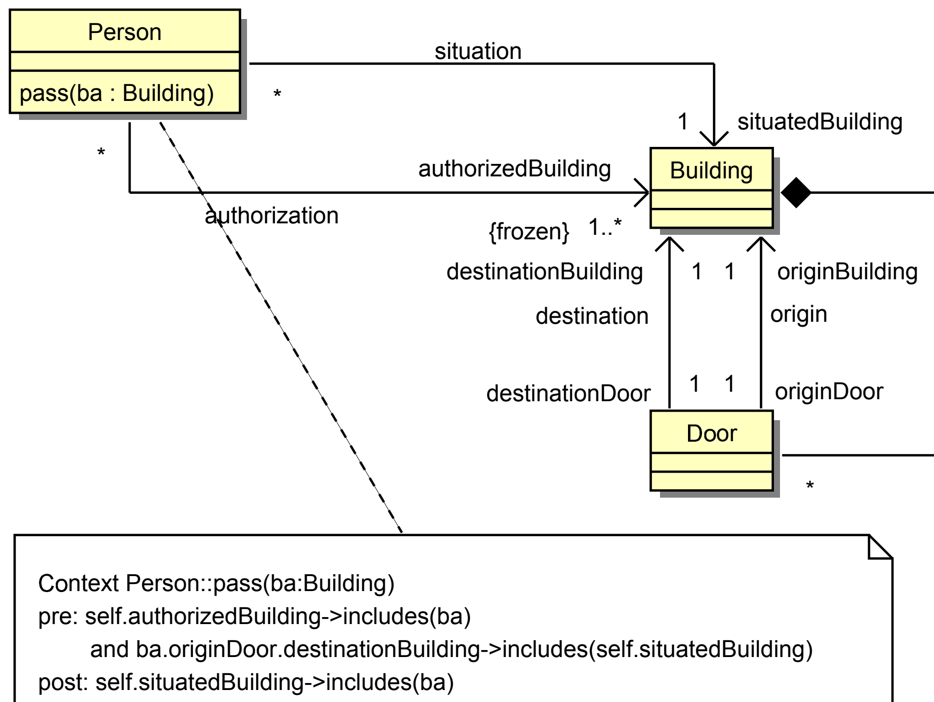
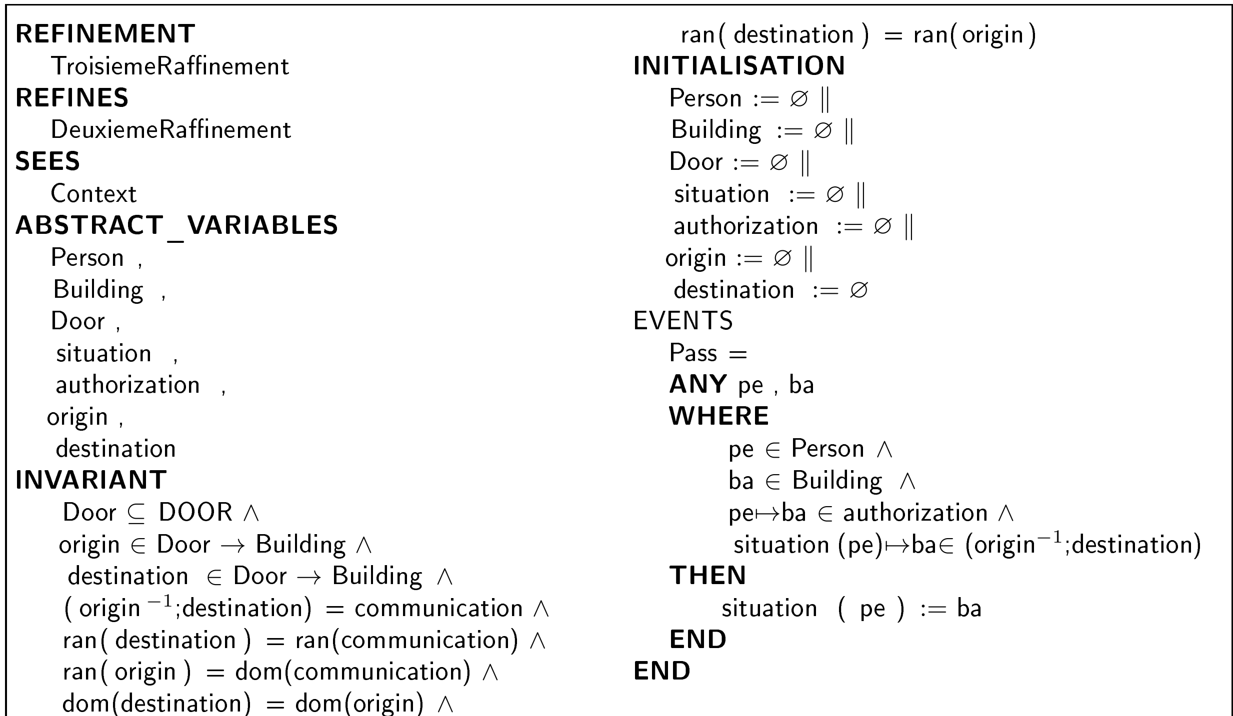


FIGURE 4.7 – Troisième raffinement

Vérification du modèle

La FIGURE 4.8 représente la transformation en B du troisième raffinement. L'invariant de la machine `TroisiemeRaffinement` est issu de l'invariant de collage du pattern `Class_Helper` (Inv_Collage_Class_Helper) (3.1).

Les obligations de preuve générées relatives à la machine `TroisiemeRaffinement` ont été toutes déchargées automatiquement (voir Tableau 4.4).



```
REFINEMENT
  TroisiemeRaffinement
REFINES
  DeuxiemeRaffinement
SEES
  Context
ABSTRACT_VARIABLES
  Person ,
  Building ,
  Door ,
  situation ,
  authorization ,
  origin ,
  destination
INVARIANT
  Door ⊆ DOOR ∧
  origin ∈ Door → Building ∧
  destination ∈ Door → Building ∧
  ( origin ⁻¹;destination) = communication ∧
  ran( destination ) = ran(communication) ∧
  ran( origin ) = dom(communication) ∧
  dom(destination) = dom(origin) ∧
  ran( destination ) = ran( origin )
INITIALISATION
  Person := ∅ ||
  Building := ∅ ||
  Door := ∅ ||
  situation := ∅ ||
  authorization := ∅ ||
  origin := ∅ ||
  destination := ∅
EVENTS
  Pass =
  ANY pe , ba
  WHERE
    pe ∈ Person ∧
    ba ∈ Building ∧
    pe↦ba ∈ authorization ∧
    situation (pe)↦ba∈ (origin⁻¹;destination)
  THEN
    situation ( pe ) := ba
  END
END
```

FIGURE 4.8 – Modélisation en **B** de l'état de troisième raffinement

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	6	0	6	0	100
pass	1	0	1	0	100
TroisiemeRaffinement	7	0	7	0	100

TABLE 4.4 – Tableau de l'état de la machine `TroisiemeRaffinement`

Quatrième raffinement

La quatrième étape de raffinement consiste à définir les fonctionnalités de la classe `Door` introduite dans l'étape précédente (FUN-Door-2). Une telle transformation nécessite la révision de la sémantique de l'opération `pass`. En effet, la propriété (FUN-PERSON) consiste à dire qu'une personne doit se présenter devant une porte pour passer d'un bâtiment à un autre. Ceci nécessite l'introduction d'une association `acceptance` entre `Person` et `Door`. Ainsi, l'opération `pass` ne doit pas prendre une instance de la classe `Building` en paramètre formel mais plutôt une instance de la classe `Door`. Pour cela nous appliquons le pattern de raffinement `Refinement_Operation` pour générer le diagramme de classes de la FIGURE 4.9. Le nouveau diagramme de classes présenté par la FIGURE 4.9 peut être généré par l'application du pattern de raffinement `Refinement_Operation`.

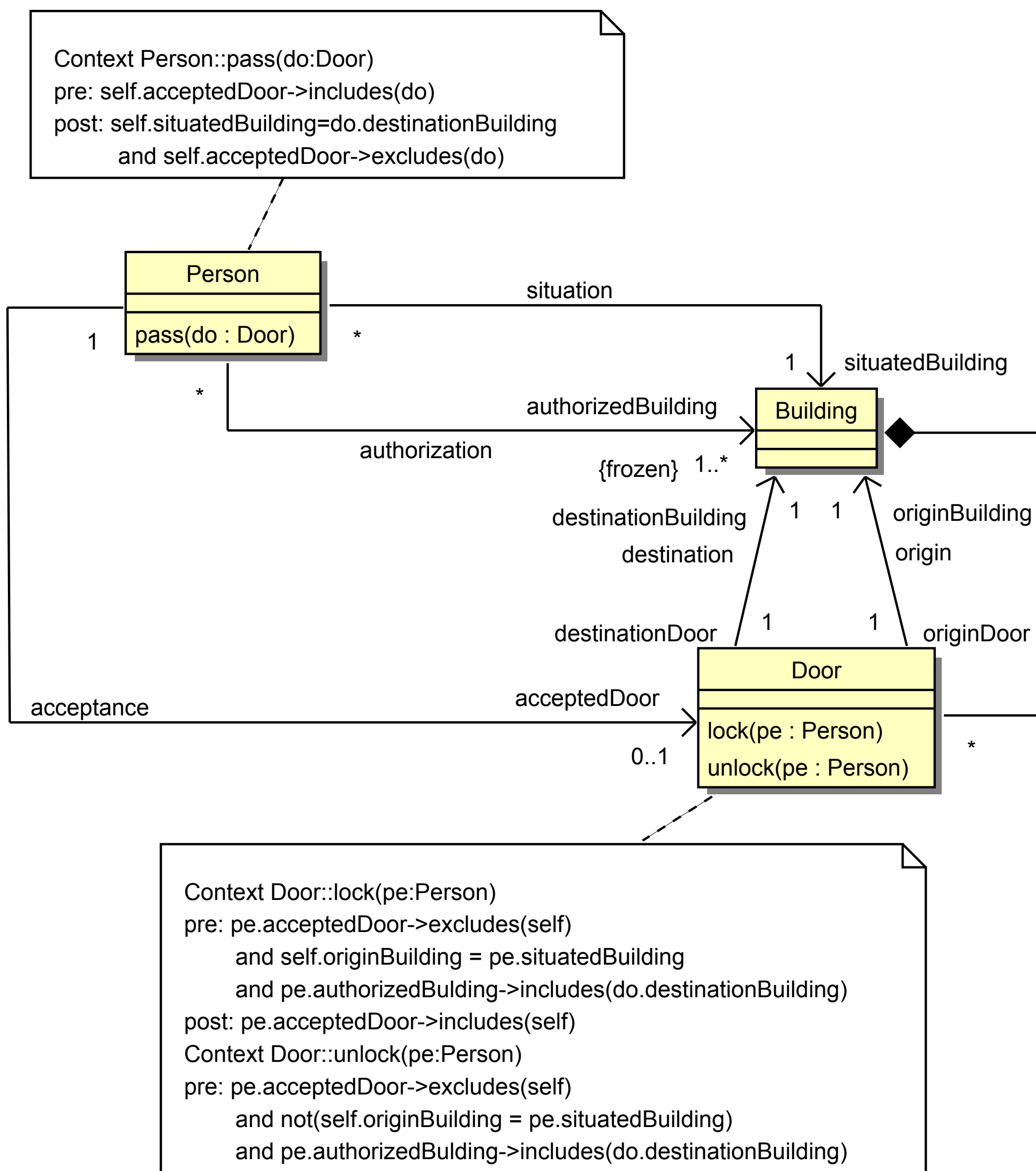


FIGURE 4.9 – Quatrième raffinement

Vérification du modèle

La FIGURE 4.10 représente la transformation en B du quatrième raffinement.

```
REFINEMENT
   QuatriemeRaffinement
REFINES
   TroisiemeRaffinement
SEES
   Context
ABSTRACT_VARIABLES
   Person ,
   Building ,
   Door ,
   situation ,
   authorization ,
   origin ,
   destination ,
   acceptance
INVARIANT
   acceptance ∈ Person ⤔ Door ∧
   (acceptance; origin ) ⊆ situation ∧
   (acceptance; destination ) ⊆ authorization
INITIALISATION
   Person := ∅ ||
   Building := ∅ ||
   Door := ∅ ||
   situation := ∅ ||
   authorization := ∅ ||
   origin := ∅ ||
   destination := ∅ ||
   acceptance := ∅
EVENTS
   Pass =
   ANY
        do
   WHERE
        do ∈ Door ∧
        do ∈ ran(acceptance)
   THEN
    situation (acceptance⁻¹(do)):=destination(do)||
    acceptance := acceptance ⩥ {do}
   END;

   lock =
   ANY
        do , pe
   WHERE
        pe ∈ Person ∧
        do ∈ Door ∧
        do ∉ ran(acceptance) ∧
        origin (do) = situation (pe) ∧
        pe ↦ destination(do) ∈ authorization ∧
        pe ∉ dom(acceptance)
   THEN
        acceptance(pe) := do
   END;

   unlock =
   ANY
        do , pe
   WHERE
        do ∈ Door ∧
        pe ∈ Person ∧
        do ∉ ran(acceptance) ∧
        ¬(origin(do) = situation (pe) ∧
        pe ↦ destination(do) ∈ authorization ∧
        pe ∉ dom(acceptance))
   THEN
        skip
   END
END
```

FIGURE 4.10 – Modélisation en **B** de l'état de quatrième raffinement

Les obligations de preuve générées relatives à la machine `QuatriemeRaffinement` ont été toutes déchargées dont trois de façon interactive et neuf de façon automatique (voir Tableau 4.5).

Cinquième raffinement

Dans cette étape, nous tenons compte des propriétés (EQU-GREENLIGHT, FUN-GREENLIGHT, EQU-REDLIGHT et FUN-REDLIGHT). Ces propriétés définissent deux nouveaux équipements avec leurs caractéristiques comme composants de la classe `Door`. L'application du pattern de raffinement `Class_Decomposition`, avec composition comme relation reliant `Door` et ses composants `GreenLight` et `RedLight`, génère le diagramme de classes présenté dans la FIGURE 4.11.

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	3	0	3	0	100
pass	5	3	2	0	100
lock	4	0	4	0	100
unlock	0	0	0	0	100
QuatriemeRaffinment	12	3	9	0	100

TABLE 4.5 – Tableau de l'état de la machine `QuatriemeRaffinement`

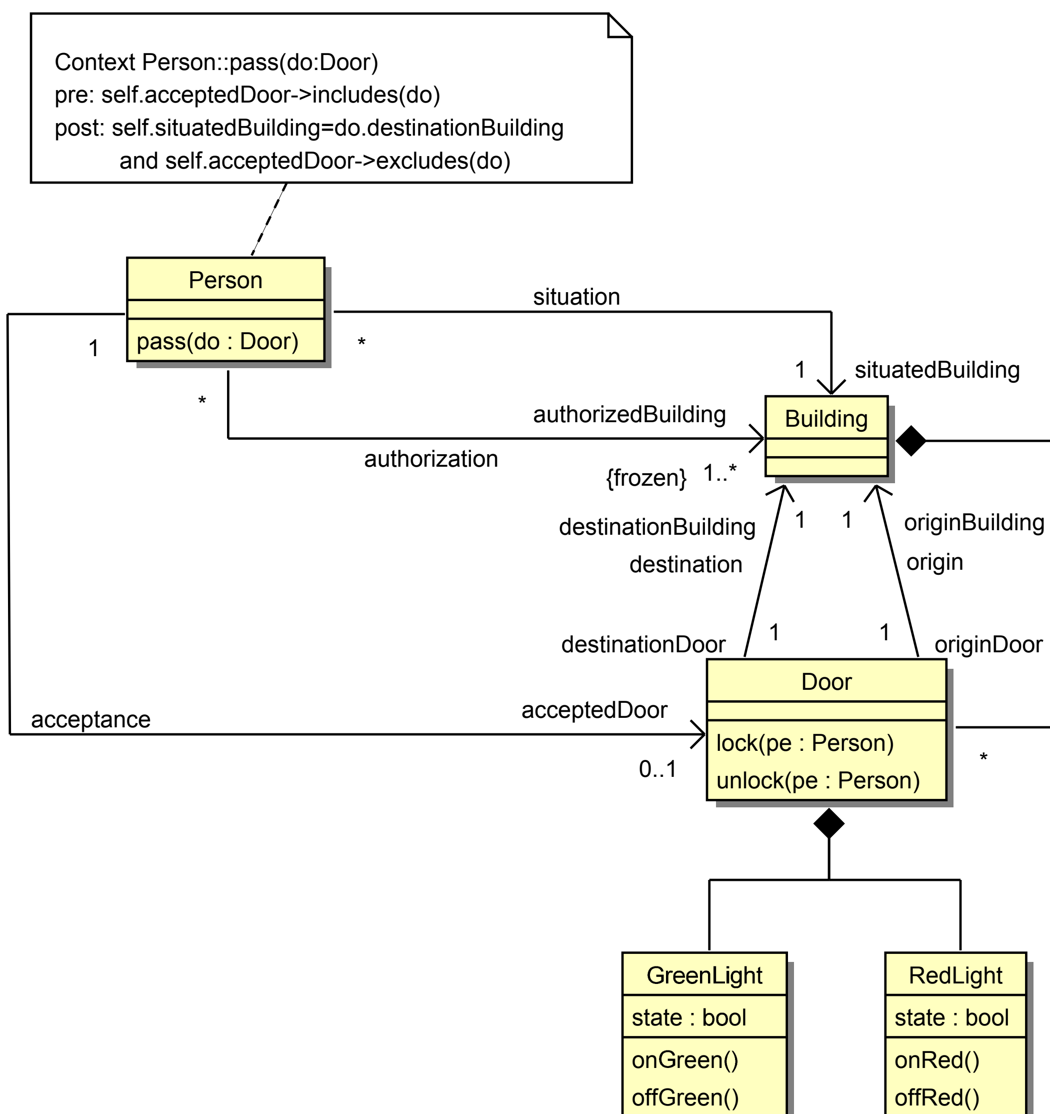


FIGURE 4.11 – Cinquième raffinement

Vérification du modèle

La Figure 4.12 représente la transformation en B du cinquième raffinement.

```
REFINEMENT
   CinquiemeRaffinement
REFINES
   QuatriemeRaffinement
SEES
   Context
ABSTRACT_VARIABLES
   Person, Building,
   Door, situation ,
   authorization ,
   origin, destination,
   acceptance, RedLight,
   GreenLight, stateGreen,
   stateRed
INVARIANT
    stateGreen ∈ Door ↔ 𝔹 ∧
        stateRed ∈ Door ↔ 𝔹 ∧
        GreenLight ⊆ Door ∧
        RedLight ⊆ Door ∧
        RedLight ∩ GreenLight = ∅ ∧
        RedLight ∪ GreenLight = Door
INITIALISATION
   Person := ∅ || Building := ∅ ||
   Door := ∅ || situation := ∅ ||
   authorization := ∅ ||
   origin := ∅ || destination := ∅ ||
   acceptance := ∅ || RedLight := ∅ ||
   GreenLight := ∅ || stateGreen := ∅ ||
   stateRed := ∅
EVENTS
   Pass = ANY do
   WHERE do ∈ Door ∧ do ∈ ran ( acceptance )
   THEN
       situation (acceptance⁻¹(do)):=destination(do) ||
       acceptance := acceptance ⩥ { do }
   END;
   lock = ANY pe, do
   WHERE
       pe ∈ Person ∧ do ∈ Door ∧
       do ∉ (ran(acceptance) ∪ RedLight) ∧
       ¬(origin(do) = situation(pe) ∧
           pe ↦ destination(do) ∈ authorization ∧
           pe ∉ dom(acceptance))
   THEN
       RedLight := RedLight ∪ {do}
   END;
   unlock = ANY pe, do
   WHERE
       pe ∈ Person ∧ do ∈ Door ∧
       do ∉ (ran(acceptance) ∪ RedLight) ∧
       origin(do) = situation(pe) ∧
       pe ↦ destination(do) ∈ authorization ∧
       pe ∉ dom(acceptance)
   THEN
       acceptance(pe) := do
   END;
   onGreen = ANY light
   WHERE
       light ∈ GreenLight ∧
       stateGreen( light ) = FALSE
   THEN
       stateGreen( light ) := TRUE
   END;
   offGreen = ANY light
   WHERE
       light ∈ GreenLight ∧
       stateGreen( light ) = TRUE
   THEN
       stateGreen( light ) := FALSE
   END;
   onRed = ANY light
   WHERE
       light ∈ RedLight ∧
       stateRed( light ) = FALSE
   THEN
       stateRed( light ) := TRUE
   END;
   offRed = ANY light
   WHERE
       light ∈ RedLight ∧
       stateRed( light ) = TRUE
   THEN
       stateRed( light ) := FALSE
   END
END
```

Figure 4.12 – Modélisation en **B** de l'état du cinquième raffinement

Les obligations de preuve générées relatives à la machine `CinquiemeRaffinement` ont été toutes déchargées dont deux de façon interactive et dix de façon automatique (voir Tableau 4.6).

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	2	0	2	0	100
pass	1	1	0	0	100
lock	3	1	2	0	100
unlock	2	0	2	0	100
onGreen	1	0	1	0	100
offGreen	1	0	1	0	100
onRed	1	0	1	0	100
offRed	1	0	1	0	100
CinquiemeRaffinment	12	2	10	0	100

TABLE 4.6 – Tableau de l'état de la machine `CinquiemeRaffinement`

Sixième raffinement

Dans cette étape, nous remarquons la ressemblance entre les deux classes `GreenLight` et `RedLight` (FUN-LIGHT). Ainsi, nous décidons de factoriser les propriétés communes entre ces deux classes. L'application du pattern de raffinement `Class_Abstraction` génère le diagramme de classes présenté par la FIGURE 4.13. Le pattern `Class_Abstraction` permet d'introduire une nouvelle classe appelée `Light` regroupant les propriétés communes entre les deux classes `GreenLight` et `RedLight`.

Vérification du modèle

Le sixième raffinement ne nécessite pas une vérification particulière. En effet, nous retrouvons la même transformation en B que celle du cinquième raffinement.

Septième raffinement

Les dernières propriétés (EQU-CARD et EQU-CARDREADER) seront prises en compte dans cette dernière étape de raffinement. Deux classes intermédiaires peuvent être introduites :

- la classe `Card` associée à chaque personne,
- la classe `CardReader` associée à chaque porte d'un bâtiment.

Ces classes sont reliées de la manière suivante : `Card` est reliée à `Person`, `Card` est reliée à `CardReader` et `CardReader` est reliée à `Door`.

L'application du pattern de raffinement `Class_Helper` à l'association `acceptance` génère le diagramme de classes de la FIGURE 4.14.

Vérification du modèle

La FIGURE 4.15 représente la transformation en B du huitième raffinement.

Les obligations de preuve générées relatives à la machine `SeptiemeRaffinement` ont été toutes déchargées automatiquement (voir Tableau 4.7).

4.4 Conclusion

Nous avons proposé une démarche de développement des diagrammes de classes UML basée sur le concept raffinement avec preuves. Notre démarche comporte quatre phases : Réécriture du cahier des charges, Stratégie de raffinement, Spécification abstraite et Raffinement.

Nous avons appliqué cette démarche sur l'étude de cas : Contrôle d'accès aux bâtiments. Ceci nous a permis de réutiliser des patterns de raffinement proposés dans le chapitre 3 comme : `Class_Helper`, `Refinement_Operation`, `Class_Abstraction` et `Class_Decomposition`. En outre, le développement conjoint UML/B nous a permis de vérifier la correction de chaque étape de raffinement.

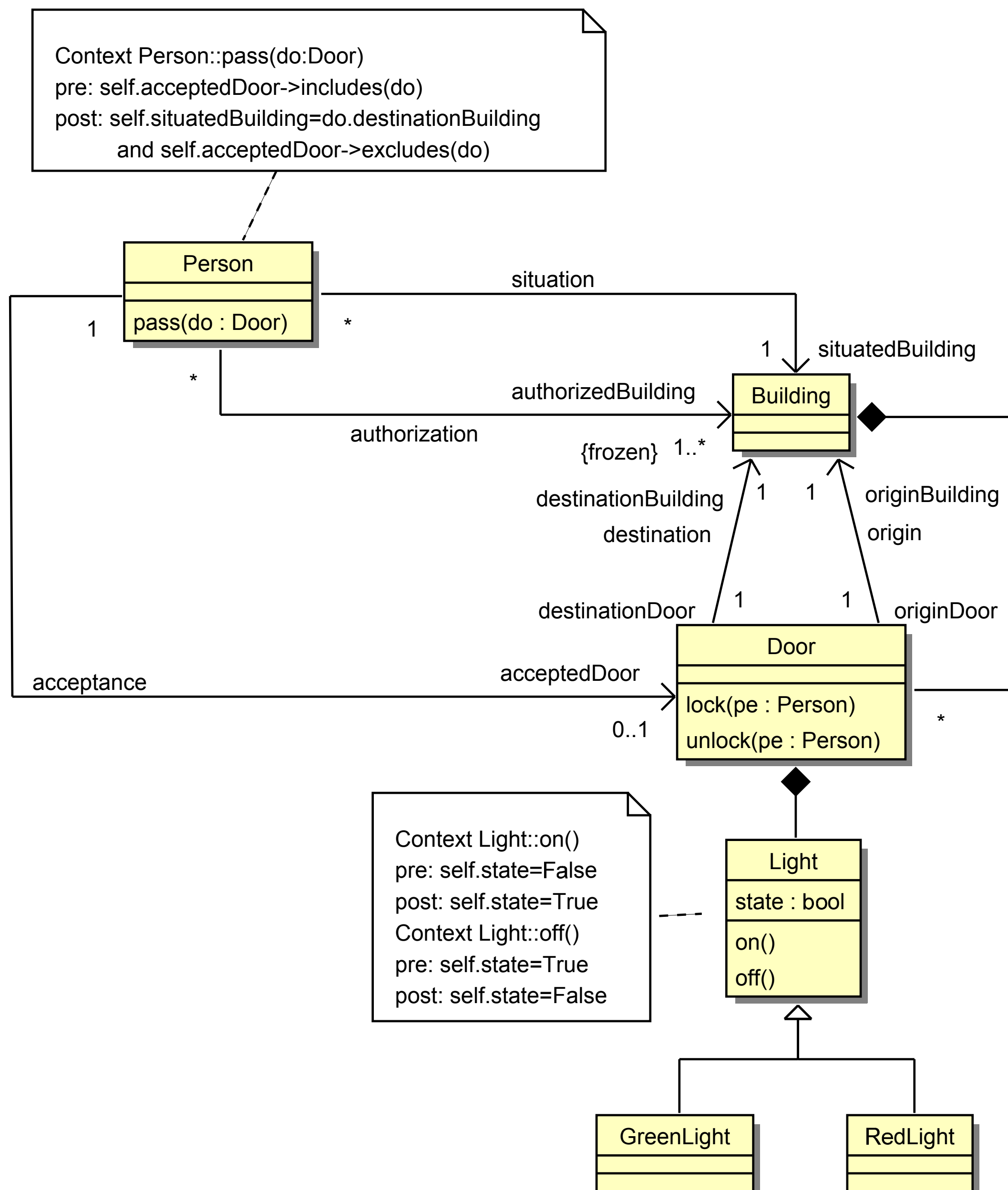


FIGURE 4.13 – Sixième raffinement

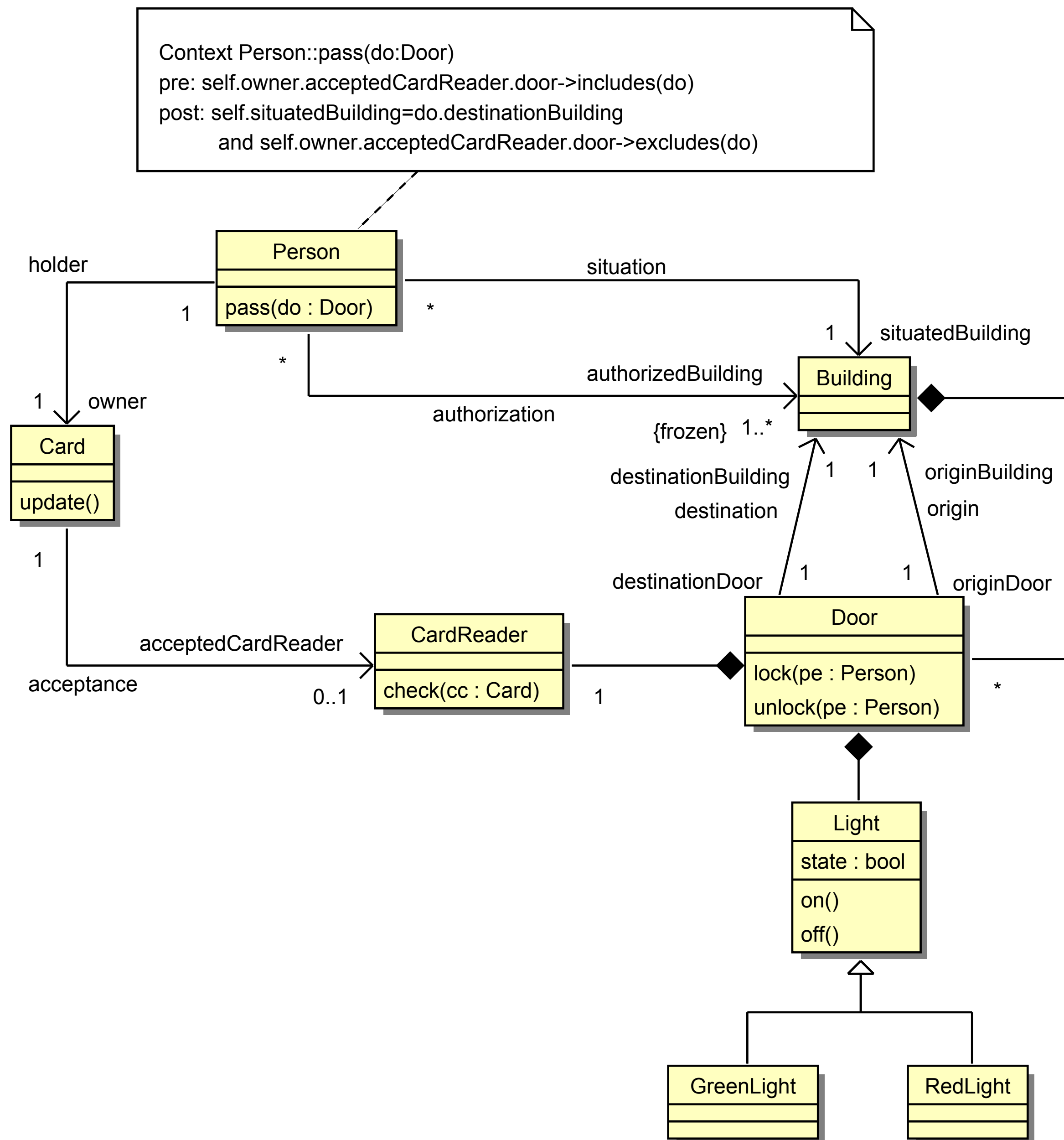


Figure 4.14 – Septième raffinement

```
REFINEMENT
   SeptiemeRaffinement
REFINES
   CinquiemeRaffinement
SEES
   Context
ABSTRACT_VARIABLES
   Person, Building, Door,
   situation , authorization ,
   origin, destination,
   acceptance1, RedLight,
   GreenLight, stateGreen,
   stateRed, Card,
   Card_Reader, holder
INVARIANT
    Card ⊆ CARD ∧
    Card_Reader ⊆ Door ∧
    holder ∈ Person → Card ∧
    acceptance1 ∈ Card ⤔ Card_Reader ∧
    (holder;acceptance1)=acceptance ∧
    ran(acceptance)=ran(acceptance1) ∧
    dom(acceptance)=dom(holder)∧
    ran(holder)=dom(acceptance1)
INITIALISATION
   Person := ∅ || Building := ∅ ||
   Door := ∅ || situation := ∅ ||
   authorization := ∅ || origin := ∅ ||
   destination := ∅ || acceptance1 := ∅ ||
   RedLight := ∅ || GreenLight := ∅ ||
   stateGreen := ∅ || stateRed := ∅ ||
   Card := ∅ || Card_Reader := ∅ ||
   holder := ∅
EVENTS
   Pass = ANY do
   WHERE
       do ∈ Door ∧
       do ∈ ran(acceptance1)
   THEN
       situation (holder⁻¹(acceptance1⁻¹(do))):=
                destination (do) ||
       acceptance1 := acceptance1 ⩥ {do}
   END;
   lock = ANY pe, do
   WHERE
       pe ∈ Person ∧ do ∈ Door ∧
       do ∉ ( ran ( acceptance1 ) ∪ RedLight ) ∧
       ¬(origin(do) = situation(pe) ∧
        pe ↦ destination(do) ∈ authorization ∧
        holder(pe) ∉ dom(acceptance1))
   THEN
       RedLight := RedLight ∪ { do }
   END;
   unlock = ANY pe, do
   WHERE
       pe ∈ Person ∧ do ∈ Door ∧
       origin(do) = situation(pe) ∧
           pe ↦ destination(do) ∈ authorization ∧
           holder(pe) ∉ dom(acceptance1)
   THEN
       acceptance1(holder(pe)) := do
   END;
   onGreen = ANY light
   WHERE
       light ∈ GreenLight ∧
       stateGreen ( light ) = FALSE
   THEN
       stateGreen ( light ) := TRUE
   END;
   offGreen = ANY light
   WHERE
       light ∈ GreenLight ∧
       stateGreen ( light ) = TRUE
   THEN
       stateGreen ( light ) := FALSE
   END;
   onRed = ANY light
   WHERE
       light ∈ RedLight ∧
       stateRed ( light ) = FALSE
   THEN
       stateRed ( light ) := TRUE
   END;
   offRed = ANY light
   WHERE
       light ∈ RedLight ∧
       stateRed ( light ) = TRUE
   THEN
       stateRed ( light ) := FALSE
   END;
   check = ANY cc, cr
   WHERE
       cc ∈ Card ∧ cr ∈ Card_Reader
   THEN skip
   END;
   update = ANY cc
   WHERE cc ∈ Card
   THEN skip
   END
END
```

FIGURE 4.15 – Modélisation en **B** de l'état de septième raffinement

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	9	0	9	0	100
pass	6	0	6	0	100
lock	1	0	1	0	100
unlock	6	0	6	0	100
onGreen	1	0	1	0	100
offGreen	1	0	1	0	100
onRed	1	0	1	0	100
offRed	1	0	1	0	100
check	0	0	0	0	100
update	0	0	0	0	100
SeptiemeRaffinment	26	0	26	0	100

TABLE 4.7 – Tableau de l'état de la machine `SeptiemeRaffinement`

Troisième partie

Approche de refactoring des spécifications UML

Introduction

Commençons par rappeler que UML définit quatre types de relations entre classes :

Relation d'association. Elle décrit un ensemble de liens, ou connexions entre classes. Dans les Langages Orientées objets (LOO), cette relation conceptuelle est connue sous le nom de relation client.

Relation de spécialisation/généralisation. C'est une relation conceptuelle qui permet à une classe appelée *sous-classe* d'hériter des caractéristiques de sa classe mère appelée *super-classe*. Dans les LOO, cette relation est connue sous le nom de relation d'héritage.

Relation de réalisation. C'est une relation dans laquelle une interface définit un contrat garanti par une classe d'implantation.

Relation de dépendance. C'est une relation qui ne nécessite pas forcément un lien entre les classes. Elle indique une utilisation entre ces classes. Lorsque cette relation est réalisée par des liens entre deux classes, elle présente l'utilisation que fait une classe d'une autre. Une classe dépend d'une autre si ses méthodes manipulent l'objet de cette classe.

De plus, UML définit plusieurs concepts. Nous détaillons, dans ce qui suit, ceux qui nous intéressent.

Délégation. Une classe peut déléguer une partie de son activité à une autre classe. En UML, le mécanisme de délégation d'opération est permis grâce à une relation de composition ou d'agrégation qui relie les deux classes.

Généricité d'une classe. Une classe UML peut avoir des paramètres génériques formels représentant des types ou des variables. En UML, les classes génériques sont appelées classes *template*. On ne peut pas utiliser un template directement, il faut d'abord l'instancier. L'instanciation implique en passant par la dépendance «bind» de lier ces paramètres génériques formels du template aux paramètres génériques réels. Ceci donne une classe concrète qui peut être utilisée exactement comme n'importe quelle classe ordinaire. Contrairement aux langages de programmation comme Eiffel [71], UML ne supporte pas la généricité contrainte exigeant l'introduction de l'héritage : les paramètres génériques formels représentant des types doivent descendre des types ascendants.

Polymorphisme. Dans le développement orienté objet, une entité variable ou un élément de structure de données peut prendre plusieurs formes, devenant attaché, lors de l'exécution, à des objets de types différents, sous contrôle d'une déclaration statique [71].

Dans cette partie, nous proposons des schémas de refactoring permettant l'adjonction des différentes relations et concepts que nous venons de décrire, dans une optique d'amélioration des facteurs qualité logiciel d'une spécification UML existante.

Certains schémas se proposent d'aider le concepteur à mieux ajouter des relations entre classes non liées et d'autre décrivent la nécessité d'introduire des concepts dans une classe donnée.

Les schémas de refactoring associés à l'héritage, l'association, la redéfinition et le polymorphisme sont paramétrés par :

- les deux classes concernées,
- les contraintes OCL attachées à chaque classe,
- les diagrammes d'états-transitions correspondants à la dynamique de chaque classe.

Tandis que les schémas de refactoring associés à la délégation, la généricité et la classe abstraite sont paramétrés par :

- la classe concernée,

- les contraintes OCL attachées à cette classe,
- le diagramme d'états-transitions correspondant à cette classe.

1 Canevas de présentation d'un schéma de refactoring

Afin d'appliquer un schéma de refactoring, la première étape consiste à identifier ses paramètres. Ensuite, il convient de s'assurer que ces paramètres vérifient un ensemble de conditions. Ensuite, nous détaillons les différentes mises à jours apportées par le schéma choisi. Enfin, nous vérifions le résultat produit par ce schéma de refactoring. Ainsi, la définition d'un schéma de refactoring est composée de quatre étapes :

1. Identification des paramètres
2. Vérification des conditions d'application
3. Evolution de la spécification
4. Correction du schéma

Une description informelle du schéma de refactoring, permettant de définir le concept introduit, débutera chaque canevas.

Une illustration de l'application du schéma sur l'étude de cas de contrôle d'accès à un bâtiment conclura chaque canevas.

1.1 Identification des paramètres

Elle présente les paramètres du schéma. Ces paramètres sont de la forme d'un ou de deux triplet(s) contenant chacun la classe concernée, les contraintes OCL attachées à cette classe et son diagramme d'états-transitions associé (voir FIGURE I) :

$$< Class, OCL_Class, STD_Class >$$

Où :
- *Class* représente la classe concernée. Elle est caractérisée par un ensemble de propriétés statiques, appelé l_attr_Class et un ensemble de propriétés dynamiques, appelé l_meth_Class.
- OCL_Class représente les contraintes OCL attachées à *Class*. Ces contraintes sont constituées des éléments suivants :
  - I_Class : l'invariant de *Classe*, condition qui doit être vérifiée pour tous les objets de la classe, à tous les moments stables [3], formalisé en OCL par :

```
Context Class
inv I_Class: condition
```

  - Pour chaque méthode $meth_Class$ de l_meth_Class :
    - P_meth_Class : la pré-condition de l'opération $meth_Class$, une condition qui doit être vérifiée avant l'exécution de $meth_Class$,
    - Q_meth_Class : la post-condition de l'opération $meth_Class$, une condition qui doit être vérifiée après l'exécution de $meth_Class$. Leur formalisation en OCL est la suivante :

```
Context Class :: meth_Class()
pre P_meth_Class: condition
post Q_meth_Class: condition
```

- STD_Class constitue le diagramme d'états-transitions qui représente la dynamique de *Class*.

3. Un "moment stable" correspond au moment qui suit la sortie d'une méthode de la classe. Pendant, l'exécution d'une méthode, l'invariant de classe, peut être temporairement violé.

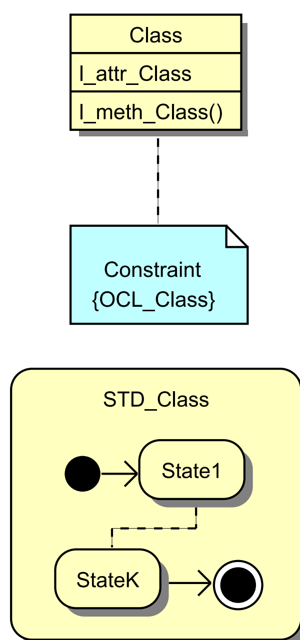


FIGURE I – Paramètres avant refactoring

1.2 Vérification des conditions d'application

Un schéma de refactoring s'applique sur les paramètres que nous venons d'identifier. Il convient de s'assurer que ces paramètres vérifient les deux conditions suivantes :

1. Cohérence entre les différents paramètres du schéma.
   Elle concerne respectivement :

   `Class` et `OCL_Class`. en vérifiant l'adéquation des contraintes OCL avec les propriétés de la classe.

   `Class` et `STD_Class`. par la possibilité d'exécuter les méthodes invoquées en séquence selon `STD_Class`.

2. Consistance de la notion introduite par le schéma (héritage, association, délégation,...) :
   Cette condition sera définie pour chaque notion, lors de la présentation du schéma concerné.

Les conditions que nous venons d'énumérer peuvent être classées en deux types :
- ceux concernant les propriétés statiques,
- ceux concernant les propriétés dynamiques.

Pour vérifier les conditions concernant les propriétés statiques ou de sûreté, nous transformons les classes, leurs propriétés et leurs contraintes OCL en des spécifications **B** en utilisant les règles de transformation systématique :
- d'UML vers **B** proposées par Meyer [74] et Ledang [56],
- d'OCL vers **B** proposées par Marcano [60, 61] et Ledang [58].

L'outil de vérification est l'***Atelier B*** [26].

Pour vérifier les conditions concernant les propriétés dynamiques ou de vivacité, nous transformons les diagrammes d'états-transitions en processus CSP en utilisant la fonction $\varphi_{UML \to CSP}$ proposée par Rasch et *al.* [84] (voir chapitre 2). L'outil de vérification est FDR2 [38].

Notons que, le recours à la traduction vers B et CSP est motivé par la non disponibilité d'outils permettant d'effectuer de telles vérification directement sur des spécifications UML. A notre connaissance, la plupart des outils de vérification de contraintes OCL n'ont pas atteint une maturité suffisante pour une utilisation fiable [98]. Cependant, de nombreux travaux de traduction de spécifications UML vers des langages formels ont été proposés, permettant d'utiliser les outils associés (voir chapitre 2).

Le processus de vérification de la cohérence des paramètres est le suivant :

– traduire `Class` en une machine B, appelée B_Class,
– traduire `OCL_Class` en des expressions invariantes dans la machine correspondante,
– ajouter des annotations de contrôle, proposées par Ifill et *al.* [51], dans la clause `ASSERTIONS` de la machine B. Pour chaque méthode $meth_i_Class$, une assertion contient deux obligations de preuve de la forme :

$$I_Class \wedge P_meth_i_Class \Rightarrow [Q_meth_i_Class](P_meth_j_Class)$$
$$I_Class \wedge P_meth_i_Class \Rightarrow [Q_meth_i_Class](P_meth_k_Class)\ ^a$$

a. inspiré des travaux de Ifill et *al.* où les méthodes $meth_j_Class$ et $meth_K_Class$ peuvent être appelées après l'exécution de la méthode $meth_i_Class$

La cohérence de la machine **B** obtenue prouve la cohérence entre $Class$ et ses contraintes OCL_Class.
La validation des assertions prouve la cohérence entre $Class$ et son diagramme d'états-transitions STD_Class.
Le processus de vérification de la consistance de la notion introduite par le schéma sera décrit dans chaque schéma.

1.3 Evolution de la spécification

Elle présente les différentes mises à jours réalisées automatiquement sur l'état de la spécification UML.

1.4 Correction du schéma

Elle concerne :

1. La préservation des propriétés des classes restructurées.
2. La préservation des comportements des classes restructurées.

Les différentes vérifications seront effectuées comme mentionnée dans la section 1.2

2 Plan

Cette partie comporte deux chapitres :
– Le chapitre 5 décrit le schéma de refactoring : introduction de la notion d'héritage, ainsi que trois schémas de refactoring sous-jacents à la notion d'héritage. Ils sont : introduction de la notion de redéfinition, introduction de la notion de classe abstraite et introduction de la notion de polymorphisme.
– Le chapitre 6 décrit les schémas de refactoring : introduction de la notion d'association, introduction de la notion de délégation et introduction de la notion de généricité.

Chapitre 5

Schéma de refactoring : introduction de la notion d'héritage

Dans l'ingénierie dirigée par les modèles, la relation d'héritage permet la réutilisabilité et l'extensibilité des classes. Le principe de cette relation est basé sur des classes, appelées *sous-classes* et classe-mère, appelée *super-classes*.

Les *sous-classes* possèdent les mêmes caractéristiques que leur *super-classes* et peuvent apporter des caractéristiques supplémentaires. UML supporte l'héritage simple et multiple. Dans la suite, nous nous limitons à l'héritage simple.

5.1 Etape 1. Identification des paramètres

Le schéma de refactoring d'introduction de la relation d'héritage est paramétré par :

$$< Class1, OCL_Class1, STD_Class1 >$$
$$< Class2, OCL_Class2, STD_Class2 >$$

Le schéma de refactoring proposé permet d'ajouter une relation d'héritage entre `Class1` et `Class2`. Pour la suite, nous désignons par :

- `Class1` la super-classe,
- `Class2` la sous-classe.

Aussi, nous supposons que les propriétés statiques et dynamiques communes entres les deux classes ont les mêmes noms.

5.2 Etape 2. Vérification des conditions d'application

5.2.1 Définition de la consistance de la relation d'héritage

L'objectif est de vérifier la consistance de la relation d'héritage à ajouter relativement à l'état de la spécification, entre :

- les classes : `Class1` et `Class2` (Définition 1),
- les contraintes OCL : `OCL_Class1` et `OCL_Class2` (Définition 2),
- les diagrammes d'états-transitions : `STD_Class1` et `STD_Class2` (Définition 3).

Définition 1

Les propriétés statiques `l_attr_Class1` et dynamiques `l_meth_Class1` de `Class1` sont aussi des propriétés de `Class2` :

I_attr_Class1 ⊆ I_attr_Class2 ∧
I_meth_Class1 ⊆ I_meth_Class2

Définition 2

1. L'invariant de `Class1` est renforcé dans `Class2`.
2. Pour chaque méthode de `Class1`, sa pré-condition est affaiblie par rapport à la pré-condition de son homologue de `Class2` ; sa post-condition est renforcée. Ces deux contraintes sont formalisées en **B** comme suit :

I_Class2 ⇒ I_Class1 ∧
(∀ m_Class1 ∈ I_meth_Class1 . ∃ m_Class2 ∈ I_meth_Class2 |
P_m_Class1 ⇒ P_m_Class2 ∧
Q_m_Class2 ⇒ Q_m_Class1)

Définition 3

Le diagramme d'états-transitions `STD_Class2` a un comportement compatible avec `STD_Class1`. Deux types de comportement sont identifiés [32] :

observé : toutes les séquences d'appels de méthodes qui peuvent être observées,

appelé : toutes les séquences d'appels de méthodes qui peuvent être appelées.

1. Un objet de la sous-classe doit également se comporter comme s'il s'agissait d'un objet de sa super-classe : Chaque séquence d'appel observable à l'égard d'une sous-classe `Class2` doit être considérée comme une séquence d'appel observable de sa super-classe `Class1`, en tenant compte que des méthodes connues par `Class1`.
2. Chaque objet instance d'une sous-classe `Class2` peut être considéré comme instance de la super-classe `Class1` : Chaque séquence appelée dans une classe donnée doit également être appelée dans l'ensemble de ses sous-classes.

5.2.2 Vérification

Elle est établie en deux temps.

Avec la méthode B (pour les vérifications de Définition 1 et Définition 2)

Cette vérification nécessite de :
- traduire la super-classe `Class1` en un modèle **B** abstrait,
- traduire la sous-classe `Class2` en un modèle **B** concret,
- traduire les contraintes OCL en des expressions invariantes dans la machine correspondante.

La vérification de la relation de raffinement entre le modèle concret et son homologue abstrait, nous permet de montrer que :
- l'ensemble des propriétés statiques et dynamiques de `Class1`, modèle abstrait, est inclus dans l'ensemble des propriétés statiques et dynamiques de `Class2`, modèle concret :

  I_attr_Class1 ⊆ I_attr_Class2 ∧
  I_meth_Class1 ⊆ I_meth_Class2

- l'invariant de `Class2` implique l'invariant de `Class1` :

  I_Class2 ⇒ I_Class1

- pour chaque méthode de `Class1`, sa pré-condition implique la pré-condition de la méthode correspondante de `Class2`,

- pour chaque méthode de `Class2`, sa post-condition implique la post-condition de la méthode correspondante de `Class1` :

$$\forall \text{ m_Class1} \in \text{l_meth_Class1} \;.\; \exists \text{ m_Class2} \in \text{l_meth_Class2} \mid$$
$$\text{P_m_Class1} \Rightarrow \text{P_m_Class2} \wedge$$
$$\text{Q_m_Class2} \Rightarrow \text{Q_m_Class1}$$

Avec le langage CSP (pour la vérification de Définition 3)

Cette vérification nécessite de :
- traduire `STD_Class1` et `STD_Class2` en deux processus CSP, `PSM_Class1` et `PSM_Class2`,
- extraire les canaux de communication, $channel_Class1$ et $channel_Class2$, à partir des ensembles de transitions de `PSM_Class1` et `PSM_Class2`.

Les règles de vérification dépendent des type de comportements identifiés, observés et appelés.

Pour un comportement observé. Les deux assertions suivantes modélisent les conditions de compatibilité des diagrammes d'états-transitions, `PSM_Class2` et `PSM_Class1` :

$$assert\ PSM_Class2\{|channel_Class2 \backslash channel_Class1|\} \sqsubseteq_{\tau} PSM_Class1$$
$$assert\ PSM_Class1 \sqsubseteq_{\tau} PSM_Class2\{|channel_Class2 \backslash channel_Class1|\}$$

Pour un comportement appelé. L'assertion suivante modélise la condition de compatibilité des diagrammes d'états-transitions, `PSM_Class2` et `PSM_Class1` :

$$assert\ PSM_Class2 \sqsubseteq_{\tau} PSM_Class1$$

Avec $\sqsubseteq_{\tau}$ dénote le raffinement CSP basé sur le modèle des traces (voir chapitre 2).

5.3 Etape 3. Evolution de la spécification

L'évolution de la spécification se décompose comme suit :

Introduction d'une relation d'héritage entre `Class1` et `Class2`, `Class1` étant la super-classe.

Mise à jour de `Class2`. Les aspects statiques et dynamiques communs aux deux classes sont supprimés de `Class2`. Ces propriétés sont obtenues par l'intersection des ensembles l_attr_Class1 et l_attr_Class2, pour les aspects statiques et par l'intersection des ensembles l_meth_Class1 et l_meth_Class2, pour les aspects dynamiques. `Class2` retrouve l'ensemble des propriétés supprimées dans sa super-classe `Class1`.

Les autres diagrammes restent inchangés. La FIGURE 5.1 présente l'état de la spécification après refactoring.

5.4 Etape 4. Correction du schéma

Les modifications apportées par l'application du schéma de refactoring préservent l'ensemble des propriétés des deux classes dans leur contexte. La suppression des aspects statiques et dynamiques de `Class2` est sans effet pour la spécification et en particulier pour `Class2`. En effet, `Class2` retrouve l'ensemble des propriétés supprimées dans sa super-classe `Class1`. Ainsi, les contraintes `OCL_Class2` liées aux méthodes restantes et à l'invariant `I_Class2` et le diagramme d'états-transitions `STD_Class2` restent inchangés. En effet, l'invariant de classe `I_Class2` englobe l'invariant `I_Class1` venant de la classe ascendante `Class1`. Ceci favorise un héritage implicite de contraintes OCL [43]. Enfin, les contraintes `OCL_Class1` et le diagramme d'états-transitions `STD_Class1` restent inchangés, puisque `Class1` n'a pas été modifiée par l'application du schéma. En outre, ce schéma de refactoring n'autorise pas la redéfinition des méthodes (voir section 5.7.1). Ainsi, il n'a y aucune vérification à faire.

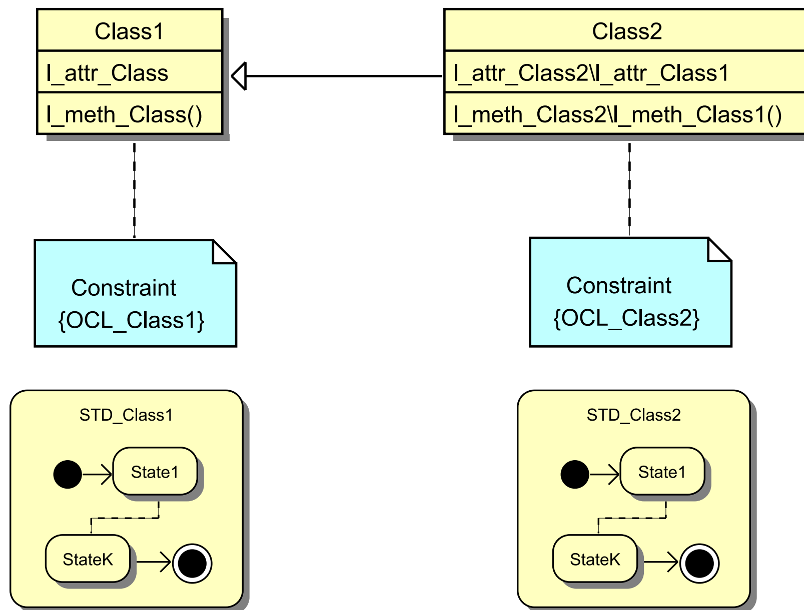


FIGURE 5.1 – Paramètres après refactoring

5.5 Etude d'un exemple

Nous considérons une modélisation OO des processus au sens de la programmation concurrente inspirée de [32]. La classe `Process` modélise un processus au sens général. La classe `PrioProcess` modélise un processus doté en plus d'une priorité qui change dynamiquement. Ainsi, une relation d'héritage peut être introduite entre ces deux classes avec `Process` comme super-classe et `PrioProcess` comme sous-classe.

5.5.1 Etape 1. Identification des paramètres

Les paramètres du schéma sont :

$$< Process, OCL_Process, STD_Process >$$
$$< PrioProcess, OCL_PrioProcess, STD_PrioProcess >$$

Classes : Les propriétés statiques et dynamiques des classes *Process* et *PrioProcess* sont présentées dans la FIGURE 5.2.

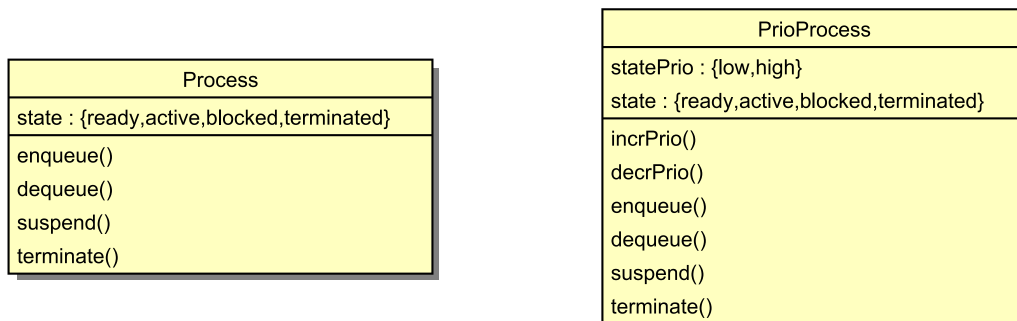


FIGURE 5.2 – Diagramme de classes avant refactoring

Contraintes OCL : Les contraintes OCL correspondant à chaque classe sont les suivants :

```
Context Process
inv state_valide : self.state = #ready
               xor self.state = #active
               xor self.state = #blocked
               xor self.state = #terminate

Context Process::state
init : ready

Context Process::enqueue()
pre : state = active or state = blocked
post : state = ready

Context Process::dequeue()
pre : state = ready
post : state = active

Context Process::suspend()
pre : state = active
post : state = blocked

Context Process::terminate()
pre : state = active
post : state = terminated
```

```
Context PrioProcess
 inv state_valide : self.state = #ready
                xor self.state = #active
                xor self.state = #blocked
                xor self.state = #terminate
inv statePrio_valide : self.statePrio = #low
                   xor self.statePrio = #high

Context PrioProcess::state
init : ready

Context PrioProcess::statePrio
init : low

Context PrioProcess::enqueue()
pre : state = active or state = blocked
post : state = ready

Context PrioProcess::dequeue()
pre : state = ready
post : state = active

Context PrioProcess::suspend()
pre : state = active
post : state = blocked

Context PrioProcess::terminate()
pre : state = active
post : state = terminated

Context PrioProcess::decrPrio()
pre : statePrio = high
post : statePrio = low

Context PrioProcess::incrPrio()
pre : statePrio = low
post : statePrio = high
```

Diagrammes d'états-transitions : Les comportements correspondant aux classes *Process* et *PrioProcess* sont présentés respectivement par les diagrammes d'états-transitions $STD_{Process}$ et $STD_{PrioProcess}$ (voir Figure 5.3 et Figure 5.4).

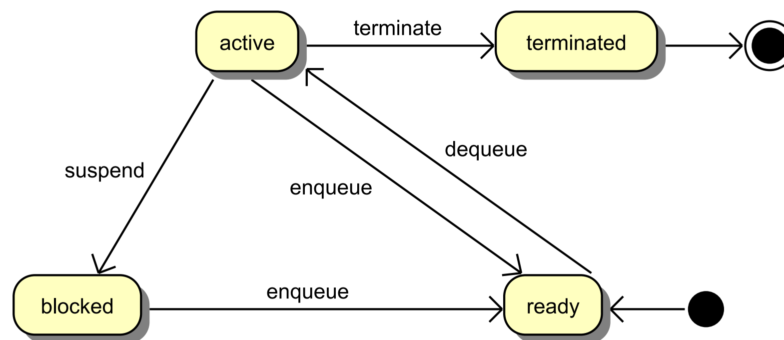


Figure 5.3 – Diagramme d'états-transitions de la classe Process

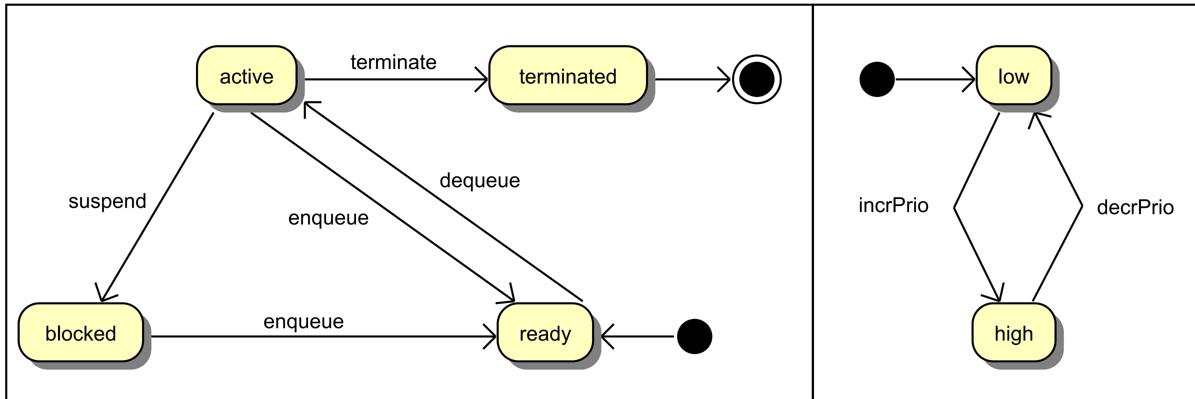


FIGURE 5.4 – Diagrammes d'états-transitions de la classe PrioProcess

5.5.2 Etape 2. Vérification des conditions d'application

Vérification avec la méthode B

Les spécifications **B** obtenues par transformation des spécifications UML sont présentées dans la FIGURE 5.5, avec la machine abstraite `Process` et son raffinement `PrioProcess`.

Les obligations de preuve données dans le tableau 5.1 ont été déchargées automatiquement.

	nPO	nPRi	nPRa	nUn	%Pr
AssertionLemmas	1	0	1	0	100
Initialisation	1	0	1	0	100
enqueue	0	0	0	0	100
dequeue	0	0	0	0	100
terminate	0	0	0	0	100
suspend	0	0	0	0	100
Process	2	0	2	0	100

TABLE 5.1 – Tableau de l'état de la machine `Process`

De même, les obligations de preuve données dans le tableau 5.2 ont été déchargées automatiquement.

	nPO	nPRi	nPRa	nUn	%Pr
AssertionLemmas	0	0	0	0	100
Initialisation	1	0	1	0	100
enqueue	1	0	1	0	100
dequeue	0	0	0	0	100
terminate	0	0	0	0	100
suspend	0	0	0	0	100
incrPrio	0	0	0	0	100
decrPrio	0	0	0	0	100
PrioProcess	2	0	2	0	100

TABLE 5.2 – Tableau de l'état de la machine `PrioProcess`

La vérification de la consistance des spécifications **B** et du raffinement prouve la cohérence des paramètres du schéma de restructuration et la consistance de la relation d'héritage à ajouter entre les classes et leurs contraintes OCL.

```
MODEL
  Types
SETS
    OBJECTS
END
```

```
MODEL
  Process
SEES
    Types
CONSTANTS
    PROCESS
PROPERTIES
    PROCESS ⊆ OBJECTS
SETS
    channel = {active, terminated, blocked, ready}
VARIABLES
    process, state
INVARIANT
    process ⊆ PROCESS ∧
    state ∈ channel
INITIALISATION
    process := ∅ ||
    state := ready
ASSERTIONS
    (state=ready ⇒ state=ready) ∧
    ((state=active ⇒ state=active) ∧
    (state=active ⇒ state=active)) ∧
    (state=blocked ⇒ (state=active ∨ state=blocked))
OPERATIONS
    enqueue = SELECT state = active ∨ state= blocked THEN state:=ready END;
    dequeue = SELECT state =ready THEN state:=active END;
    terminate = SELECT state =active THEN state:=terminated END;
    suspend = SELECT state =active THEN state:=blocked END
END
```

```
REFINEMENT
  PrioProcess
REFINES
    Process
SEES
  Types
SETS
    channelPrio = {low,high}
VARIABLES
    prioProcess, process, state, statePrio
INVARIANT
    process = process ∧
    prioProcess ⊆ process ∧
    state = state ∧
    statePrio ∈ channelPrio
INITIALISATION
    prioProcess := ∅ ||
    process := ∅ ||
    state := ready ||
    statePrio := low
ASSERTIONS
    (statePrio=high ⇒ statePrio=high) ∧
    (statePrio=low ⇒ statePrio=low)
OPERATIONS
    enqueue = SELECT state=active ∨ state=blocked THEN state:=ready END;
    dequeue = SELECT state=ready THEN state:=active END;
    terminate = SELECT state=active THEN state:=terminated END;
    suspend = SELECT state=active THEN state:=blocked END;
    incrPrio = SELECT statePrio=low THEN statePrio:=high END;
    decrPrio = SELECT statePrio=high THEN statePrio:=low END
END
```

FIGURE 5.5 – Modèles B

Vérification avec le langage CSP

Les processus, `PSM_Process` et `PSM_PrioProcess` présentent les traductions CSP respectives des diagrammes d'états-transitions, `STD_Process` et `STD_PrioProcess`.

$$
\begin{array}{l}
PSMA = \sqcap_{i\in\{ready\}} P_i = Ready \\
Ready = \Box_{(e,t)\in\{(dequeue,active)\}} e \rightarrow P_t = dequeue \rightarrow Active \\
Active = \Box_{(e,t)\in\{(terminate,terminated),(suspend,blocked),(enqueue,ready)\}} e \rightarrow P_t = \\
\qquad terminate \rightarrow Terminated \Box suspend \rightarrow Blocked \Box enqueue \rightarrow Ready \\
Blocked = \Box_{(e,t)\in\{(enqueue,ready)\}} e \rightarrow P_t = enqueue \rightarrow Ready \\
Terminated = \sqcap_{i\in\{STOP\}} P_i = STOP
\end{array}
$$

Avec $PSM_Process = PSMA$ nous avons maintenant une traduction CSP de la machine d'état de la classe $Process$. Après simplification, $PSM_Process$ ressemble à ceci :

$$
\begin{array}{l}
PSM_Process = dequeue \rightarrow (suspend \rightarrow enqueue \rightarrow PSM_Process \\
\Box enqueue \rightarrow PSM_Process \Box terminate \rightarrow STOP)
\end{array}
$$

De même, nous obtenons le processus CSP de la machine d'état de la classe $PrioProcess$:

$$
\begin{array}{l}
PSM_PrioProcess = PSMB1 ||| PSMB2 \\
PSMB1 = PSM_Process \\
PSMB2 = incrPrio \rightarrow decrPrio \rightarrow PSMB2
\end{array}
$$

Les diagrammes d'états-transitions ci-dessus définissent des comportements observés. Ainsi, nous devons montrer que :

$$
\begin{array}{l}
PSM_PrioProcess\{|channelPrioProcess \backslash channelProcess|\} \sqsubseteq_{\tau} PSM_Process \\
PSM_Process \sqsubseteq_{\tau} PSM_PrioProcess\{|channelPrioProcess \backslash channelProcess|\}
\end{array}
$$

Ces deux assertions sont facilement prouvées à l'aide de l'outil ***FDR2*** comme présenté dans la FIGURE 5.6 [4]. Ainsi, nous prouvons la consistance de la relation d'héritage à ajouter entre `Process` et `PrioProcess`.

5.5.3 Etape 3. Evolution de la spécification

L'application du schéma de refactoring génère le diagramme de classes décrit dans la FIGURE 5.7, les autres diagrammes restent inchangés.

5.5.4 Etape 4. Correction du schéma

Comme l'indique la section 5.4, il n'a y aucune vérification à faire.

5.6 Etude de cas

Nous illustrons le processus de refactoring sur l'étude de cas de contrôle d'accès à un bâtiment. A titre d'exemple, nous partons du diagramme de classes présenté par la FIGURE 2.9.

5.6.1 Etape 1. Identification des paramètres

Les classes `Light` et `GreenLight` partagent l'attribut *state* et les méthodes on et off. Ainsi, une relation d'héritage entre ces deux classes peut être ajoutée au diagramme de classes, avec `Light` comme super-classe et `GreenLight` comme sous-classe. Les paramètres du schéma sont :

$$
\begin{array}{l}
< Light, OCL_Light, STD_Light > \\
< GreenLight, OCL_GreenLight, STD_GreenLight >
\end{array}
$$

4. Avec, $PSMC = PSM_PrioProcess\{|channelPrioProcess \setminus channelProcess|\}$

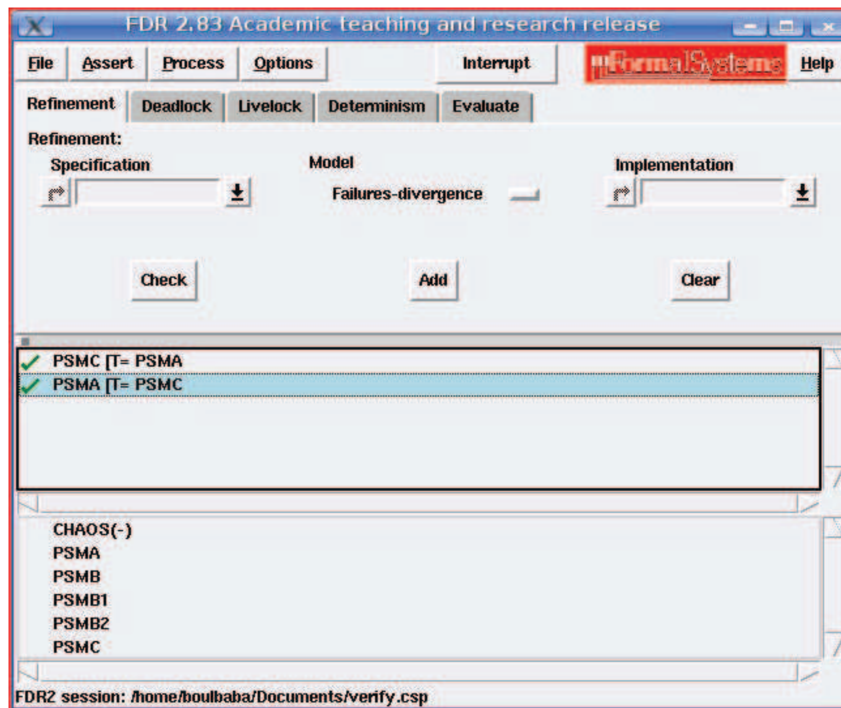


FIGURE 5.6 – Vérification avec FDR2

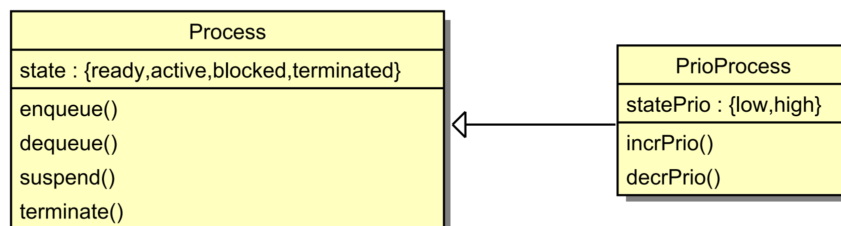


FIGURE 5.7 – Diagramme de classes après refactoring

5.6.2 Etape 2. Vérification des conditions d'application

Vérification avec la méthode B

Les spécifications **B** obtenues par transformation des spécifications UML sont présentées dans la FIGURE 5.8, avec la machine abstraite `Light` et son raffinement `GreenLight`.

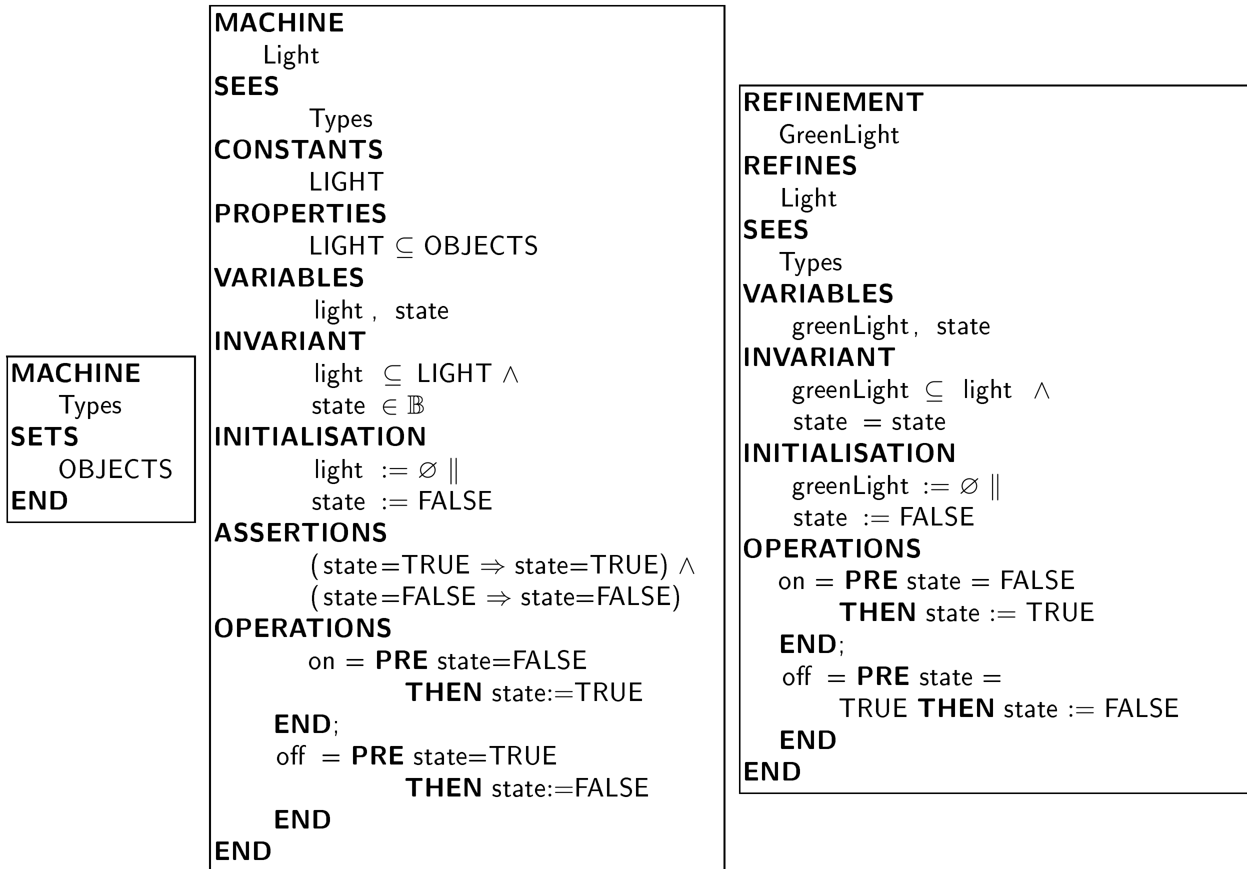


FIGURE 5.8 – Modèles B

Les obligations de preuve données dans le tableau 5.3 ont été déchargées automatiquement.

	nPO	nPRi	nPRa	nUn	%Pr
AssertionLemmas	0	0	0	0	100
Initialisation	1	0	1	0	100
on	0	0	0	0	100
off	0	0	0	0	100
Light	1	0	1	0	100

TABLE 5.3 – Tableau de l'état de la machine `Light`

De même, les obligations de preuve données dans le tableau 5.4 ont été déchargées automatiquement.

La vérification de la consistance des spécifications **B** et du raffinement prouve la cohérence des paramètres du schéma de restructuration et la consistance de la relation d'héritage à ajouter entre les classes et leurs contraintes OCL.

	nPO	nPRi	nPRa	nUn	%Pr
Initialisation	1	0	1	0	100
on	0	0	0	0	100
off	0	0	0	0	100
GreenLight	1	0	1	0	100

TABLE 5.4 – Tableau de l'état de la machine GreenLight

Vérification avec le langage CSP

Les processus, PSM_Light et PSM_GreenLight présentent les traductions CSP respectives des diagrammes d'états-transitions, STD_Light et STD_GreenLight.

$$PSM_Light = on \rightarrow off \rightarrow PSM_Light$$
$$PSM_GreenLight = on \rightarrow off \rightarrow PSM_GreenLight$$

A partir de ces deux processus CSP, nous devons prouver l'assertion suivante :

$$assert\ PSM_GreenLight \sqsubseteq_{\mathcal{T}} PSM_Light$$

Ceci est facilement prouvé à l'aide de l'outil ***FDR2***. Ainsi, nous prouvons la consistance de la relation d'héritage à ajouter entre Light et GreenLight.

De même, nous ajoutons une relation d'héritage entre Light et RedLight, avec Light comme classe ascendente et RedLight comme classe descendante.

5.6.3 Etape 3. Evolution de la spécification

L'application du processus de refactoring sur le diagramme de classes présenté par la FIGURE 2.9 génère le diagramme de classes donné dans la FIGURE 5.9, les autres diagrammes restent inchangés.

5.6.4 Etape 4. Correction du schéma

Comme l'indique la section 5.4, il n'a y aucune vérification à effectuer.

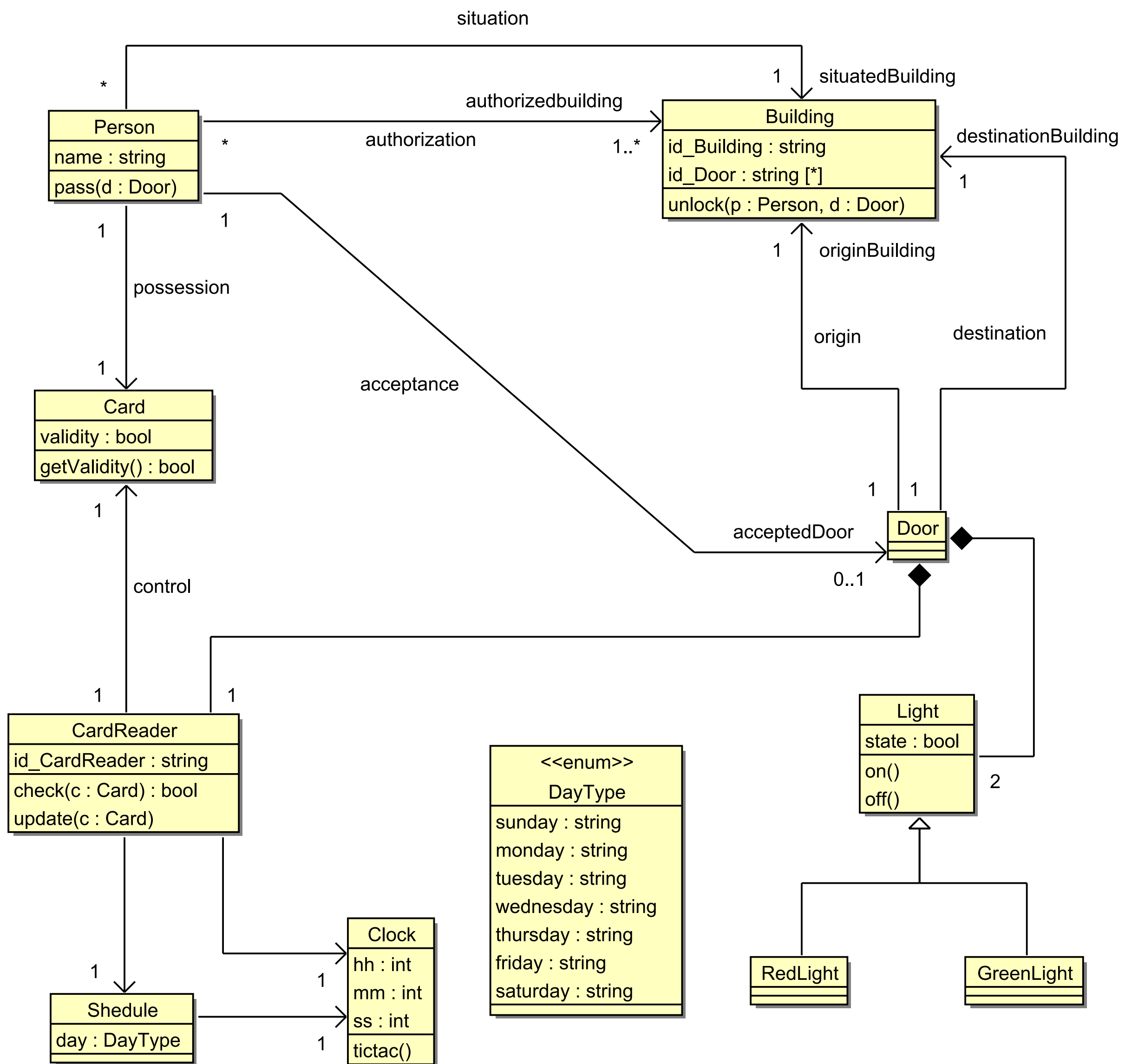


FIGURE 5.9 – Diagramme de classes après refactoring

5.7 Autres schémas de refactoring

Dans la suite, nous présentons des schémas de refactoring sous-jacents à la notion d'héritage. Ils sont : introduction de la notion de redéfinition, introduction de la notion de classe abstraite et introduction de la notion de polymorphisme.

5.7.1 Schéma de refactoring : introduction de la notion de redéfinition

Une redéfinition devrait changer l'implantation d'une opération UML –souvent pour des raisons d'efficacité : tenir compte du nouveau contexte– mais pas sa sémantique exprimée par des contraintes OCL.

Etape 1. Identification des paramètres

Le schéma de refactoring d'introduction de la notion de redéfinition est paramétré par (voir Figure 5.10) :

$$< SuperClass, OCL_SuperClass, STD_SuperClass >$$
$$< SubClass, OCL_SubClass, STD_SubClass >$$

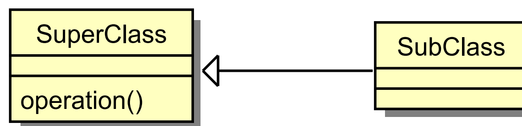


Figure 5.10 – Paramètres avant refactoring

Le schéma de refactoring proposé permet de redéfinir une opération appelée `operation` de `SuperClass` dans `SubClass`.

Etape 2. Vérification des conditions d'application

A ce stade, nous vérifions que la cohérence des paramètres du schéma comme l'indique la section 1.2. Il n'y a pas de conditions particulières pour l'application de ce schéma de refactoring.

Etape 3. Evolution de la spécification

L'évolution de la spécification consiste à redéfinir `operation` dans `SubClass`.

Les autres diagrammes restent inchangés. La Figure 5.11 présente l'état de la spécification après restructuration.

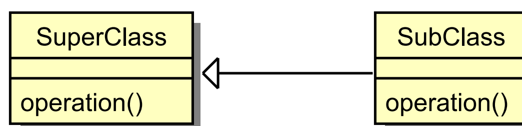


Figure 5.11 – Paramètres après refactoring

Etape 4. Correction du schéma

Pour la méthode `operation` redéfinie dans `SubClass`, sa pré-condition peut être affaiblie par rapport à la pré-condition de son homologue de `SuperClass` ; sa post-condition peut être renforcée. Ces deux contraintes sont formalisées en **B** comme suit :

P_operation_SuperClass ⇒ P_operation_SubClass ∧
Q_operation_SubClass ⇒ Q_operation_SuperClass)

Ainsi, la vérification de la correction du schéma est similaire à la vérification de la **définition 2** de la définition de la consistance de la relation d'héritage.

5.7.2 Schéma de refactoring : introduction de la notion de classe abstraite

Les classes abstraites constituent un outil crucial dans l'utilisation des méthodes orientées objet lors des phases d'analyse et de conception. D'ailleurs, elles sont souvent utilisées par les patrons de conception de GoF [36]. Les contraintes OCL (pré, post, invariant) sont applicables aux opérations abstraites permettant aux classes abstraites d'être spécifiées précisément. Les classes abstraites fournissent le mécanisme d'abstraction souhaité permettant de décrire les noeuds intermédiaires d'une classification.

Etape 1. Identification des paramètres

Le schéma de refactoring d'introduction de la notion de classe abstraite est paramétré par :

$$< Class, OCL_Class, STD_Class >$$

Le schéma de refactoring proposé permet de créer une classe abstraite appelée `AbstractClass` et d'ajouter une relation d'héritage entre `AbstractClass` et `Class`.

Etape 2. Vérification des conditions d'application

A ce stade, nous vérifions que la cohérence des paramètres du schéma comme l'indique la section 1.2 (voir Introduction de la troisième partie). Il n'y a pas de conditions particulières pour l'application de ce schéma de refactoring.

Etape 3. Evolution de la spécification

L'évolution de la spécification se décompose comme suit :

Création de `AbstractClass`. Une classe abstraite est introduite dans la spécification initiale.

Introduction d'une relation d'héritage entre `AbstractClass` et `Class`, `AbstractClass` étant la super-classe.

Abstraction des méthodes de `Class`. Les opérations rendues abstraites dans `AbstractClass` restent inchangées dans `Class`.

La Figure 5.12 présente l'état de la spécification après refactoring.

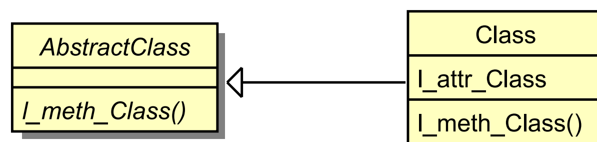


Figure 5.12 – Paramètres après refactoring

Etape 4. Correction du schéma

L'objectif est de vérifier la consistance de la relation d'héritage ajoutée entre `AbstractClass` et `Class`. Pour cela nous utilisons les **définitions 1**, **2** et **3** de la définition de la consistance de la relation d'héritage.

Ainsi, la vérification de la correction du schéma est similaire à la vérification de **définition 1, 2 et 3** de la définition de consistance de la relation d'héritage.

5.7.3 Schéma de refactoring : introduction de la notion de polymorphisme

Le polymorphisme est un outil puissant permettant l'obtention des architectures OO élégantes et extensibles. Il est largement utilisé dans les patterns de conception de Gof [36]. Le polymorphisme suppose l'héritage.

Etape 1. Identification des paramètres

Le schéma de refactoring d'introduction de la notion de polymorphisme est paramétré par (voir FIGURE 5.13) :

$$< Class1, OCL_Class1, STD_Class1 >$$
$$< Class2, OCL_Class2, STD_Class2 >$$
$$< Class3, OCL_Class3, STD_Class3 >$$

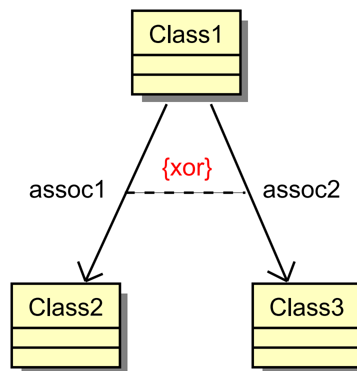


FIGURE 5.13 – Paramètres avant refactoring

Le schéma de refactoring proposé permet de créer une classe abstraite appelée `Class4`, d'ajouter deux relations d'héritage entre `Class4` et `Class2` et entre `Class4` et `Class3`, de supprimer les relations d'association entre `Class1` et `Class2` et entre `Class1` et `Class3` et de les remplacer par une seule association entre `Class1` et `Class4`.

Etape 2. Vérification des conditions d'application

Un objet de type `Class1` est attaché :
- soit à un objet de type `Class2`,
- soit à un objet de type `Class3`.

Ceci est formalisé en **B** par :

$$\mathrm{dom}(\mathrm{assoc1}) \cap \mathrm{dom}(\mathrm{assoc2}) = \varnothing$$

La vérification est effectuée avec la méthode **B** en ajoutant la formalisation en **B** de la définition donnée ci-dessus dans la clause **INVARIANT** de la machine correspondante.

Etape 3. Evolution de la spécification

L'évolution de la spécification se décompose comme suit :

Création de `Class4`. Une classe abstraite est introduite dans la spécification initiale.

Introduction de deux relations d'héritage entre `Class4` et `Class2` et `Class4` et `Class3`, `Class4` étant la super-classe.

Introduction d'une relation d'association entre `Class1` et `Class4`, avec un sens de navigation allant de `Class1` vers `Class4`.

Suppression des relations d'association entre `Class1` et `Class2` et entre `Class1` et `Class3`.

Ainsi, nous introduisons la notion de polymorphisme en proposant à `Class1` d'utiliser un même nom de méthode de `Class4` pour plusieurs types d'objets différents (`Class2` ou `Class3`).

La FIGURE 5.14 présente l'état de la spécification après refactoring.

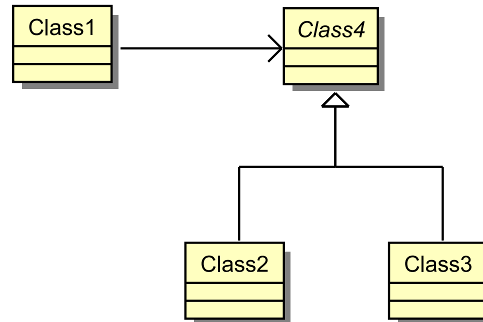


FIGURE 5.14 – Paramètres après refactoring

Etape 4. Correction du schéma

L'objectif est de vérifier la consistance des relations d'héritage ajoutées entre `Class4` comme super-classe et `Class2` et `Class3` comme sous-classes. Pour cela nous utilisons les **définitions 1**, **2** et **3** de la définition de la consistance de la relation d'héritage.

Ainsi, la vérification de la correction du schéma est similaire à la vérification de **définition 1, 2 et 3** de la définition de consistance de la relation d'héritage.

5.8 Conclusion

Nous avons proposé un schéma de refactoring permettant l'introduction d'une relation d'héritage entre deux classes UML existantes. De plus, nous avons accordé un soin particulier à la vérification de la préservation des propriétés de sûreté et vivacité après le refactoring en utilisant B et CSP. En outre, nous avons introduit des schémas de refactoring sous-jacents à la notion d'héritage : introduction de la notion de redéfinition, introduction de la notion de classe abstraite et introduction de la notion de polymorphisme. Enfin, nous avons illustré les schémas de refactoring proposés sur des exemples plus ou moins bien ciblés.

Dans le chapitre suivant, nous allons proposer des schémas de refactoring liés aux notions UML : association, délégation et généricité.

Chapitre 6

Schémas de refactoring : introduction des notions d'association, de délégation et de généricité

Dans ce chapitre, nous proposons trois schémas de refactoring. Le premier permet l'introduction d'une association entre deux classes existantes appartenant à un diagrammes de classes. Le second aide à lutter contre les classes monstres. Quant au troisième, il permet de paramétrer une classe existante.

6.1 Schéma de refactoring : introduction de la notion d'association

La relation d'association est une connexion sémantique entre deux classes. Elle peut être binaire ou n-aire, et peut être nommée. Une association est caractérisée par :

- *sa multiplicité* qui sert à préciser le nombre minimum et maximum d'instances de chaque classe dans la relation liant deux ou plusieurs classes,
- *sa navigabilité* qui indique comment accéder d'une classe à une autre. Si la relation est entre les classes `Class1` et `Class2` et que seulement `Class2` est navigable, alors on pourrait accéder à `Class2` à partir de `Class1` mais pas l'inverse, on dit que l'association est mono-directionnelle. Par défaut, la navigabilité est bi-directionnelle.

En UML, une classe peut utiliser une ou plusieurs propriétés statiques (attributs) et dynamiques (méthodes) d'une ou plusieurs autres classes. Pour cela, la classe doit posséder une relation d'association avec les classes consultées, avec un sens de navigation allant vers ces classes.

Dans cette section, nous nous limitons à l'introduction d'une association mono-directionnelle lors de l'appel d'une ou plusieurs méthodes.

6.1.1 Etape 1. Identification des paramètres

Le schéma de refactoring d'introduction de la notion d'association est paramétré par :

$$< Class1, OCL_Class1, STD_Class1 >$$
$$< Class2, OCL_Class2, STD_Class2 >$$

Hypothèse : Les associations entre les classes indiquent des invocations de méthodes, c'est-à-dire si la classe $Class1$ a une association avec la classe $Class3$, alors la classe $Class1$ appelle une méthode de la classe $Class3$ (voir FIGURE 6.1).

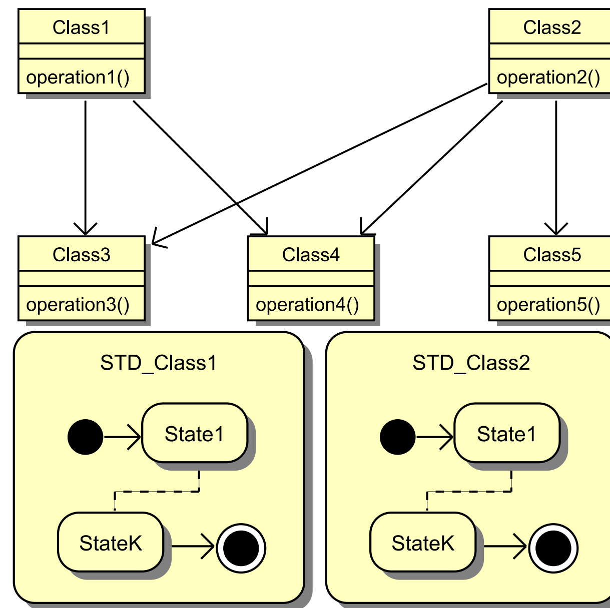


FIGURE 6.1 – Paramètres avant refactoring

6.1.2 Etape 2. Vérification des conditions d'application

Nous supposons que le comportement de le classe `Class2` est en partie réalisé par la classe `Class1`. Ainsi, nous supposons que la classe `Class2` délègue son travail aux classes reliées de la façon suivante : `Class3` exécute `operation3`, puis `Class4` exécute `operation4` enfin `Class5` exécute `operation5`. De la même manière, `Class1` délègue son travail aux classes reliées de la façon suivante : `Class3` exécute `operation3`, puis `Class4` exécute `operation4`.

Le comportement d'une classe est modélisé par un diagramme d'états-transitions et la détection des appels des méthodes est vérifiée à l'aide des processus CSP.

Les comportements des classes `Class1` et `Class2` sont respectivement définis par les processus `PSM_Class1` et `PSM_Class2`.

$$PSM_Class1 = Class3!operation3 \rightarrow Class3_r?x \rightarrow Class4!operation4 \rightarrow Class4_r?x$$
$$\rightarrow STOP$$
$$PSM_Class2 = Class3!operation3 \rightarrow Class3_r?x \rightarrow Class4!operation4 \rightarrow Class4_r?x$$
$$\rightarrow Class5!operation5 \rightarrow Class5_r?x \rightarrow SKIP$$

Ainsi, pour vérifier l'existence d'un appel de méthode, nous devons trouver une substitution textuelle des processus CSP que nous venons de décrire.

6.1.3 Etape 3. Evolution de la spécification

L'évolution de la spécification se décompose comme suit :

Introduction d'une relation d'association entre `Class1` et `Class2`, avec un sens de navigation allant de `Class2` vers `Class1`.

Suppression des associations entre `Class2` et `Class3` et entre `Class2` et `Class4`.

La FIGURE 6.2 présente l'état de la spécification après refactoring.

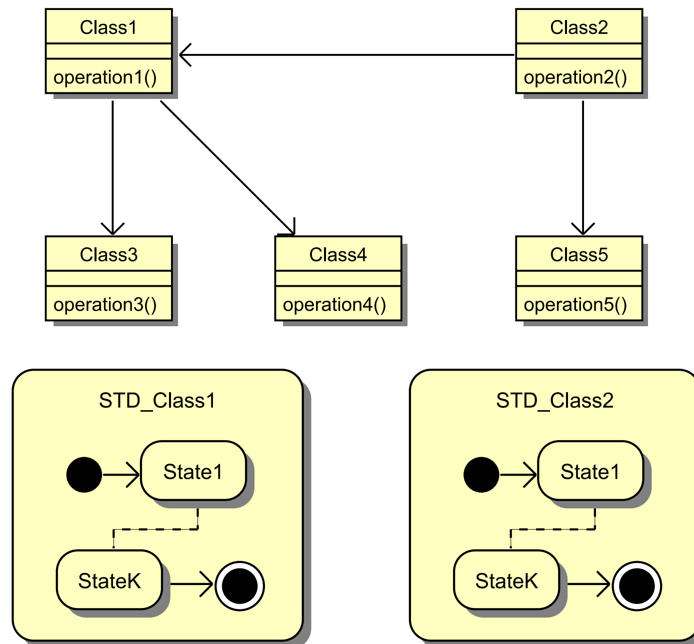


FIGURE 6.2 – Paramètres après refactoring

6.1.4 Etape 4. Correction du schéma

Les modifications, apportées par l'application du schéma de refactoring proposé, préservent l'ensembles des propriétés des deux classes, du fait que nous n'avons pas modifié les aspects statiques et dynamiques des classes `Class1` et `Class2`.

Les modifications apportées aux diagrammes d'états-transitions `STD_Class1` et `STD_Class2` nécessitent une vérification de la préservation des comportements de ces deux diagrammes par rapport à leurs correspondants du niveau abstrait.

Ainsi, il suffit de montrer que :

$$assert\ PSM_Class1_r \sqsubseteq_{\tau} PSM_Class1$$
$$assert\ PSM_Class2_r \sqsubseteq_{\tau} PSM_Class2$$

Avec $\sqsubseteq_{\tau}$ dénote le raffinement CSP basé sur le modèle des traces (voir chapitre 2).

6.1.5 Etude d'un exemple

A titre d'exemple, nous partons du diagramme de classes du système SAAT (Software Architecture Analysis Tool) [99]. SAAT est un outil utilisé pour calculer les paramètres sur des modèles UML. Ces paramètres peuvent ensuite être utilisés pour analyser le modèle de défauts potentiels ou anti-patterns.

Classes : Le diagramme de classes (voir FIGURE 6.3), correspondant au système SAAT, est constitué des classes : $Saat$, DB, $Stat$, $DBCreate$, $Parser$, $DBFill$, $DBCheck$, $Analyse$, $StatCalc$ et $StatFilter$. Les associations entre ces classes indiquent des invocations de méthodes, c'est-à-dire si une classe A a une association avec la classe B, alors la classe A appelle une méthode de la classe B.

La FIGURE 6.3 présente le diagramme de classes initial correspondant au système SAAT. La classe $Saat$ délègue son travail aux classes associées d'une façon séquentielle. Tout d'abord la base de données est créée ($DBCreate.create()$), un fichier d'entrée est analysé ($Parser.parse()$) et les données sont insérées dans la base de données ($DBFill.fill()$). Après l'insertion des données, la base de données remplie est vérifiée ($DBCheck.check()$), ensuite, les données seront analysées ($Analyse.analyse()$) et les statistiques

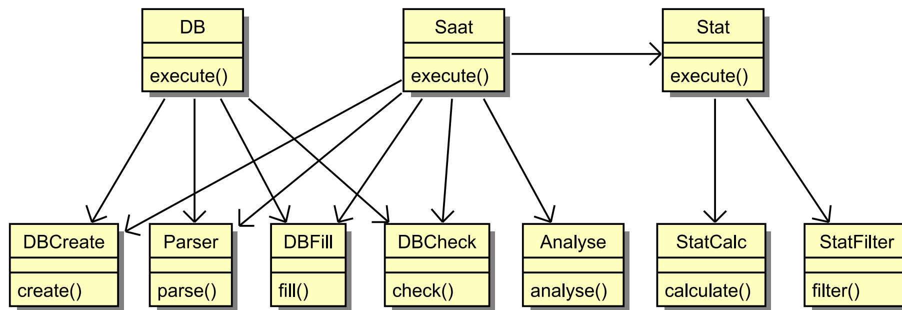


FIGURE 6.3 – Diagramme de classes avant refactoring

sont calculées ($StatCalc.calculate()$) puis filtrées ($StatFilter.filter()$) en fonction des critères définis par l'utilisateur.
Contraintes OCL : Il n'a y pas de contraintes OCL importantes attachées aux différentes classes.
Diagrammes d'états-transitions : Les comportements correspondant aux différentes classes sont présentés par les FIGURES 6.4, 6.5, 6.6 et 6.7.

Dans ce diagramme, nous avons une situation où on peut appliquer le schéma de refactoring : introduction de la relation d'association entre *Saat* et DB.

Etape 1. Identification des paramètres

Les paramètres du schéma sont :

$$< Saat, OCL_Saat, STD_Saat >$$
$$< DB, OCL_DB, STD_DB >$$

Etape 2. Vérification des conditions d'application

Les processus CSP suivants correspondent aux transformations systématiques des diagrammes d'états-transitions de toutes les classes du système SAAT.

```
PSM_DBCreate     = DBCreate?x → create() → DBCreate_r!create →
                   PSM_DBCreate
PSM_Parser       = Parser?x → parse() → Parser_r!parse → PSM_Parser
PSM_DBFill       = DBFill?x → fill() → DBFill_r!fill → PSM_DBFill
PSM_DBCheck      = DBCheck?x → check() → DBCheck_r!check →
                   PSM_DBCheck
PSM_Analyse      = Analyse?x → analyse() → Analyse_r!analyse →
                   PSM_Analyse
PSM_StatCalc     = StatCalc?x → calculate() → StatCalc_r!calculate →
                   PSM_StatCalc
PSM_StatFilter   = StatFilter?x → filter() → StatFilter_r!afilter →
                   PSM_StatFilter
PSM_Saat         = (DBCreate!create → DBCreate_r?x → Parser!parse
                   → Parser_r?x → DBFill!fill → DBFill_r?x →
                   DBCheck!check → DBCheck_r?x) → Analyse!analyse →
                   Analyse_r?x → Stat!execute → Stat_r?x → SKIP
PSM_DB           = (DBCreate!create → DBCreate_r?x → Parser!parse
                   → Parser_r?x → DBFill!fill → DBFill_r?x →
                   DBCheck!check → DBCheck_r?x) → STOP
PSM_Stat         = Stat?x → StatCalc!calculate → StatCalc_r?x →
                   StatFilter!filter → StatFilter_r?x → Stat
```

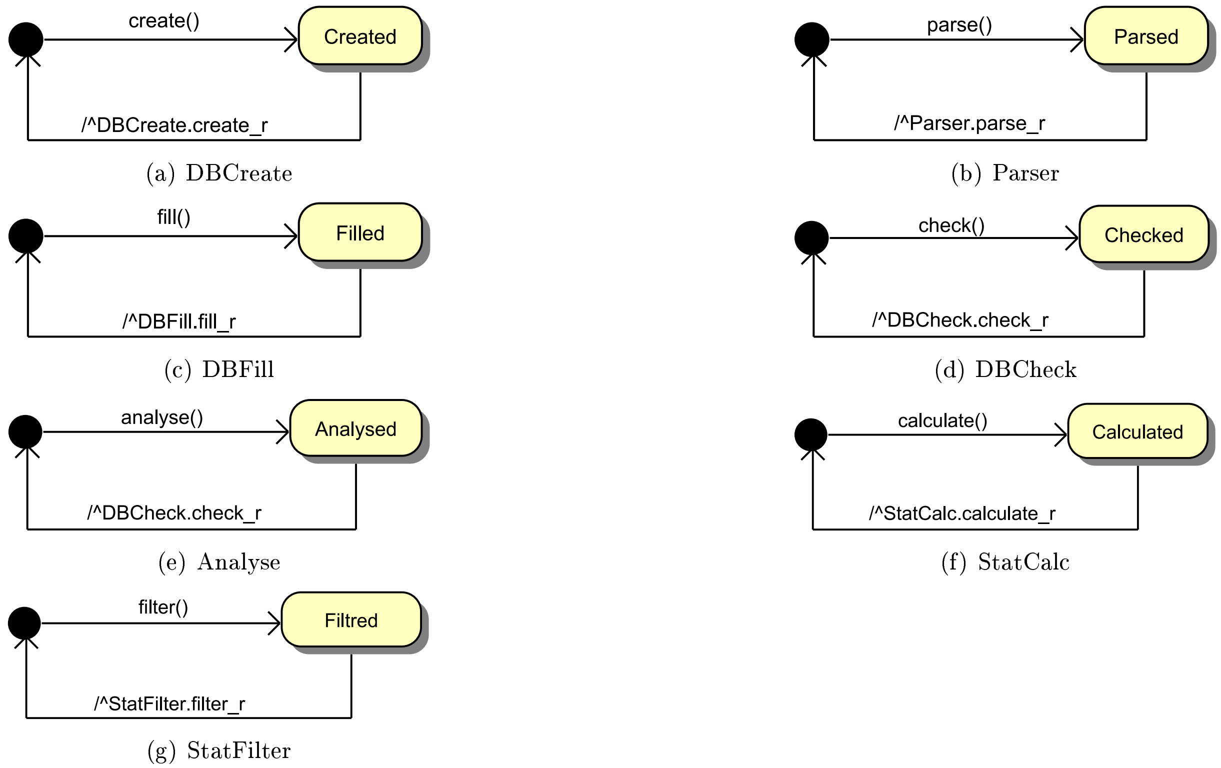


FIGURE 6.4 – Diagrammes d'états-transitions des classes : DBCreate, Parser, DBFill, DBCheck, Analyse, StatCalc et StatFilter

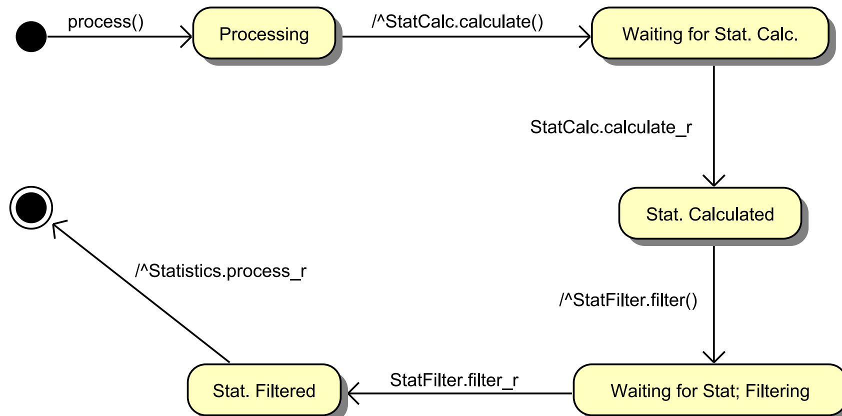


FIGURE 6.5 – Diagramme d'états-transitions de Statistics

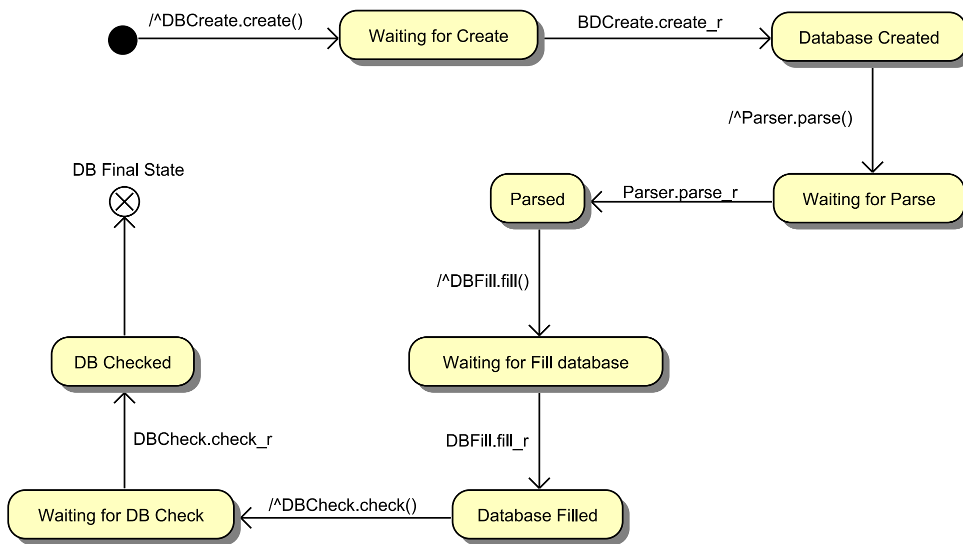


FIGURE 6.6 – Diagramme d'états-transitions de DB avant refactoring

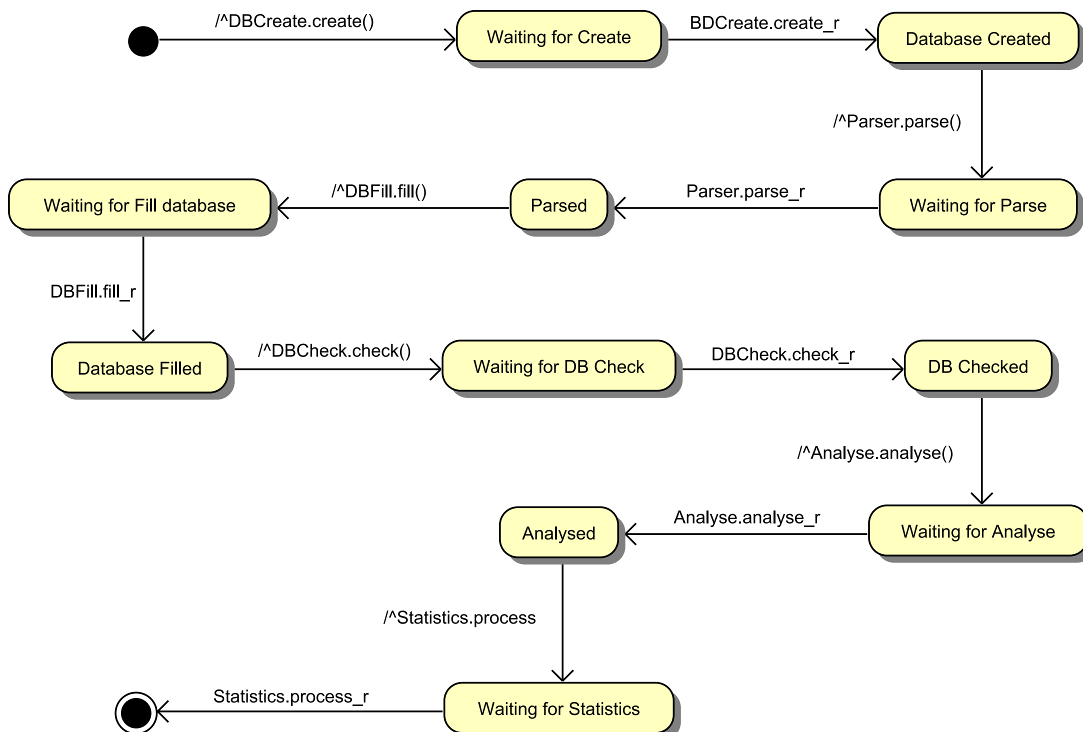


FIGURE 6.7 – Diagramme d'états-transitions de Saat avant refactoring

A partir de ces processus CSP, nous déduisons que le processus PSM_Saat demande l'exécution de la séquence d'appel de méthodes du processus PSM_DB. Ainsi, une relation d'association entre ces deux classes s'avère nécessaire.

Etape 3. Evolution de la spécification

L'application du processus de refactoring sur le diagramme de classes présenté par la Figure 6.3 génère le diagramme de classes proposé dans la Figure 6.8.

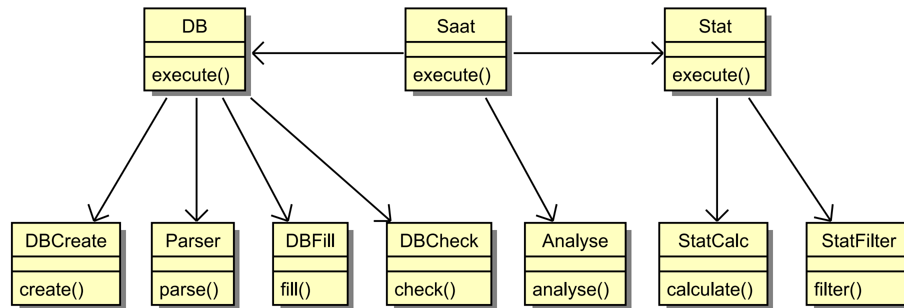


Figure 6.8 – Diagramme de classes après refactoring

De plus, des modifications au niveau des diagrammes états-transitions des classes Saat et DB sont présentées dans les Figures 6.9 et 6.10.

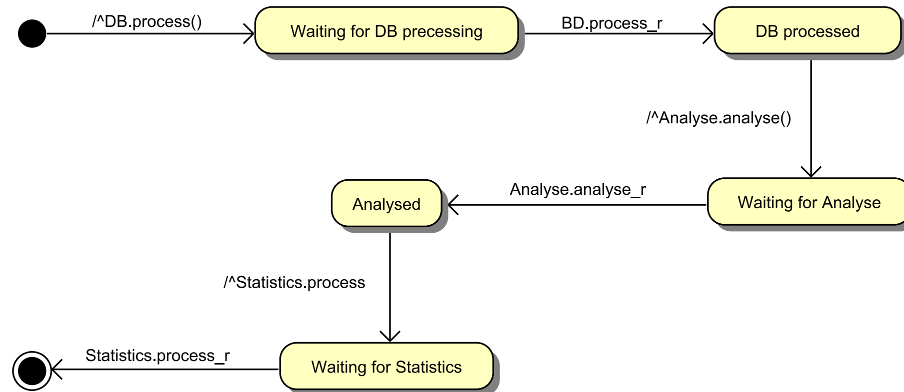


Figure 6.9 – Diagramme d'états-transitions de Saat après refactoring

Les diagrammes d'états-transitions correspondantes aux autres classes restent inchangés.

Etape 4. Correction du schéma

Après refactoring, nous devrons vérifier la conservation du comportement du système SAAT. Pour cela, nous appliquons les règles de vérifications décrites dans la section 6.1.4.

```
PSM_Saat_r  = DB!process → DB_r?x → Analyse!analyse →
              Analyse_r?x → Stat!process → Stat_r?x → SKIP
PSM_DB_r    = DB?x → (DBCreate!create → DBCreate_r?x →
              Parser!parse → Parser_r?x → DBFill!fill →
              DBFill_r?x → DBCheck!check → DBCheck_r?x) →
              DB_r!process → DB
```

A l'aide de l'outil ***FDR2***, nous prouvons facilement la consistance de la relation d'association à ajouter entre Saat et DB.

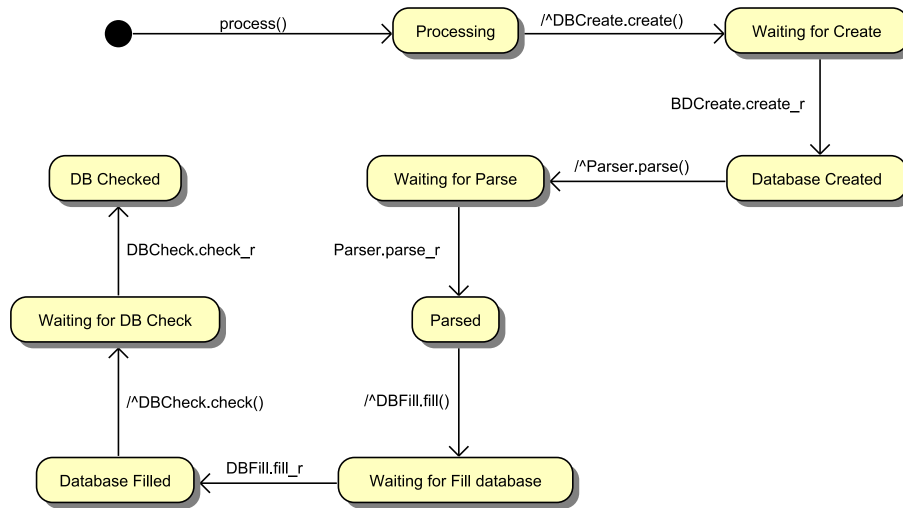


FIGURE 6.10 – Diagramme d'états-transitions de DB après refactoring

6.2 Schéma de refactoring : introduction de la notion de délégation

La construction de diagrammes de classes se fait pas à pas et conduit, dans bon nombre de cas, au développement de classes complexes. Ce type de classes pose des problèmes. En effet, une classe complexe modélise généralement un concept qui encapsule d'autres concepts. Ceci représente un handicap pour l'évolution et la réutilisation de telles classes.

Dans cette section, nous présentons le schéma de refactoring : introduction de la notion de délégation [15]. Nous proposons d'effectuer une partition entre les différents concepts encapsulés dans la classe identifiée comme complexe. Chaque concept encapsulé sera extrait de cette classe complexe et modélisé dans une nouvelle classe reliée par une relation de composition avec la classe complexe. Bien entendu, les propriétés des classes créées seront extraites de la classe complexe.

6.2.1 Etape 1. Identification des paramètres

Le schéma de refactoring d'introduction de la notion de délégation est paramétré par :

$$< Class, OCL_Class, STD_Class >$$

L'identification de la classe complexe doit être suivie par l'extraction d'un concept encapsulé. Ainsi, nous devons identifier les propriétés statiques et dynamiques de ce concept. Cette opération peut être guidée par :
- la recherche de l'identificateur du concept, par exemple un mot clé,
- l'examen de la multiplicité des différents attributs de la classe avec celle de l'identificateur du concept.

Les attributs relatifs au concept à extraire de `l_attr_Class` ont le profil suivant :
- `idConcept : concept[x..y]`
  Cet attribut dénote l'identificateur du concept à extraire. Sa multiplicité `[x..y]` est utilisée comme guide pour l'identification des autres attributs reliés au concept à extraire,
- `attributeJ : typeJ[x..y]`
  Ces attributs modélisent les autres caractéristiques relatives au concept à extraire.

Les méthodes relatives au concept à extraire sont de type modification et consultation. Leur identification est guidée par la présence de l'identificateur du concept en paramètre de ces méthodes. Les méthodes identifiées de `l_meth_Class` sont de la forme :

```
«update» + operationN(idConcept : concept, arg : list_arg)
«query» + operationM(arg : list_arg) : concept
«query» + operationK(idConcept : concept, arg : list_arg) : typeK
```

6.2.2 Etape 2. Vérification des conditions d'application

Le schéma de refactoring proposé est basé sur la notion de délégation, utilisation particulière de la relation d'association. L'idée consiste à redistribuer le contenu d'une classe par le déplacement d'un ensemble d'attributs et de méthodes associés à un concept dans une nouvelle classe, au sens type abstrait de données.

Ainsi, il n'a y aucune vérification à faire.

6.2.3 Etape 3. Evolution de la spécification

L'évolution de la spécification se décompose comme suit :

Création d'une nouvelle classe. Une nouvelle classe, appelée `ComponentOfClass`, modélisant le concept à extraire est créée. Cette nouvelle classe récupère, avec modification, l'ensemble des propriétés identifiées dans l'étape 1. Si `ComponentOfClass` existe déjà dans le diagramme de classes, l'ensemble des propriétés identifiées dans l'étape 1 sera ajouté, avec modification, à cette classe.

Attributs : Ils proviennent de `Class`. Leur type est inchangé et leur multiplicité est égale à 1. La multiplicité sera prise en compte dans la relation entre classes (voir ci-dessous).

```
idConcept : concept
attributeJ : typeJ
```

Méthodes : Elles sont décrites à partir des méthodes de modification et d'interrogation identifiées dans `Class`, lors l'étape précédente. Elles donnent naissance à trois types de méthodes :

- *Consultation.* Le profil de ces méthodes est obtenu par recopie du profil des méthodes d'interrogation avec suppression de l'identificateur du concept, `idConcept`, de la liste des paramètres. Leur profil est de la forme :

  ```
  «query» + operationK (arg : list_arg) : typeK
  ```

- *Modification.* Le profil de ces méthodes est obtenu par recopie du profil des méthodes de modification ayant l'identificateur du concept dans la liste des paramètres, et suppression de cet identificateur :

  ```
  «update» + operationN (arg : list_arg)
  ```

- *Création.* Le profil de ces méthodes est obtenu par recopie du profil des méthodes de consultation ayant l'identificateur du concept comme paramètre résultat dans leur signature :

  ```
  «constructor» + operationM (arg : list_arg)
  ```

Addition de la relation de composition. Une relation de composition est établie entre `Class` et `ComponentOfClass` avec `ComponentOfClass` comme classe composante et `Class` comme classe agrégat. Le sens de navigation va de `Class` vers `ComponentOfClass`. La cardinalité est égale à `x..y`, celle de la multiplicité du concept dans la spécification initiale, du côté de `ComponentOfClass`.

Mise à jour de `Class`. Cette étape consiste à effectuer les mises à jour nécessaires de `Class`, après extraction du concept :

- la suppression des aspects statiques identifiés dans l'étape 1, ces aspects ayant été reportés dans la composante `ComponentOfClass`,
- le profil des méthodes déléguées reste aussi dans `Class`, leur définition est reportée en partie voire totalement au niveau de `ComponentOfClass` (voir ci-dessous).

Mise à jour de OCL_Class et création d'OCL_ComponentOfClass. Un diagramme de classes peut être accompagné d'invariants exprimés en termes de contraintes OCL. Le refactoring du diagramme nécessite de faire évoluer les contraintes OCL si elles portent sur la partie refactorisée [63].

Afin d'avoir des spécifications OCL cohérentes avec le nouveau diagramme, les expressions contenant des propriétés statiques ou dynamiques identifiées dans l'étape 1 seront modifiées de la façon suivante [5] :

- Elles sont réécrites dans Class en utilisant les règles de réécriture présentées dans la TABLE 6.1.

OCL_Class initiales	OCL_Class restructurées
Context Class	
self.attribute$_i$->forAll (attr : type$_i$ | condition$_i$(attr))	self.componentOfClass->forAll (c : ComponentOfClass | condition$_i$(c.attribute$_i$))
self.attribute$_i$[j]	self.componentOfClass.at[j].attribute$_i$

TABLE 6.1 – Règles de réécriture des contraintes dans le contexte de Class

- Elles sont transférées vers ComponentOfClass en utilisant la règle présentée dans la TABLE 6.2.

OCL_Class initiales	OCL_ComponentOfClass restructurées
Context Class	Context ComponentOfClass
self.attribute$_i$->forAll (attr : type$_i$ | condition$_i$(attr))	condition$_i$(attribute$_i$)

TABLE 6.2 – Règle de réécriture des contraintes dans le contexte de ComponentOfClass

Mise à jour de STD_Class et création d'STD_ComponentOfClass. Afin de montrer l'évolution de STD_Class et la création de STD_ComponentOfClass, nous supposons que Class possède une seule opération op, celle qui va être transférée vers sa composante. STD_Class, avant refactoring, est présenté par la FIGURE 6.11.

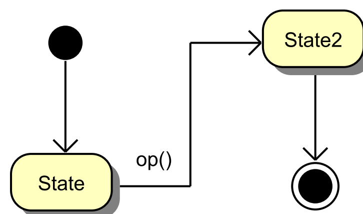


FIGURE 6.11 – STD_Class avant refactoring

Après refactoring, Class demande à ComponentOfClass l'éxécution de op. Ainsi, STD_Class sera restructuré de la façon suivante (voir FIGURE 6.12) et STD_ComponentOfClass sera spécifié de manière à répondre à la demande de Class (voir FIGURE 6.13).

Un schéma illustratif des paramètres du schéma après refactoring est proposé dans la FIGURE 6.14.

6.2.4 Etape 4. Correction du schéma

A cette étape, nous devons vérifier la consistance de la relation de composition entre Class et ComponentOfClass. Cette relation est induite de l'invocation des méthodes de ComponentOfClass par

5. La composition entre Class et ComponentOfClass est assimilée à un ensemble ordonné (OrderedSet en OCL2). Ceci autorise l'utilisation de la fonction at offerte par OrderedSet.

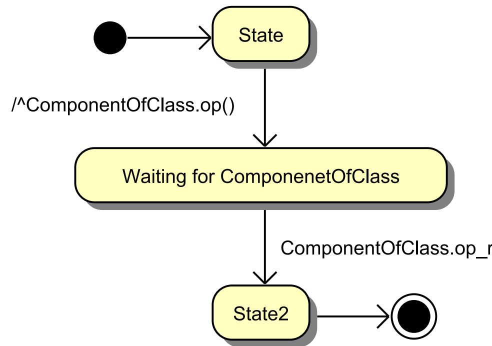


FIGURE 6.12 – STD_Class après refactoring

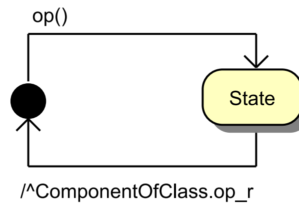


FIGURE 6.13 – STD_ComponentOfClass

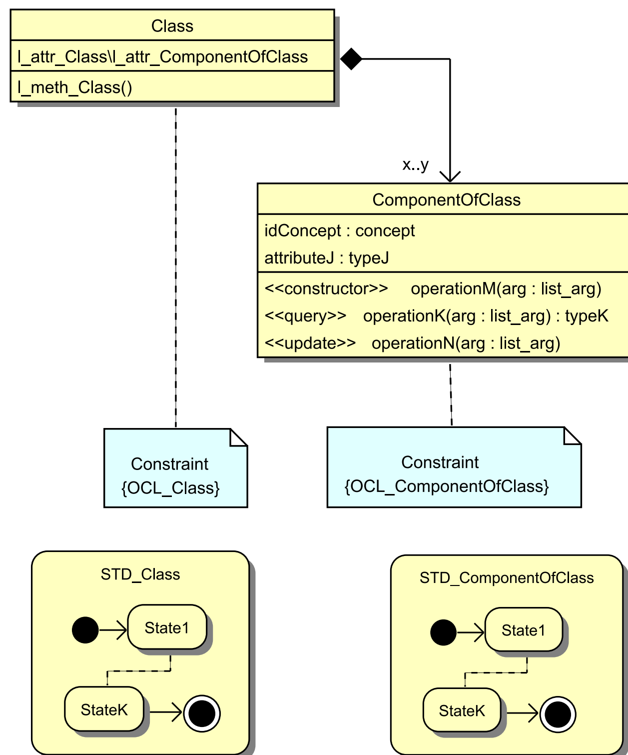


FIGURE 6.14 – Paramètres après refactoring

`Class`. Pour cela, nous utilisons les règles de vérification de la consistance de la relation d'association (voir section 6.1.2).

6.2.5 Etude de cas

Dans ce qui suit, nous illustrons le schéma de refactoring proposé sur l'étude de cas du contrôle d'accès à un bâtiment. Nous partons du diagramme de classes décrit dans la FIGURE 5.9.

Etape 1. Identification des paramètres

Nous remarquons que la classe `Building` est une classe complexe qui encapsule des notions relatives aux portes. Ce diagramme peut être restructuré à l'aide du schéma de refactoring proposé.

Les paramètres du schéma sont :

$$< Building, OCL_Building, STD_Building >$$

Identification des propriétés à extraire.

Attributs relatifs au concept à extraire : Une porte est identifiée par `id_Door`.

Les méthodes relatives au concept à extraire : La présence du concept dans les méthodes de la classe `Building` nous amène à identifier la méthode `unlock`.

Etape 2. Vérification des conditions d'application

Comme l'indique la section 6.2.2, il n'a y aucune vérification à faire.

Etape 3. Evolution de la spécification

L'application du processus de refactoring sur le diagramme de classes génère le diagramme de classes donné dans la FIGURE 6.15.

Les contraintes OCL attachées aux classes `Building` et `Door` ont évolué de la façon suivante :

```
Context Door::unlock(p : Person)
pre P_unlock_Door : self.originBuilding = p.situatedBuilding and
                    p.authorizedBuilding->includes(self.destinationBuilding)
post Q_unlock_Door : p.acceptedDoor = self
```

Les expressions OCL ont été simplifiées, dues au changement de contexte d'utilisation : une collection de portes est gérée dans le contexte `Building`, alors que seul une porte (`self`) est concerné dans le contexte `Door`.

`STD_Building` est modifié de la manière suivante (voir FIGURE 6.16).

Le nouveau diagramme `STD_Door` est présenté par la FIGURE 6.17.

Etape 4. Correction du schéma

Après une vérification similaire à celle de la consistance de la relation d'association (voir section 6.1.2), nous déduisons l'exactitude des modifications apportées par le schéma de refactoring proposé.

6.3 Schéma de refactoring : introduction de la notion de généricité

Un des grands empêchements à la conception de logiciels *réutilisables* est dû aux types des variables. La généricité des types de variables apporte une solution à ce problème. En effet, à partir de ces types génériques, on peut créer des classes génériques qui constituent des germes de classes qui peuvent être réutilisés dans différents contextes. Il s'agit précisément d'un schéma ou un patron pour générer autant de

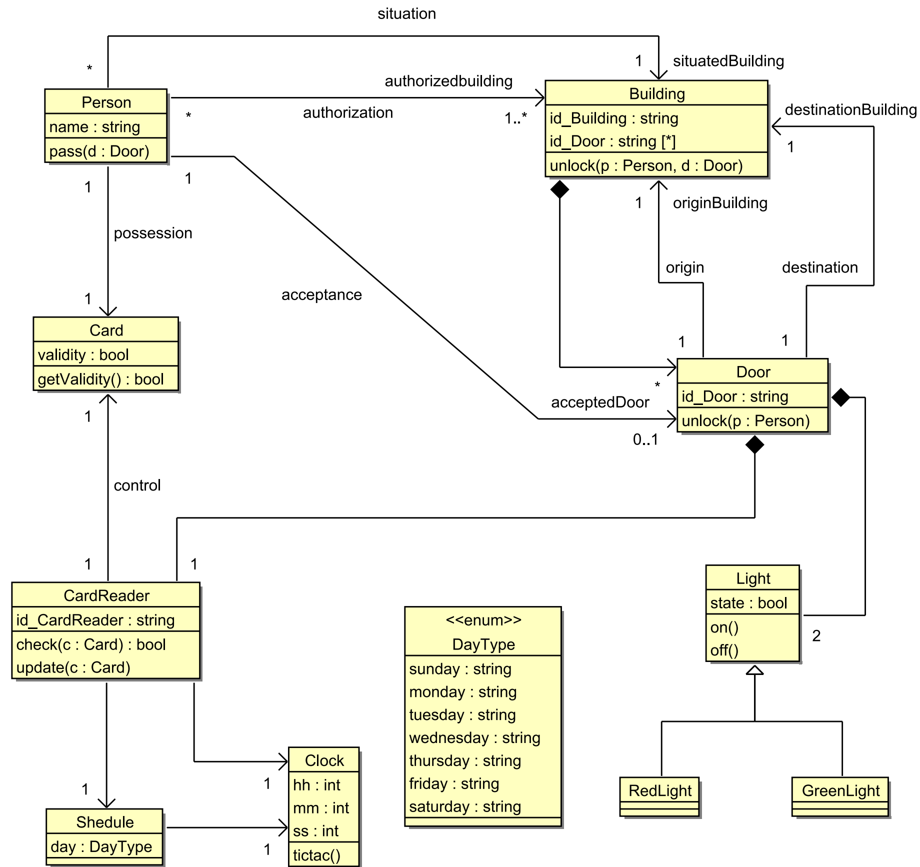


FIGURE 6.15 – Diagramme de classes après refactoring

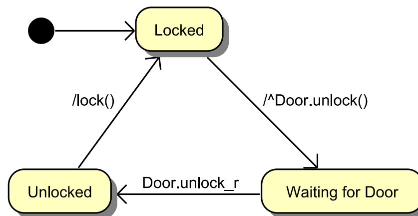


FIGURE 6.16 – `STD_Building` après refactoring

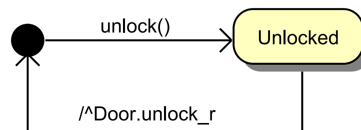


FIGURE 6.17 – `STD_Door`

classes réelles que nécessaire. Une classe réelle est obtenue à partir d'une classe générique en substituant chacun des paramètres de la classe générique par des types réels. Ceci est connu sous le nom de dérivation générique.

Dans cette section, nous proposons un schéma de refactoring permettant de transformer une classe existante en une classe générique.

6.3.1 Etape 1. Identification des paramètres

Le schéma de refactoring d'introduction de la notion de généricité est paramétré par (voir FIGURE 6.18) :

$$< Class1, OCL_Class1, STD_Class1 >$$

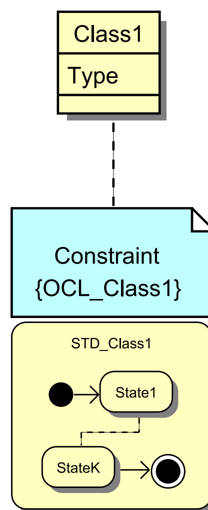


FIGURE 6.18 – Paramètres avant refactoring

Supposons que la spécification OCL de `Class1` est modélisée de la façon suivante :

```
OCL_Class1

Context Class1
attribute : Type

Context Class1::op(x:Type)

Context Class1::op():Type
```

6.3.2 Etape 2. Vérification des conditions d'application

A partir d'une classe réelle, on peut créer une classe générique si et seulement si la réalisation des services offerts par cette classe ne dépend pas d'un type particulier. Par exemple, nous pouvons citer les structures de données conteneurs, implantées de manière indépendante des éléments qu'elles contiennent.

La vérification des conditions d'application consiste à identifier une répétition d'un `Type` dans les types des propriétés statiques et dans les paramètres formels des opérations offertes par une classe `Class1`.

6.3.3 Etape 3. Evolution de la spécification

L'évolution de la spécification se décompose comme suit :

Création de `Class[G]` : Afin d'obtenir une classe générique, `Class[G]` hérite toutes les propriétés statiques et dynamiques de `Class1` et les modifications suivantes doivent être apportées :

- `Type` est remplacé par le paramètre générique `G` dans tout le typage des aspects statiques et des paramètres formels des aspects dynamiques de `Class1`,
- les relations, quelles que soient leurs natures, restent inchangées tant pour la navigabilité que pour la multiplicité.

Création de `Class2` et suppression de `Class1` : Après la suppression de `Class1` et pour conserver toutes les propriétés statiques et dynamiques de la spécification UML initiale, nous créons une classe, appelée `Class2`, identique à la classe `Class1`. Ensuite, nous créons un lien de réalisation entre `Class[G]` et `Class2`.

Création de `OCL_Class[G]` et de `OCL_Class2`. Elles sont réécrites en utilisant les règles de réécriture :

OCL_Class2	OCL_Class_G[G]
Context Class2 attribute : Type	Context Class_G attribute : G
Context Class2::op(x:Type)	Context Class_G::op(x:G)
Context Class2::op():Type	Context Class_G::op():G

TABLE 6.3 – Règles de réécriture de `OCL_Class_G[G]` et de `OCL_Class2`

Création de `STD_Class[G]` et de `STD_Class2`. La classe générique `Class[G]` a le même diagramme d'états-transitions que son homologue `Class1`, puisque les transitions et les états du diagramme d'états-transitions de la classe `Class_G` sont enchaînées de la même façon (même nombres et même ordres) que la classe réelle `Class1`. De même, pour `STD_Class2`.

La FIGURE 6.19 présente `Class[G]` après le refactoring.

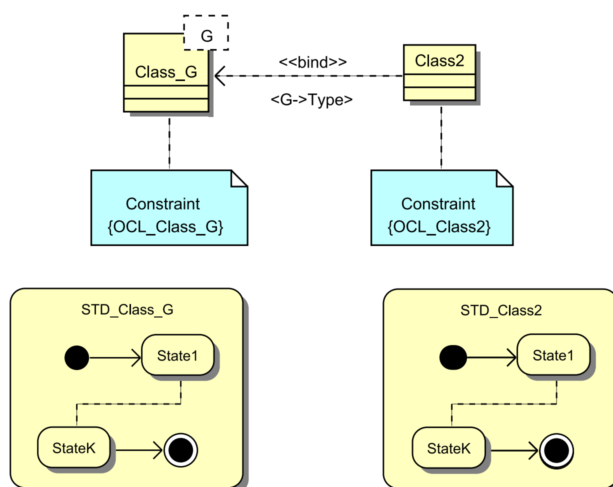


FIGURE 6.19 – Paramètres après refactoring

Ainsi, `Class[G]` devient une classe générique indépendante de l'argument `Type` défini dans sa première présentation.

6.3.4 Etape 4. Correction du schéma

Les modifications apportées par l'application du schéma de refactoring doivent préserver la conception initiale. Pour cela, il suffit de montrer que le comportement de la classe `Class2` préserve le comportement de la classe `Class1`. Comme l'indique l'étape 3, les deux classes `Class1` et `Class2` sont identiques. Ainsi, il n'y a aucune vérification à effectuer.

6.3.5 Etude de cas

Nous illustrons le schéma de refactoring : introduction de la notion de généricité sur l'étude de cas du contrôle d'accès à un bâtiment. Nous partons du diagramme de classes présentée dans la FIGURE 6.15.

Etape 1. Identification des paramètres

Les paramètres du schéma sont :

$$< CardReader, OCL_CardReader, STD_CardReader >$$

Etape 2. Vérification des conditions d'application

Nous remarquons que la classe `CardReader` dépend du type `card`, ainsi une application du schéma de refactoring : introduction de la notion de généricité est nécessaire.

Etape 3. Evolution de la spécification

L'application du processus de refactoring sur le diagramme de classes de la FIGURE 6.15 génère le diagramme de classes proposé dans la FIGURE 6.20, les autres diagrammes restent inchangés.

Les contraintes OCL attachées à la classe `Reader_G` et générées par l'application du schéma de refactoring sont les suivantes :

```
Context Reader_G::check(g:G)
pre P_check_Reader_G : shedule.day <> sunday and
    clock.hh > 8 and clock.hh < 18
post Q_check_Reader_G : g.getValidity()
```

Etape 4. Correction du schéma

Comme l'indique la section 6.3.4, il n'a y aucune vérification à faire. En effet, il suffit de substituer le paramètre générique `G` par `card` dans la nouvelle classe `Reader_G` pour obtenir la classe `CardReader`.

6.4 Conclusion

Dans un premier temps, nous avons proposé un schéma de refactoring permettant d'introduire une association de type invocation entre deux classes existantes. Celles-ci sont reliées par des associations de type invocation à d'autres classes communes. La vérification de la correction de ce schéma de refactoring est obtenue moyennant une relation de raffinement entre deux processus CSP issus des diagrammes d'états-transition associés aux deux classes concernées. Dans un deuxième temps, nous avons établi un deuxième schéma de refactoring permettant de faire face aux classes complexes voire monstres. L'idée de ce schéma est d'extraire méthodiquement la notion (ou le concept) englobée au sein de la classe complexe. Enfin, nous avons introduit un troisième schéma de refactoring permettant de paramétrer une classe existante.

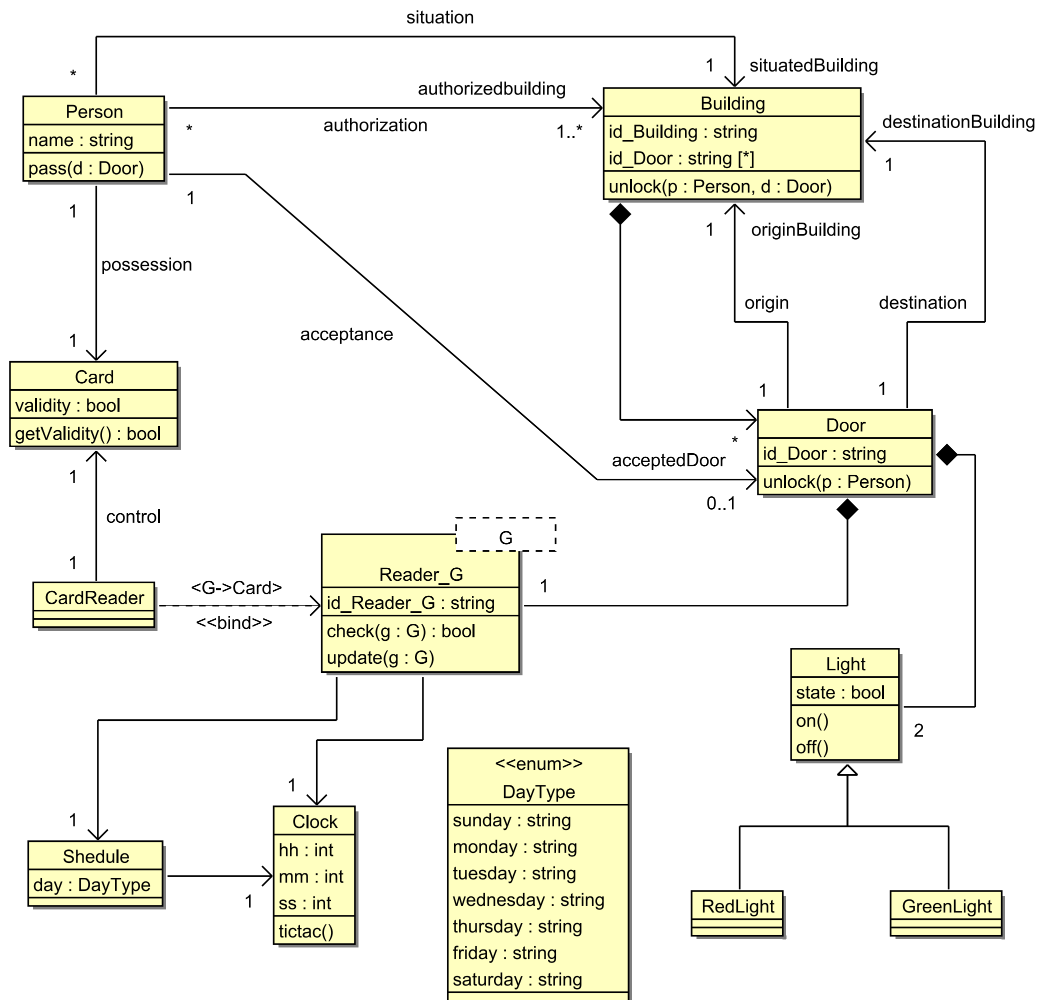


FIGURE 6.20 – Diagramme de classes après refactoring

Conclusions et perspectives

1 Raffinement

1.1 Bilan

La technique de raffinement permet l'obtention de logiciels corrects par construction. Mais l'application de cette technique pose des problèmes aussi bien dans un cadre formel que semi-formel.

Dans un cadre formel comme B, Event-B, CSP, Z et Object-Z, bien que la relation de raffinement soit bien définie et outillée (génération des obligations des preuves, prouveur interactif, model-checker et animateur), un spécifieur éprouve des difficultés plus ou moins importantes à identifier les différents niveaux d'abstraction (stratégie de raffinement "optimale") et mener à terme le processus de raffinement. Des solutions, basées sur le concept de pattern – à l'instar des patterns de conception dans le monde OO – permettant de guider le spécifieur lors du processus de raffinement commencent à apparaître couvrant aussi bien le raffinement vertical visant des domaines d'application comme les systèmes réactifs [3, 55, 95] que le raffinement horizontal dans le cadre de B. Par exemple l'outil BART [86] associé à B propose des règles de raffinement pouvant être utilisées en étapes finales de la phase Raffinement vertical d'un processus formel de développement.

Dans un cadre semi-formel celui d'UML, le raffinement - hormis la construction rudimentaire «refine» – n'est pas supporté et a fortiori ne peut pas être vérifié.

Dans cette thèse, nous avons apporté une contribution permettant aux spécifieurs UML de tirer profit de la technique de raffinement, sous la forme d'un catalogue de 7 patterns de raffinement :

1. Introduction d'une classe intermédiaire : `Class_Helper`.
2. Réification d'un attribut : `Class_Attribute`.
3. Enrichissement d'une association : `Class_Association`.
4. Décomposition d'une classe : `Class_Decomposition`.
5. Introduction d'une nouvelle entité : `Class_NewEntity`.
6. Raffinement de contrôle d'une classe : `Refinement_Operation`.
7. Abstraction d'une classe : `Class_Abstraction`.

Ces patterns permettent une construction incrémentale des diagrammes de classes UML (deuxième partie, chapitre 3).

De plus, nous avons proposé une démarche de développement des diagrammes de classes UML basée sur le concept raffinement avec preuves. Notre démarche comporte quatre phases : Réécriture du cahier des charges, Stratégie de raffinement, Spécification abstraite et Raffinement.
Nous avons appliqué cette démarche sur l'étude de cas : Contrôle d'accès aux bâtiments. Ceci nous a permis de réutiliser des patterns de raffinement proposés. En outre, le développement conjoint UML/B nous a permis de vérifier la correction de chaque étape de raffinement (deuxième partie, chapitre 4).

Le Tableau I permet de positionner notre contribution vis-à-vis des approches existantes déjà étudiées dans le chapitre 1. En fait, nous reprenons le Tableau 1.1 en lui ajoutant une ligne signalant les points forts et faibles de la contribution préconisée.

1.2 Perspectives

Pour faire face aux points faibles de notre approche relative à l'intégration du concept raffinement dans UML, nous pourrions envisager les deux prolongements directs suivants :

- proposer une certaine automatisation de l'application des patterns de raffinement en s'inspirant des travaux autour d'Event-B [45] ;

Approches de	Points forts	Points faibles
N. Belloir et *al.*	– une définition de contrats de transformation de modèles – la prise en compte des contraintes OCL – la vérification est effectuée au niveau méta (M2)	– la vérification nécessite l'écriture de fonctions utilitaires souvent complexes en OCL en utilisant la construction `def`
L. C. Briand et *al.*	– 47 changements atomiques dont 31 raffinements atomiques applicables sur les diagrammes de classes – la composition des raffinements – la détection des raffinements via l'outil VIATool	– la non prise en compte des aspects sémantiques – des problèmes de cohérence
N. Correa et *al.*	– une extension d'UML sous forme d'un profil – la prise en compte des contraintes OCL	– la description des raffinements des modèles UML concerne uniquement l'enrichissement syntaxique d'UML
W. L. Low et *al.*	– des règles de raffinement des diagrammes de classes – la vérification est effectuée au niveau méta (M2) – la prise en compte des contraintes OCL	– les règles de raffinement concernent uniquement le concept relation d'UML : association, composition et généralisation/spécialisation
R. Van Der Streaten et *al.*	– la formalisation d'une opération de raffinement de classes – la prise en compte du comportement d'une classe – la vérification via la traduction d'UML vers un formalisme logique basé sur la logique du premier ordre – un prototype sous forme d'un plug-in	– une unique opération de raffinement : introduction d'une sous-classe
L. Zhao et *al.*	– un ensemble de règles permettant d'étendre un diagramme de classes par décomposition et adjonction des classes – une définition formelle de la notion de raffinement structurel en termes de transformations de graphes	– la non prise en compte des contraintes OCL – des règles de raffinement qui se limitent à la décomposition et l'adjonction des classes
B. Hnatkowska et *al.*	– des règles liées au raffinement structurel – des règles liées au raffinement comportemental – l'utilisation de la théorie des ensembles – un prototype sous forme d'un plug-in	– des règles de raffinement qui se limitent aux diagrammes de collaboration et de communication
C. Pons et *al.*	– 5 patterns de raffinement – l'utilisation des structures de raffinement venant des méthodes formelles (Object-Z) – la prise en compte des contraintes OCL – la découverte des raffinements cachés – la documentation entre deux modèles UML successifs en utilisant la dépendance stéréotypée "refine"	– des insuffisances liées à OCL concernant le mapping entre les deux modèles (abstrait et concret)
B. Ben Ammar et *al.*	– un ensemble des patterns de raffinement – une bonne couverture des concepts statiques d'UML : classe, relation et contraintes OCL – une assistance méthodologique combinant UML/B – une vérification formelle des propriétés de sûreté	– pas d'assistance automatique d'application ou de réutilisation des patterns de raffinement – pas de prise en compte des aspects dynamiques d'UML

TABLE I – Contribution de raffinement proposée vis-à-vis des approches existantes

– enrichir les patterns de raffinement proposés en tenant compte des aspects dynamiques UML décrits par des diagrammes de séquence, d'états-transition ou d'activité.

Les patterns de raffinement proposés favorisent plutôt l'identification des classes d'analyse qui modélisent des concepts métier issus du cahier des charges. Deux nouvelles orientations pourraient être explorées :

– proposition des patterns de raffinement orientés conception en récupérant et adaptant des idées venant des patterns de GoF,
– proposition des patterns de raffinement orientés implantation en se servant de la modélisation orientée objet des structures de données universelles en Eiffel [70].

2 Refactoring

2.1 Bilan

La technique de refactoring permet d'améliorer la qualité du logiciel notamment sur les plans de l'extensibilité, de la réutilisabilité et de l'efficacité. Elle est fréquemment appliquée sur le code. Le problème central du refactoring réside dans la préservation du comportement suite à une restructuration. Cette thèse plaide en faveur de l'utilisation de la technique de refactoring à une étape avancée du développement du logiciel. En effet, les 7 schémas de refactoring proposés sont appliqués aussi bien sur des diagrammes de classes, que sur des contraintes OCL et des diagrammes d'états-transitions en vue d'obtenir des modèles UML de qualité, c'est-à-dire corrects, extensibles, réutilisables et efficaces :

1. Introduction de la notion d'héritage.
2. Introduction de la notion de redéfinition.
3. Introduction de la notion de classe abstraite.
4. Introduction de la notion de polymorphisme.
5. Introduction de la notion d'association.
6. Introduction de la notion de délégation.
7. Introduction de la notion de généricité.

La préservation du comportement est confiée aux outils de vérification formelle associés à B et CSP (troisième partie).

En guise de preuve de faisabilité, nous avons appliqué avec succès ces schémas de refactoring sur plusieurs exemples illustratifs.

Le TABLEAU II permet de positionner notre contribution vis-à-vis des approches existantes déja étudiées dans le chapitre 1. En fait, nous reprenons le TABLEAU 1.2 en lui ajoutant une colonne évaluant notre contribution vis-à-vis des critères retenus.

2.2 Perspectives

Le canevas adopté de description des schémas de refactoring proposés favorise le développement d'un outil d'automatisation de ces schémas.

Les schémas de refactoring proposés sont élémentaires. Ils couvrent l'introduction de la notion d'héritage, d'association, de polymorphisme, de classe abstraite, de délégation et de généricité. Il est donc intéressant de les étendre, en vue de proposer des schémas de refactoring élaborés. Une piste à creuser est le mariage entre le refactoring et les patterns d'analyse ou de conception notamment ceux de GoF. En effet, un modèle UML existant peut être amélioré par une opération de refactoring introduisant un pattern

Approche de	S. Markovic et *al.*	P. Gorp et *al.*	M. V. Kempen et *al.*	T. Mens et *al.*	S. Markovic et *al.*	G. Sunyé et *al.*	A. Correa et *al.*	B. Ben Ammar et *al.*
Prise en compte de diagramme de classes	oui	partielle	non	oui	oui	oui	non	oui
Prise en compte de diagramme d'états-transition	non	non	oui	oui	non	oui	non	oui
Prise en compte de contraintes OCL	oui	oui	non	non	oui	non	oui	oui
Préservation du comportement	transformation de modèle formalisée en QVT	méta-modélisation	UML vers des processus CSP	UML vers des graphes	grammaires de graphes	réécriture	réécriture	B et CSP
Outil	supportant QVT	moteur de requête OCL	supportant CSP	Fujaba pour la transformation de graphes	formalisme basé sur les grammaires de graphes	non	non	non
Détection du refactoring	non	design smells	non	le refactoring le mieux adapté	non	non	OCL smells	non

TABLE II – Contribution de refactoring proposée vis-à-vis des approches existantes

de conception ou d'analyse. Celle-ci pourrait être réalisée par **composition** de schémas de refactoring existants.

L'identification du concept que l'on cherche à extraire est un problème ouvert (voir Schéma de refactoring : Introduction de la notion de délégation). Une piste consiste à s'inspirer des résultats obtenus dans le cadre du refactoring de code, avec la détection des défauts structurels, "bad smells". [9] utilise les diagrammes UML pour les détecter.

Bibliographie

[1] J. R. Abrial. Etude Système : méthode et exemple. Technical report, ClearSy System Engineering, Octobre 1998.

[2] J. R. Abrial. Train systems. In *RODIN Book*, pages 1–36, 2006.

[3] J. R. Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[4] J. R. Abrial and D. Cansell. Click'n prove : Interactive proofs within set theory. In *TPHOLs*, pages 1–24, 2003.

[5] J. R. Abrial and T. S. Hoang. Using design patterns in formal methods : An event-b approach. In *ICTAC*, pages 1–2, 2008.

[6] J.R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996.

[7] AFADL'2000. Etude de cas : système de contrôle d'accès. In *Journées AFADL, Approches formelles dans l'assistance au développement de logiciels*, 2000. Actes LSR/IMAG.

[8] K. Allem and T. Mens. Refactoring des modèles : concepts et défis. In *Proc. IDM 2007*. Hermes Science Publications, Lavoisier, 2007.

[9] D. Astels. Refactoring with UML. In *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering (XP)*, pages 67–70, 2002. Alghero, Sardinia, Italy.

[10] B-Core. *The B-Toolkit home page. http ://www.b-core.com/btoolkit.html*, 2002.

[11] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook : Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[12] I. Bayley. Formalising design patterns in predicate logic. *Software Engineering and Formal Methods, International Conference on*, 0 :25–36, 2007.

[13] B. Ben Ammar, M. T. Bhiri, and J. Souquières. Control access case study : an incremental development of UML specifications. Technical report, LORIA, 2007.

[14] B. Ben Ammar, M. T. Bhiri, and J. Souquières. Quelques patrons de raffinement pour le développement de diagrammes de classes UML. In *6ème atelier sur les Objets, Composants et Modèles dans l'ingénierie des Systèmes d'Information, OCM-SI, couplé avec le 15ème congrès INFORSID*, Perros-Guirec France, 2007.

[15] B. Ben Ammar, M. T. Bhiri, and J. Souquières. Schéma de refactoring de diagrammes de classes basé sur la notion de délégation. In *7ème atelier sur l'Evolution, Réutilisation et Traçabilité des Systèmes d'Information, ERTSI, couplé avec le XXVIème congrès INFORSID*, Fontainebleau France, 2008.

[16] B. Ben Ammar, M. T. Bhiri, and J. Souquières. Incremental development of uml specifications using operation refinements. *Innovations in Systems and Software Engineering*, 4 :259–266, 2008.

[17] B. Ben Ammar, M. T. Bhiri, and J. Souquières. Modélisation événementielle pour la construction de diagrammes de classes. *RSTI - ISI*, 13 :131–155, 2008.

[18] R. Binder. *Testing Object Oriented Systems : Models, Patterns and Tools*. Addison Wesley.

[19] L. C. Briand, Y. Labiche, L. O'Sullivan, and M. M. Sówka. Automated impact analysis of uml models. *J. Syst. Softw.*, 79(3) :339–352, 2006.

[20] L. C. Briand, Y. Labiche, L. O'Sullivan, and M. M. Sówka. Automated traceability analysis for UML model refinements. Technical report, Carleton University, TR SCE-06-06, Version 2, August 2006.

[21] L. C. Briand, Y. Labiche, and T. Yue. Automated traceability analysis for uml model refinements. *Inf. Softw. Technol.*, 51(2) :512–527, 2009.

[22] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture : a system of patterns*, volume 1. John Wiley and Sons, 1996.

[23] R.-D. Kutsche C. Pons. Traceability across refinement steps in uml modeling. In *in : Proceedings of the Workshop on Software Model Engineering, in conjunction with UML'04*, 2004.

[24] D. Cansell and D. Méry. Event b. Hermes-Science Lavoisier, 2006.

[25] ClearSy. *BEditor tool homepage. http ://www.methode-b.com/php/outils_b_plugin_b_editor_fr.php*, 2007.

[26] Clearsy. *Atelier B tool homepage. http ://www.atelierb.societe.com*, 2008.

[27] A. Correa and C. Werner. Refactoring object constraint language specifications. *Software and Systems Modeling*, 6(2) :113–138, June 2007.

[28] N. Correa and R. Giandini. A uml extension to specify model refinements, 2006.

[29] J. Davies and C. Crichton. Concurrency and refinement in the unified modeling language. *Formal Aspects of Computing*, V15(2) :118–145, November 2003.

[30] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8) :453–457, 1975.

[31] N. Belloir E. Cariou and F. Barbier. Contrats de transformation pour la validation de raffinement de modèles. In *5èmes journées sur l'Ingénierie dirigée par les Modèles (IDM)*, Nancy France, 2009.

[32] J. Ebert and G. Engels. Specialization of Object Life Cycle Definitions. Technical report, Koblenz University, January 1997.

[33] M. Fowler. *Analysis Patterns : Reusable Object Models (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, October 1996.

[34] M. Fowler. *Refactoring : Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999.

[35] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring : Improving the Design of Existing Code.* Addison-Wesley Professional, June 1999.

[36] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, Boston, MA, January 1995.

[37] F. Gervais, M. Frappier, and R. Laleau. *Vous avez dit raffinement ?* Technical Report 829, CEDRIC, Paris, France, March 2005.

[38] M. Goldsmith. *FDR2 User's Manual version 2.82*, June 2005.

[39] P. Gorp, H. Stenten, T. Mens, and S. Demeyer. Towards automating source-consistent UML refactorings. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language*, volume 2863 of *Lecture Notes in Computer Science*, pages 144–158. Springer-Verlag, 2003.

[40] S. Graeme. *The Object-Z specification language.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[41] M. Guyomard. Spécification et raffinement en B : deux exemples pédagogiques. *ZB2002 4th International B Conference, Education Session Proceedings*, Janvier 2002.

[42] H. Habrias and C. Stoquer. Une sémantique formelle pour le raffinage en UML. In *XII Colloque National de la Recherche en IUT, CNRIUT'06*, Brest, France, June 2006.

[43] R. Hennicker, H. Hussmann, and M. Bidoit. On the precise meaning of ocl constraints. pages 415–418. 2002.

[44] B. Hnatkowska, Z. Huzar, and L. Tuzinkiewicz. Refinement of uml collaborations. *Int. J. Appl. Math. Comput. Sci.*, 16(1) :155–164, 2006.

[45] T. S. Hoang, A. Furst, and J. R. Abrial. Event-b patterns and their tool support. *Software Engineering and Formal Methods, International Conference on*, 0 :210–219, 2009.

[46] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, October 1969.

[47] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8) :666–677, 1978.

[48] C. A. R. Hoare. *Communicating sequential processes.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[49] C. A. R. Hoare. *Communicating sequential processes.* Electronic edition edited by Jim Davies, 2004.

[50] A. Idani, Y. Ledru, and M.-A. Labiadh. Ingénierie dirigée par les modèles pour une intégration efficace de uml et b. In *INFORSID 2009*, Toulouse, May 2009.

[51] W. Ifill, S. A. Schneider, and H. Treharne. Augmenting b with control annotations. In Jacques Julliand and Olga Kouchnarenko, editors, *B*, volume 4355 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 2007.

[52] R. Laleau. *Conception et développement formels d'applications bases de données.* PhD thesis, CEDRIC (CNAM), University of Evry, 2002. Habilitation à diriger des recherches.

[53] K. Lano, K. Androutsopoulos, and D. Clark. Refinement patterns for uml. *Electr. Notes Theor. Comput. Sci.*, 137(2) :131–149, 2005.

[54] K. Lano, D. Clark, and K. Androutsopoulos. Uml to b : Formal verification of object-oriented models. In *IFM*, pages 187–206, 2004.

[55] T. Lecomte, D. Méry, and D. Cansell. Patrons de conception prouvés. *Génie Logiciel - Magazine de l'ingéniérie du logiciel et des systèmes*, pages 14–18, 2007.

[56] H. Ledang. Automatic Translation from UML Specifications to B. In *ASE '01 : Proceedings of the 16th IEEE international conference on Automated software engineering*, page 436. IEEE Computer Society, 2001.

[57] H. Ledang. *Traduction Systématique de spécifications UML vers B.* Thèse de doctorat, LORIA - Université Nancy 2, Novembre 2002.

[58] H. Ledang and J. Souquières. Integration of uml and b specification techniques : Systematic transformation from ocl expressions into b. In *APSEC '02 : Proceedings of the Ninth Asia-Pacific Software Engineering Conference*, page 495, Washington, DC, USA, 2002. IEEE Computer Society.

[59] B. Liskov and S. Zilles. Programming with abstract data types. *SIGPLAN Not.*, 9(4) :50–59, 1974.

[60] R. Marcano and N. Levy. Transformation rules of OCL constraints into B formal expressions. In Jan Jürjens, María Victoria Cengarle, Eduardo B. Fernandez, Bernhard Rumpe, and Robert Sandner, editors, *Critical Systems Development with UML – Proceedings of the UML'02 workshop*, pages 155–162. Technische Universität München, Institut für Informatik, 2002.

[61] R. Marcano and N. Levy. Using B formal specifications for analysis and verification of UML/OCL models. In Ludwik Kuzniarz, Gianna Reggio, Jean Louis Sourrouille, and Zbigniew Huzar, editors, *Blekinge Institute of Technology, Research Report 2002 :06. UML 2002, Model Engineering, Concepts and Tools. Workshop on Consistency Problems in UML-based Software Development. Workshop Materials*, pages 91–105. Department of Software Engineering and Computer Science, Blekinge Institute of Technology, 2002.

[62] S. Markovic. *Model refactoring using transformations.* PhD thesis, Lausanne, 2008.

[63] S. Marković and T. Baar. Synchronizing Refactored UML Class Diagrams and OCL Constraints. In *1st Workshop on Refactoring Tools , ECOOP07 Conference Workshop, July 31, 2007, Berlin, Germany*, pages 15–17. TU Berlin Technical Report, ISSN 1436-9915, 2007.

[64] S. Marković and T. Baar. Refactoring ocl annotated uml class diagrams. *Software and Systems Modeling*, 7(1) :25–47, February 2008.

[65] T. Mens. On the use of graph transformations for model refactoring. In Joost, editor, *Generative and transformational techniques in software engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 215–254. Springer, 2006.

[66] T. Mens and P. Van Gorp. A taxonomy of model transformation. In *Proc. International Workshop on Graph and Model Transformation (GraMoT 2005)*, volume 152. Elsevier.

[67] T. Mens, G. Taentzer, and D. Müller. Challenges in model refactoring. In *Proc. 1st Workshop on Refactoring Tools*. University of Berlin, 2007.

[68] T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *Software and Systems Modeling*, pages 269–285, September 2007.

[69] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2) :126–139, February 2004.

[70] B. Meyer. *Reusable software : the Base object-oriented component libraries*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[71] B. Meyer. *Conception et programmation orientées objet*. Eyrolles, 2000.

[72] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 2nd edition, March 2000.

[73] E. Meyer. *Developpements formels par objes : utilisation conjointes de B et d'UML*. Thèse de doctorat, LORIA - Université Nancy 2, Mars 2001.

[74] E. Meyer and J. Souquières. A Systematic Approach to Transform OMT Diagrams to a B Specification. In *FM '99 : Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume I*, pages 875–895. Springer-Verlag, 1999.

[75] M. Mohamed and M. Romdhaniand K. Ghedira. Classification des approches de refactorisation des modèles. In *4ème Journées sur l'Ingénierie Dirigée par les Modèles (IDM)*, Mulhouse France, 2008.

[76] J. Moores. CCSP – a Portable CSP-based Run-time System Supporting C and occam. In B.M.Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering series*, pages 147–168, Amsterdam, the Netherlands, April 1999. WoTUG, IOS Press.

[77] OMG. *UML 2.0 OCL Specification*, Juin.

[78] OMG. *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006.

[79] W. F. Opdyke. *Refactoring : A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[80] C. Pons. Heuristics on the Definition of UML Refinement Patterns. *Springer-Verlag Berlin Heidelberg*, 2006.

[81] C. Pons and D. Garcia. An ocl-based technique for specifying and verifying refinement-oriented transformations in mde. In *MoDELS*, pages 646–660, 2006.

[82] C. Pons, R. S. Giandini, G. Pérez, P. Pesce, V. Becker, J. Longinotti, and J. Cengia. Pampero : Precise assistant for the modeling process in an environment with refinement orientation. In *UML Satellite Activities*, pages 246–249, 2004.

[83] C. Pons, G. A. Perez, R. Giandini, and R. Kutsche. Understanding Refinement and Specialization in the UML. In *2nd International Workshop on MAnaging SPEcialization/Generalization HIerarchies (MASPEGHI 2003)*, 2003.

[84] H. Rasch and H. Wehrheim. Checking Consistency in UML Diagrams : Classes and State Machines. In E. Najm, U. Nestmann, and P. Stevens, editors, *Formal Methods for Open Object-based Distributed Systems*, volume 2884 of *LNCS*, pages 229–243. Springer, 2003.

[85] Refactoring Community. *Refactoring home page. http ://www.refactoring.com*, 2005.

[86] A. Requet. Bart : A tool for automatic refinement. In *ABZ*, page 345, 2008.

[87] A. W. Roscoe. *Model-checking CSP*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994.

[88] A. W. Roscoe. Csp and determinism in security modelling. In *SP '95 : Proceedings of the 1995 IEEE Symposium on Security and Privacy*, page 114, Washington, DC, USA, 1995. IEEE Computer Society.

[89] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[90] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1 : Foundations.* World Scientific, 1997.

[91] J. B. Scattergood. *Tools for CSP and Timed CSP.* Ph.d. thesis, Oxford University Computing Laboratory, 1998.

[92] C. Snook and M. Butler. U2b - a tool for translating uml-b models into b. In *in : UML-B Specification for Proven Embedded Systems Design*, 2004.

[93] J. M. Spivey. The Z notation : a reference manual. *Prentice-Hall International Series In Computer Science*, page 155, 1989.

[94] R. Van Der Straeten, V. Jonckers, and T. Mens. A formal approach to model refactoring and model refinement. *Software and Systems Modeling*, 6(2) :139–162, June 2007.

[95] W. Su, J. R. Abrial, R. Huang, and H. Zhu. From requirements to development : Methodology and example. In *ICFEM*, pages 437–455, 2011.

[96] G. Sunyé, D. Pollet, Y. Le Traon, and J.-M. Jézéquel. Refactoring UML models. In *Proceedings of UML 2001*, volume 2185 of *LNCS*, pages 134–148. Springer Verlag, 2001.

[97] Formal Systems. *Formal Systems homepage. http ://www.fsel.com/index.html.*

[98] A. Toval, V. Requena, and J. L. Fernández. Emerging ocl tools. *Software and Systems Modeling*, 2(4) :248–261, December 2003.

[99] M. van Kempen, M. Chaudron, D. Kourie, A. Boake, and A. Boake. Towards proving preservation of behaviour of refactoring of uml models. In *SAICSIT '05 : Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, pages 252–259, Republic of South Africa, 2005. South African Institute for Computer Scientists and Information Technologists.

[100] A. van Lamsweerde. *Requirements Engineering - From System Goals to UML Models to Software Specifications.* Wiley, 2009.

[101] Y. Lu W. Shen and W.L. Low. Extending the uml metamodel to support software refinement. In *Proceedings of the Workshop on Consistency Problems in UML-Based Software Development, in conjunction with UML*, 2002.

[102] P.H. Welch, N.C.C. Brown, J. Moores, K. Chalmers, and B. Sputh. Integrating and Extending JCSP. In Steve Schneider, Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, volume 65 of *Concurrent Systems Engineering*, pages 349–370, Amsterdam, The Netherlands, July 2007. WoTUG, IOS.

[103] N. Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4) :221–227, 1971.

[104] L. Zhao, X. Liu, Z. Liu, and Z. Qiu. Graph transformations for object-oriented refinement. *Form. Asp. Comput.*, 21(1-2) :103–131, 2009.

Résumé

La spécification de systèmes complexes est une tâche difficile qui ne peut être accomplie en une seule étape. Dans les méthodes formelles, le concept de raffinement a donné lieu à de nombreux travaux dans lesquels la preuve de la correction entre les différents états de spécifications joue un rôle important. L'activité de refactoring consiste à restructurer un modèle en vue d'améliorer certains facteurs de qualité, tout en préservant la cohérence de ce modèle.

Cette thèse préconise l'utilisation de deux techniques de raffinement et de refactoring afin d'établir des modèles UML de qualité c'est-à-dire corrects par construction, extensibles, réutilisables et efficaces. En outre, elle plaide en faveur de l'utilisation conjointe UML (semi-formel) et B, Event-B et CSP (formels).

Les principales contributions de cette thèse sont : proposition des patterns de raffinement de diagrammes de classes UML/OCL afin de guider le concepteur lors de la modélisation statique de son application et proposition des schémas de refactoring des modèles UML décrits par des diagrammes de classes, des contraintes OCL et des diagrammes d'états-transitions afin d'aider le concepteur lors de la restructuration des modèles UML.

Mots-Clés :

raffinement, refactoring, préservation du comportement, correction du raffinement, modélisation, patterns de raffinement, schémas de refactoring.

Abstract

Specifying complex systems is a difficult task which cannot be done in one step. In the framework of formal methods, the refinement is a key feature to incrementally develop more and more detailed models, preserving correctness in each step. The refactoring activity consists in restructuring a model in order to improve its quality, preserving the consistency of this model.

This thesis advocates the use of both refinement and refactoring technics in order to build a high quality UML models ie correct by construction, scalable, reusable and efficient. It also helps in the joint use of UML (semi-formal) and B, Event-B and CSP (formal).

The main contributions of this thesis are : First, a proposal of the refinement patterns of the UML/OCL class diagrams to guide the designer during the static modeling application ; Second, a proposal of the model refactoring patterns described by the UML class diagrams, OCL constraints and state diagrams, in order to assist the designer during the restructuring of UML models.

Keywords :

refinement, refactoring, preserving behavior, correction of refinement, modeling, refinement patterns, refactoring patterns.